An experiment on the interoperability of DEVS implementations

Stephen Lombardi
Gabriel A. Wainer

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON, K1S 5B6, Canada

slombardi@connect.carleton.ca
gwainer@sce.carleton.ca

Bernard P. Zeigler

ACIMS - Arizona Center for Integrated Modeling and Simulation
Department of Electrical and Computer Engineering
The University of Arizona,
Tucson, AZ 85715.

zeigler@ece.arizona.edu

ABSTRACT: *The DEVS formalism defines a theory for discrete-events systems specification. It is a formal approach to build the models, using a hierarchical and modular approach. DEVS formal nature showed to be useful for easy reuse of models that have been validated. In this way, the security of the simulations can be improved, reducing the testing and maintenance times, and improving the productivity of the development process. The discrete-event nature of the formalism also allows reducing the execution times of complex simulations. In this work we will discuss the results of an experiment on the interoperation between two existing DEVS environments (namely, CD++ and DEVS/C#), in an effort within the DEVS Standardization study group. This work would provide the basis for future discussion on the standardization effort, by providing an actual experimental result on sharing of DEVS models developed by different teams using different DEVS simulation engines, permitting discussing the basic issues involved in this effort. We will present the basic API provided by the engines, how to use them to provide interoperability at the level of the models, and a detailed discussion on interoperation of the underlying simulators.*

1. Introduction

In recent years, we have witnessed tremendous advances in model building and simulation execution thanks to the improvements in software and hardware technology. The definition of the High Level Architecture (HLA) standard [1] raised fundamental issues, such as model credibility and interoperation. The HLA focuses on the interoperation of existing geographically dispersed simulation assets. However, the HLA does not address how to solve the problem of creating models to be executed in the simulation environment. Current practices in development still use ad-hoc techniques, trying to encapsulate models, simulators and experimental frames into tightly coupled packages. As a result, testing, maintenance and software reuse become difficult tasks [2].

At present, there is a need to solve these problems, enabling interoperability (including digital and analog simulations), model reuse (using centralized or distributed repositories), while keeping high performance in the model execution. There are different efforts addressing these issues, for instance, the Base Object Model specifications, C4ISR, Extensible Modeling and Simulation Framework, Simulation Conceptual Modeling, etc. [3]. Other efforts consider the use of widely used standards like the UML, or simulation languages including support for execution on the RTI. Our proposal, instead, is based on the use of the DEVS formalism [4].

DEVS (Discrete Event systems Specifications) allows modular description of models that can be integrated using a hierarchical approach. DEVS has been proved to be a universal representation for all discrete event models and has been successfully used in previous efforts in model interoperability (see, for instance, [5, 6]) providing ease for reuse of simulation models. Another advantage of using DEVS is that different existing techniques (Bond Graphs, Cellular Automata, State Charts, Partial Differential Equations, Petri Nets, Queuing models, Timed Automata, etc.) have been mapped to DEVS. This permits sharing information at the level of the model, and different submodels can be specified using different techniques, while keeping independence at the level of the simulation engine. Existing DEVS tools have showed their ability to execute this wide variety of models with high performance in standalone or distributed environments.

DEVS has a theoretical foundation which makes it in principle independent of various programming languages and hardware platforms. There is a wide variety of groups working on extensions to the DEVS formalism, with several modeling tools based on these extensions. The goal of SISO DEVS Study Group [7] is to find a core of the DEVS formalism that is suitable for standardization.

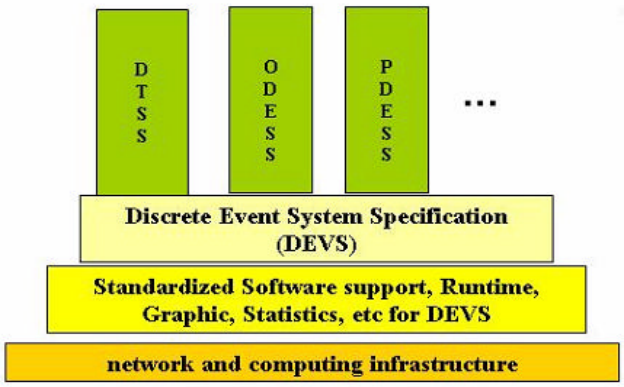


Figure 1.1 Standardization at the right level. DTSS: Discrete Time System Specification; ODESS: Ordinary Differential Equation System Specification; PDESS: Partial Differential equation System Specification

The primary objective of this effort is to support interoperability at the right level. In Figure 1, we can see that DEVS allows interchange of information at the level of models developed in different paradigms (Discrete-Time, Differential Equations, etc). The DEVS specification model is constructed on top of existing standard software support (like the HLA). This organization supports model composability and technology independence (for instance, we can replace the software support by a different type of middleware without modifying the models). This organization has proved to successfully include all modeling paradigms that are being widely used in academia, industry and government, enabling research collaboration between experts in these different areas.

