Standardizing DEVS model representation

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 in the previous chapter, the idea of *Standardizing DEVS model representation* is to allow a platform-independent DEVS model representation to be executed by a DEVS-based simulator. In this case, a DEVS model is executed in a single processor or parallel/distributed simulation environment. This allows model reusability without the need for performing long-distance distributed simulation. Typically, models are stored in repositories and retrieved as needed. This type of interoperability is very important to achieve, because often modelers already have or can easily install a DEVS simulator on their machine. However, it is much more difficult for them to retrieve and reuse already existing models that were intended to run on a specific DEVS environment. Therefore, we need some way to provide access and share these numerous models. Having a platform independent DEVS model representation format allows us to go a step further in automatic model transformations and generate entire local copies of distant models.

Using transformations based on code parsing, we can generate the XML description of existing models written for a specific framework. These files can be stored in *model repositories*, accessible through Internet, so that anyone (with the proper authorizations) can access them [1]. When facing a problem to which Modeling & Simulation (M&S) will be applied, the modeler starts by searching for an existing model that could fulfill the requirements. If there is none, the modeler can write their own but still take advantage of existing models by reusing them in their coupled models. To do so, the users download a model description from a repository in their framework, and then they use generic transformations to generate the code for the model. After this, they simulate their models and those of other users on their own simulation platform.

Compared to distributed simulation interoperability, this approach has the benefit that it drastically reduces the communications with remote servers. In fact, after the model description has been downloaded, there is no more need for accessing the network (hence the "off-line" appellation). However, the drawback of this is that a given model can have several unsynchronized versions all over the world. A solution to this issue could be to use some sort of update feed, but this is still more complicated than simply invoking a service without caring about the underlying implementation, which can continuously improve and evolve. Simultaneously, knowing the details about the implementation can provide insight into the structure of the model (which is an important factor for model-based publishing [2].

To sum up both interoperability solutions exposed here, we can say that the simulator-based approach (i.e., via distributed simulation middleware, as discussed in Section **Error! Reference source not found.**) is aimed at coupling distributed or local models over remote servers, whereas the model-based approach aims at integrating distributed models in a local simulation.

Although the DEVS mathematical formulation is universal and well known, representing complex models with DEVS is still a difficult task (and, it is not easy to understand).

Consequently, there are a number of alternatives to represent DEVS models with different notations and tool support. In the following sections, we will introduce model-based efforts by different M&S groups around the world that have tackled the problem of standardizing DEVS models and their simulators.

2 DEVSML

DEVSML is a novel way of writing DEVS models in XML. DEVSML is built on JAVAML (an XML implementation of JAVA). JAVAML is used to specify the behavioral logic of atomic models and the structure of coupled models. DEVSML models are transformable back and forth between Java and DEVSML, with the objective to provide interoperability between various models and to reuse and create dynamic scenarios.

The layered architecture of DEVSML is shown in Figure 1. A DEVS model can be provided as a set of platform-specific components, as a set of DEVSML components, or as a combination of both (see layer 1 in the Figure). Next, the final composed model can be transformed into a full DEVSML representation (layers 2 and 3). To do so, the user must provide the platform from which their model originates (e.g. DEVSJAVA, CD++, etc.). Finally, using layer 4, the DEVSML model can be transformed to any of the compatible DEVS platforms (layer 5)

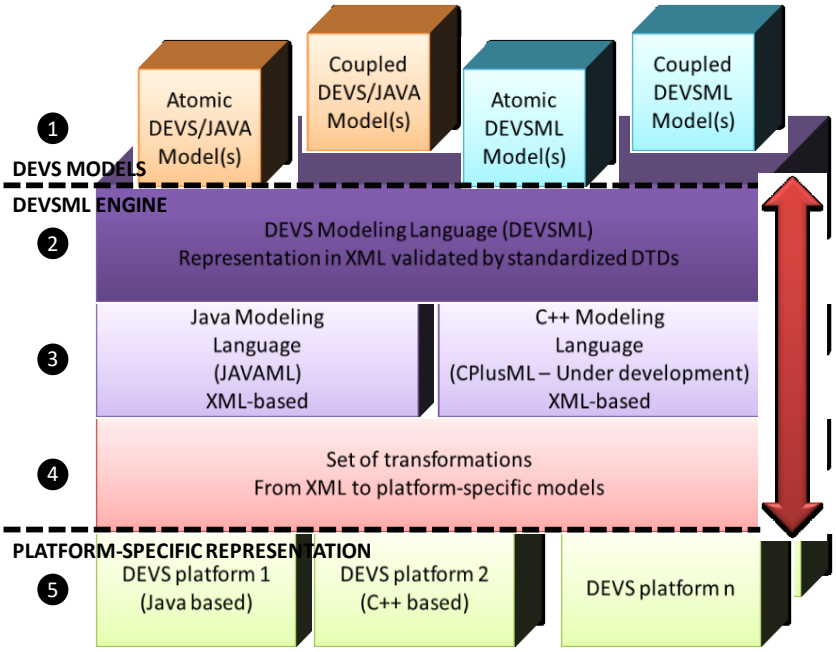

Figure 1: DEVSML layered architecture

2.1 DEVS Structure

To ensure that the final DEVSML complex model is "DEVS compliant", a single atomic *Document Type Definition (DTD)* and a single coupled DTD are defined. Thus, every DEVSML atomic or coupled model may be validated against the DEVSML DTDs. Both DTDs are listed below:

```
<!-- DEVS ATOMIC MODEL -->
<!ENTITY % variable-info
        "name CDATA #REQUIRED
         type CDATA #REQUIRED">
<!ELEMENT atomic
(inputs,outputs,states,ta,deltint,deltext,deltcon,lambda,services?,java-specific?)>
<!ATTLIST atomic
```

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

Figure 2: DEVSML atomic DTD

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

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

Figure 3: DEVSML coupled DTD

Both DTDs define the structure and behavior of a DEVS model. As Figure 2 and Figure 3 show, both atomic and coupled components are formed by standard DEVS structural definitions. In

the following code snippet (Figure 4), we show how the structure of a Lorentz Attractor coupled model may be defined. Note that typically, a DEVS coupled model consists of structural definitions, so no JavaML code is needed in the transformation from Java to DEVSML.

```
<devs>
 <scenario>
  <coupled name="root" model="LorentzAttractor" simulator="xdevs" host="127.0.0.1">
   <inputs/>
   <outputs/>
   <components>
    <atomicRef name="fxu" model="Function" simulator="xdevs" host="127.0.0.1"/>
    <atomicRef name="integrator" model="Integrator"
               simulator="xdevs" host="127.0.0.1"/>
    <atomicRef name="gxu" model="Function" simulator="xdevs" host="127.0.0.1"/>
    <atomicRef name="scope" model="Scope" simulator="xdevs" host="127.0.0.1"/>
   </components>
   <internal_connections>
    <connection component_from="fxu" port_from="out"
                component_to="integrator" port_to="in"/>
    <connection component_from="integrator" port_from="out"
                component_to="fxu" port_to="x"/>
        ...
```

Figure 4: Code snippet for a Lorentz Attractor coupled model

The structure definition of both atomic and coupled DEVS components is easy to describe in XML following the DTD shown in Figure 2 and Figure 3.

2.2 DEVS behavior

The behavior of a DEVS model is limited to atomic components, that is, the body of the transition and output functions, as well as the time advance function. When the behavior of an atomic DEVS component is generated, all this code is transformed to JavaML. However, platform-specific sentences are transformed to platform-independent ones, extending the JavaML specification with new tags such as *platform-specific*, *state-assignment*, *port-read*, etc. For example, if an atomic component contains DEVSJAVA code such as:

```
void deltint() {
 holdIn("standby", 3.5); // DEVSJAVA library function call
}
```

It is transformed to DEVSML using the following extension of JavaML:

```
<deltint>
 <block>
  <platform-specific>
   <state-assignment op="=">
    <lvalue>
     <var-ref name="phase"/>
    </lvalue>
    <literal-string value="standby"/>
   </state-assignment>
   <state-assignment op="=">
    <lvalue>
     <var-ref name="sigma"/>
    </lvalue>
    <literal-number kind="float" value="3.5"/>
   </state-assignment>
  </platform-specific>
 </block>
</deltint>
```

Using these special tags, a DEVSML model can be transformed to any supported DEVS simulation engine. The main disadvantage of using this methodology is the portability between different programming languages (for example, between Java and C++). Every programming

language has its native as well as external libraries to simplify applications code, (e.g., containers, numeric libraries, etc). With DEVSML, transformations between DEVS simulation engines developed in different programming languages would be possible only if no specialized functions or libraries were used. However, the DEVSML reverse engineering provides full interoperability between models developed in the same programming language which is a great advantage when models from different repositories are used to compose bigger coupled models using DEVSML seamless integration capabilities. Furthermore, all the DEVSML files generated can be stored in a Library for reuse. The composed integrated model is complete in every respect as it contains behavior and is ready for simulation. Based on the information contained in the DEVSML model description, the corresponding simulator is called; after the simulator has instantiated the model, the model executes with the designated simulator. DEVSML provides new concepts for easily integrating reverse engineering into a DEVS/XML standard providing both interoperability and reuse. Note that in this example two DTDs were created instead of two XML Schemas (the reason is that JavaML is defined in terms of DTDs; the DTD definition is included in the DEVS/XML standard).

