Standardizing DEVS Simulation Middleware

Gabriel A. Wainer, Khaldoon Al-Zoubi, Olivier Dalle, David R.C. Hill, Saurabh Mittal, José L. Risco Martín, Hessam Sarjoughian, Luc Touraille, Mamadou K. Traoré, Bernard P. Zeigler.

1 Introduction

As discussed earlier in Chapter 16, there are two different interoperability objectives that one must consider when standardizing DEVS environments: **(1)** *Standardizing DEVS model representation* to allow a platform-independent DEVS model representation that can be executed by any DEVS-based simulation tool. **(2)** *Standardizing Interoperability Middleware* to interface different simulation environments and allow synchronization for the same simulation run across a distributed network regardless of their model representation.

This chapter focuses on approaches by different groups to *standardize the simulation middleware*. All of the implementations discussed in this chapter are based on a Service Oriented Architecture (SOA) design, which employs the concept of deploying services so that they can be invoked by clients. This concept is applied in CORBA and SOAP/REST Web-services.

This kind of middleware is of interest in order to overcome current distributed simulation challenges and to meet future expectations in this area [1][2]. A standardized DEVS simulation protocol should enable different DEVS implementations to simulate the same DEVS model hierarchy partitioned between various DEVS engines in distributed fashion. Moreover, each DEVS domain in this distributed system should be able to execute its legacy models and, thus, perform distributed simulation experiments between different heterogeneous models and engines. The middleware designs showed in this chapter offer simulation resources as a set of services that can be invoked by simulators, and where these simulators act as peers (i.e., clients and servers at the same time) to each other to synchronize a simulation session.

The designs presented in this chapter show that different methods can be employed for DEVS simulation synchronization. One option is to expose the actual DEVS simulators and coordinators (and to take care of synchronization at that level). A second option considers placing the DEVS coordinators and simulators of a domain behind a wrapper so that this wrapper routes all information to the appropriate coordinator/simulator. Further, the synchronization protocols can pass all simulation data between distributed entities in the form of procedure parameters (relying on CORBA or SOAP-based Web-services), or can pass simulation data between distributed entities in the form of XML messages.

2 DEVS/SOA

As discussed in the previous chapter, DEVS/SOA is a DEVS simulation platform implemented in both Java and .NET programming environments. It manages messages in XML format, which enables the publication of web services (including the publishing of simulators and even atomic or coupled models as web services). As discussed earlier, there are now many libraries for expressing DEVS models around the world. All have efficient implementations for executing the DEVS protocol and most of them provide the advantages of Object Orientation (such as encapsulation, inheritance, and polymorphism). In order to simplify notation, we use the term DEVS/JAVA to denote a DEVS library implemented in Java (for example DEVSJAVA or xDEVS) and DEVS/NET to denote a DEVS library implemented in any language supported by .NET (for example ADEVS in C, CD++ in C++ and DEVS.NET in C#).

2.1 DEVS/SOA simulation layer

The DEVS/SOA simulator integrates the common DEVS/SOA atomic model interface pictured at the bottom of Figure 1 to support multi-framework interoperability. This modeling interface plus a customized adapter allow the aggregation of DEVS/JAVA native models (xDEVS, MicroSim, DEVSJAVA, etc) in parallel with DEVS/NET based models (ADEVS, CD++, DEVS.NET, etc). A noteworthy achievement of a DEVS modeling common interface is the presence of an embedded "translator" that is in charge of mapping incoming messages to XML and outgoing messages the other way around.

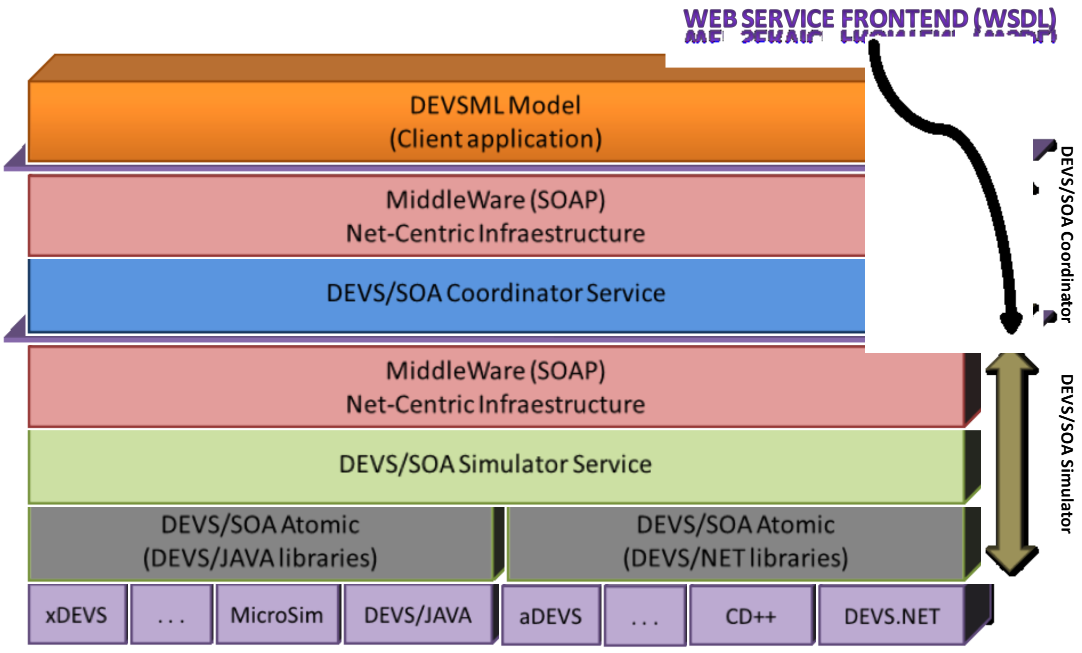


Figure 1. DEVS/SOA architecture

The communication between a given DEVS/SOA atomic models and its corresponding Simulator are detailed in an XML format. Therefore, the essential duty of the aforementioned DEVS modeling interface translator is to bind the platform independent XML data types to the message objects exchanged between the DEVS/SOA simulator and the DEVS atomic models.

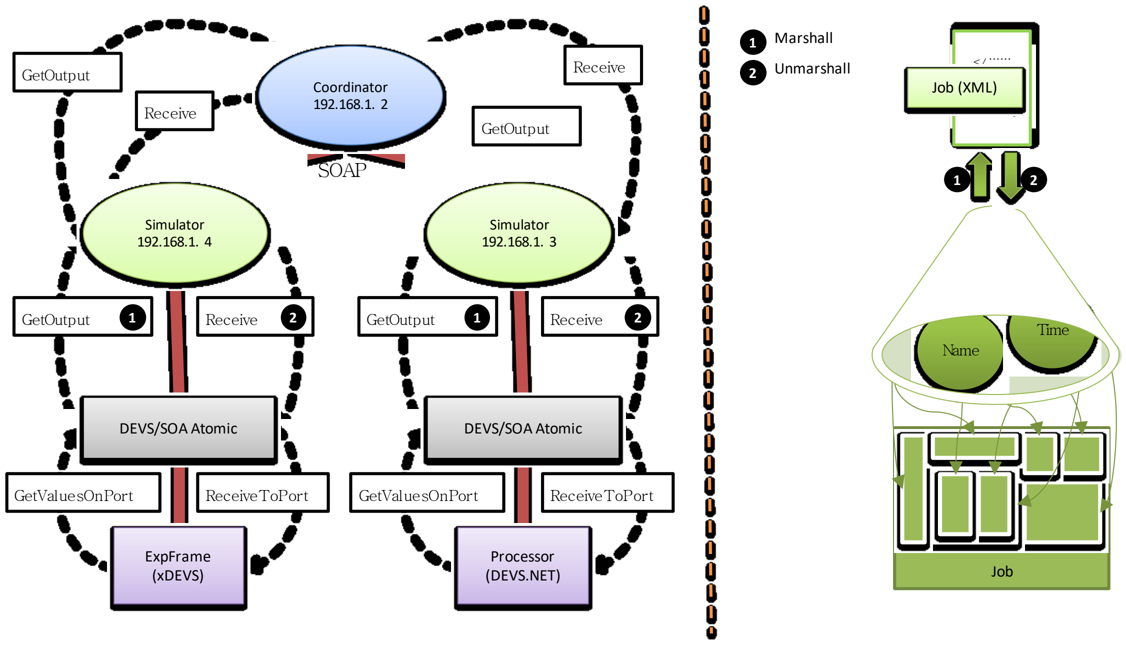


Figure 2. DEVS/SOA XML Message Mapping (centralized in root coordinator)

Figure 2 illustrates the message passing process of the coupled model consisting of an Experimental Frame (*ExpFrame,* which generates jobs to be processed, and it computes the performance obtained when processing those jobs) and a *Processor* (which models the activity of a server that we want to simulate) using DEVS/SOA . Dashed lines represent the model dataflow

i. The *ExpRrame* DEVS/SOA atomic model requests an output message from a platform specific model (xDEVS or DEVS.NET) using *GetValuesOnPort* and *GetOutput*;
ii. a simulator requests an XML formatted output message from the DEVS/SOA atomic model using *GetOutput*;
iii. the Coordinator requests an XML formatted output message from a Simulator and sends it to another simulator through the corresponding ports coupling;
iv. a simulator receives an XML formatted message using *Receive*;
v. the DEVS/SOA atomic model receives a platform specific message;
vi. the platform specific *Processor* model receives the value of the message.

It should be mentioned that each operation embodies a specific input or output port. The right-hand side of Figure 2 symbolizes the translation procedure that converts a DEVS message (DEVS model input or output) built from class objects into an XML based document (marshall), and vice versa (unmarshall). The corresponding steps (1) and (2) can be seen on the left-hand side of the figure. Because of this XML serialization, DEVS model inputs and outputs are sent through the web within SOAP.

Figure 3 depicts the DEVS/SOA simulator interface. The DEVS/SOA simulator is implemented as a classic DEVS simulator, with the difference that data are managed in XML. Thus, the input and output of a function can be declared using standard data types (`string` and `double`) and the publication of such interfaces as web services can be easily performed.

```
public interface Simulator{
  public void setModel(String XmlModel); // Receive the corresponding DEVSML part
  void initialize(double t); // Typical DEVS initialization function
  double getTN(); // Returns the time of the next event
  void deltfcn(double t); // Encapsulate the call to deltint, deltext and deltcon in the model
  void lambda(double t); // Output function
  String[] getOutput(String portName); // Returns the values at output ports as XML
  void receive(String portTo, String[] xmlMessage); // Receive messages in XML format
}
```

Figure 3. Simulator interface

Technically, these operations, expressed in the WSDL file of the DEVS/SOA Simulator service, wrap the classic DEVS simulation protocol with the purpose of achieving a qualified and standardized DEVS simulation.

