

Interfacing and Coordination for a DEVS Simulation Protocol Standard

Khaldoon Al-Zoubi

Gabriel Wainer

Department of Systems and Computer Engineering
Carleton University Centre for Visualization and Simulation (V-Sim)
Ottawa, ON K1S-5B6 Canada

kazoubi@connect.carleton.ca , gwainer@sce.carleton.ca

ABSTRACT

The DEVS formalism has been adopted and developed independently by many research teams, which led to various DEVS implementation versions. Consequently, different DEVS implementations currently vary at many levels such as in the type of simulation (parallel, distributed, embedded, etc.) or in the used programming language or computer platform. Interfacing various DEVS implementations becomes a necessity, particularly in today's world where sharing resources in distributed fashion have been rapidly gaining momentum. Here, we propose a mechanism enable interfacing and coordination between different DEVS implementations to cooperate among each other, towards a standardization effort. The proposed DEVS protocol is implemented using Web-Services technology as the communication framework to exchange control and simulation messages (contained by SOAP messages). This will allow executing simulations for the same model hierarchy in an open grid distributed environment. Therefore, the main objective of the proposed protocol is to enable different DEVS implementations to interface and coordinate among each other to simulate the same model structure across their domains. The protocol objective is achievable with minimum design changes to each DEVS implementation, mainly by hiding the detailed implementation behind a wrapper and focusing only on the exchanged messages. Further, the simulation cycle is greatly simplified by enclosing all diverse models across domains by outer coupled model, hence simulating single distributed coupled model. Furthermore, the proposed protocol is flexible enough to adjust to any new coordination schemes or communication framework may be used in the future.

Keywords

DEVS Standard, Distributed Simulation, Web-services.

1. INTRODUCTION

Modeling and simulation (M&S) plays an important role in studying complex natural and artificial systems. Discrete Event System Specification (DEVS) [13] is a modeling and simulation formalism that has been used to study such discrete event systems. It provides means for modeling the system as hierarchal

components, each of which has input and output ports to interact with other components and with the external environment. The success using DEVS in the field of M&S has inspired researchers to define DEVS-based extensions (e.g. Cell-DEVS [11] is an extension that 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).

Over the years, the DEVS formalism has evolved from its original discrete-event conception, and it has been adapted and modified independently by many research teams. Various DEVS implementations exist (see [10] for a list). Each of them vary in aspects such as the programming language used, underlying computer platforms, simulation extensions (e.g. standalone, parallel, distributed) and modeling extensions (each DEVS implementation uses different ways to write/construct models). The need to overcome incompatibilities between different DEVS implementation becomes a necessity as a result of today's world where interoperability issues must be resolved before resources integration and coordination becomes achievable in an open marketplace sharing grid environment (where resources can be deployed and consumed over the WWW). Here, we outline a proposal for a DEVS simulation protocol standard, which enables interfacing and coordination between different DEVS implementations to cooperate to carry out simulation (in discrete virtual time) for the same distributed model hierarchy. The protocol aims mainly on achieving interoperability with minimum required changes to the internal design and software implementation of each DEVS version. Therefore, it increases the protocol success chances since various DEVS teams are not expected to change their internal design and implementation in a way that jeopardizes their existing DEVS tools integrity.

The paper is organized as follows: Section 2 introduces the P-DEVS formalisms and provides brief background about web services (WS) and commonly used WS technologies. It also discusses the differences between the proposed protocol and other related DEVS works. Section 3 outlines the main objective and requirements of the proposed DEVS protocol. Section 4 describes the overall communication framework. Section 5 describes the DEVS protocol messages and their format, simulation coordination among different DEVS domains and the main simulation cycle.