Such a standard would also allow modelers to reason about the validity of model composition independently of the underlying simulation middleware technology. Similarly, simulation developers can integrate their DEVS simulation engines using a component based simulation standard that promotes the construction of verifiable, large-scale simulation systems. Finally, this standard would be a stepping stone toward realization of a standard for expression of DEVS models themselves [8].

In this work we will discuss the results of an experiment on the interoperation between two existing DEVS environments within the activities of the SISO DEVS SG (namely, CD++ and DEVS/C#). This work would provide the basis for future discussion on the standardization effort, by providing an actual experimental result on sharing of DEVS models developed by different teams using different DEVS simulation implementations. This permits discussing the basic issues involved in this effort. We show the basic APIs provided by both implementations, how to use them to provide interoperability at the level of the models, and how to integrate the underlying simulators. The models are split between the two simulators, and they execute in an on-line fashion, having both engines active and sharing execution results in real-time.

2. Background

In this section we will explore the theory behind the formalisms and implementations used in the interfacing of CD++ and DEVS C#.

A **model** is a set of rules, instructions, equations and behaviour reacting to input and generating output according to those rules, instructions and equations. Models can be simulated based on a number of different formalisms, but this case deals with the Discrete Event System Specification (DEVS) formalism [4]. Models transition between states based on inputs received or internal stimuli such as timer expiration. A model's behaviour is the set of all possible data generated by following its rules and instructions.

A **simulator** is a system with the capability to execute a model, thereby generating its behavior. Simulators differ in capability as follows:
(i) dedicated to a model or a small group of closely related models
(ii) able to simulate the behavior of models belonging to a related field (for example, plant growth models)
(iii) contain logic to execute models adhering to a single formalism (the DEVS formalism for instance)
(iv) have the ability to simulate models adhering to more than one formalism

DEVS is an increasingly accepted framework for understanding and supporting the activities of modeling and simulation. DEVS is a sound formal framework based on generic dynamic systems, including well defined coupling of components, hierarchical, modular construction, support for discrete event approximation of continuous systems and support for repository reuse. DEVS theory provides a rigorous methodology for representing models, and presents an abstract way of thinking about the world withcompletely indepedent of the simulation mechanisms, underlying hardware and middleware.

A real system modeled with DEVS is described as a composite of submodels, each of them being behavioral

(atomic) or structural (coupled). A DEVS atomic model is can be informally described as in Figure 1.

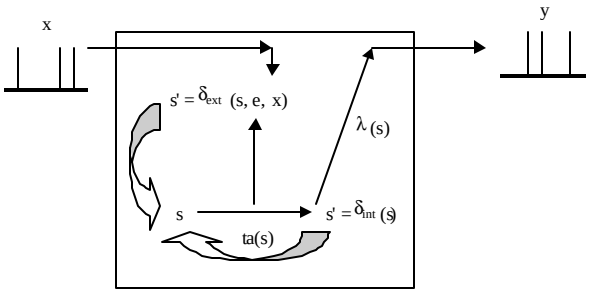


Figure 2.1. Informal description of an atomic model.

A DEVS atomic model has the structure:

$$M = X, S, Y, s_{int}, s_{ext}, l, ta$$

Where:

X is the set of inputs
S is the set of states
Y is the set of outputs
d_{int} is the internal transition function
d_{ext} is the external transition function
d_{con} is the confluent transition function
l is the output function
ta is the time advance function

The value of the time advance function can be any positive real number, zero or infinity. If ta is equal to zero, the system is changing states. If ta is infinity, the system is in a passive state and will not change states without external stimuli. The internal transition function is triggered by an elapsed wait time equal to that supplied by the ta function. The external transition function is triggered by input from an external source. When all models in a simulation are in a passive state, the simulation has ended. The confluent function allows the modeler to specify what happens when both an external input and an internal transition are about to occur.