Finally, Figure 4 shows the DEVS/SOA Coordinator interface. Likewise, as pictured in Figure 1, the Coordinator service layer together with a web service frontend invokes DEVS simulation operations against the Simulator service layer. Again, the communication is driven by the standard SOAP protocol and the operations are detailed in the Simulator service WSDL file allowing platform-independent interactions. Moreover, assuming that a coordinator is already provided by the client application with the root DEVS model based on DEVSML syntax, the initial coordination task comprehends parsing the DEVSML document looking for DEVS atomic or coupled models and their connections. Each atomic or coupled model supplies the coordinator with the associate remote simulator location in order to enable communications among them. The connections among the models let the coordinator be aware of the entire distributed circuit to carry out proper performance analysis. After the communication between the coordinator (coordination service layer) and the respective simulators (simulation service layer) is set up, the coordinator proceeds to activate and initialize all simulators by feeding them with their proper DEVSML based model definition. Next, the coordinator continues with the simulation initialization within the DEVS/SOA simulation protocol.

```
public interface Coordinator{
  public void setModel(String XmlModel); // Initialize the model with the DEVSML file
  public String[] simulate(int numIterations); // Simulate a given number of cycles
}
```

Figure 4. DEVS/SOA Coordinator interface

As Figure 1 illustrates, the client application requires a web service frontend to provide access to the Coordinator service layer. The baseline communication is settled within the SOAP protocol and the operations are detailed in the Coordinator WSDL file. This arrangement provides platform independent interactions. Assembled with a web service framework, the client application acquires the capacity to look over the web for the coordinator service hosted at the remote location stated at the DEVSML root model (see **Error! Reference source not found.**). Once the connection is established, the web client application activates the remote coordinator by supplying a model specified using DEVSML, and it subsequently executes the simulation according to specific parameters. As soon as the simulation completes, the client application receives a summary of the overall simulation performance and cumulative results. The aforementioned operations are specified inside the DEVS/SOA Coordinator WSDL file.

When composing a DEVS/SOA model, one important simplification, which is not mandatory, is that every coupled model can work as an atomic model [3]. For example, Figure 5 shows that the root model can be simulated in two different ways: with each atomic model on a different computer (as seen on the left-hand side of the figure), or with the Experimental Frame on one computer (i.e., the coupled model *ExpFrame* shown on the right-hand side figure, which is the composition of *Generator* and *Transducer* atomic models on the left) and the *Processor* on another (as seen on the right-hand side).

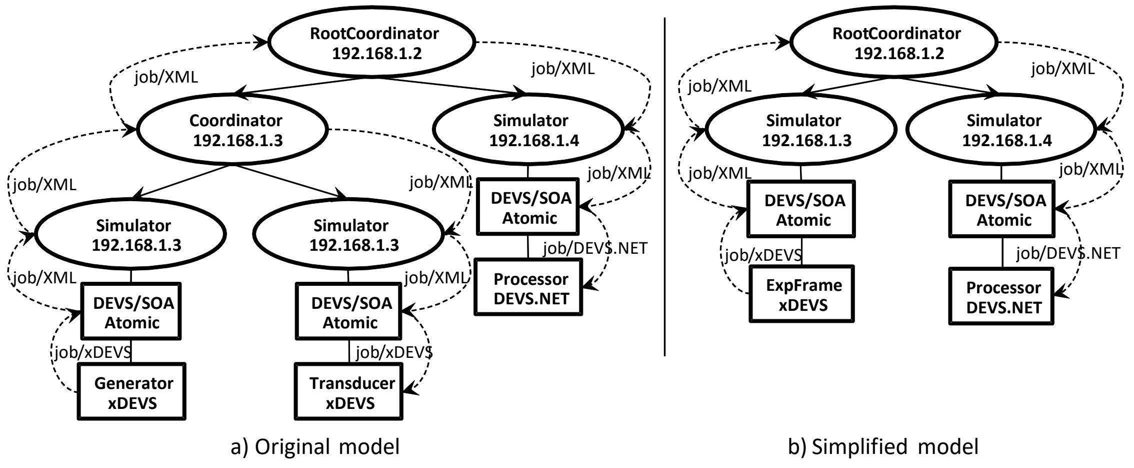


Figure 5. Simplification of DEVS models

2.2 DEVS/SOA application

As stated above, to build a web service-based infrastructure, both the coordinator and simulators are published as web services. Thus, we have two Coordinators (DEVS/SOA Coordinator and stub) and two Simulators (DEVS/SOA Simulator and stub).

On the server side, all the Coordinators and the Simulators must be placed in a directory accessible by the selected web server (Apache Tomcat and Axis2 to allow web service development, or MS Internet Information Server with .NET), as well as all the models to share by every particular server and their corresponding DEVS libraries to allow interoperability. For example, for a particular case where Apache Tomcat and Axis 2 is being used, the server side content comprises:

- TOMCAT_HOME/webapps/axis2/WEB-INF/services:
  - Simulator.aar: Axis2 archived web service and stub of DEVS/SOA Simulator.
  - Coordinator.aar: Axis2 archived web service and stub of DEVS/SOA Coordinator.
- TOMCAT_HOME/shared/lib:
  - xdevs.jar: xDevs M&S API.
  - devsjava.jar: DESJAVA M&S API.
  - Several xDEVS and DEVSJAVA models.

Since both web services and stubs are installed in the server, they are able to act both as a simulation services and as clients to another server. For a fully Symmetrical Service architecture design, refer to [3]. By this symmetrical design, when a distributed DEVS/SOA is executed on multiple machines (say on a server farm), each machine can serve as a DEVS coordinator as well as a DEVS simulator facilitating the recursive DEVS hierarchical design principles.

Similarly, the client application contains just two executable classes (the DEVS/SOA coordinator and simulator stubs), namely *DevsSoaSimulator.jar* and *DevsSoaCoordinator.jar*. A graphical UI client for DEVS/SOA was implemented in [3]. To execute the model, the user must compose an XML file describing the distributed architecture (as depicted in **Error! Reference source not found.**) and run the simulation as follows (in the Java client application):

```
// xmlCoupledModelAsString is a DEVSML file, which has been previously loaded
CoordinatorServiceInterface service = new
        CoordinatorService("http://localhost:8080/devsoa/Coordinator",true);
service.setCoupledModel(xmlCoupledModelAsString);
String[] response = service.simulate(numIterations);
```

or using the client Coordinator as a executable file:

```
java -jar DevsSoaCoordinator.jar  -file=ef-p.xml -numIter=110
```

To this end, the software required in a DEVS/SOA Java Axis2 client application is:

- Java Development Kit (JDK): Version 1.4 or later.
- Axis2: Versión 1.4.1 or later standard binary distribution.

To conclude, the purpose of DEVS/SOA is to support distributed simulation and interoperability. The distributed arrangement of the ongoing simulations enables the user to partition the original model and distribute it between several processors or cores in the same computer, among several computers connected through Internet, or between both. In the same manner, the user is able to compose a complex DEVS model using different submodels that may be hosted on different computers. Furthermore, the interoperability quality allows the user to compose a complex DEVS model with different submodels implemented using distinct DEVS M&S frameworks or libraries. Every computer involved in the M&S process must act as a repository providing DEVS models (implemented for a specific DEVS library), as a server (providing Simulators and Coordinators as web services), and as a client (communicating with the Coordinator on any machine).

3 Distributed DEVS Simulation Protocol (DDSP)

In [6], a flexible and scalable XML-based message-oriented mechanism was developed with the goal to allow interoperability between different DEVS implementations. The main objective of the protocol is to enable different DEVS implementations to interface and coordinate among each other to simulate the same model structure across their diverse domains. To do so, the developed simulation protocol uses SOAP-based Web-Services technology as the communication framework to exchange control and standardized simulation XML messages.

3.1 Introduction to DDSP

The idea of DDSP is to provide interoperability with minimum design changes to each DEVS implementation, mainly by hiding the detailed implementation behind a *wrapper* (i.e., a SOAP-engine port) and focusing only on the exchanged XML messages. This point is important because various DEVS implementations are different (even if they are implemented with the same programming language). Interfacing the same tool implementation in a parallel/distributed environment can require weeks of programming and debugging by programmers who understand that tool implementation very well. One cannot expect interfacing different DEVS implementations that were developed by different independent teams to be internally structured the same. Further, different teams have extended their tools over the years to accommodate different optimizing algorithms or modeling technique. For example, the Cell-DEVS extension [7] allows for representing each cell in the cell space as a DEVS model that is only activated when it receives external inputs from its neighboring cells. CD++ [8] provides an environment for DEVS and Cell-DEVS models. However, it extends the software design into different C++ classes to implement both DEVS and Cell-DEVS. Figure 6 shows a fragment of the design of distributed CD++ (DCD++). The figure clearly shows that this version of the simulation software uses a specific implementation to simulate Cell-DEVS models (using the *AtomicCell* and *CoupledCell* methods). The various DEVS versions have in common that coordinators synchronize coupled models, and simulators execute atomic models where the simulation is advanced according to the DEVS theory rules. However, each DEVS version provides different software design and implementations. In fact, the internal implementation for a DEVS coordinator, for instance, can vary between parallel, standalone, and distributed for the same DEVS tool because each of these DEVS coordinators can use a different algorithm to coordinate its children.

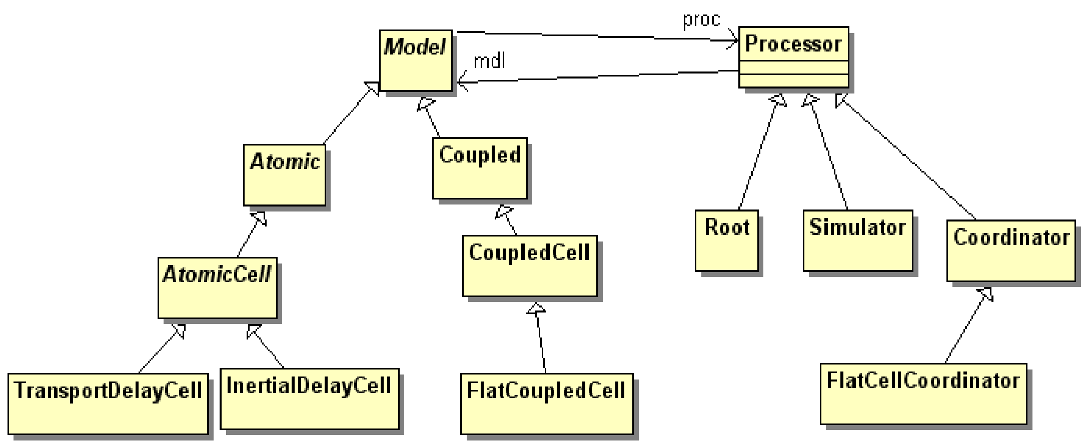


Figure 6: Snippet of the DCD++ Model/Simulators hierarchy

Hiding the internal implementation increases the protocol chances of success since various DEVS teams are not expected to change their internal design and software implementation in a way that jeopardizes their existing DEVS tools integrity. Further, they can have full freedom to extend/change their own internal software implementation. This is because a DEVS tool is always in conformance with the standards as soon as it handles the standardized XML simulation messages correctly.