2. Background and Related Works

Discrete Event System Specification (DEVS) [13] is M&S specification that is aimed to study discrete event systems. The model consists of components connected together through external port(s), as shown in Figure 2, where events are exchanged among models via those ports. Obviously as in any discrete-event simulation, the models being simulated changes state only at discrete points in time, upon the occurrence of an event. The P-DEVS formalism [4] expresses a system as a number of connected behavioral (atomic) and structural (coupled) components. The basic building component of DEVS models is the atomic DEVS model. A P-DEVS atomic model is formally defined as:

$$M = \langle X, Y, S, \text{dint}, \text{dext}, \text{dcon}, ?, \text{ta} \rangle$$

At any given time, an atomic model is in some state $s \in S$. It stays in state s for the time period specified by the state time advance function $\text{ta}(s)$. Now when the atomic model state life time expires, the model then outputs value $?(s) \in Y$, and changes its state as indicated by the internal transition function $\text{dint}(s)$. A P-DEVS model uses bag of inputs (X_b) to exploit parallelism in the system, hence execute multiple concurrent events simultaneously. Nevertheless, the model also changes its state as defined by the external transition function $\text{dext}(s, e, X_b)$, if the atomic model receives one or more external events $x \in X$ before the expiration of $\text{ta}(s)$, merging the functionality of multiple external transitions into a single one. A confluent transition function (dcon) is used to conclude the model's next state via resolving the collisions when receiving external events and internal transitions simultaneously. A P-DEVS coupled model is formally defined as:

$$N = \langle X, Y, D, \{M_d \mid d \in D\}, \text{EIC}, \text{EOC}, \text{IC} \rangle$$

The external input coupling (EIC) specifies the connections between external and component inputs, while the external output coupling (EOC) describes the connections between component and external outputs. The connections between the components themselves are defined by the internal coupling (IC).

All of the various DEVS versions share in common that there is a coordinator (coupled simulator) to simulate a coupled model and an atomic simulator to simulate an atomic model. Thus, DEVS separates simulation layer from the modeling layer. However, each DEVS version provides different software design and implementation for the same notion. For example Figure 1 shows simplified portion of the DCD++ [10] coordinators/simulators hierarchy where the Simulator simulates an atomic model and the Coordinator simulates a coupled model.

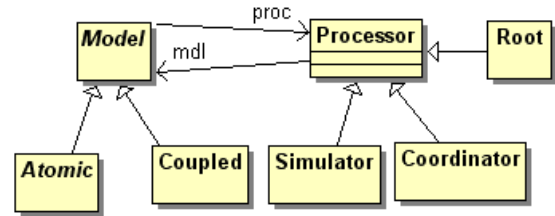


Figure 1: Simplified DCD++ Model/Simulators hierarchy

Web services are group of standards and languages aiming to facilitate developing, publishing, and discovering web-enabled applications. In other words, a web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-understandable format (specifically Web Service Description Language WSDL [5]). Client systems interact with the web service in a manner prescribed by its description using SOAP [6] messages, typically conveyed using HTTP with XML serialization in conjunction with other web-related standards [2]. WSDL [5] documents include information for the web service clients to know the operations it offers, parameters required to invoke an operation, and return type. The major elements of any WSDL document are type, message, portType, binding, port, and service elements. SOAP [6] plays an important role in any web service transaction. It is the messaging protocol used to convey information to and from the web service. It was designed to decentralized communication among multiple parties. The structure of SOAP messages is based on XML. Once the web server receives the HTTP request containing the SOAP message (i.e. as in the case of any other HTTP request), the SOAP message is extracted from the HTTP request and forwarded to SOAP engine, which is responsible for processing messages and converting the request(s) into a method call(s) that the service implementation code can understand.

The basic abstract simulator presented in [13] has been extended into varied parallel/distributed versions, a full list is provided in [10]. Further, the presented standard proposal here differs from the one presented in [7] in a number of ways summarized as follows: the proposal in [7] was based on the design and implementation that underlie the DEVSJAVA [7] tool. In other words, it defines how java interfaces (i.e. equivalent to C++ abstract classes) are structured and implemented. Further, the work in [7] does not define a number of important issues that is necessary in order to *implement* the standards for example: what the required coordination messages are needed, how models are distributed across different domains, how messages are passed through the network, how messages are formatted, etc. In contrast, the presented proposal here not only answers (with reasonable details) the above issues, but also hides specific DEVS implementation behind wrappers, focuses only on exchanged

messages (better for scalability and portability), and simplified simulation by enclosing all inter-domain models in one outer model (hence it becomes an issue of simulation one distributed coupled model).