A **coupled model** has a composite structure in that it can be made up of other models. Through these input and output ports, all interaction between models is mediated. Coupled models make it possible to model more complex systems. A coupled model consists of a set of input ports from which external events are received, a set of output ports through which the model can send outputs outside the system

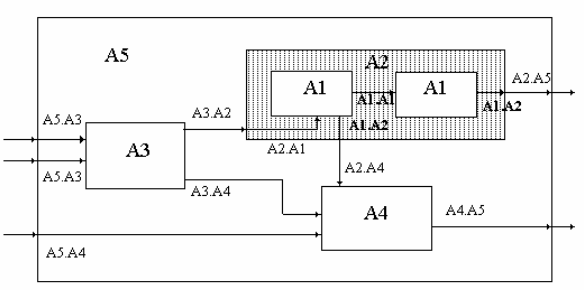


Figure 2.2 Informal description of a coupled model.

Coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the model's interfaces. The model's coupling defines how to convert the outputs of a component into inputs for other peers, and to convert inputs/outputs of the components to the exterior of the model [4].

CD++ [9] is a modeling environment that was defined using the specifications presented in the previous section, and the basic simulation techniques introduced in [4]. DEVS Atomic models can be programmed and incorporated onto a class hierarchy programmed in C++. Coupled models can be defined using a built-in specification language. CD++ makes use of the independence between modeling and simulation provided by DEVS, and different simulation engines have been defined for the platform: a stand-alone version, a Real-Time simulator, and a Parallel simulator. CD++ is built as a class hierarchy of models related with simulation processing entities.

```
class Atomic : public Model  {
public:
virtual ~Atomic();     // Destructor

protected:       //Kernel services
Time nextChange();
Time lastChange();
holdIn(AtomicState::State &, Time &);
passivate();
ModelState* getCurrentState() ;
sendOutput(Time &time, Port &port, Value value);

//User defined functions.
initFunction();
externalFunction(ExternalMessage & );
internalFunction(InternalMessage & );
outputFunction(CollectMessage & );
string className() const
};      // class Atomic
```

Figure2.3 The Atomic Class

DEVS Atomic models can be programmed and incorporated onto the *Model* basic class hierarchy using C++. A new atomic model is created as a new class that inherits from the *Atomic* base class. The state of a model is defined in the *AtomicState* class. When creating a new atomic model, a new class derived from *Atomic* has to be created. *Atomic* is an abstract class that declares a model's API and defines some service functions the user can use to write the model. The *Atomic* class provides a set of services and requires a set of functions to be redefined: **`nextChange()/lastChange()`** return the time until the next internal transition/since the last state change; **`holdIn(state, Time)`**: tells the simulator that the model remains in a *state* during a given *Time.* It corresponds to the ta(s) function of DEVS. The **`passivate()`** function sets the next internal transition time to infinity. The model will only be activated again if an external event is received, while **`getCurrentState()`**: returns the current model's phase. The **`sendOutput(Time, port, value)`** transmits an output message through the specified *port*.

Any newly defined class should override the following functions: **`initFunction()`** (invoked at the beginning the simulation), **`externalFunction (ExternalMessage &)`** (δ_{ext} function), **`internalFunction(InternalMessage &)`** (δ_{int} function) and **`outputFunction(const CollectMessage&)`** (λ function of the DEVS formalism).

Once an atomic model is defined, it can be combined with others into a multicomponent model using a specification language specially defined with this purpose. The coupled model at the higher level is always named `[top]`. Four properties must be configured: components, output ports, input ports and links between models. The following syntax is used:

`Components`: `name1[@atomicClass1] name2 ...` Lists the components of the coupled model (atomic or coupled). For atomic models, an instance and a class name must be specified, allowing a coupled model to use more than one instance of a given atomic class. For coupled models, only the model name must be given, and it must be defined as another group in the same file.

Out: portname1 portname2 ... Enumerates the model's output ports (optional clause).

In: portname1 portname2 ... Enumerates the input ports (optional clause).

`Link`: `source[@model] destination[@model]`. It describes the internal and external coupling scheme. If the name of the model is not included, the default will be the coupled model currently being defined.

DEVS C# is a DEVS engine created in the University of Arizona's ACIMS laboratory A revised version of the engine will be released later this year under the new title DEVS .NET. DEVS C# is a real time, parallel DEVS engine programmed in C# .NET. Models created in DEVS C# can be embedded in aspx web pages or exposed as web services due to .NET's service-oriented nature. DEVS C# is a self contained environment that allows users to easily model systems on a desktop computer.