The DEVS Distributed Simulation Protocol also supports interfacing DEVS legacy models. This allows a modeler to assemble and simulate heterogeneous DEVS models that were originally intended to run in a specific DEVS environment. The protocol expects each tool to react to expected messages (with a standardized format constructed as XML documents) in order to correctly synchronize and carry out simulation of the overall model (which is spread over different domains). Having a message-oriented protocol that hides implementation detail (behind wrappers) and focuses only on the information needed (within exchanged messages) has many advantages. To summarize a few:

- Maintainability: Protocol changes are only applied to the protocol messages rather than to every DEVS implementation,
- Scalability: the contents of the XML message being exchanged are easy to add (or remove) by adding (or removing) the Remote Procedure Calls (RPCs) interface, and
- Testing: local testing is easy to perform by each group before executing integration testing between different DEVS domains. The general rule is that if a DEVS implementation can interface with itself via exchanging XML standardized messages, it should be able to interface with a different DEVS implementation using the same standardized messages (in case both implementations conform to the standardized messages and rules).

3.2 Web-Service DEVS Wrapper

Each DEVS implementation should execute its own specific models. This requirement enables both the utilization of hundreds of legacy models for each DEVS tool as well as the integration with other models in different DEVS tools. This requirement is essential to make DEVS standards attainable because we can never expect all legacy models to be rewritten. This requirement is satisfied by enclosing all models in a single outer model and making each DEVS tool responsible for simulating its specific models. For example, in Figure 7, *Coupled1* can be in DEVSJAVA while *Coupled2* can be in DCD++. In this case, the main DEVS domain owns the Root coordinators and simulates both heterogeneous models, giving the impression of simulating a single distributed heterogeneous DEVS model.

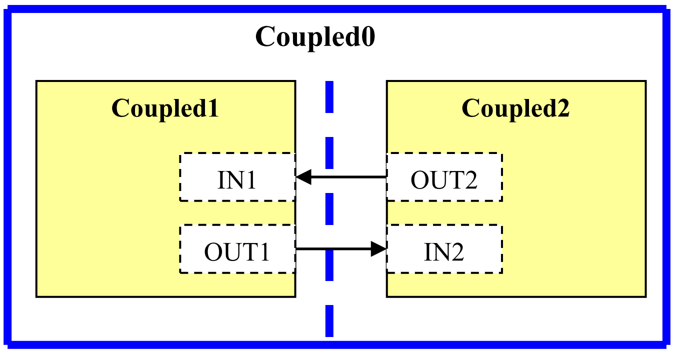


Figure 7: Coupled model partitioned across DEVS Domains

As shown in Figure 7, both coupled models interface without worrying about how the other implementation performs the simulation internally. Therefore, coupled models are viewed as black boxes with input/output ports. However, it is still possible for a DEVS implementation to know more details about the model structure in other domains, depending on the level of detail that is made available to the domains when the structure is distributed, as described in the XML structure document.

The concept is to have each DEVS implementation use a single communication entry point, implemented as a DEVS-Wrapper (Figure 8). Therefore, a coupled model may physically be partitioned among different machines within a DEVS implementation domain, but other DEVS domains "believe" the coupled model actually exist on the machine through which they communicate with the coupled model. The DEVS-Wrapper is actually a Web-service port that exposes a number of stubs (interfaces), allowing other DEVS domains to invoke them in an RPC-style mechanism, as discussed in the communication framework section. Therefore, the DEVS-wrapper interfaces are described in WSDL document allowing domains to construct the necessary stubs.

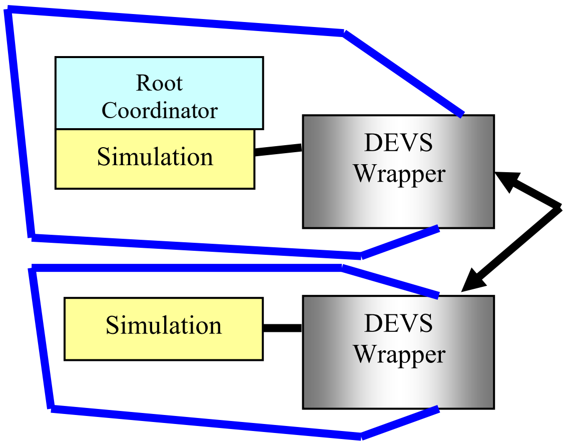


Figure 8: Connecting Two DEVS Domains

This requirement simplifies the coordination via one single Web-service port described in a single WSDL document. This approach does not require the DEVS-wrapper operations to be exposed in a separate Web-service port or to be merged with other existing ports, but it leaves this decision to individual teams since it is a software design issue rather than a standardization one. Further, the DEVS-wrapper port needs to be described by a WSDL document where other domains can use the standardized interfaces. The DEVS-Wrapper component is expected to perform the following tasks:

- Translate incoming standardized simulation messages to specific domain simulation messages.

- Transmit simulation messages to other DEVS domains according to the DEVS standards.
- Route incoming simulation messages to the correct models/ports within its domain.

The protocol should minimize its dependency on the communication framework, requiring few (or no) changes to the standardized simulation messages if one needs to move the simulation protocol to other communication engines in the future. In DCD++, this requirement is implemented by sending all simulation messages as XML documents in SOAP attachments via exposed DEVS-wrapper interfaces. Therefore, if the communication mechanism changes, those same XML documents can still be transmitted without changes.

Although SOAP messages are standardized XML documents, they are hidden from the Web-service applications and are only seen as programming stubs with input/output parameters. Consequently, a DEVS standard would need to define its own XML messages. Eventually, the standard is realized by the programming code, hence if it only relies on SOAP XML messages, all DEVS interfaces become a matter of simply gluing programming remote procedures together. This makes the standard extremely sensitive to changes since programming language procedures by nature are sensitive to many factors such as the order, type, and number of parameters that are passed into them. Therefore, the standard can never claim to be using XML messages for communication because the SOAP messages are only handled by the SOAP engine (i.e., the communication layer below a Web-service application that is responsible for SOAP messages handling), and this engine is enabled to invoke the appropriate service stub.

The DDSP implementation uses Web-services technology to transfer standardized simulation messages between different domains. All messages are transmitted through SOAP/HTTP engines, hence wrapped within SOAP and HTTP envelopes, as shown in Figure 9, where a DEVS Wrapper communicates with other DEVS domains by invoking the deployed-service stubs in a remote procedure call style. Simulation messages are passed into those stubs as SOAP attachments in the form of XML documents.

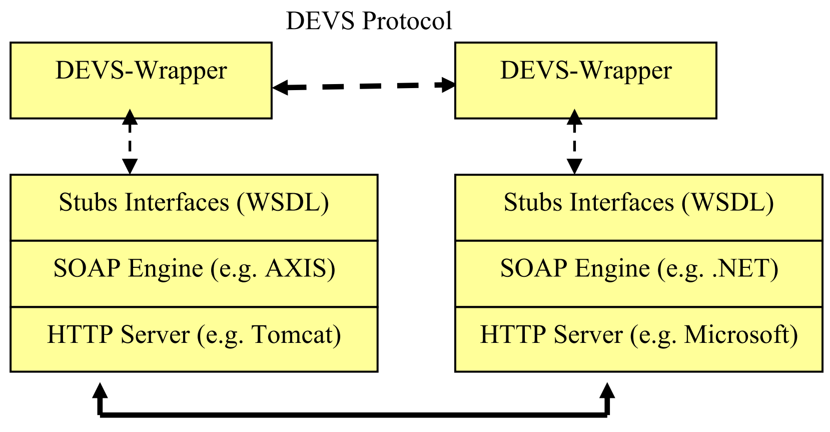


Figure 9: Connecting Domains using Web-Services

Stubs are constructed from the deployed WSDL document by the service provider (other DEVS domains). To support SOAP-based web-services, each DEVS domain should have the following engines:

- HTTP Server (e.g., Tomcat [9]).
- SOAP Engine (e.g., AXIS[10]).
- XML parser: the proposed protocol is not making any assumptions regarding a specific XML parser.

A DEVS-Wrapper is actually a Web-service port connected to the AXIS SOAP engine that encapsulates the necessary operations, allowing different DEVS domains to communicate with each other and clients to activate DEVS simulation services. The standard here assumes that individual DEVS domains provide their own interfaces for clients allowing them to invoke the services of the individual DEVS domains (such as user authentication, file submission, etc.). The remaining discussion in this section focuses only on the communication between various DEVS domains.

Figure 10 shows the AXIS Java interface for the DEVS-wrapper operations. They are described in WSDL documents, so that other AXIS Java communication classes may be constructed by various DEVS domains (these operations are stubs that allow DEVS domains to communicate with each other).

```
import javax.activation.DataHandler;

public interface DEVSWrapperType extends java.rmi.Remote {

  public DataHandler retrieveResultFile(int SupportiveSession);
  public boolean startSimulation(int SupportiveSession);
  public boolean isSimRunning(int session);
  public boolean StopSimulation(int session);
  public boolean setDEVSXML(int session, String filename,
             DataHandler file);
  public boolean deleteSession(int SupportiveSession);
  public int createSupportiveSession(int MainSession);
}
```

Figure 10: DEVS-Wrapper AXIS-Port Services

The DEVS-Wrapper services in Figure 10 can be summarized as follows:

- *retrieveResultFile* : Is used to retrieve simulation result files from a support DEVS domain.
    - Input: The support DEVS domain session number.
    - Output: The results file.
- *startSimulation* : It starts the simulation on a support DEVS domain. In this case the simulation engine starts and waits for the first simulation message from the main DEVS domain.
    - Input: The support DEVS domain session number.
    - Output: True on success; false otherwise.
- *isSimRunning* : It checks if a simulation is running on a DEVS domain.
    - Input: The support DEVS domain session number.
    - Output: True if simulation is running. Otherwise false.
- *StopSimulation* : It stops the simulation normally on a support DEVS domain.
    - Input: The support DEVS domain session number.
    - Output: True on success. Otherwise false.
- *setDEVSXML* : It sends an XML document to a DEVS domain. XML document is either configuration file or a simulation message.
    - Input:
        - A DEVS domain session number.
        - XML document file name.
        - The actual XML file.
    - Output: True on success; false otherwise.
- *deleteSession* : It deletes a simulation session on a support DEVS domain. This operation releases all resources used for this session.
    - Input: The support DEVS domain session number.
    - Output: True on success. Otherwise false.
- *createSupportiveSession* : It creates a simulation session on a support DEVS domain. This operation allocates all necessary resources needed for this session. The support and main DEVS domains are accepted to bind both session numbers together. This gives each DEVS domain the freedom to allocate its session numbers without worrying about possible conflict with other DEVS domain session numbers.