3. Objective and Requirements

The main objective for developing a standardized DEVS simulation protocol is to enable different DEVS implementations to coordinate among each other to carry out simulation for the same model hierarchy that is partitioned between various DEVS version domains. Therefore, the simulation protocol proposes an answer to the following question: How to coordinate/synchronize simulation for the same DEVS model structure that is distributed over diverse DEVS implementation domains? In short, the answer is by coordinating via exchanging standardized DEVS messages.

The protocol does not need to know a DEVS tool internal software design and implementation, and is not attempting to standardize how a DEVS tool implements its internal modeling and simulation software. However, the proposed protocol expects each DEVS tool to react to incoming messages with the 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). Hiding internal implementation details, and focusing only on the information needed has many advantages such as:

- *Maintainability*: Protocol changes only applied on the protocol messages rather than on every DEVS implementation,
- *Scalability*: Easier to add/remove exchanged messages, and
- *Testing*: Easier to perform local testing by each group before executing integration testing between different DEVS domains.

The DEVS Protocol not only needs to meet its objective (by interfacing different DEVS implementations) but also to be effective and usable by diverse teams (even beyond the DEVS community). In order to achieve these goals, these requirements are proposed:

1. Each DEVS implementation should be able to execute its own specific models. It is impractical to convert the large number of existing DEVS models to a standard format (however, in the future, we envision DEVS models that could be defined in a standard format and executed with different DEVS implementations). For example, as shown in Figure 2, coupled #1 model can be specific to DEVJSJAVA [1] while coupled #2 can be specific to DCD++ [10]. In this case, both DEVJSJAVA and DCD++ coordinate between themselves to interface both coupled models without worrying about how the other DEVS implementation does it internally. Therefore, each DEVS implementation, in this case, views other coupled

model as black-box with input/output ports. On the other hand, it is still possible for a DEVS implementation to know more details about the model structure (not just one black-box coupled model) in other DEVS domains. This depends on the level of details the domains are told when the model structure is distributed among different domains (to be discussed in detail in Section 5.1).

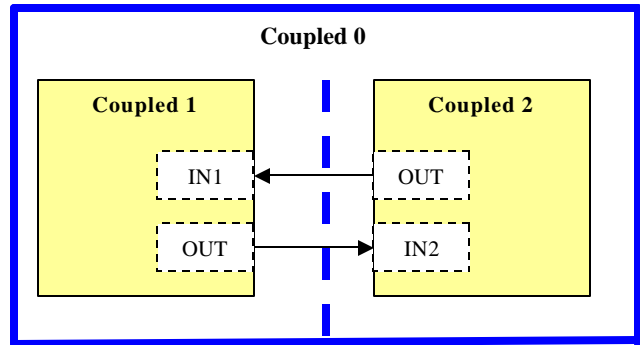


Figure 2: A Coupled model partitioned across Two DEVS Domains

2. Each DEVS implementation uses a single communication entry point. We this component a *DEVS-Wrapper* (shown in Figure 3). 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 that it communicates with. This requirement also simplifies security issues in a DEVS domain. The DEVS-Wrapper component is expected to perform the following tasks:
 - To translate incoming standardized simulation messages to specific domain simulation messages.
 - To transmit simulation messages to other DEVS domains according to the DEVS standards, and
 - To route incoming simulation messages to the correct models/ports within its domain.

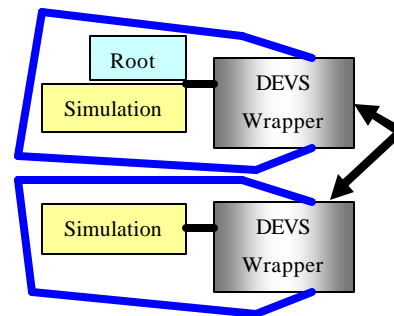


Figure 3: Connecting Two DEVS Domains

- The simulation protocol should minimize its dependency on the communication framework as much as possible. In this way, it should require minimum (or no) changes to the standardized simulation messages if one needs to move the simulation protocol to other communication engines in the future. In our proposal, this requirement is implemented by sending all simulation messages as XML documents using SOAP engine that can transmit files as SOAP attachments. Therefore, if the communication mechanism changes, those same XML documents can still be transmitted without changes.
- Only one Master DEVS domain will be in charge of driving the overall simulation. This master domain creates and owns the *Root Coordinator* as shown in Figure 4 while other DEVS domains become slaves and only react to messages from the master or other slave domains. The Master domain is the one that was selected by the user to initialize and start the simulation session.