In DEVS C#, models are written as C# classes that extend the Atomic class. The Atomic class, as discussed above contains a set of rules governed by the DEVS formalism's atomic level. A DEVS C# model consists of input and output ports, a constructor function, an initialization function, internal and external transition functions and an output function. The initialization function is invoked by the simulator at the beginning of the simulation and serves to put the model in its initial state and sets its initial context. The internal function is invoked when the time advance expires. The external function is invoked when a message is received at an input port. The output function outputs messages on one or more of the model's output ports. Similar to CD++, the DEVS C# Atomic class provides a set of services to all models extending the class. The `hold(time)` function tells the simulator that the model stays in its current state for the *time* specified. The `passivate()` function sets the model's next internal transition time to infinity. The `TimeNext` and `TimeLast` properties get/set the time of the next event and the time of the previous event respectively. The `TimeCurrent` property gets/sets the current time, The following is the code for a simple timer model:

```
public class SimpleTimer : Atomic{
   public SimpleTimer() { }
   // Initialize the SPTimer
      public override void init(){
         hold(3.3);              }
   // Internal transition function.
      public override void delta_int(){
         hold(3.3);                   }
   // External transition function.
      public override void delta_ext(double e,
         Bag<PortValue> x) {}
   // Output function
   public override void
   output_func(Bag<PortValue> y){
      Console.WriteLine("Alarm at " +
         TimeCurrent);
   }
}
```

Figure 2.4 The Simple Timer model

The above C# class extends the base type Atomic. For this simple case, the constructor is empty. For more complex models, member variables may be set and helper functions can be invoked. In the *init* function, the hold(time) function is called specifying when the first internal transition will occur. The external transition function is empty in this case, meaning that external events (messages) are ignored. In this model, state change only occurs due to timer expiration. The internal transition function calls the hold(time) helper function, setting the time until the next internal transition. The output function displays the time of the alarm, utilizing the Atomic classes TimeCurrent property to get the current time.

3. Interfacing CD++ and DEVS C#

As a DEVS simulation proceeds, models in the simulation change their state based on internal or external events [4]. External events are sent between models as messages. Messages sent by a model are based on events it has encountered, both internal and external. These messages trigger the external transition function of the model receiving them. The external transition function of the receiving model could result in more messages being passed to other models (although not directly [10]). Simply put, we must have message passing among models in order to simulate behavior in a system. A message includes the source model and port names, and the destination model and port names as well as the message body.

The simulation in CD++ is carried out by *Processors* that drive the simulation by exchanging messages. Two types of *Processors* exist:

1. *Simulators:* drive the simulation of atomic models, and
2. *Coordinators:* drive the execution of coupled components and coordinate the activities of all their dependant children.

A *simulator* object manages an associated *atomic* object, handling the execution of its δ_{int} (internal transition function), δ_{ext} (external transition function) and λ (output function). A *coordinator* object manages an associated coupled object.

The following figure shows a sample model with a few components:

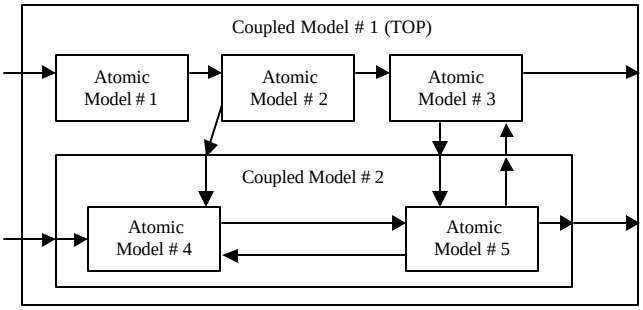


Figure 2.5 Sample model

The figure shows a sample model whose topmost component has three atomic submodels (*Atomic Models #1, #2* and *#3*) and one coupled model (*Coupled Model #2*). That inner-coupled component is formed by two atomic components (*Atomic Models #4* and *#5*). The corresponding *model hierarchy* for the depicted sample is shown below:

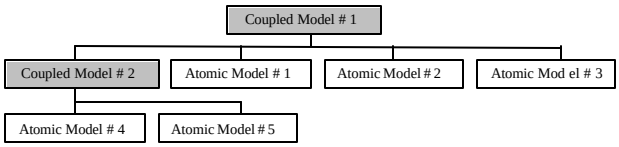


Figure 2.6 Hierarchical models' hierarchy

The processor hierarchy corresponding to this example is shown in the following figure.