- Input: The Main DEVS domain session number.
- Output: The support session number or -1 on failure.

Initially, the main DEVS domain creates support simulation sessions and establishes full connections with all support domains. Each DEVS domain should know each participant DEVS-Wrapper port URI and its associated session number for all participant DEVS domains. This allows DEVS domains to have multiple concurrent simulation sessions using the same Web-service port.

The main DEVS domain opens a session with all relevant support domains, and it broadcasts this information to support domains in one XML document (using the method setDEVSXML). The simulation session document contains the main domain session number (which all support DEVS domains know upon invoking *createSupportiveSession* by the main domain), and the support URIs paired with their session number, as shown in Figure 11

```
<Sessions ver="1.0">
   <Session Type="Main">
      <Number>123</Number>
      <URI>http://…</URI>
   </Session>
   <Session Type="Supportive">
      <Number>1000</Number>
      <URI>http://…</URI>
   </Session>
   …
</Sessions>
```

Figure 11: Domain-Simulation Sessions XML Binding Document Example

After receiving the XML document in Figure 11, each domain should be able to send messages on a session to any other domain.

The main principle followed here is to enclose all various DEVS domain heterogeneous models within a single coupled model. This simplifies the simulation coordination, as each DEVS domain hides its internal activities and coordinates with other DEVS domains. This approach has been adopted in the variant of DEVS/SOA developed by [12] [13] which supports interoperability across different web service platforms using the XML namespace concept to be described next.

3.3 Model Structure XML Document

The Model structure XML document (shown in Figure 12) is initially submitted by the modeler to the main DEVS domain to describe how the overall model is structured so that each DEVS version can identify which models belong to its domain. Further, from this document, the main machine can identify the participant support domains.

The model structure document contains enough information to allow different domains to create local models, coordinators (i.e., coupled model processor), and simulators (i.e., atomic model processor). It also includes data on how they will relate to other models in different domains.

The main DEVS domain must pass this document before it starts the simulation (i.e., before invoking service *startSimulation* on support domains). The model structure document contains the following information (see Figure 12):

- model names,
- model type (coupled/atomic),
- model input/output ports,
- coupled models internal submodels and their ports connections,

- models domain URIs, and
- coupled models synchronization algorithms used (e.g., the Head/Proxy Coordinator discussed earlier).

The DEVS models hierarchy can easily be mapped into this XML document. For example, assume two models connected with each other as in Figure 7 (two DEVS domains where each model is specific to its domain implementation). In this case, the two models would be enclosed within an outer model (*Coupled0*), resulting in the XML document shown in Figure 12. This XML document also serves as an agreement contract between various implementations on the used synchronization schemes. For example, the coordination scheme that is used can be set by the COUPLED_SYNC field to simulate a distributed coupled model across various domains. In this way, the standard can easily adopt any new schemes that may appear in the future.

```
<MODEL_STRUCTURE ver=”1.0”>
 <COUPLED_SYNC>
    <scheme ver=”1.0”>HeadProxy</scheme>
 </COUPLED_SYNC>
 <Models>
   <Model Type=”Coupled”>
     <Name> Coupled0 </Name>
       <Components>
        <Name Type=”Coupled”>Coupled1</Name>
        <Name Type=”Coupled”>Coupled2</Name>
       </Components>
     <URI>http://… </URI>
     <LINKS>
        <LINK>
          <FROM>
            <Component>Coupled1</Component>
           <Port>OUT1</Port>
          </FROM>
          <TO>
            <Component>Coupled2</Component>
           <Port>IN2</Port>
          </TO>
        </LINK>
        …
     </LINKS>
     …
   </Model>
   <Model Type=”Coupled”>
     <Name> Coupled1 </Name>
     <Ports>
       <Port Type=”in”>IN1</Port>
       <Port Type=”out”>OUT1</Port>
     </Ports>
     <URI>http://… </URI>
     …
   </Model>
   <Model Type=”Coupled”>
     <Name> Coupled2 </Name>
     <Ports>
       <Port Type=”in”>IN2</Port>
       <Port Type=”out”>OUT2</Port>
     </Ports>
     <URI>http://… </URI>
     …
   </Model>

  </Models>
    …
</MODEL_STRUCTURE>
```

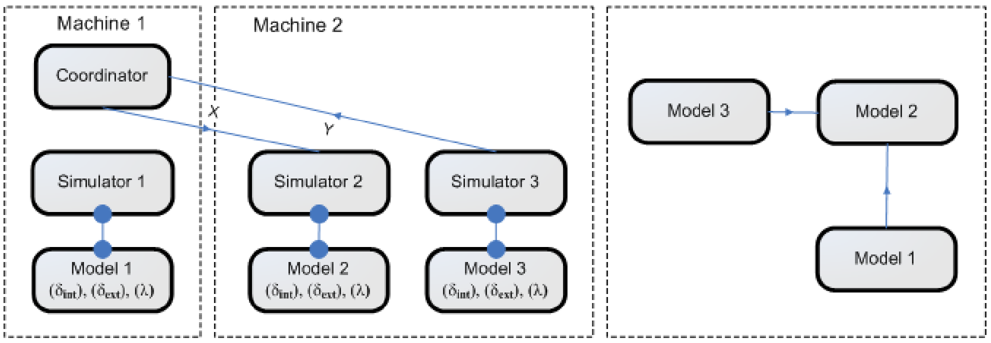
Figure 12: XML Model Structure Document Example

The simplest way of structuring a DEVS model is to have one coupled model at each of the DEVS domains connected to one other via their input/output ports, where each coupled model views the coupled models in other domains as “black boxes”. Even with this simple scenario, another top-coupled model should then be created to wrap all coupled models across various domains. Therefore, there will be at least one coupled model partitioned across DEVS domains. By having one Coordinator simulating a single coupled model distributed over the network, it becomes a performance bottleneck (because of the number of messages exchanged between the parent Coordinator and its children). For this reason, we propose to adopt a Head/Proxy Coordinator structure. Other algorithms can be adopted if they are scalable and it is needed. The Head/Proxy extends the coordinator concept, as follows:

- Head Coordinator: it is in charge of simulating the entire coupled model. It coordinates the internal models that exist in its domain and (via Proxy Coordinators) the other internal models that exist in other domains.
- Proxy Coordinator: it acts as an agent on behalf of the Head Coordinator to simulate the internal submodels of a coupled model that exist in its DEVS domain. A Proxy Coordinator passes *all* the unknown messages to its Head Coordinator; however, a Proxy Coordinator usually passes *only one* message to its head Coordinator on behalf of the coupled model internal partitions its domain (which is possibly distributed among different machines in the same domain).

Note that the domain that owns the first internal model as structured in the XML model structure document will create the Head coordinator for the parent and other domains will create proxy coordinators. For example, the domain that owns Coupled 1 in Figure 13 creates the Head coordinator for the outer model Coupled 0 while the others create Proxy coordinators. Note further that the main domain always owns the Root coordinator and drives the overall simulation. This is not related to the Head/Proxy algorithm. Therefore, it is possible for the top-level model to have its Head coordinator in a support domain since this depends on how the modeler described it in the XML structure document. However, the modeler should structure the top-level model to have its Head coordinator in the main DEVS domain to be near the Root coordinator in the main DEVS domain for performance reasons.

Using a single Coordinator adds unnecessary overhead if two child simulators want to exchange messages and are running on a machine different from their coordinator. As shown in Figure 13, Simulator 3 sends an output message that is to be translated into external message to Simulator 2, which resides on the same machine as its sibling Simulator 3. Therefore, sending this message to the coordinator, it ends up being transmitted twice as remote messages because the coordinator is running on a machine different from the source and destination of the message.


Machine 1
Machine 2
Coordinator
X
Y
Simulator 1
Simulator 2
Simulator 3
Model 1 $(\delta_{int}), (\delta_{ext}), (\lambda)$
Model 2 $(\delta_{int}), (\delta_{ext}), (\lambda)$
Model 3 $(\delta_{int}), (\delta_{ext}), (\lambda)$
Model 3
Model 2
Model 1
Figure 13: superfluous messages exchange in distributed simulation

The above-described problem could have been avoided if there is a coordinator responsible for message routing locally in each machine. Therefore, having a proxy coordinator on Machine 2 (in Figure 13) causes the message from Simulator 3 to Simulator 2 to be sent locally, thereby improving the performance of the simulator. Further, one DONE message is sent to the Head

Coordinator (on Machine 1) from the Proxy Coordinator (on Machine 2) on behalf of Simulator 2 and Simulator 3.

3.4 Format and Content of Messages

Simulation messages are constructed as XML documents and sent to other domains as SOAP attachments (using the AXIS stub setDEVSXML where the receiver session number is one of its parameters). Therefore, any changes in the simulation messages will be made to the message XML document rather than to the input/output parameters of the AXIS stub, thereby increasing scalability and portability.

The simulation message types are listed as follow (note that the specific simulation phases are discussed in the next section):

- *Init (I)*: Simulation starts when the Init message is passed to the top-coupled model Coordinator, which then pushes it downward to its children.
- *Collect (@)*: it is used to start the collection phase. The top model Coordinator propagates it downward.
- *Internal (*)*: it is used to start the transition phase.
- *Done (D)*: it is used by Coordinators to identify which children need to be simulated at this phase. It is used by the Root Coordinator to advance the simulation time and switch simulation phases.
- *External Message (X)*: Messages from the environment, or as a result of output messages.
- *Output Message (Y)*: Generated during the collection phase.