2.3 A DEVSML application

Figure 5 shows the Java application that implements DEVSML principles. It contains two simulators, namely, xDEVS and DEVSJAVA, demonstrating the validation of DEVSML atomic and coupled models with the same Atomic and Coupled DTDs. It converts any atomic/coupled model from their Java implementation to DEVSML and *vice versa*. It also validates any DEVSML model description and integrates any coupled DEVSML description into a composite DEVSML coupled model ready to be simulated with the target simulator. The tool also generates a Java code library from a composite DEVSML coupled model. It contains various web services in operation. The five web services that are publicly offered are: (1) convert Java model to DEVSML; (2) convert DEVSML to Java code; (3) validate the existing DEVSML model; (4) integrate coupled and atomic DEVSML models towards a portable 'Composite' Coupled DEVSML file that is simulatable at any remote server; and (5) simulate the Composite Coupled file and send console messages at the Server to the Client window providing evidence that the simulation is running. The users can select their own Source and Target directories and can choose their implementation, that is, Java code and Simulator compatibility. The Server application checks for compatibility as well.

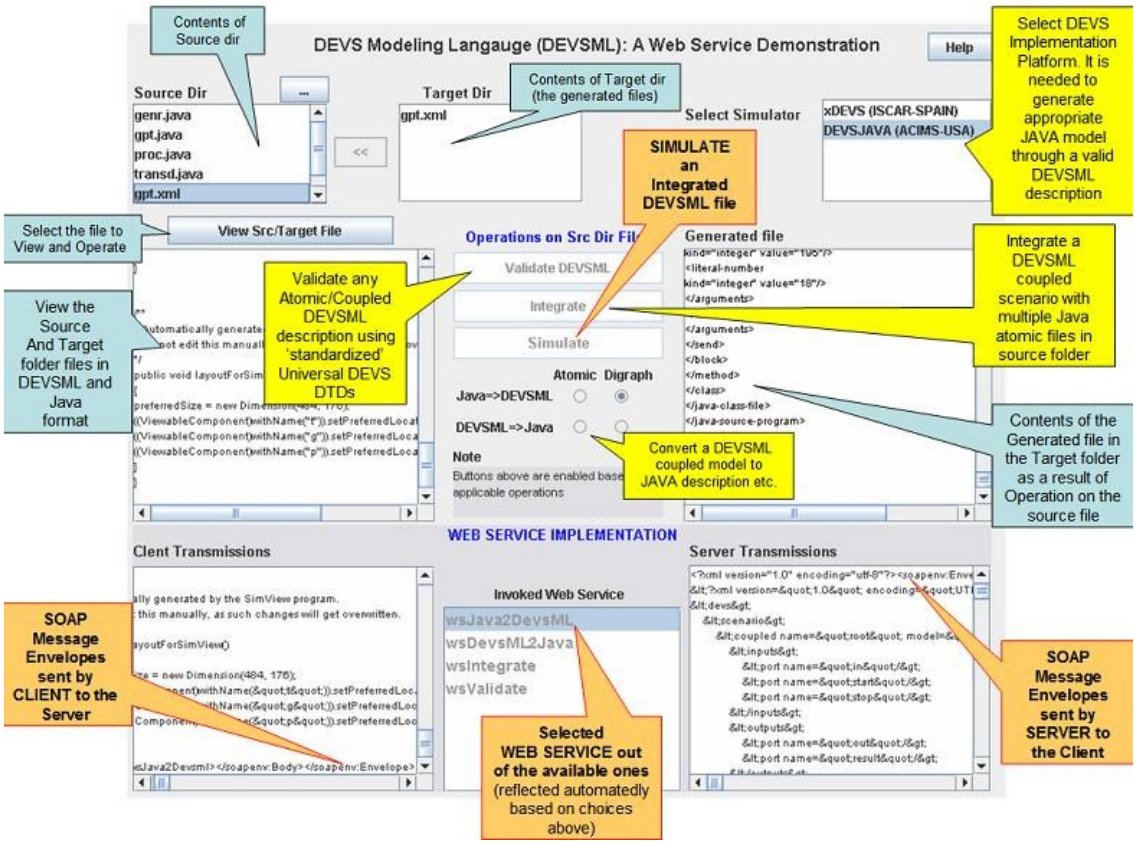

Figure 5: Client application snapshot implemented as an applet

To conclude, DEVSML partially solves the interoperability in DEVS/XML, and fully supports the reuse between different simulation platforms by means of reverse engineering.

3 DEVS/SOA

This section presents DEVS/SOA in detail. We start by introducing how DEVS/SOA works in general terms, from a user point of view. Next, we present an overview and the software architecture of DEVS/SOA [3] and its implementation using web standards. Finally, we show how the DEVS/SOA framework is able to integrate several DEVS and non-DEVS models.

As stated before, there are many libraries for expressing DEVS models. All have efficient implementations for executing the DEVS protocol and provide advantages of Object Oriented frameworks 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#).

First of all, DEVS/SOA is another DEVS simulation platform. Implemented in both Java and .NET programming environments, DEVS/SOA includes a modeling library as well as a simulation library. The main difference with respect to other DEVS simulation libraries is that the modeling package (including both atomic and coupled classes) are “adapters” of other DEVS simulation engines. For instance, as seen in Figure 27, a DEVS/SOA atomic component can be written in DEVSJAVA or xDEVS (as well as its own DEVS/SOA native atomic component). Thus, DEVS/SOA allows interoperability among different DEVS simulation engines using diverse web services infrastructures.

Figure 6. Heterogeneous DEVS/SOA model

The same situation happens in the DEVS/SOA .NET infrastructure. The "magic" of interoperability between DEVS/SOA Java and DEVS/SOA .NET resides in the message-passing interface. As Figure 28 shows, when a simulator must send a message to another one, all the objects (or entities) included in the message are passed from one simulator to the other in XML format. To this end, some intermediate libraries such as JAXB in Java and native .NET transformations are used to transform an object representation to its XML equivalent (both at the model level, not at the simulator level). Managing messages in XML format makes possible the publication of web services, giving DEVS/SOA the ability to publish simulators (and even atomic or coupled models) as web services.