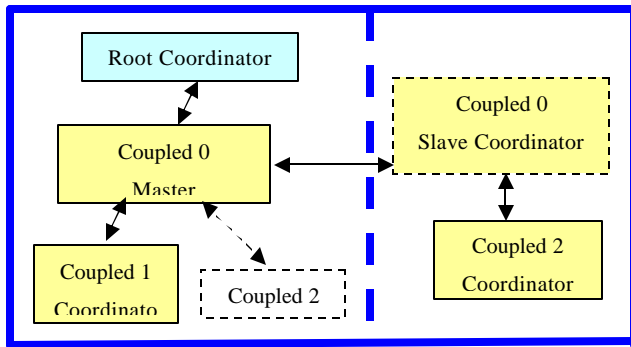


Figure 4: Coupled #0 Split between Two DEVS Versions.

4. Communication Framework

We propose to use Web-services technology to transfer the standardized simulation messages between different DEVS domains. All exchanged messages are transmitted through SOAP/HTTP engines, hence wrapped within SOAP and HTTP envelopes, as shown in Figure 5.

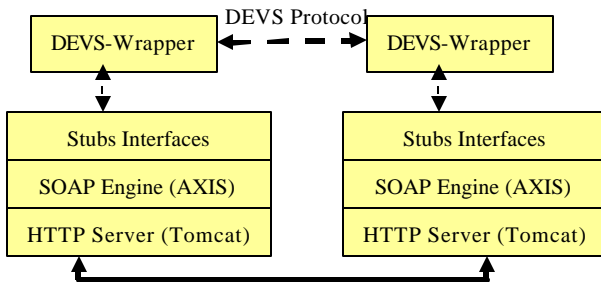


Figure 5: Connecting DEVS Domains using Web-Services

As we can see, a DEVS-Wrapper communicates with other DEVS domains by invoking the deployed-service stubs in a remote procedure call (RPC) style where the simulation messages are passed into those stubs as SOAP attachments. The Stubs are constructed from the deployed WSDL document by the service provider (other DEVS domains in this case). For example, tool *WSDL2Java* is used in the Apache AXIS server environment to convert a deployed WSDL document to the necessary Java classes including the needed stubs. Therefore, the needed communication engines for each DEVS domain:

- HTTP Server (Tomcat [3] in our case).
- SOAP Engine (AXIS [12] in our case).
- XML parser: the proposed protocol is not making any assumptions regarding which XML parser to use. Simple API for XML (SAX) [9] and Java Architecture for XML Binding (JAXB) [8] are examples of current used XML parsers.

Each DEVS implementation is expected to deploy the following basic services via a WSDL document so that a master DEVS domain can use them to create/start a simulation session (other supporting services may be needed):

- *CreateNewSession (Master session)*: To login to a DEVS domain and create a simulation session. It returns the opened slave session number. Each DEVS domain needs to bind its session number with other relevant DEVS domains session numbers, since the same simulation session may have different numbers in different domains (more on this point in section 5).
- *StopSimulation (session)*: To abort current simulation without closing the session.
- *CloseSession (session)*: To delete a current session.
- *ReceiveDEVSMML (session, filename, attachment)*: To receive XML documents related to the simulation. The document can be a simulation message used for synchronization or simulation/modeling configuration. This message can be sent from any domain to any other domain without involving the master domain for obvious performance grounds; hence it actually depends on the simulation progress as discussed in Section 5.

In addition, each DEVS implementation is expected to create the necessary services to allow a user to use it as the master simulation domain (the one driving the whole simulation). For example, the user is expected to have a service to start the simulation once all necessary models are spread over different DEVS domains.