Table 1 shows all possible fields in an XML message document. All fields are not required to be sent with each message type. However, if the sender chooses to send all fields in a message the required fields (based on the message type) are the only ones that the receiver must consider. The Next-change-time element is used by DONE messages to inform the parent Coordinator about the next expected internal change (in turn, the parent Coordinator passes a DONE message to its parent including the minimum next change of its model children, whether local or in other domains). Eventually only one DONE message is received by the Root Coordinator (in the main domain), which then starts another simulation phase. All Coordinators (including Root) use this message to know which children branches should be involved in each simulation cycle. This prevents many unnecessary message transmissions across the network.

Table 1: Simulation Message XML Fields

Element	Format	Allowed Values	Comments

MessageType	Character	I, @, D, X, Y, *	I = INIT,
			@ = Collect,
			D = Done,
			X = External,
			Y = Output,
			* = Internal.

Time	String Hours:Minutes:Secs:mSec	Numbers separated by colon (“:”)	Example: 08:50:00:00
SrcModel	String	Known Model Name	Source Model
DestModel	String	Known Model Name	Destination Model
Port	String	Known Port Name	Destination Port.
Value	C++/Java double	N/A	Mandatory only for External and Output messages.

NextChange	See Time element	See Time element	Next Change Time. Mandatory only for DONE messages.
IsFromProxy	Java boolean	True or False	Mandatory only for DONE messages if Head/Proxy Algorithm is used. This allows Head to synchronize its Proxies.

Figure 14 shows an example of an INIT message from model *Coupled0* to port IN of model *Coupled2*. In this example, the sender domain chose to send all fields; however, the receiver must only use the fields relevant to the INIT message.

```
<Message ver=”1.0”>
   <MessageType>I</MessageType>
   <Time>00:00:00:00</Time>
   < SrcModel>Coupled0</SrcModel>
   < DestModel>Coupled2</DestModel>
   <Port>IN</Port>
   <Value>-1.0</Value>
   <NextChange>00:00:00:00</NextChange>
   <IsFromProxy>false</IsFromProxy>
</Message>
```

Figure 14: Initialization Simulation Message with All Fields Example

The developed protocol in this section has simplified the simulation by wrapping all distributed models across various DEVS domains in one single coupled model; hence it becomes the responsibility of coupled Coordinators to locate their children (i.e., internal models) in order to pass them the needed simulation messages (perhaps by having a database that stores each model description along with its domain URI). Further, simulation messages can be specific to a certain domain when they are exchanged within the domain itself, but when they must exit to another domain, the DEVS-Wrapper (discussed in the communication section) translates them to the standardized XML message documents and passes them as SOAP attachments using the AXIS stub setDEVSXML to other domains DEVS-Wrappers. For example, as shown in Figure 15 a DEVS domain does not need to use the standards within its domain. However, when a message must travel to another domain, it has to be translated first to the standard format so that it can cross the DEVS protocol bridge.

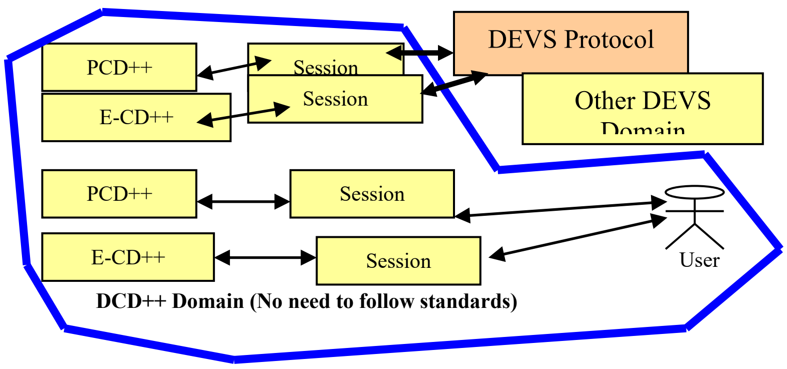


Figure 15: An Internal Look of a DEVS Domain

4 Shared Abstract Model

In [14], an approach for on-line model-based interoperability named the *Shared Abstract Model* (SAM) was defined. The Shared Abstract Model is used to specify an *Abstract Model Interface*, shown in Figure 16. The specification of the Abstract Model Interface is based on the DEVS Atomic Model formalism. The interface is specified in terms of OMG-IDL, and it is executed using CORBA. Based on the Abstract Model Interface definition, the SAM approach requires a

Model Proxy (corresponding to the atomic stub of Figure 16). Then, models have to be wrapped into model adapters that will make their interfaces match the standard one. This approach is predominantly aimed at integrating existing legacy models or models specified in different DEVS implementations (and for simulation engines responsible for executing a non-native model implementation). Writing the model adapters for the Shared Abstract Model can be a tedious task, notably because of the use of generic messages that have to be converted before being processed by the model. Fortunately, this task of writing the two adapters (one for atomic and one for coupled models) must be done only once for each DEVS engine implementation. The two adapters can then be used for all models specified within the DEVS engine implementation.

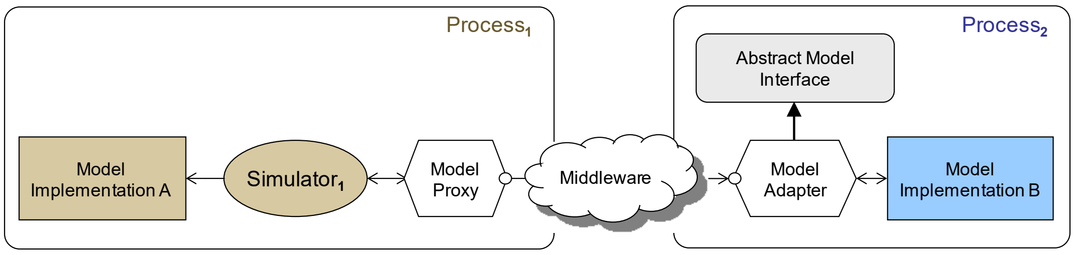


Figure 16: The Shared Abstract Model concept for a simulator executing non-native models using Proxy Model, Adapter Model, and the Abstract Interface Model

Considering the DEVS-Suite simulation engine, its simulator (Simulator$_1$ in Figure 17) can directly execute Model Implementation A (i.e., there is no need for syntactic translation from one programming language to another). However, the same simulator cannot execute the Model Implementation B, which is implemented for direct execution using the ADEVS simulator (Simulator$_2$ in Figure 17). The Model Proxy and Model Adapter are used to overcome the syntactical differences between the Java and C++ programming languages which also necessitates Process$_1$ and Process$_2$ to communicate with one another (i.e., send and receive messages). The Model Proxy translates the method invocations of the Simulator$_1$ to those of the Abstract Model Interface. The Simulator$_1$ then can use the Model Adapter to execute the transition, time advance, and output functions defined for Model Implementation B. The inheritance relationship from Model Adapter to the Abstract Model Interface allows Model Implementation B and other models that are developed in other programming languages to be uniformly executed using Simulator$_1$. Therefore, a simulator can execute its own models (e.g., the arrow from Simulator$_2$ to Model Implementation C), models that are developed for other variants of parallel DEVS simulators (e.g., the arrows from Simulator$_1$ to Model Implementation B), and any model that can be wrapped inside an atomic DEVS model but does not have its own simulator.

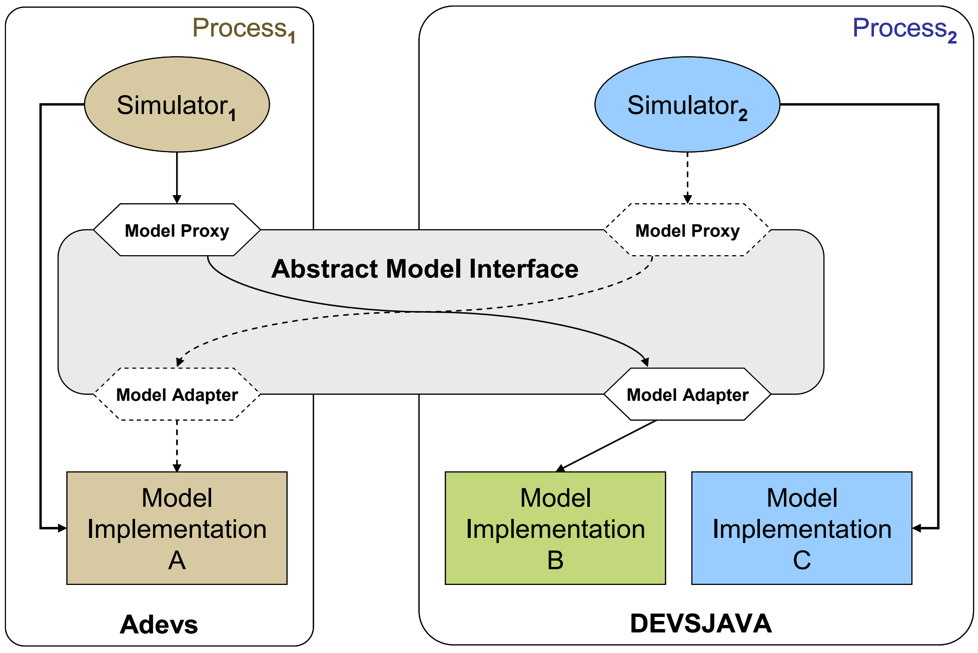


Figure 17: Example of the Shared Abstract Model for ADEVS and DEVS-Suite

The specifications for the Model Proxy and Model Adapter are straightforward since their operations have a one-to-one relationship to those that are defined for the Abstract Model Interface. However, to support message mappings between different simulation engines, it is necessary to develop modules that can translate one kind of message to another kind. Figure 18 and Figure 19 show the implementations of the Model Proxy and Model Adapter for DEVS-Suite. The listings exclude implementations for error handling.

```
interface DEVS {// OMG-idl (CORBA)
    // start of simulation
    double doInitialize()

    // time of next internal transition without input messages
    // value also returned by doInitialize and state transition functions.
    double timeAdvance()

    // produce outputs for current simulation time
    // output does not have any side effect (no state change)
    Message outputFunction()

    // internal state transition without input messages
    double internalTransition()

    // external state transition with input messages
    double externalTransition(in double e, in Message msg)

    // input message is received at the time of internal state transition
    double confluentTransition(in Message msg)
};

Message: bag { inputPort -> value }
```

Figure 18: The Abstract Model Interface specification

```
void initialize() {
    ta = devsMod.doInitialize();
    if(ta == devsBridge.DEVS.TA_INFINITY)
          passivate();
    else
          holdIn("active",ta);
}

//External Transition Function
void deltext(double e, MessageInterface x){
   MsgEntity[] msg = trans.devs2CorbaInputs(x);
   ta = devsMod.externalTransition(e, msg);
   if(ta == devsBridge.DEVS.TA_INFINITY)
          passivate();
   else
          holdIn("active",ta);
}
//Internal Transition Function
void deltint() {
   ta = devsMod.internalTransition();
   if(ta == devsBridge.DEVS.TA_INFINITY)
          passivate();
   else
          holdIn("active",ta);
}
//Confluent Transition Function
void deltcon(double e, MessageInterface x){
   MsgEntity[] msg = trans.devs2CorbaInputs(x);
   ta = devsMod.confluentTransition(msg);
   if(ta == devsBridge.DEVS.TA_INFINITY)
          passivate();
   else
          holdIn("active",ta);
}
//Output Function
MessageInterface out() {
   MsgEntity[] msg = devsMod.outputFunction();
   MessageInterface devsMsg = trans.corba2DevsOutputs(msg);
   return devsMsg;
}
```

Figure 19: Proxy implementation for the DEVS-Suite atomic model

Figure 19 shows an example where an Abstract Model Interface is defined for DEVS-Suite and ADEVS simulation engines. A pair of Model Proxy and Model Adapter is defined (shown in solid lines) such that ADEVS Model Implementation B can be simulated using the DEVS-Suite Simulator$_1$. Using the same Abstract Model Interface with another pair of Model Proxy and Model Adapter (shown in dotted lines), Model Implementation A can be simulated using the ADEVS Simulator$_2$.