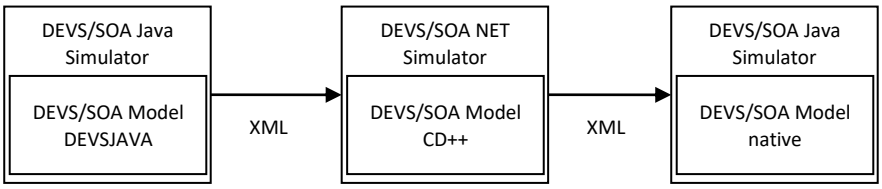

Figure 7. Interoperability in DEVS/SOA

How to compose a DEVS/SOA model? As stated before, each DEVS M&S library is developed using a particular programming language (Java, C, C++, C#, etc.). Since DEVS/SOA provides interoperability, the composition of the desired DEVS model at the client application level is completed using a platform independent mechanism. Therefore, our approach describes DEVS coupled models exploiting the XML description format. Figure 8 shows the DEVSML representation of a DEVS coupled model including an Experimental Frame and a Processor [12] using three different platforms: xDEVS, DEVSJAVA, and DEVS.NET. The DEVS/XML notation used to define the structure of DEVS/SOA distributed and interoperable models is based on DEVSML (see Section **Error! Reference source not found.**). The root coupled element determines the root model identification and the localization of the DEVS web coordination service. The children nodes of the root element represent DEVS atomic models, connections among DEVS models, or other DEVS coupled models (in a recurrent manner). The data included for each atomic model item are: its name, the platform where the model can be simulated, the class that implements the model that must be instantiated, the localization of the DEVS web simulation service that has access to the model implementation class repository, and finally, the names of the input and output ports of a model as well as the class types of their messages (as internal elements). The connection items designate links between a model output and a model input port.

```
<?xml version="1.0" encoding="UTF-8"?>
 <coupled name="efp" host="http://192.168.1.2:8080/axis2/services/Coordinator">
  <coupled name="ef" platform="devsjava" class="Ef“ host="http://192.168.1.4:8080/axis2/services/Simulator">
   <atomic name="generator" platform="xdevs" class="Generator“ host="http://192.168.1.4:8080/axis2/services/Simulator">
    <inport name="stop" class="Job"/>
    <outport name="out" class="Job"/>
   </atomic>
   <atomic name="transducer" platform="xdevs" class="Transducer“ host="http://192.168.1.4:8080/axis2/services/Simulator">
    <inport name="solved" class="Job"/>
    <inport name="arrived" class="Job"/>
    <outport name="out" class="Job"/>
   </atomic>
   <inport name="in" class="Job"/>
   <outport name="out" class="Job"/>
   <connection atomicFrom="ef" portFrom="in" atomicTo="transducer" portTo="solved"/>
   <connection atomicFrom="generator" portFrom="out" atomicTo="ef" portTo="out"/>
   <connection atomicFrom="generator" portFrom="out" atomicTo="transducer" portTo="arrived"/>
   <connection atomicFrom="transducer" portFrom="out" atomicTo="generator“ portTo="stop"/>
  </coupled>
  <atomic name="processor" platform="devs.net" class="Processor“ host="http://192.168.1.3:3158/Simulator.asmx">
   <inport name="in" class="Job"/>
   <outport name="out" class="Job"/>
  </atomic>
  <connection atomicFrom="ef" portFrom="out" atomicTo="processor" portTo="in"/>
  <connection atomicFrom="processor" portFrom="out" atomicTo="ef" portTo="in"/>
</coupled>
```

Figure 8. EFP DEVS/SOA model structure (DEVSML)

Summarizing, from a user's perspective, the use of DEVS/SOA M&S implies:

- Compose a DEVS model according to the localization of the original submodels.
- Write a DEVSML file as a result of the modeling phase.
- Partition the composed model either manually or through the DEVS/SOA automated mechanism
- Deploy the models
- Run the distributed simulation.
- Wait for the results.

The main advantage of DEVS/SOA with respect to other distributed M&S methods is that the original models remain encoded in the native programming language related to the selected DEVS library. Therefore, from a user point of view, no middleware is needed (although of course DEVS/SOA is built using web services technologies), and, most important, the software engineer does not need special distributed programming skills.

3.1 DEVS/SOA software architecture

In DEVS/SOA, an implementation of the DEVS formalism is developed within SOA so as to provide DEVS M&S services over the WWW. DEVS/SOA allows distributing simulations over multiple processors in the same computer, over Internet, or over both. A DEVS coupled model is split into different submodels that run on multiple computers in a distributed manner. For this purpose, the DEVS/SOA coordinator deals with simulators hosted anywhere on the web. Every DEVS/SOA simulator is responsible for managing its corresponding DEVS atomic model. Communications between simulators and coordinators are performed using standard technologies. As a result, DEVS models can be composed in a transparent, open, and scalable way. Another advantage of this architecture is the interoperability among DEVS submodels. Since the communication among simulators is performed in a standard way such as XML, the associated DEVS submodels can be implemented using different DEVS simulation engines.

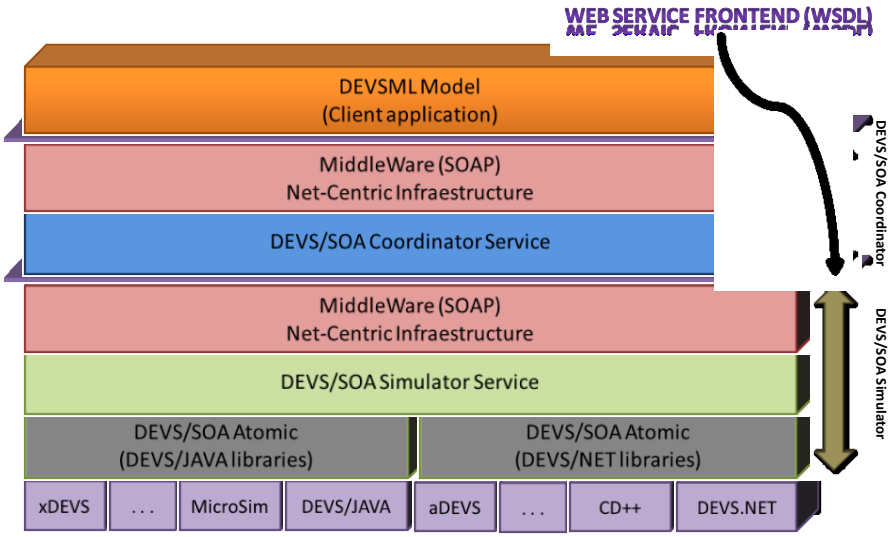

Figure 9. DEVS/SOA architecture

The proposed DEVS/SOA architecture, whose basics are captured in Figure 9, embraces three levels of interoperability between DEVS M&S elements. The outer level comprehends the interaction between the client application layer and the DEVS coordinator service. It is implemented using the SOAP communication protocol according to the operations detailed at the Coordinator WSDL file. Next, a similar interoperability level is encountered as the communication between the DEVS coordinator and simulation services. Again, the SOAP protocol specification is applied for exchanging structured information between both types of services. The operations offered by the Simulator service are encoded in the Simulator WSDL file. Finally, the third level is at the junction of a DEVS simulator with its associated native DEVS model, responsible for implementing the DEVS/SOA atomic models. The messages among different native DEVS models are expressed in XML. In the following, these three levels (modeling, simulation, and client application) are described in detail.

3.2 DEVS/SOA modeling layer

Figure 10 shows excerpts of the DEVS/SOA atomic interface, base class, and the DEVSJAVA atomic adapter. The former follows a classic DEVS atomic structure. In addition, the given interface includes functions to receive input data in XML and to obtain the output in XML (both implemented in the DEVS/SOA atomic model). The DEVS/SOA atomic model (class AtomicBase) implements the previous interface, including functions `getOutput` and `receive`, which send and receive data in XML. Finally, the DEVSJAVA adapter inherits from this DEVS/SOA JAVA atomic model and implements the DEVS/SOA atomic interface. By using these classes, heterogeneous DEVS/SOA and DEVSJAVA models can coexist in a simulation. Thus, to allow interoperability with other DEVS simulation platforms, a new adapter (only one class per simulation framework) must be implemented.

```
public interface AtomicInterface {
        // DEVS protocol
        void initialize();
        double ta();
        void deltint();
        void deltext(double e);
        void deltcon(double e);
        void lambda();
        // Source code is omitted for brevity ...
        Ports getInput();
        Ports getOutput();
        String[] getOutput(String portName); // Get values at output port in XML
```

```
        void receive(String portTo, String[] xmlValue) throws Exception; // Receive input data in XML
}

public class AtomicBase implements AtomicInterface {
        // Source code is omitted for brevity ...
        public String[] getOutput(String portName) {
                Port port = output.get(portName);
                Set<Object> values = port.getValues();
                String[] xmlValues = new String[values.size()];
                int i = 0;
                for(Object value : values) {
                        StringWriter writer = new StringWriter();
                        Result result = new StreamResult(writer);
                        JAXB.marshal(value, result);
                        String xmlValue = writer.toString();
                        xmlValues[i++] = xmlValue;
                }
                return xmlValues;
        }
        public void receive(String portTo, String[] xmlValues) throws Exception {
                Port port = this.getInput().get(portTo);
                for(int i=0; i<xmlValues.length; i++) {
                        if(xmlValues[i]!=null) {
                                StringReader reader = new StringReader(xmlValues[i]);
                                Object value =
                                        JAXB.unmarshal(reader, Class.forName(port.getPortClass()));
                                this.getInput().get(portTo).addValue(value);
                        }

                }
        }
        // Source code is omitted for brevity ...
}

public class AtomicDevsJava extends AtomicBase implements AtomicInterface {
        private genDevs.modeling.atomic model;
        public AtomicDevsJava(Element xmlAtomic) throws Exception { // Constructor
                super(xmlAtomic);
                // It processes the corresponding portion of the input DEVSML file ...
        }
        // Source code is omitted for brevity ...
        public void deltext(double e) { // external transition function
                genDevs.modeling.message msg = buildMessage();
                model.deltext(e, msg);
        }
        public void lambda() {
                genDevs.modeling.message msg = model.out();
                genDevs.modeling.ContentIteratorInterface itr = msg.mIterator();
                while(itr.hasNext()) {
                        genDevs.modeling.ContentInterface devsJavaPort = itr.next();
                        Port port = output.get(devsJavaPort.getPortName());
                        if(port!=null)
                                port.addValue(devsJavaPort.getValue());
                }
        }
        // Source code is omitted for brevity ...
        public genDevs.modeling.message buildMessage() {
                genDevs.modeling.message msg = new genDevs.modeling.message();
                Map<String, Port> ports = input.ports;
                Iterator<String> itr = ports.keySet().iterator();
                while(itr.hasNext()) {
                        String portName = itr.next();
                        Set<Object> values = ports.get(portName).getValues();
                        for(Object value : values) {
                        genDevs.modeling.content con = model.makeContent(portName, (entity)value);
                        msg.add(con);
                        }
```

```
                }
                return msg;
        }
}
```