5. Simulation Protocol

The simplest way of structuring a DEVS model is to have one coupled model at each DEVS domain connected to each other via

their input/output ports where each coupled view other domain coupled models as black-boxes. Even with this simple scenario, another outer top coupled model should then be created to *wrap* all coupled models across various domains. Therefore, they will be at least one coupled model partitioned across DEVS domains. By having one Coordinator to simulate a single coupled model that is distributed over large distances becomes a performance bottleneck due to the number of exchanged messages across the network between the parent Coordinator and its children's (Coordinators and/or atomic simulators). For this reason, we propose to adopt a Master/Slave Coordinator structure [10] (the standards can also adopt other algorithms, if needed, as it should to meet its *scalability* requirement). The Coordinator concept is extended in two ways:

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

Creating a top coupled model to wrap up all coupled models across various DEVS domains, allows the simulation to be carried out like if it was performed by one DEVS domain with a single coupled model where its internal models are distributed locally and across various DEVS domains. This simplifies simulation synchronization and logic. Assume, for example (as shown in Figure 4) Coupled #0 consists of two internal coupled models: Coupled #1 (exists in a DEVS version domain) and Coupled #2 (exists in another DEVS domain). In this case, one of the DEVS versions will create the Master Coordinator to simulate the whole coupled model (coupled 0 in this case), and the other DEVS version will create the Slave Coordinator to simulate Coupled #2 on behalf of the Master Coordinator. Of course, it is the responsibility of the Master Coordinator to synchronize the simulation for both: its internal local models and Slave Coordinators. For example, if coupled #2 is passive at a specific simulation time, the Master Coordinator does not need to communicate with its Slave Coordinator at that time cycle and it only must simulate Coupled # 1.

The master/slave structure may require some changes in some of the existing DEVS implementation particularly the ones that do not support distributed simulation. However, for those that already support distribute simulation they may just need to map standard messages to their internal ones. Further, the standard

assumes that other synchronization algorithms may be supported in the future other than the Master/Slave.

We also need to decide which domain must create a Master Coordinator, and which one is expected to create a Slave Coordinator. The proposed solution is that the domain that owns the first listed internal model locally (as assembled in the model structure XML document discussed in section 5.1) of a coupled model creates the Master Coordinator and the rest of the DEVS domains create the Slave Coordinators.

A typical scenario for a user (to start the simulation) is to

1. Collect diverse DEVS model descriptions,
2. Construct an XML document to describe the structure of the overall model, including port connections,
3. Open a session with the master DEVS domain which then opens a session with all relevant DEVS domains (using the interface method *CreateNewSession*). Once the master domain opens/collects all session numbers from slave domains, it passes the information to slave domains in one XML document (using the interface *ReceiveDEVSMML*). This is important in a sense that a simulation message may sent from a model in a slave domain to a model in another slave domain. In this case, the receiver slave domain can figure out its correct internal simulation session for the incoming message. The simulation session XML document contains the following information: Master domain session, and slave URIs paired with their session number, as in the following example:

```
<DomainSessions>
<Session Type="Master">
  <Number>123</Number>
  <URI>http://...</URI>
</Session>
<Session Type="Slave">
  <Number>1000</Number>
  <URI>http://...</URI>
</Session>
...
</DomainSessions>
```

4. Submits the model structure XML document (section 5.1) to the master DEVS domain which then sends it to all slave domains (using interface *ReceiveDEVSMML*), and
5. Starts the simulation from the master DEVS domain.

5.1 Model Structure XML Document

The Model structure XML document is initially submitted by the user to the master DEVS domain to describe how the overall model is structured so that each DEVS version knows which models that belong to its domain. The master DEVS domain passes this document (as SOAP attachment) to other domains via invoking service interface *ReceiveDEVSMML*. At this point we

assume that each domain already has the models that it will simulate (i.e. its specific domain models). The model structure document will contain enough information to allow different DEVS domains to create all of their local models, Coordinators and atomic simulators and how they will relate to other models in different DEVS domains. Afterward, all slave DEVS domains will be waiting for the first simulation message from the master DEVS domain (which is the *Init* message to initialize all models).

The model structure XML document also serves as an agreement contract between various implementations on the used synchronization schemes. For example element COUPLED_SYNC in the XML document can be set to the used coordination scheme to simulate a distributed coupled model across various domains. In this way, the standard is not limited to one idea or algorithm version and can easily adopt any new schemes may appear in the future.