In order for one simulation engine to execute a coupled model that is implemented for by another simulator, it is necessary to also account for coupled models and so Model Adapters for coupled models are needed as well. Rather than specifying the Model Adapter as the coordinator, the Model Adapter is defined based on the Abstract Interface Model (see Figure 18). Examination of the Model Adapter for DEVS-Suite coupled model enforces the coordinator's logic (see Figure 20). The comments in Figure 18 relate the association defined between the Abstract Interface Model and the Model Adapter. This formulation uses the closure under coupling property which allows treating an atomic and coupled model as a basic DEVS model component. Thus, the correctness of the simulation cycle of every DEVS coupled model remains legitimate—that is, the executor (either a simulator for atomic models or a coordinator for coupled models) guarantees the correct ordering of events and transmission of events among hierarchical models in concert with the method invocations of the Abstract Model Interface.

```
//Initialize simulator
double doInitialize(){
   coord.initialize();
   return timeAdvance();
}

//query for time to next event (1. nextTN and 2. outTN)
```

```
double timeAdvance(){
   return coord.tN() - coord.tL();
}

// ComputeIO is called (5. applyDelt)
   double internalTransition(){
   coord.DeltFunc(coord.tN(), [empty set]);
   return timeAdvance();
}

// ComputeIO is not called (5. applyDelt)
double externalTransition(e, x){
   coord.DeltFunc(coord.tL() + e, x);
   return timeAdvance();
}

// ComputeIO is called (5. applyDelt)
   double confluentTransition(x){
   coord.DeltFunc(coord.tN(), x);
   return timeAdvance();
}

// 3. getOut and 4. returnOut
MsgEntity[] outputFunction(){
   coord.ComputeIO(coord.tN());
   return coord.getOutputs();
}
```
Figure 20: Model Adapter implementation for the DEVS-Suite coupled model

5 RESTful Interoperability Simulation Environment (RISE)

Interoperating applications that have been developed independently and that interact with each other is not a trivial task, since this interaction involves not only passing remote messages, but also synchronizing them (interpreting messages and reacting to them correctly). This fact further applies to interoperating DEVS-based tools in order to synchronize the same simulation run. The value proposition, however, of such interoperability is that it enables a plug-and-play middleware approach, which is an appropriate method to interface independently-developed software applications [15]. The Plug-and-play type of interoperability is already applied by the World Wide Web (WWW) network. The principles of the Web interoperability have been recently called the Representational State Transfer (REST) style [16]. These RESTful Web Services [17] has been gaining attention with the advent of Web 2.0 [18] and the concept of mashups (grouping various services from different providers presented as a bundle).

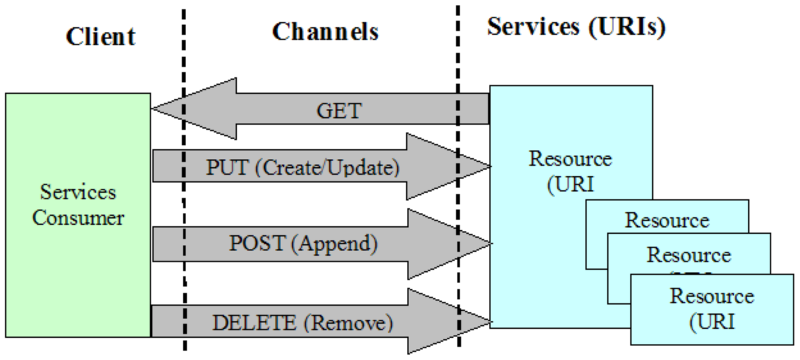


Figure 21: Uniform Channels for RESTful Resources

The RESTful Web-services lightweight approach hides internal software implementation (in “black boxes” called resources). Each resource exposes *uniform* channels (connectors) and describes connectivity semantics between resources in the form of *messages* (usually XML). RESTful services are distributed across a set of connected resources where each resource is named with a URI (similar to a website). Service consumers connect with those resources via standardized virtual uniform channels where semantic messages and the corresponding methods are assigned to those resources. In RISE, the channels are the HTTP methods shown in Figure 21: GET channel (to read a resource entirely or partially), PUT channel (to create a new resource or update existing data), POST channel (to append new data to a resource), and DELETE channel (to remove a resource). Resources use those channels to transfer their data (or potentially, their data representation) among each other, hence transferring their representational state, as specified by the name of the Representational State Transfer style [16]. REST exposes all services as URIs, hides internal implementation, employs message-oriented synchronization semantics (i.e., XML), and accesses each service (URI) via standardized channels. These are the ingredients for plug-and-play interoperability even at runtime, and they are being used on the WWW every day. A detailed study of the current and future challenges of distributed simulation algorithms and middleware is provided in [15].

Other approaches, such as CORBA or SOAP-based Web-services, expose functionalities in heterogeneous RPCs that often reflect internal implementation and describe semantics as procedure parameters. The RPC style literally splits software implementation across the distributed environment. It is worth noting that the SOAP-based Web-services transfer all RPC representations (as SOAP XML messages) via the HTTP POST channel. This overloading of the POST channel has resulted in making connectors that were once standardized uniformly into a more heterogeneous interface, which is more complex to use. RPC-style is heterogeneous in a sense that they are programming procedures invented by different programmers. Of course, the XML SOAP standard is powerful enough to describe those RPCs. However, applications interoperability is realized as RPC-style in another software layer above the SOAP handling layer (usually called SOAP engine) which converts RPCs from/to SOAP messages. For example, Figure 9 shows a typical SOAP-based Web-services protocol stack while Figure 10 shows RPCs exposed within a port.

In recent years a RESTful middleware application has been developed called RESTful Interoperability Simulation Environment (RISE), formally known as RESTful-CD++ [19][20], that has provided promising results in this area. RISE also allows any application or device attached to the Web to be in the simulation loop at runtime, using Web 2.0 mashup concepts. RISE middleware serves as a container to support concrete services; hence, concrete services are plugged into the middleware. In this case, concrete services are wrapped and accessed through URIs at the middleware level, rendering the middleware independent of any specific service. This allows additional services to be plugged into the middleware without affecting other existing services. This is similar to adding additional services or links to a regular website. The distributed CD++ (DCD++) simulation package was plugged into the middleware. In this case, multiple CD++ instances can perform distributed simulation session across the Web where the simulation model is split among those CD++ distributed instances, enabling each to simulate its portion of the model, as shown in Figure 22. The simulation manager, shown in Figure 22, manages a CD++ instance by handling, for instance, the geographic existence of model partitions, XML synchronization simulation messages, and synchronization algorithms. The simulation manager is seen externally as a URI (e.g., similar to web site URIs). The distributed CD++ instances synchronize among each other via sending simulation XML messages (wrapped in HTTP envelopes) to each other's URIs via an HTTP POST channel. RESTful DCD++ is described in [19][20].

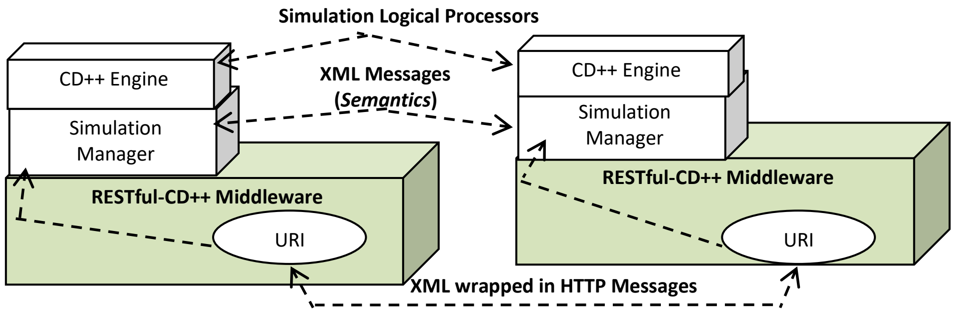


Figure 22: DCD++ Simulation Session between two Online Simulation Engines

RESTful applications APIs, including RISE, are expressed as URI templates [21] that can be created at runtime. Variables in URI templates (written within braces {}) are assigned at runtime by clients before a request is sent to the server, enabling clients to name their URIs at the server side. For example, username in template <*.../users/ {username}*> can be substituted with any string to obtain the actual URI instance (such as <*.../users/user1*> or <*.../users/user2*>). Further, URIs may include query variables to define the request scope by appending them to a URI after the question mark "?". For instance, a request via the GET channel to URI <http://www. google.com/search?q=DEVS> would instruct the Google search engine to return information only about keyword "DEVS". As another example, RISE middleware [19][20] defines the simulation framework URI template as */cdpp/sim/workspaces/{userworkspace}/ {servicetype}/{framework}*, where *{userworkspace}* is a specific workspace. The workspace allows users to define their specific URI hierarchy while avoiding naming conflicts. The *{servicetype}* is the selected simulation service (e.g., DCD++), allowing a client to use different services simultaneously. The *{framework}* is the simulation experiment framework; hence, a user may create multiple experiment frameworks that use the same simulation service. The experiment is configurable by its owner, for instance, to have different simulation partitions conduct the same simulation session. To further illustrate, the <*...>/cdpp/sim/workspace/Bob/DCDpp/MyModel* URI indicates that the user workspace belongs to user Bob, and the servicetype is DCDpp (which selects the distributed CD++ engine). The framework is named MyModel, which is the name of the simulation experiment. In this case, the modeler may select a different simulation engine (instead of DCDpp) or a different framework (instead of MyModel), because these variables are assigned at runtime according to the API URI template. Therefore, URI templates enable modelers to name their URIs without being in conflict with other users. The RESTful-CD++ API is fully described in [19].