Figure 10. DEVS/SOA JAVA atomic interface, base class, and the DEVSJAVA adapter

4 DML: DEVS Markup Language

The DEVS Markup Language DML [4] is a design that standardizes the representation of DEVS models (which can be defined in different DEVS platforms), and at the same time is as generic and flexible as possible. The idea is to use an XML representation of a specific programming language, for example the Java Markup Language, as done in DEVSML. This approach has the advantage that many tools are available to manipulate and transform the source code and the XML representation. However, it induces a strong coupling between the model and a particular language, therefore restricting its deployment scope.

To enlarge this scope, a more generic representation could be employed, such as Metal [5] or Object Oriented XML (O2XML) [6]. These markup languages enable a representation of most object-oriented programming languages by specifying a common denominator between them. Thereby, an XML document can be transformed into source code in any object-oriented target language. Though this solution seems attractive, it prevents the modeler from using certain language features or libraries (e.g., pseudo-random numbers generation with the Stochastic Simulation library [7]; matrix operations using Boost uBLAS, etc.). Consequently, such a high-level language with a small set of features would reduce the universality of the standard, and make the representation of legacy models very difficult as they are usually written in a specific programming language and use advanced language functionalities and libraries.

To resolve this issue, one should let the modelers use their favorite language and delegate the task of interpreting them to simulators or proxies. That would imply developing an architecture in which a model is composed of several modules in multiple languages, this model being itself simulated by an application possibly written in yet another language. Even though this architecture is conceivable (using technologies such as JNI, RMI, CORBA, and Web Services), the complexity and the loss of performance would be high.

DML tries to reach a compromise between generality and flexibility, by defining an XML representation of source code.

4.1 DEVS structure

Describing the structure of DEVS models is done by translating the set of formalism into an XML representation. Input and output ports, parameters, and state variables are characterized by a name and a type. To allow the deployment of the model on any simulator, this type should be as generic as possible. However, in real-world applications, the modeler may need to use certain types defined in particular language libraries. To satisfy these apparently contradictory requirements, three different kinds of types must be considered:

- intrinsic types, which are natively supported by all object-oriented languages (e.g., integer, string, float, etc.),
- custom types, which are types that are defined by the modeler in a language independent manner, and
- language dependent types, which have to be bound to a particular type, for each targeted languages. This notion, close to the one of Abstract Data Types, lets the modeler conceive algorithms using abstract types that must be matched to library or language types in order to generate the source code on the target platform.

Figure 11 shows an example of the use of these different types. In this example, the state variable *sigma* is a floating-point number, whereas *phase* is a more complex type, namely, an enumeration with two possible values: *busy* or *idle*. Finally, the variable *queue* has an abstract type, which we bind to specific types in Java and C++. Therefore, to make this model deployable on a simulator written in another language, the modeler has to specify what type can be used as a queue in this language. We can also imagine having repositories of abstract types along with their bindings in numerous programming languages, to allow their reusability in several models.

```
<state>
  <variable>
    <name>phase</name>
    <type kind="custom">procPhase</type>
  </variable>
  <variable>
    <name>buffer</name>
    <type kind="languageDependent">Queue</type>
  </variable>
  <variable>
    <name>sigma</name>
    <type>double</type>
  </variable>
</state>
...
<type id="procPhase" xsi:type="enum">
  <value>busy</value>
  <value>idle</value>
</type>
<type id="Queue" xsi:type="languageDependent">
  <implementation lang="java">
    java.util.LinkedList<String>
  </implementation>
  <implementation lang="c++">
    std::queue<std::string>
  </implementation>
</state>
```

Figure 11: Sample of XML representation of state variables

The XML Schema for coupled DEVS models includes ports, components, and coupling declarations, as well as a tie-breaking function in the case of classic DEVS. Components are described by their name and the type of model they instantiate. The couplings are pairs of ports, if necessary qualified by their component's name. Finally, the *Select* function is described by ordered sets indicating priorities among components.

4.2 DEVS Behavior

One of the main issues in standardizing behavior of atomic DEVS models boils down to the following question: How can we describe application logic in a language-independent manner? Several efforts have been directed towards providing an XML representation of source code. Some of them are oriented towards a sole programming language, for example, JAVAML [8], PascalML [9], or CppML [10], while others propose vocabularies that contain a common denominator between object-oriented languages [6][10].

None of these frameworks is generic or open to extension with language dependent features. DML proposes a compromise between both approaches, by drawing inspiration from the concept of weaving used in Aspect Oriented Programming [11], or more simply from the C and C++ macros. The main concept is to use a pseudo-language such as O2XML so that the major part of the code can be reused across different simulators, while supporting the possibility to include language specific code snippets. This concept is illustrated in Figure 12, where generic programming constructs are represented in XML, and use of libraries is abstracted and tied to

the specific implementations in several languages. This semi-generic language can also be used to create custom types (i.e., classes, enumerations), which become usable in the variables and ports declarations.

```
<if>
  <test>
    <binaryExpr op="equality">
      <lhs>
        <var>phase</var>
      </lhs>
      <rhs>
        <enumValue>
          <enum>procPhase</enum>
          <value>idle</value>
        </enumValue>
      </rhs>
    </binaryExpr>
  </test>
  <then>
    <codeSnippet id="enqueue" queue="buffer" element="job" />
  </then>
</if>
...
<codeSnippet id="enqueue">
  <parameter name="queue" />
  <parameter name="element" />
  <impl lang="java">
    queue.offer(element);
  </impl>
  <impl lang="c++">
    queue.push(element);
  </impl>
</codeSnippet>
```

Java

```
if (phase == procPhase.idle)
{
  queue.offer(element);
}
```

C++

```
if (phase == procPhase::idle)
{
  queue.push(element);
}
```

Figure 12: Representation of model dynamics using a semi-generic language

This approach has two advantages:

- It allows the use of programming language features and libraries.
- It permits the representation of legacy models written for a specific simulator.

By limiting the language-dependent code snippets to a minimum, the migration of models between platforms becomes easier, and thereby their portability and reusability are enhanced.

4.2.1 Parameterization and initialization

The above description of DEVS models is not sufficient to produce an executable specification. Indeed, models can have parameters, which need to be set prior to simulation, and their initial state must be specified in some way.

Figure 13 depicts how a model, regarded as a template, can be instantiated into several parameterized models depending on the values assigned to each parameter. After that, the state must be initialized to obtain a ready-to-simulate model. A simulator can then be fed with this

executable model, along with an input trajectory if need be, and produce the output of the simulation.