Some of the model structure XML document information are: Model names, types (coupled or atomic), input/output ports, internal models and their descriptions (e.g. nested coupled models), ports connections, model's domain URI, used synchronization algorithms (e.g. Master/Slave Coordinator structure discussed previously in this section) and the model original specific DEVS tool.

The DEVS models hierarchy can easily be mapped into XML document. For example assume two models connected with each other as shown in Figure 2 across two DEVS domains where each model is specific to its domain implementation. According to the adopted approach here the two models will be enclosed with another outer model (coupled #0 in this example). In this example the model structure XML document can be structured as follows. Note that the domain that owns first internal model ("Coupled1" in this example) will create the Master coordinator for parent "Coupled0" where other domains create slave coordinators for it:

```
<MODEL_STRUCTURE>
  <COUPLED_SYNC>
    <scheme ver="1.0">MasterSlave</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>
  </Models>
</MODEL_STRUCTURE>
```

```
</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>
```

5.2 Messages Format and Contents

Once all DEVS implementations receive the model structure document and create the necessary software structures and processes, the simulation starts by the master DEVS domain by sending the *Init* simulation message to the highest coupled model which then propagates downward throughout the model hierarchy across DEVS.

All of the simulation messages are constructed as XML documents which sent to other domains as SOAP attachment using the AXIS stub (interface) *ReceiveDEVSMML*. Therefore any future changes in the simulation messages will be done to the message XML document rather than to the input/output parameters of the AXIS stub, hence increase protocol scalability and portability. The messages XML document contains the following information (note that "Time" indicates the simulation virtual time):

- *Session ID*: The receiver domain session Id. This enables various DEVS domains to run multiple simulation sessions simultaneously with other DEVS domains.

- *Simulation Message Type* (Init, Collect, Internal, External, Output and Done as discussed in section 5.3)
- *Next Change Time*: is only used by “Done” messages type to inform parent Coordinator of the next expected internal change (where in turn a parent Coordinator passes one “Done” message to its parent with the minimum next change of its internal local/other domains model children). Eventually only one “Done” message is received by the *Root Coordinator* (in the master domain) which then starts another simulation phase as discussed section 5.3. All Coordinators (including the Root) use this message to know which children branches should be involved in each simulation cycle, consequently prevents many unnecessary message transmissions across the network.
- Message sending Time.
- Source Model.
- Destination Model.
- Source Port.
- Destination Port.
- Value.
- *IsFromSlaveDomain*: True if the message is sent from a slave domain.

The following XML description shows an example of constructing an *Init* message from port “out” of Coupled0 to port “in” of coupled2 during the simulation session #123. It is the responsibility of the sending DEVS domain to pack the correct session number of the receiver domain as described earlier in this section.

```
<SimulationMessage Type="Init">
  <Session>123</Session>
  <Time>...</Time>
  <Source Port="out">Coupled0</Source>
  <Destination Port="in">Coupled2
</Destination>
  ...
  ...
</SimulationMessage>
```

5.3 Coordination: Messages and Phases

The Parallel-DEVS (P-DEVS) algorithms [4] were mainly introduced to solve the serialization problem with the original DEVS algorithm. The main additions in P-DEVS are the message bags (used to hold multiple input messages arriving to the model and multiple output messages generated by the model), and the confluent transition function (dconf) (which defines the behavior of the model when it receives an external message while being

scheduled for internal transition). Therefore, the simulation can be divided into three phases:

- *Initialization*: it only happens at the first simulation cycle (usually at time Zero) and starts when the highest coupled model receives the *Init* message (e.g. coupled #0 in Figure 4). Afterwards, the *Init* message propagates downward in the model hierarchy until it executes every initialization method of every atomic model. In response, a “Done” message propagates upward in the model hierarchy where each Coordinator (coupled simulator) calculates the minimum next change of its children and passes it in one “Done” message to its parent. Once all “Done” messages arrive at the highest coupled model (e.g. coupled #0 in Figure 4), it passes it to the *Root Coordinator* which updates the simulation clock as required and starts the simulation “Collection” phase.
- *Collection*: The *Root Coordinator* starts this phase by sending a *Collect* message to the highest coupled model, which in turn, passes it to all of its children. In this phase all of the output messages are triggered and may be passed by internal Coordinators to their destined models as external messages (i.e. only inserted in message bags in this phase). This phase ends when the *Root Coordinator* receives a “Done” message from the highest coupled model.
- *Transition*: The *Root Coordinator* starts this phase by sending an *Internal* message to the highest coupled model, which in turn, passes it to all of its children. In this phase, all of the collected external messages in the message bags (which done in the previous collect phase) are passed downward in the model hierarchy. Once the atomic models level is reached, the appropriate atomic operations are executed by their simulators, based on:
 - An Internal event was scheduled or not, and
 - External messages exist in the bag or not.