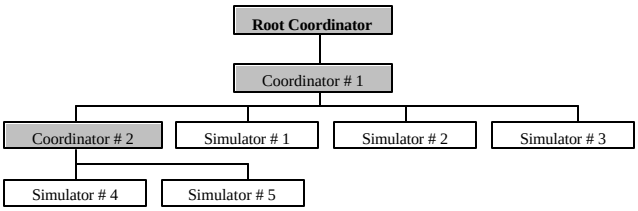


Figure 2.7. Processors' hierarchy (hierarchical approach)

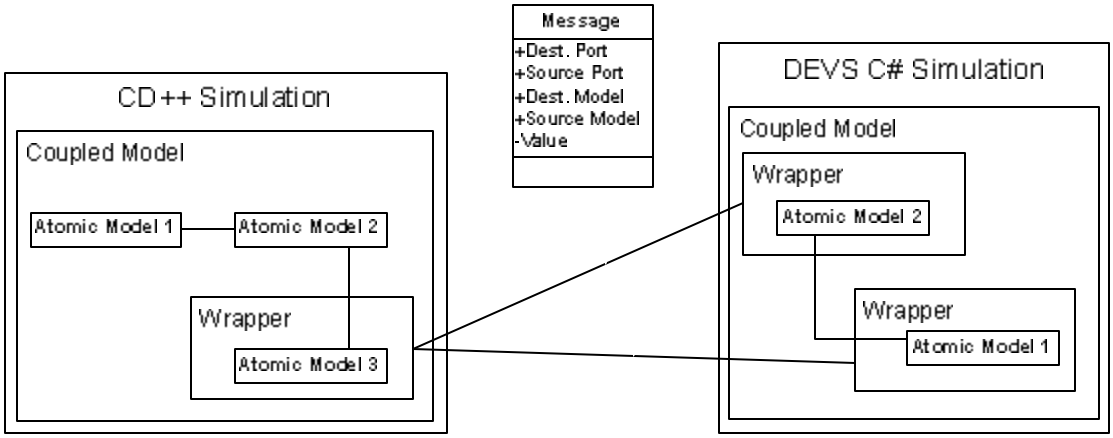
Only one *root coordinator* exists in a simulation. It manages global aspects of the simulation. It is involved with the topmost-coupled component, which has the highest level in the model hierarchy. Moreover, the *root coordinator* maintains the global time, and it starts and stops the simulation process. Lastly, it receives the output results that must be sent to the environment.

A split simulation is a source system whose components have been broken into two or more groups prior to execution. These groups of components (component groups hereafter) will run under separate simulators which may or may not be implemented using the same simulation engine (CD++ or DEVS C# specifically). There are different mechanisms that have been used for making these simulators interact. One way is to split the

execution of the coordinators/simulators between multiple processors within a single simulation engine. This is the approach taken by DEVS/HLA [5], DEVS/Corba [11] or Parallel CD++ [12]. Another alternative is to construct a model wrapper to make the two distributed simulators interact at a higher level of communication. This approach was used in [13], where the author created a wrapper for CD++ using the HLA as middleware, enabling multiple CD++ simulation engines to interact at the level of the top model. A wrapper is a software piece that hides the component and provides a means of communication with components modeled in the other environment.

Here, we extend this concept, permitting to compose a coupled model consisting of both CD++ and DEVS C# components and then execute that coupled model in distributed fashion. Some of the source system's components are modeled in CD++ and run in the CD++ simulation environment. The remaining components of the source system are modeled and simulated in DEVS C#. Any component that needs to be coupled to a component modeled in the other environment is encapsulated in a wrapper. These wrappers communicate with each other (in this case, via a TCP connection). Wrappers send messages between components that would have no means of communication otherwise. These wrappers make the interfacing of CD++ and DEVS C# possible. After receiving a message from another component's wrapper, the receiving wrapper must pass the message to its component so that the simulation can progress. CD++ wrappers and DEVS C# wrappers have different means of passing a received message to their encapsulated component. At this point, we have used TCP/IP sockets as the communication mechanism, as the focus of our approach is on the modeling aspects of the experience. This is the first successful effort in making two independent DEVS engines, developed by completely isolated teams of developers, to execute simultaneously and interchanging information in runtime. Mapping the results of this effort to other middleware (HLA, SOA, Corba, MPI) is straightforward: we need to adapt the calls made to the current API, and to incorporate calls to the corresponding middleware. As these advanced middleware provide advanced services, an extension would permit improving the simulation aspects of the experiments. The following examples serve as a proof of concepts of these ideas, focusing our experiments at the modeling level, and leaving advanced simulation aspects for future implementations.