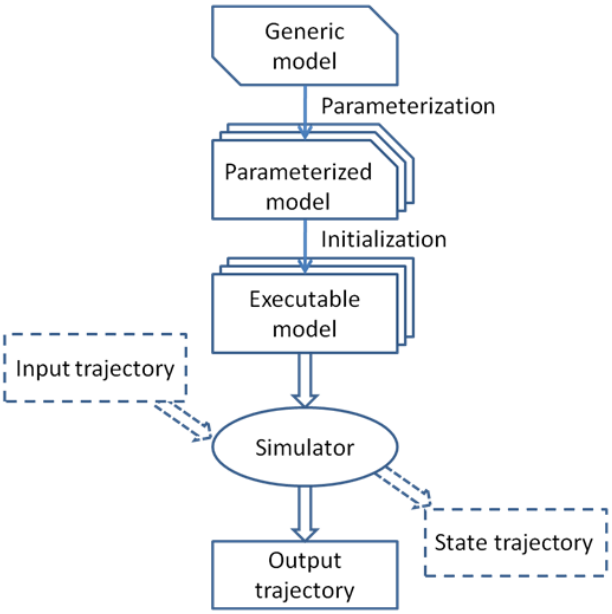

Figure 13: Parameterization and initialization of a model

To perform these operations, two approaches are possible depending on the degree of freedom required. The simplest is to restrict parameterization and initialization to pairs of variable identifiers and values. However, these operations can sometimes be more complex than mere assignments (e.g., for file manipulation or random number generation). Therefore, in DML these two activities are described in a similar fashion as the model dynamics, using the source code representation outlined in the previous paragraph.

Consequently, the modeler needs to define two functions for each atomic model in the hierarchy: one to set the parameters values, and the other one to initialize the state. Ideally, these functions should be kept separate from the model definition, so that it can be easily reused with different scenarios.

In order to simulate a DEVS model, four specifications are needed: structure, dynamics, parameterization, and initialization. Given these specifications and if required an input trajectory, a simulator can then produce results representing the model behavior.

4.2.2 Model behavior

The behavior of a model is normally fully represented by its output trajectory. However, it is common to desire its state trajectory and, in the case of coupled models, the behavior of its components as well. These two aspects can be described by sets of values annotated with a timestamp. Figure 14 shows a sample state trajectory of a coupled model, namely the *pipeSimple* defined in [12].

The XML schema for this representation of a trajectory can be deduced from the model structure: each variable or port becomes an XML tag accepting a value in accordance with its type. Another solution would be to have a tag *variable* with an attribute containing the name of the variable.

In the case of variables of complex type, the value at a given time can be obtained by serializing the object from memory, as is done when exchanging data between web services. This supposes

providing an XML Schema describing the structure of the object, for later use (e.g., in a visualization tool). For custom types, this schema can be deduced from the type definition made in the model specification; for language dependent types, the schema can often be automatically generated from the source code.

```
<behavior id="pipeSimple">
  <state>
    <snapshot time="0">
      <component id="p0">
        <phase>idle</phase>
        <queue />
        <sigma>infinity</sigma>
      </component>
      <!-- other components -->
    </snapshot>
    <snapshot time="3">
      <component id="p0">
        <phase>busy</phase>
        <queue>
          <elem>job1</elem>
        </queue>
        <sigma>10</sigma>
       </component>
       <!-- and so on -->
```

Figure 14: Sample state trajectory

This XML representation of trajectories has the valuable advantage that it can be easily processed by a machine while remaining human readable. However, because of the potentially huge size of data generated, it may be necessary to switch to a more compact representation, such as the eXternal Data Representation standard [13].

5 DCD++

Distributed CD++ (DCD++) **Error! Reference source not found.Error! Reference source not found.** supports simulation over SOA in two versions: SOAP-based Web-service **Error! Reference source not found.**[14] and RESTful Web-services **Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.**. The Distributed DEVS Simulation Protocol (described in Section 5.2) is based on SOAP DCD++ whereas RISE (described in Section 5.1) is based on RESTful DCD++. Both versions provide modelers with the complete required cycle to run a simulation experiment. This includes submitting a model to be simulated (i.e., usually partitioned over different machines) and retrieving simulation results. Both versions decouple model representations from simulation, hence different simulation engines can collaborate in the same simulation run. This enables different model representations to participate in the same simulation run, hence allowing DEVS simulation engines to execute legacy models and standardized model representations. In this section, we discuss the modeling structure and partitions for both approaches followed by DEVS model behaviors.

5.1 RESTful Interoperability Simulation Environment (RISE)

RESTful Interoperability Simulation Environment (RISE) provides lightweight thin middleware based on RESTful Web-services. RISE is independent of any simulation environment and provides interoperability at three levels. First, the *interoperability framework architecture* level provides the URI templates API that allows modelers to create simulation/modeling environment (including modeling configuration, distributing simulations, starting simulation and retrieving results). Second, the *model interoperability* level provides XML rules for binding different models together. Third, the *simulation synchronization* level provides high-level

simulation algorithms and synchronization channels. The first and third levels are discussed in Chapter 18 whereas the second level is discussed in this Section.

RISE Experiment framework is a URI created by the modeler (see URI templates API in Chapter 18) to contain a domain, where a domain is a simulation environment that contains a model and simulation engine. RISE completely wraps a domain interior, hence simplifying interoperability between heterogeneous domains. RISE modeling interoperability level assumes an entire model is placed in a domain, hence a domain at this level can be viewed as a model wrapped within a URI (e.g. …/myFireModel), as shown in Figure 15. RISE treats all models (domains) as black boxes with input and output ports. In other words, each model views other heterogeneous models as part of its environment. It is worth to note that a model (within a domain) may be partitioned among different processors, simulated by conservative/optimistic algorithms according to DEVS/none-DEVS formalism. However, at the RISE layer the model is seen as a URI (which contains both the model and the simulation engine that simulates it).

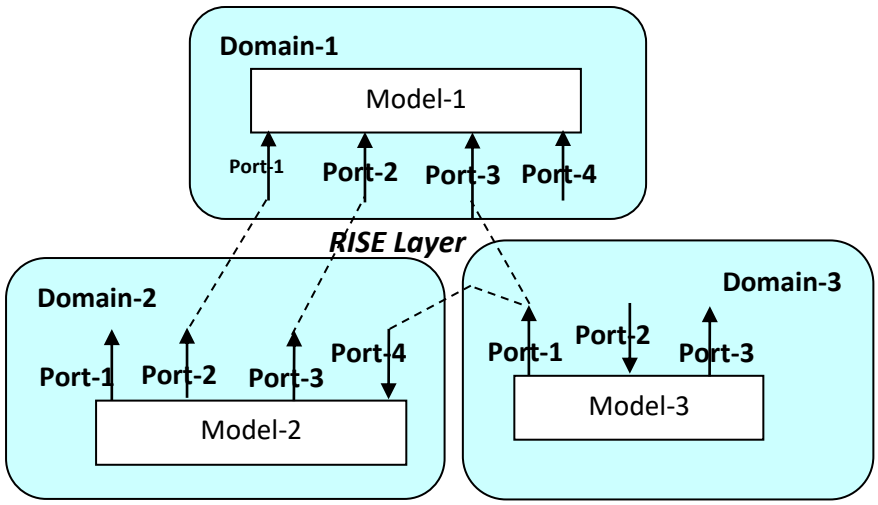

Figure 15: RISE Domain Models Interconnections

Modelers need to configure the experiment frameworks by sending XML configuration documents via PUT channels to their URIs (e.g. …/MyFireModel). This represents models interconnections shown in Figure 15 in the XML document. Note that the model interconnections map is converted into routing tables by simulation engines (see Chapter 18), allowing them to route messages during simulation according to the model URI and message input port. Further, plug-and-play capability of RESTful Web-services takes this configuration to another level. This plug-and-play level allows modelers to overwrite model interconnection during simulation runtime, allowing simulation engines to create routing tables. Thus, models (without prior knowledge) may join or leave a simulation session at runtime. In such a case, the XML configuration is sent to the simulation engine URI (…/MyFireModel/simulation).