The RISE standards approach is derived from the lessons learned of the RISE middleware. The RISE standards approach divides the entire simulation space into domains. Each domain wraps a DEVS model and DEVS-based simulation engine to simulate that model. Each domain is accessed via three URIs (i.e., API) to exchange semantics (i.e., synchronization and configuration) as standardized XML messages. Thus, a domain's interior is fully hidden, which makes the standard easier to understand and to support. This is because each domain only needs to be able to transmit/handle the standardized XML messages according to the approved rules while they are free to change whatever they need within their domain without affecting other domains. The RISE approach achieves this at three levels: **(1)** the *interoperability framework architecture* level, **(2)** The *model interoperability* level, and **(3)** the *simulation synchronization* level. These aspects are summarized next.

The *interoperability framework architecture level* provides the URI template (API) that allows modelers to create a simulation environment (including distributing simulations, starting simulation, and retrieving results). RISE requires three RESTful resources (URIs) for each domain so that other domains and modelers can use them to setup and conduct simulations. The focus here is on the parts of the URI template that are relevant to the RISE standard. The main functionality of those URIs is left to design for specific domains (the RISE standard may be part of different services provided by a specific domain). These resources (URIs) are described as follows:

1. ***…/{framework}***: represents a simulation environment in a domain. It is named by the modeler upon creation. The modeler uses this URI to submit all necessary information to execute simulation in that domain such as the simulation model and the RISE XML configuration. This URI is the parent of the other two needed resources described next. This resource uses HTTP channels as follows: The PUT channel is used to create and/or update the resource with the XML configuration document (for instance, inter-connections of the different simulation model ports across domains). The DELETE channel is used to remove this resource. POST is used to submit, as a zip file, all necessary scripts related to the model that is supposed to run on this domain. GET is used to read a simulation status on that domain as an XML document.
2. ***…/{framework}/simulation***: represents active simulation in a domain. The modeler uses this URI to start/abort simulation, and to manipulate simulation during runtime. This resource uses HTTP channels as follows: The PUT channel (with a null message) is used to create this resource. DELETE is used to abort simulation. POST is used by simulation engines in domains to exchange simulation XML synchronization messages.
3. ***…/{framework}/results***: is automatically created by a domain upon successfully completing the simulation, allowing retrieval of the simulation results.

The *model interoperability level* provides XML rules for combining different models. This XML document is provided via the PUT channel to resource *…/{framework}*. This is a straightforward step, because of the assumption that each domain contains an entire model with external ports. In this case, the modeler defines an interconnection between ports analogous to a DEVS coupled model. It is worth noting that this is different from RESTful DCD++ in the sense that DCD++ partitions a single model across the distributed environment. On the other hand, the developed approach here is placing an entire model in each domain. This is because it aims at interoperating heterogeneous environments with many implementation differences, and, therefore, the more flexible, practical, and powerful interoperability is achieved when hiding implementation. This makes sense because the heterogeneity devil resides in the software design and implementation details. For example, Figure 23 shows two models placed at two different domains. In this case, the model is wrapped in URI …/{framework}: The first model URI is …/Domain1 and the second model URI is …/Domain2. In order to conduct different simulation session experiments, different URI frameworks are needed in a given domain. Each model in Figure 23 has two external ports connected to the other model ports. This interconnection is shown in the XML document in Figure 24. For example, Lines 7-10 show the connection link of port OUT1 (at …/Domain1) to port IN1 (at …/Domain2). The XML document also shows other configuration such as "Type" at Line 3 is set to "C", indicating that the simulation will be synchronized according to a RISE conservative based algorithm. Likewise, the "Type" attribute can be set to "O" to conduct optimistic synchronization. Line #5 selects the main domain, which is mainly needed to manage the conservative-based simulation.

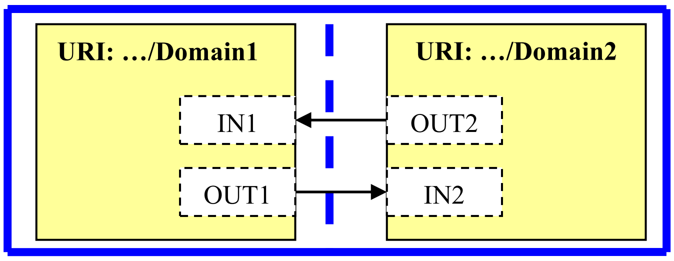


Figure 23: RISE Models Interconnection across DEVS Domains

```
1 <ConfigFramework>
2      …
3      <RISE Version="1.0" Type="C">
4       <Domains>
5         <Main><URI>…/Domain1</URI></Main>
```

```
        <Links>
          <Link>
<From><Port>OUT1</Port><URI>…/Domain1</URI></From>
<TO><Port>IN2</Port><URI>…/Domain2</URI></TO>
          </Link>
          <Link>
<From><Port>OUT2</Port><URI>…/Domain2</URI></From>
<TO><Port>IN1</Port><URI>…/Domain1</URI></TO>
          </Link>
        </Links>
      </Domains>
    </RISE>
    …
</ConfigFramework>
```

Figure 24: RISE XML Configuration corresponding to Figure 23

The *simulation synchronization level* provides high-level simulation algorithms (i.e., conservative/optimistic) and synchronization channels in order to carry out simulation among different domains. The modeler starts the simulation via the main domain (i.e., using the PUT channel to create URI *…/{framework}/simulation*). Consequently, the main domain starts simulation, in the same way, on all other domains, as shown in Figure 25. Afterward, all simulation engines at different domains are ready to exchange XML simulation messages to synchronize the simulation session. All of the simulation messages are sent to a domain via URI *…/{framework}/simulation* using the POST channel.

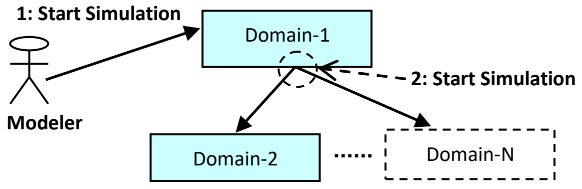


Figure 25: Starting Simulation Overview

The conservative-based approach expects the main domain to create the RISE Time Manager (RISE-TM) to manage time advancement of the entire space. This is not required for the optimistic-based approach where domains may directly send messages to each other, provided domains would detect and correct any error because of receiving a *straggler* message. Hiding this detail allows moving algorithm complexity to the interior of domains, while the RISE standard layer simply comprises channels to exchange simulation messages. On the other hand, the conservative approach requires more handling at RISE, since it owns the RISE-TM component. Note that the RISE-TM URI is the same as the main domain URI. Thus, the synchronization between the main domain simulation and the RISE-TM is specific to the internal implementation of the simulation software. However, they are separated in the discussion here for clarity. RISE-TM executes a simulation cycle in the following steps, as shown in Figure 26: **(1)** Execute all events in all domains at the current RISE time. This starts a new simulation cycle with the current or newly calculated RISE time. RISE-TM always starts the first phase with time zero. The domains must always execute all events with current RISE time, if any, and respond to the RISE-TM with the following information: all external messages generated for other domains stamped with RISE time (or larger) and its next time. The next time is the time of next event in a domain larger than RISE time. If no more events exist, this value is then set to "-1", indicating infinity. **(2)** Once RISE-TM receives all replies from relevant domains, it calculates the next RISE time and starts a new simulation cycle. Further, the RISE-TM merges all generated external messages and passes them to all relevant domains at the beginning of a simulation cycle. Note that the new simulation cycle may be a continuation of the current simulation cycle since external messages may be stamped with current RISE time. Note further that the RISE-TM stops simulation if it calculates a new RISE time to be infinity.

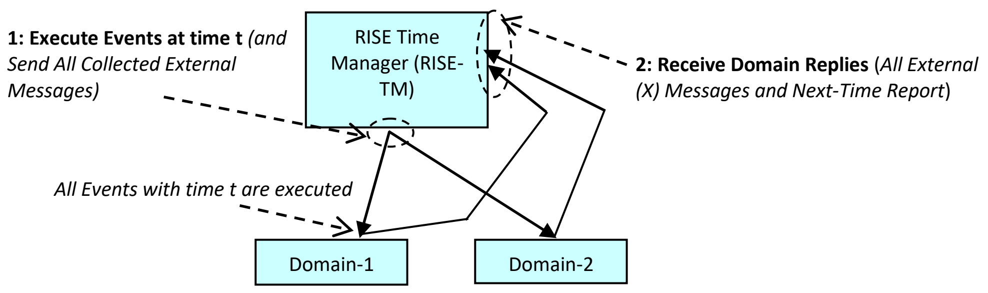

1: Execute Events at time t *(and Send All Collected External Messages)*
RISE Time Manager (RISE-TM)
2: Receive Domain Replies (*All External (X) Messages and Next-Time Report*)
All Events with time t are executed
Domain-1
Domain-2

Figure 26: RISE Conservative-based Simulation Cycle at Time t

Figure 27 shows a domain-2 response message to RISE-TM (i.e., step #2 at Figure 26). Line #2 indicates the message source domain URI, hence allowing RISE-TM to wait for all replies. Lines 3-15 contain all newly generated simulation events by the source domain. For instance, Lines 5-10 show a generated event at port *IN1* in *Domain1* with value 9. Line 14 specifies the minimum time of all enclosed external messages from the source domain. RISE-TM must include this time when calculating next RISE time. Line 16 specifies the time of the next event of *Domain1*. RISE-TM must include this time when calculating next RISE time. Figure 28 shows an example of a message sent by RISE-TM to all relevant domains. In this case, the new RISE-TM is calculated (i.e., Line #2), hence all events with this time must be executed at this cycle. Lines 3-17 forward all generated external messages. At this point, it becomes the responsibility of a domain to forward events to appropriate models through specified ports.

```
1 <RISE Version="1.0">
2 <URI>…/Domain2</URI>
3  <XEvents>
4    <MessagesCount>2</MessagesCount>
5       <XEvent>
6          <Time>00:00:01:000</Time>
7          <Port>IN1</Port>
8          <Value>9</Value>
9          <URI>…/Domain1</URI>
10       </XEvent>
11       <XEvent>
12          … … …
13       </XEvent>
14    <Time>00:00:01:000</Time>
15  </XEvents>
16  <Next>00:00:03:000</Next>
17 </RISE>
```