As the above phases show that simulation is carried out in the same manner regardless of models distribution across domains as soon as each DEVS domain reacts to the simulation messages with the expected messages. This is achieved mainly because all model fragments across various DEVS domains have been enclosed by a single coupled model, which is treated by the master domain as a single coupled model that is partitioned across the network.

The simulation messages are listed as follow:

- *Init*: To initialize all models. The simulation starts when *Init* message is passed to the highest coupled model Coordinator which then pushes it downward to its children. In response, a “Done” message is triggered with the minimum next internal change time.

- *Collect*: Used by the Root Coordinator to start the collection phase by passing this message to the highest coupled model Coordinator which then propagates it downward. In response a “Done” message is expected to indicate the end of the collection phase.
- *Internal*: Used by the Root Coordinator to start the *transition* phase. In response a “Done” message is expected to indicate the end of the transition phase.
- *Done*: Used for synchronization by Coordinators to identify which children need to be simulated at current phase. Also, used by the Root Coordinator to advance the simulation time and switch simulation phases.
- *External Message*: Messages from simulation environment or as a result of output messages. No “Done” message is expected in response to this message.
- *Output Message*: Generated during the collection phase. No “Done” message is expected in response to this message.

The main simulation is quite simple, since the real work is distributed among internal Coordinators of the coupled models. A possible implementation is as follows (assuming all simulation messages get queued first in a global queue):

```
While (simulation is running)
{
  If (unprocessed messages exist in queue)
  {
    Get first message from queue;
    If (message belongs to my DEVS domain)
    {
      // Destination is either Root
      // Coordinator, coupled Coordinator
      // or atomic simulator
      Send message to its Destination;
    }
    Else
    { // going to another DEVS domain
      Send message to my CPP-Wrapper;
    }
  }
}
```

Based on the above simulation loop, the Root Coordinator (which exists only in the master DEVS domain) receives simulation messages like any other Coordinator. The main function of the Root Coordinator is performed when it receives the “Done” message: it advances the simulation clock (i.e. the received “Done” message contains the next change time), starts the collection/transition phase or stops the entire simulation. The Root Coordinator will receive its first “Done” message to indicate the end of the initialization phase. A possible implementation of the Root Coordinator is as follows (note that Root simulation *Next Phase* is initialized to *Collect*):

```
Root Coordinator::ReceiveDoneMessage ()
{
  If (Next Phase == Transition)
  { // Start transition phase
    Next Phase = Collect;
    Send Internal Msg to highest model;
  }
  Else
  {
    If (next Time <= STOP_TIME)
    {
      Send Stop to all;
    }
    Else
    {
      While (envExternal == NextEventTime)
      { // to be executed at this time
        Send environment external event;
      }
      If (Next Event is NOT external)
      { // Start the Collect Phase
        Next Phase = Transition;
        Send Collect Msg to highest model;
      }
      Else
      { // Start transition phase
        Next Phase = Collect;
        Send Internal Msg to highest model;
      }
    }
  }
}
```

The proposed protocol standard here 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 on locating their children (i.e. internal models) in order to pass to them the needed simulation messages (perhaps by having a database which 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 need to leave to another domain, the *DEVS-Wrapper* (shown in Figure 3) translates them to the standardized XML message documents and passes them as SOAP attachments using the AXIS stub *ReceiveDEVSMML*. For example, as shown Figure 6, a DEVS domain does not need to use the standards within its domain. However, when a message needs to travel to another domain, it must be translated first to the standard format so that it can cross over the DEVS protocol bridge.