In Figure 15, Model-2 influences Model-1 via ports-2 and 3 of Model-2. Model-3 influences both Model-1 and Model-2 via Port -1 of Model-3. This implies that the output ports of a model can generate RISE external messages addressed to their connected input ports during simulation, thereby influencing the receiving models. Note that Port-1and Port-3, of Model-3 and Port-4 of Model-1 are not used by RISE. These ports can still be used by modelers, for instance, to influence a model dynamically during active simulation. Figure 16 shows the XML configuration document for the example in Figure 15. This document needs to be submitted to each domain (typically constructed and submitted by client software). The RISE configuration must be the enclosed element <RISE> body (which starts on line 4 in Figure 16). The domains are free to define their own specific configuration outside the <RISE> block. Line #3 shows RISE protocol version, and simulation type where “C” type stands for Conservative-based and “O” type for Optimistic-based simulation. Lines 4-32 define domains (models) configurations. Line #5 defines the Main domain URI. The Main domain is needed during active simulation,

discussed in Chapter 18. Lines 6-31 define RISE ports interconnections. For example, Lines 7-12 define the connection from port #2 in URI <…/Domain-2> to port #1 in URI <…/Domain-1>.

```
1 <ConfigFramework>
2     …
3     <RISE Version="1.0" Type="C">
4      <Domains>
5       <Main><URI>…/Domain-1</URI></Main>
6        <Links>
7           <Link>
8            <From><Port>Port-2</Port>
9              <URI>…/Domain-2</URI></From>
10           <TO><Port>Port-1</Port>
11           <URI>…/Domain-1</URI></TO>
12          </Link>
13          <Link>
14           <From><Port>Port-3</Port>
15            <URI>…/Domain-2</URI></From>
16           <TO><Port>Port-2</Port>
17            <URI>…/Domain-1</URI></TO>
18          </Link>
19          <Link>
20           <From><Port>Port-1</Port>
21             <URI>…/Domain-3</URI></From>
22           <TO><Port>Port-4</Port>
23             <URI>…/Domain-2</URI></TO>
24          </Link>
25          <Link>
26           <From><Port>Port-1</Port>
27             <URI>…/Domain-3</URI></From>
28           <TO><Port>Port-3</Port>
29             <URI>…/Domain-1</URI></TO>
30           </Link>
31       </Links>
32      </Domains>
33    </RISE>
34    …
35 </ConfigFramework>
```

Figure 16: RISE Model Domain Configuration

RISE is independent of any modeling representation or simulation engines. For example, CD++ uses RISE to simulate different distributed model partitions. In this case, the CD++ domain is viewed as a single model that is executable by a CD++ engine. In the RESTful DCD++ **Error! Reference source not found.Error! Reference source not found.**), the modeler partitions the DEVS or Cell-DEVS model among different RISE instances. Note that the CD++ interior domain is hidden behind a single URI, as discussed previously, even though it is using other RISE middleware instances to simulate its own domain model partitions. RISE in this case is a server with an IP address and port number, hence RISE is usually placed on a machine by itself. However, multiple instances of RISE may be placed on the same machine by assigning different port numbers, usually for testing purposes. For example, assume a modeler created experiment framework *<.../cdpp/sim/ workspaces/Bob/DCDpp/Producer_Consumer>*. The RISE server then creates experiment *Producer_Consumer*. Workspace *Bob* and service *DCDpp* are created, if they do not already exist. *DCDpp* service indicates that the simulation is using the DCD++ engine to execute the model. Assume further that the modeler wants to place atomic model "*Producer*" on a server while atomic model "*Consumer*" is placed on a different server. In this case, the modeler needs to send the following partitioning (Figure 17) to the experiment URI:

```
<DCDpp>
 <Servers>
```

```
  <Server IP="10.0.40.8" PORT="8080">
    <MODEL>Producer</MODEL></Server>
  <Server IP="10.0.40.9" PORT="8282">
    <MODEL>Consumer</MODEL></Server>
   </Servers>
 </DCDpp>
```

Figure 17: RESTful DCD++ Model Partitions by a Modeler

Though it is easier for a modeler to partition a model in terms of IP address and port number, the simulation is conducted among URIs (which allows multiple users and experiments to be performed simultaneously). Thus, the main server must convert the modeler documents into the format shown in Figure 18. In this case, the main server assigns an ID to every server in the distributed simulation where "0" is its own ID. It further creates a name for each server (i.e., a combination of the server IP and port number). This name is the server user account on all other servers. This is needed because all messages during simulation are authenticated according to a user name and password. Furthermore, it assigns a URI for each model partition. The model partition at the main server is assigned the URI created by the modeler while other server partitions are assigned slightly modified URIs. This is because the main server needs to create experiment partitions on other supportive servers similar to other clients. This also ensures URI uniqueness on all servers, since supportive servers on an experiment may be used as main servers by other modelers for other experiments.

```
<DCDpp>
 <Servers>
  <Server ID="0" NAME="10_0_40_8_8080">
    <URI>…/workspaces/Bob/DCDpp/Producer_Consumer</URI>
    <MODEL>Producer</MODEL></Server>
  <Server ID="1" NAME="10_0_40_9_8282">
    <URI>…/workspaces/10_0_40_8_8080/DCDpp/Producer_Consumer_Bob</URI>
    <MODEL>Consumer</MODEL></Server>
 </Servers>
 </DCDpp>
```

Figure 18: RESTful DCD++ Model Partitions by the Main Server

5.2 Modeling using the Distributed DEVS Simulation Protocol

In [14], a flexible and scalable XML-based message-oriented mechanism was developed to allow interoperability between different DEVS implementations based on the SOAP-based DCD++ **Error! Reference source not found.** simulation package. This section focuses on the modeling part while simulation synchronization is described in Chapter 18. The interoperability is achievable with minimum design changes to each DEVS implementation, mainly by hiding the detailed implementation behind a *wrapper* (i.e., a SOAP port) and focusing only on the exchanged XML messages. This method assumes the simulation is conducted among different domains where a domain contains a DEVS model (with an engine capable of executing that model). The model in this case represents a complete reusable feature that a modeler wants to plug into an overall model hierarchy. The main idea behind modeling here is wrapping all models across domains within a single DEVS coupled model, which can be simulated by the main domain simulation engine while other domains only simulate their portion of the overall coupled model. For example, Coupled 1 and Coupled 2 are wrapped within Coupled 0, as shown in Figure 19. This information is described in the Model structure XML document.

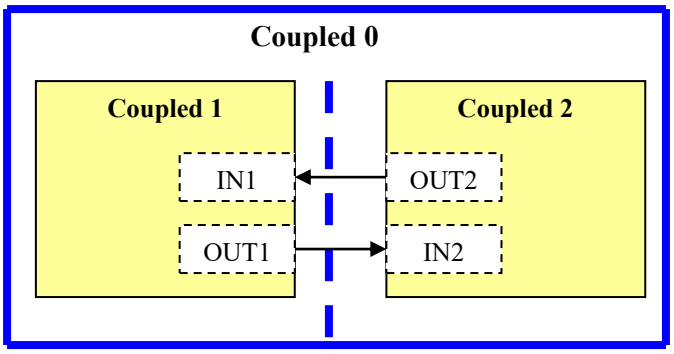

Figure 19: Coupled model partitioned across DEVS Domains

The Model structure XML document (shown in **Error! Reference source not found.**) 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 domain can identify the participant supportive 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. The model structure document contains the following information (**Error! Reference source not found.**):

- 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 in Chapter 18).

The DEVS models hierarchy can easily be mapped into this XML document. For example, assume two models connected with each other as in Figure 19 (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 (Coupled 0), resulting in the XML document shown in **Error! Reference source not found.**. This XML document also serves as an agreement contract between various implementations on the used synchronization schemes. For example, the COUPLED_SYNC keyword can be used to choose different coordination schemes 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 20: XML Model Structure Document Example

A given domain can use any model representation, since each domain should contain a simulation engine capable of executing that model partition. For example, the SOAP-based DCD++ **Error! Reference source not found.** partitions a model among different machines where each machine is wrapped within a SOAP port. Figure 21 shows an example of such partition. Lines 2-11 list all machines participating in the distributed simulation. Each machine has a rank (i.e., id) and a SOAP-port URI. Lines 12-19 describe the model partitions. In this example, a 30x30 Cell-DEVS fire coupled model (i.e., each cell is an atomic model) is split between two machines.

```
1 <Grid>
2  <MACHINES>
3     <MACHINE>
4           <MACHINE_RANK>0</MACHINE_RANK>
5           <MACHINE_URI>http://.../CDppPortType</MACHINE_URI>
6     </MACHINE>
7     <MACHINE>
8           <MACHINE_RANK>1</MACHINE_RANK>
9           <MACHINE_URI> http://.../CDppPortType</MACHINE_URI>
10    </MACHINE>
11 </MACHINES>
12 <MODEL_PARTITIONS>
13    <PARTITION machine="0">
14          <ZONE>fire(0,0)..(14,29)</ZONE>
15    </PARTITION>
16    <PARTITION machine="1">
17          <ZONE>fire(15,0)..(29,29)</ZONE>
18    </PARTITION>
19 </MODEL_PARTITIONS>
20 </Grid>
```

Figure 21: SOAP-based DCD++ Cell-DEVS Model Partitioning

5.3 CD++ DEVS Model Structure and Behavior

CD++ DEVS modeling behavior is the same for both SOA versions. This is mainly because DCD++ separates SOA (RESTful or SOAP Web-services) from the actual modeling/simulation environment. This separation allows CD++ to be portable to other platforms other than SOA and to keep the door open for algorithms optimization and other improvement. In other words, separation between simulation environment and Web-services allows both parts to evolve independently with less complexity.