Figure 27: RISE Domain XML Document Response to RISE-TM Example

```
1 <RISE Version="1.0">
2  <Time>00:00:01:000</Next>
3  <XEvents>
4    <MessagesCount>1</MessagesCount>
5       <XEvent>
6          <Time>00:00:01:000</Time>
7          <Port>IN1</Port>
8          <Value>9</Value>
9          <URI>…/Domain1</URI>
10       </XEvent>
17  </XEvents>
18 </RISE>
```

Figure 28: RISE-TM Message to Start a Cycle

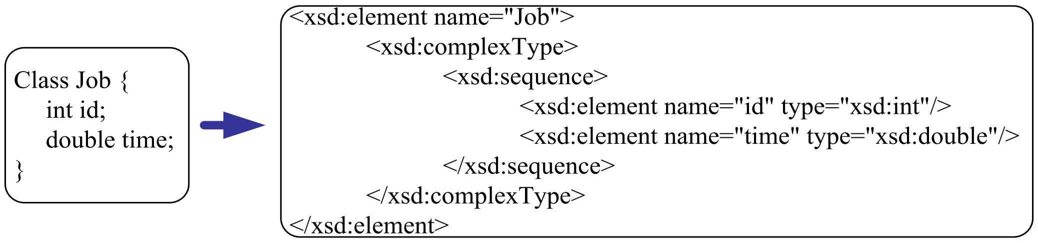
6 DEVS Namespaces

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 encapsulated in a SOAP message. The XML document is created in conformance with a type of schema in WSDL. The data types in WSDL are only 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 contents, each of which includes a port name and a value. In the DEVS formalism, values are defined as abstract sets that are not further constrained. Therefore, different DEVS simulation environments can have different class representations and associated object instance representations for values. To overcome this problem, a DEVS message is converted to an XML document at the web service level. This approach requires that different DEVS environments can translate back and forth between their internal value representations and a common XML representation. The *namespace* is the concept in XML that enables services to access a Schema employed by other services and thereby to parse documents to extract data corresponding to the instances of the Schema.

Thus, in order to interoperate DEVS simulator services in different platforms or languages, the namespace concept can be used to provide information about DEVS model messages. This gives rise to the *DEVS namespace* to support interoperability of DEVS simulator services on different web service platforms [12].

The DEVS namespace is an indicator of a schema document for types of messages that are used in DEVS models. The types are expressed in an element of XML Schema that describes the structure of the XML document. XML Schema assigns a unique name to each element. For example, if the name of the element is *Job,* then *Job* element is unique in the schema document. The uniqueness of a type provides clarity for message passing between systems that need to interoperate.

Figure 29 illustrates 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, the class *Job* 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.



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

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

Figure 29: Conversion of *Job* class to schema data type

Conversion of a class to a schema is performed by a service provider. For example, in a Java environment, the JAXB library performs this conversion and in addition supports dynamic invocation in which data are bound from a class instance to a corresponding document. The schema document resulting from the conversion is registered into a DEVS namespace storage to access through the network.

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 XML. A common XML message is designed 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 30: The structure of the XML message

Figure 30 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 an identifier for 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.

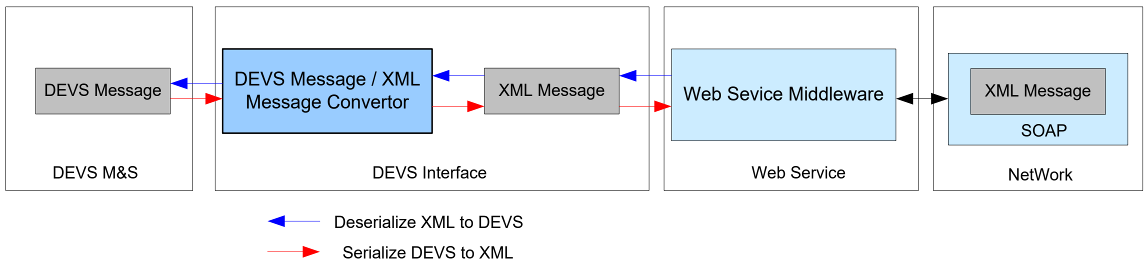


Figure 31. The DEVS message and XML message in the web service.

Figure 31 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 handles the DEVS messages, and the DEVS interface converts DEVS messages to XML messages. The web service then generates a SOAP message including the XML messages. This procedure is called serialization. The opposite procedure converts XML messages to DEVS messages and is called deserialization.

Seo [13] created a web service called *NamespaceService* through which a Schema of a DEVS simulator service is registered and browsed. 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 code for *NamespaceService* web service, which can help easily invoke operations. It displays the response of the operations. Any DEVS developer who uses a Java based environment or .Net based environment can use the GUI to register a schema. If a developer wants to browse the DEVS namespace storage, the developer 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. This concept can be extended to support browsing based on search criteria such as metadata about which applications have used the schema, the kinds of operations supported, and so on.

7 SUMMARY

This chapter introduced different methods to standardize DEVS simulation middleware. Theis middleware supports interfacing different simulation environments along with synchronization for distributed simulations. Most of the software introduced in this chapter is based on a Service Oriented Architecture, with Web Services deployed such that they can be easily invoked by clients.

The DEVS/SOA distributed simulation platform manages distributed simulation through the interchange of XML messages and the publication of web services. The DEVS/SOA simulator is based on a net-centric infrastructure that permits coordinating different atomic models written in varied tools and libraries. The environment provides a well-defined interface for atomic model Simulators and coupled model Coordinators that can be remotely invoked. Client applications are based on DEVSML models.

DDSP provides interoperability with minimum design changes to the relevant DEVS engines by providing a wrapper SOAP interface and XML messages for simulation synchronization. DDSP hides the internal implementation of the simulation tools (improving interoperability and supporting DEVS legacy models). The protocol implementation uses Web-services to interchange simulation messages, and a DEVS wrapper is employed to call service stubs using a remote procedure call.

The Shared Abstract Model defines an abstract Model Interface based on DEVS atomic models, and executed using CORBA. The system uses a model Proxy and a model Adapter to distribute the workload depending on the source and destination of the simulation messages, using the abstract model interface to standardize this communication.

RISE is a RESTful Interoperability Simulation Environment based on Representational State Transfer (REST). RISE uses RESTful web services and messages (in XML) to transfer information between simulation engines. Uniform channels are used to expose the simulation resources, based on the methods that are widely used for the World Wide Web (and the HTTP protocol). This approach hides the internal implementation and it uses message-oriented synchronization based on XML.

Finally, we introduced DEVS namespaces, which permit quickly finding the service data types and operations available. The namespace enables the services to access an XML Schema employed by other services, and to parse documents to extract data for that Schema. The namespace indicates the type of messages used in the model, and it assigns a unique name to each element. In order to browse the namespace storage, a namespace service is available to register and browse Schemas for DEVS simulation services.

REFERENCES

[1] S. Strassburger, T. Schulze, R. Fujimoto “Future trends in distributed simulation and distributed virtual environments: results of a peer study”. Proceedings of Winter Simulation Conference (WSC 2008). Miami, FL, USA. 2008.

[2] C. Boer, A. Bruin and A. Verbraeck “A survey on distributed simulation in industry”. Journal of Simulation. Vol. 3, No. 1, pp. 3–16. March 2009.

[3] S. Mittal, J. L. Risco-Martín, and B. P. Zeigler (2009), “DEVS/SOA: A Cross-Platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process,” Simulation, vol. 85, no. 7, pp. 419-450.

[4] A. Moreno, J. L. Risco-Martín, E. Besada, S. Mittal, and J. Aranda (2009), "DEVS/SOA: Towards DEVS Interoperability in Distributed M&S," in Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications: IEEE Computer Society.

[5] C. Seo and B. P. Zeigler (2009), "Automating the DEVS modeling and simulation interface to web services," in Proceedings of the 2009 Spring Simulation Multiconference San Diego, California: Society for Computer Simulation International.

[6] K. Al-Zoubi, and G. Wainer, "Interfacing and Coordination for a DEVS Simulation Protocol Standard," in The 12-th IEEE International Symposium on Distributed Simulation and Real Time Applications, Vancouver, British Columbia, Canada, 2008.

[7] G. Wainer; N. Giambiasi "Timed Cell-DEVS: modeling and simulation of cell spaces". In Discrete Event Modeling & Simulation: Enabling Future Technologies. Springer-Verlag. 2001.

[8] G. Wainer, "Discrete-Event Modeling and Simulation: A Practitioner's Approach". CRC press, Taylor & Francis Group. Boca Raton, Florida. 2009.

[9] Apache Tomcat. Available via http://tomcat.apache.org/. [Accessed July, 2008].

[10] Apache Axis. Available via http://ws.apache.org/axis/. [Accessed July, 2008].

[11] B. Zeigler; P. Hammods "Modeling & Simulation-Based Data Engineering: Pragmatics into Ontologies for Net-Centric Information Exchange". Academic Press. 2007.

[12] C. Seo and B.P. Zeigler. "DEVS Namespace for Interoperable DEVS/SOA". Proceedings of the 2009 Winter Simulation Conference.

[13] C. Seo. "Interoperability between DEVS Simulators using Service Oriented Architecture and DEVS Namespace". Doctoral Dissertation, ECE Dept, Univ. Arizona.Tucson.

[14] T. Wutzler and H.S. Sarjoughian, "Interoperability among Parallel DEVS Simulators and Models Implemented in Multiple Programming Languages," Simulation, vol.83, no 6, pp 473-490, 2007.

[15] G. Wainer G.; K. Al-Zoubi "An Introduction to Distributed Simulation". Chapter 11, Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains. Banks C., Soklowski J. (Editors). Wiley. New Jersey, 2010.

[16] R. T. Fielding "Architectural Styles and the Design of Network-based Software Architectures", Doctoral dissertation, University of California, Irvine, 2000. Available at: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Accessed October 2008.

[17] L. Richardson, S. Ruby "RESTful Web Services", O'Reilly Media, Inc., Sebastopol, California. 2007.

[18] T. O'Reilly "What Is Web 2.0". <http:// www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>. Accessed May 2009.

[19] K. Al-Zoubi; G. Wainer "Performing Distributed Simulation with RESTful Web-Services Approach". Proceedings of the Winter Simulation Conference (WSC 2009). Austin, TX, USA. 2009.

[20] K. Al-Zoubi; G. Wainer "Using REST Web Services Architecture for Distributed Simulation". Proceedings of Principles of Advanced and Distributed Simulation PADS 2009, Lake Placid, New York, USA. 2009.

[21] J. Gregorio "URI Templates". <http://bitworking.org/projects/URI-Templates/>. Accessed October 2008.