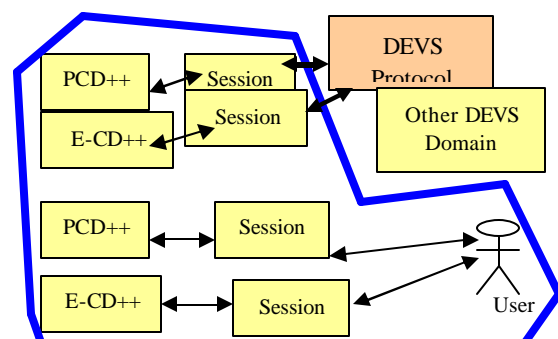


Figure 6: An Internal Look of a DEVS Domain

6. Conclusions

The main objective of the proposed DEVS standard protocol is to enable different DEVS implementations to interface and coordinate among each other to simulate the same model structure across their domains. The standard has made some requirements and assumptions in order to make the proposed protocol achievable and acceptable by different teams. Some of these requirements that minimum design changes are expected to each DEVS implementation, mainly by hiding the detailed implementation behind a wrapper and focusing only on the exchanged information that is needed to perform simulation and coordination among distributed models. Further, this proposal has simplified the simulation logic by enclosing all diverse models by one single coupled model simulated at the master DEVS domain, hence it becomes the responsibility of the coupled model Coordinators to find the models that they want to send them messages without worrying about other details such as constructing messages in XML documents or where which specific DEVS implementation is simulating the other models. In fact, when we got into section 5 which discusses simulations the reality of diverse DEVS implementation almost disappeared.

The DEVS simulation protocol was also discussed in reasonable details to show the exchanged messages format and contents. Further, we described the overall simulation coordination showing each DEVS domain role in the phases of each simulation cycle. In addition, the master/slave structure was proposed to coordinate a coupled model simulation in the distributed environment in order to reduce the number of exchanged messages across the network. However, the standards does not limit itself to one algorithm, hence more schemes may be added in the future and used easily by including this information in the exchanged XML documents.

The proposed protocol assumed the usage of web-services technology as the communication framework. However, the proposed protocol takes into account that the DEVS simulation messages should easily be ported into different communication architecture in future, if needed to do so. This is accomplished by

constructing all simulation messages in XML documents so that any changes in the protocol messages will be done to those XML documents rather than to the web-services specific communication interfaces.

7. REFERENCES

- [1] ACIMS software site: <http://www.acims.arizona.edu/SOFTWARE/software.shtml> .
- [2] Alonso, G. Web services : concepts, architectures and applications. Springer. 2003.
- [3] Apache Tomcat. Available via <http://tomcat.apache.org/> . [Accessed July, 2008].
- [4] Chow, A.; Zeigler, B. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism". Proceedings of the Winter Computer Simulation Conference. Orlando, FL. USA. 1994.
- [5] Christensen, E; Curbera, F.; Meredith, G.; Weerawarana, S." Web Service Description Language (WSDL) 1.1". March, 2001. Available via <http://www.w3.org/TR/wsdl> . [Accessed July, 2008].
- [6] Gudgin, M.; Hadley, M.; Mendelsohn, N.; Moreau, J.; Nielsen, H. "SOAP Version 1.2 Part 1: Messaging Framework". June, 2003. Available via <http://www.w3.org/TR/soap12-part1/> . [Accessed July, 2008].
- [7] Hu, X.; Zeigler B. "A Proposed DEVS Standard: Model and Simulator Interfaces, Simulator Protocol". January 2008.
- [8] Java Architecture for XML Binding (JAXB). Available at <http://java.sun.com/developer/technicalArticles/WebServices/jaxb/> . [Accessed July, 2008].
- [9] Simple API for XML (SAX). Available at <http://www.saxproject.org/> . [Accessed July, 2008].
- [10] Wainer, G.; Madhoun, R.; Al-Zoubi, K. "Distributed Simulation of DEVS and Cell-DEVS Models in CD++ using Web-Services". Accepted for publication in *Simulation, Practice and Experience*; Elsevier. 2008.
- [11] 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
- [12] Web Services-Axis. Available via <http://ws.apache.org/axis/> . [Accessed July, 2008].
- [13] Zeigler, B.; Kim, T.; Praehofer, H. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press. 2000