CD++ represents its coupled models in textual format by connecting ports of models together similar to what has been discussed so far. The coupled model representation is parsed and its information is loaded into a coordinator where it is simulated. For example, Figure 22 shows a CD++ coupled model for a barbershop. In this example, "top" coupled model is defined in lines 2-27. Lines 4-7 lists the models enclosed in the "top" coupled model. In this case, Line #5 shows an atomic model instance named "reception" defined in C++ class "Reception". This allows more instances of an atomic model to be used, if needed. Line #6 declares a coupled model named "Barber" and its definition should come later. Lines 8-13 define "top" model input/output ports. Lines 14-25 define port connections for the "top" model. For instance, Lines 15-23 define a connection from input port "newcust" of the "top" model to port "newcust" of the "reception" atomic model. Lines 29-32 define the "Barber" coupled model in a similar way to the "top" model. Lines 33-39 initialize certain values for the "reception" atomic model.

```
1 <Models>
2    <Model Type="Coupled">
3      <Name>top</Name>
4      <Components>
5         <Name Type="Atomic" Class="Reception">reception</Name>
6         <Name Type="Coupled">Barber</Name>
7      </Components>
8      <Ports>
9         <Port Type="in">newcust</Port>
10        <Port Type="in">next</Port>
11        <Port Type="out">cust</Port>
12        <Port Type="out">finished</Port>
13     </Ports>
14     <LINKS>
15        <LINK>
16          <FROM>
17            <Port>newcust</Port>
18          </FROM>
19          <TO>
20            <Component>reception</Component>
21            <Port>newcust</Port>
22          </TO>
23        </LINK>
24        …
25     </LINKS>
26     …
27   </Model>
28
29   <Model Type="Coupled">
30     <Name>Barber</Name>
31    …
32   </Model>
33   <Model Type="Atomic">
34     <Name>reception</Name>
35     <Parameter>
```

```
36          <Name>numberofChairs</Name><Value>3</Value>
37       </Parameter>
38       …
39    </Model>
40  </Models>
```

Figure 22: CD++ Coupled Model Example

The CD++ provides a template for modelers where they need to overwrite four C++ methods, as shown in Figure 23. Methods initFunction, externalFunction, internalFunction, and outputFunction are wrappers to initialize model specific values, handle a message from a port, handle an internal transition, and generate outputs, respectively. The new atomic model is then inherited from the shown Atomic class in Figure 23. During simulation, those methods are invoked by the atomic model "simulator" according to the DEVS formalism.

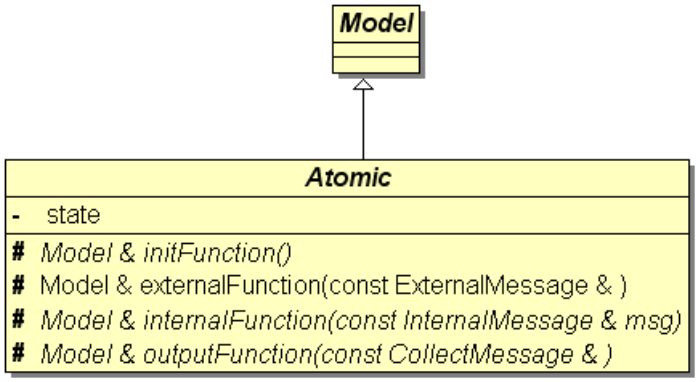

Figure 23: CD++ Atomic Model Template

6 CoSMoS

The design of CoSMoS is based on the separation of model specification and simulator protocol. This partitioning was used to develop the concept of the *Shared Abstract Model* (which will be fully discussed in the following chapter). The concept of interoperability, however, cannot be completely divorced from model specification. One simulation engine can correctly execute a model that is developed for another simulation engine if and only if both simulation engines adhere to the same mathematical specification. The implication is to have a syntactical standard for model specification that conforms to the mathematical specification.

Model specification should consist of (i) visual syntax, (ii) database syntax, and (iii) programming syntax. To achieve this objective, the Component-based System Modeling (CoSMo) framework has been developed [15][16][15]. The CoSMo has a unified framework that supports logical, visual, and database persistent modeling. It is aimed at discrete-event, discrete-time, continuous-time, and cellular automata specifications. The goal is to provide a mechanism for visual model development, and for storing logical models in databases, while also providing support and management facilities for developing a family of interrelated models. Such capabilities are useful for individual users and crucial for teams of users, for example for purposes of software/hardware co-design [18].

The top down concept of CoSMo has a visual syntax for logical models, which serves as the basis for storing logical specification of models in alternative database and flat files. Given the stored logical models, they can be translated to different programming languages for different target simulation engines. Other approaches to interoperability are generally defined in terms of programming syntax such as C++, Java, and XML—standardization is based on the traditional software engineering concepts and techniques. For example, simulation tools such as DEVS-Suite render a visual representation of models from implemented code that persist in flat files.

In CoSMo, visual and database persistence of models are proposed to be integral to standardization.

The CoSMo environment supports developing Parallel DEVS models and generating partial code for execution using DEVS-Suite. For coupled DEVS models, it generates complete code. For atomic models, it generates partial code. For the transition, output, and time advance functions only template code is generated. Other translators can be developed in a straightforward manner for other simulators such as ADEVS. As shown in Figure 24, the separation of a modeling engine and a simulation engine has important implications for standardization particularly when standardization is not at the level of XML [15] and DEVSML [19], or high-level programming languages such as Java. In effect, the CoSMo Simulator environment (called CoSMoS) is built from CoSMo and DEVS-Suite. Users can develop models using well-formed visual syntax that in turn allows generating "standardized" models stored in relational databases which in turn can be used to generate XML-based and high-level programming code. Standardization, from the perspective of the CoSMoS framework, enhances visual model development, model management via databases as well as reducing errors in programming code.

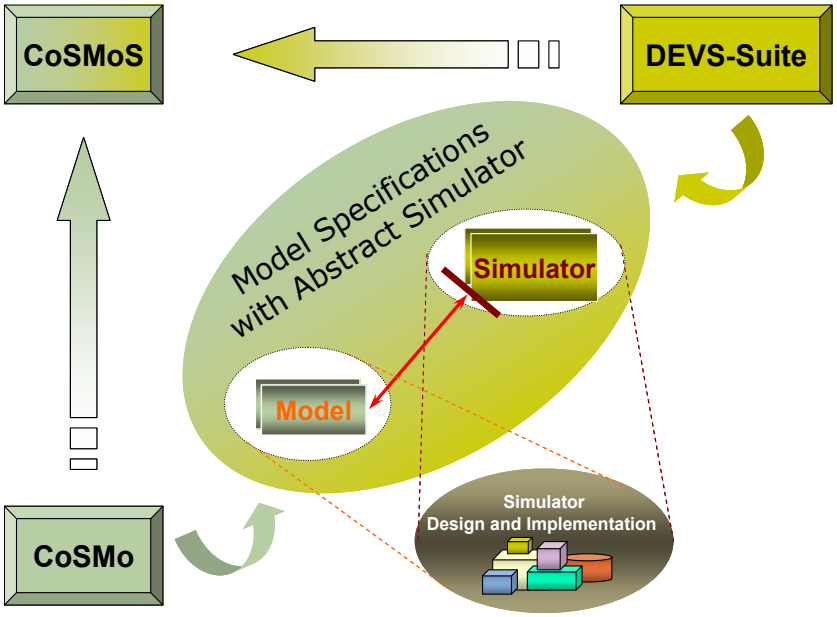

Figure 24: separation of a modeling engine and a simulation engine

The underlying concept of CoSMoS can enable interoperability among multiple simulation engines. The CoSMoS architecture and realization supports adding translators as plug-ins. The CoSMoS framework also aids in supporting interoperating simulators given current and new generations of middleware technologies such as service-oriented computing. As a result, the XML Schema translator developed for CoSMo can be used to support SOA-based distributed simulation. For example, the DEVS/SOA environment [19] can be a target simulation engine and thus writing models at the level of DEVSML, one can visually develop models that are logically correct with respect to the parallel DEVS formalism. Therefore, users can automatically generate code for atomic and coupled models given target simulation engines. The structures of the atomic and coupled models as well as their relationships are formulated as a set of tables and relations in a relational database. The axioms of CoSMo also provide a set of rules for creating a family of models such that composition and specialization relationships among different model designs do not conflict with one another.

The visual, and programming elements for DEVS mathematical specifications, and the support repositories, must account for the behaviors of atomic and coupled models. A key obstacle to

achieving this goal is the lack of visual notation for atomic models. Different authors have proposed and used variations of the Statecharts formalism to represent states and functions of atomic models. The visual statechart elements (states, function, and transitions) are adapted such that internal, external, and output functions can be distinguished (solid, small dashed, and large dashed lines are used). In other work [20], a notation that is also derived from Statecharts is developed for specifying the conditionals and distinguishing between transition functions with at=0 and ta>0. From the perspective of making DEVS accessible to a broader community of users, the UML visual notation is proposed for modeling the behavior of atomic models [21]. In this work, the developed DEVS/UML models are executed using a parallel DEVS simulation engine. Unfortunately, developing a simple visual notation suitable for complete behavior specification of atomic model remains challenging since Java and other such programming languages allow arbitrary logic and, compared to visual notation, can capture complete details of a model dynamics. Therefore, existing notations and tools can only support generating partial and complete source code for atomic models. The CoSMo modeling engine can be extended once a standardized visual notation for atomic models becomes available. The extension of the relational database and model translators to target programming code will be straightforward.

Finally, it is noted that although use of generic modeling formalisms is important, it is equally important to provide users with user-friendly, yet formally grounded, capabilities for developing simulation models. In DEVS, as in other general-purpose M&S approaches, domain knowledge must be incorporated into the generic atomic and coupled model specifications. It is only then that the model dynamics of a system can be simulated and evaluated. For certain application domains, such as electrical systems, visual notation (and thus programming code templates) are available. However, for other application domains, such as biological and social-ecological systems, universal agreements do not exist. Nonetheless, standardization for the generic atomic and coupled DEVS models is highly important and undoubtedly necessary toward developing domain-specific standards and more generally enabling a new kind of model verification and simulation validation.

7 SUMMARY

This chapter has focused on various methods for standardizing DEVS model representations, which is necessary to enable platform-independent DEVS models to interoperate with each other. The concept relies on defining a generic representation that other tools can use to understand how a model is created, which allowis sharing of existing models. Most of the examples in this chapter are based on XML descriptions that can be transformed and manipulated to achieve interoperability.

We showed how DEVSML (an XML notation built on JAVAML) can be used to transform different compatible DEVS platforms. DEVS coupled models are built using Document Type Definitions, and the atomic models are transformed into a combination of JavaML code with special tags.

DEVS/SOA includes a model library and a simulation library (implemented in Java and .Net). Interoperability is achieved by using standardized XML for message passing, and the ability to publish simulators, atomic, and coupled models as web services.

DML, the DEVS Markup Language, standardizes the DEVS model representation using an XML representation of specific languages (e.g., JavaML), and provides the common denominator of most object-oriented languages in order to allow the use of the features of language libraries. Modelers can use their favorite programming language, which is interpreted by a simulation proxy.

DCD++ supports simulation on SOAP-based or RESTful Web Services (also called the Distributed DEVS Simulation Protocol, DDSP, and the RESTful Interoperability Simulation Environment, RISE). In both cases, different simulation engines can interoperate to execute a complete simulation. DDSP introduces an XML-based environment to interoperate different DEVS implementations, using SOAP-based messages. RISE provides an interoperability framework (and API based on URIs so the modelers can create models, simulations, and experiments), a model interoperability layer (providing XML rules to combine different models together, and simulation synchronization algorithms.

Finally, CoSMoS provides a visual syntax, a programming syntax, and a database syntax that allow the models to be visually created, to be stored in a persistent data store, and overall make implementation and interoperability of components easy. CoSMoS supports the creation of models and generates code that can (partially) be executed on the DEVS-Suite environment.

These different efforts show the varied possibilities in achieving model interoperability, sharing, and reuse, based on a DEVS-based methodology.

REFERENCES

[1] R. Chreyh, G. Wainer. "CD++ Repository: An Internet Based Searchable Database of DEVS Models and Their Experimental Frames". 2009 Spring Simulation Conference - March 2009.

[2] Pieter J. Mosterman, Don Bouldin, and Andrzej Rucinski, "A Peer Reviewed Online Computational Modeling Framework," paper ID pmo131763 in proceedings of the Canadian Design Engineering Network (CDEN) 2008 Conference, Halifax, Nova Scotia, July 27-29, 2008.

[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] L. Touraille, M.K. Traore, D.R.C. Hill. "A Mark-up Language for the Storage, Retrieval, Sharing and Interoperability of DEVS Models". Proceedings of the 2009 ACM/SCS Spring Simulation Multiconference, SpringSim 2009, San Diego, CA, USA, March 22-27, 2009

[5] M. Lemos. "MetaL: An XML-based Meta-Programming Language". http://www.meta-language.net/

[6] M. Wiharto, P. Stanski. "An Architecture for Retargeting Application Logic to Multiple Component Types in Multiple Languages". Fifth Australasian Workshop on Software and System Architectures, 2004.

[7] P. L'Ecuyer, L. Meliani, J. Vaucher. "SSJ: A Framework for Stochastic Simulation in Java". Proceedings of the 2002 Winter Simulation Conference.

[8] G. J. Badros, "JavaML: a markup language for Java source code," Computer Networks, vol. 33, no. 1, pp. 159-177, 2000.

[9] G. McArthur, J. Mylopoulos, S.K. Ng. "An Extensible Tool for Source Code Representations Using XML". Proceedings of the Ninth Working Conference on Reverse Engineering. 2002.

[10] E. Mamas, K. Kontogiannis. "Towards Portable Source Code Representations Using XML". Proceedings of the Seventh Working Conference on Reverse Engineering. 2000.

[11] G. Kiczales et al. "Aspect-Oriented Programming". Proceedings of the European Conference on Object-Oriented Programming. 1997.

[12] B. Zeigler; T. Kim; H. Praehofer. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press. 2000.

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

[14] 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.
[15] H.S. Sarjoughian, and R. Flasher, (2007), System Modeling with Mixed Object and Data Models. DEVS Symposium, Spring Simulation Multi-conference, Norfolk, VA, USA.
[16] H.S. Sarjoughian, V. Elamvazhuthi, (2009), “CoSMoS: A Visual Environment for Component-Based Modeling, Experimental Design, and Simulation”, Proceedings of the International Conference on Simulation Tools and Techniques, March, 1-9, Rome, Italy.
[17] W. Hu and H. S. Sarjoughian, (2005), “Discrete event simulation of network systems using distributed object computing,” Proceedings of the Intl. Symposium on Performance Evaluation and Telecom. Systems, San Diego, CA, pp. 884-893.
[18] M. Traoré, (2009), “A Graphical Notation for DEVS”, High Performance Computing & Simulation Symposium, Proceedings of the Spring Simulation Conference, March, San Diego, CA, ACM Press.
[19] S. Mittal, J. L. Risco-Martín, and B. P. Zeigler, “DEVSML: automating DEVS execution over SOA towards transparent simulators,” in Spring Simulation Multiconference, SpringSim 2007, Norfolk, Virginia, USA, 2007, pp. 287-295.
[20] D. Huang, H.S. Sarjoughian, (2004), “Software and Simulation Modeling for Real-time Software-intensive System”, The 8th IEEE International Symposium on Distributed Simulation and Real Time Applications, 196-203, October, Budapest, Hungary.
[21] J. Mooney, H.S. Sarjoughian, (2009), “A Framework for Executable UML models”, High Performance Computing & Simulation Symposium, Proceedings of the Spring Simulation Conference, March, San Diego, CA, ACM Press.