The following diagram represents a simple split simulation comprised of two component groups



component groups are made with TCP connections over which messages are passed.

Both CD++ and DEVS C# send messages between simulations in a similar manner. In each atomic model's output function, in addition to sending the message to the simulator, the message is passed to the component's wrapper which will forward it to another wrapper. This creates an explicit, loose coupling between the component groups. The following code fragment from a generator component modeled in DEVS C# shows how this coupling is achieved:

```
// Output function
public override void output_func(Bag<PortValue>
   y){
   y.Add(new PortValue(portOut,
   m_count.ToString()));
   m_wrapper.send(portOut.Name, "transducer",
   "arriv",
      m_count.ToString());
}
```

Figure 3.2 DEVS C# output finction

First, the bag of port values (here called "y") passed by reference to the function is appended to include a message from the generator's "portOut" port containing the job number that has been generated. The DEVS C# simulator will use the coupling defined to deliver this message to all of its intended recipients. Next, we have a message being sent from the wrapped component to a component in another component group. This is done by invoking the wrapper's send function. When a component modeled in CD++ needs to send a message to another component group, a similar call is made from the component's output function to its wrapper. The `setWrapper(Wrapper)` function is invoked when a split simulation will be run. This function encapsulates the model in the *Wrapper*.

Received messages are handled differently in CD++ wrappers than they are in DEVS C# wrappers. In CD++ wrappers, all messages are routed from the wrapper directly to the atomic model. The wrapper calls the atomic model's external function, passing the received message as the argument. The following is a fragment of the CD++ wrapper's *receive* function, showing how received messages are handled:

```
m_model.externalFunction( receivedMessage );
```

The variable m_model is the component encapsulated by the wrapper. The variable receivedMessage is the message received from another wrapper.

In contrast, DEVS C# wrappers have a reference to their component's simulator. This means messages can be injected directly into the simulation by the wrapper. This results in the receipt of the message by its intended recipient models. The following is a fragment from the DEVS C# wrapper's listen function, which listens for and handles messages as they arrive:

```
PortValue pv = new PortValue(port, value);
m_wrappedSim.inject(pv);
```

The first line shows the creation of a PortValue, using the port and value received from the other wrapper. The second line shows the injection of the PortValue into the simulator, here named m_wrappedSim.

Prior to initiating the split simulation, each of the DEVS C# wrappers must know the IP address of the CD++ wrapper to which they will connect. Currently, only the default of localhost is used (this means all component groups are running on the same computer.) Upon execution of the DEVS C# simulation, each wrapper will try to connect to the CD++ wrapper specified. If a connection fails, the DEVS C# wrapper retries to connect until the user aborts the attempt or until a connection is made. When a connection can be made, two sockets are created with the CD++ wrapper. Two sockets are used so that there can be asynchronous communication between the two models (i.e. both models can be sending a message at the same time). Upon execution of a CD++ wrapper, two listening sockets are created and the wrapper waits for a connection from a DEVS C# wrapper. After a connection has been established and both sockets are ready for communication, the simulation is initiated and started. Messages are passed between the wrappers until the simulation is completed. At this time, the model where completion has been decided or detected sends out a termination message and all wrapper connections are closed.

4. AN EXAMPLE OF APPLICATION

The Generator, Processor, Transducer (GPT) model is a simple coupled model composed of three atomic models (components) each with a simple purpose. The generator component creates jobs and sends them out on the out port. The processor accepts jobs on an input port, processes them for a given time and then forwards them on an output port. The transducer accepts jobs from the

generator on the *arrived* input port and notes the generation time. Jobs processed by the processor are forwarded to the transducer's *solved* input port. The transducer notes the time the job was solved and calculates the elapsed time. This time is used later when calculating throughput. In this simple coupled model the internal coupling is as follows: the generator's output port is coupled to the processor's input port and the transducer's arrived port. The processor's output port is coupled to the transducer's solved port. The transducer's output port is coupled to the generator's stop port. The GPT model also has external coupling to receive events from other systems and provide events to other systems. The GPT model's start and stop ports are coupled to the generator to control job generation. The coupled model's out port is coupled to the processor's out port so that jobs are forwarded externally as well as to the transducer. The coupled model's result port is coupled to the transducer's out port so that the simulation's final result will be forwarded to other systems.

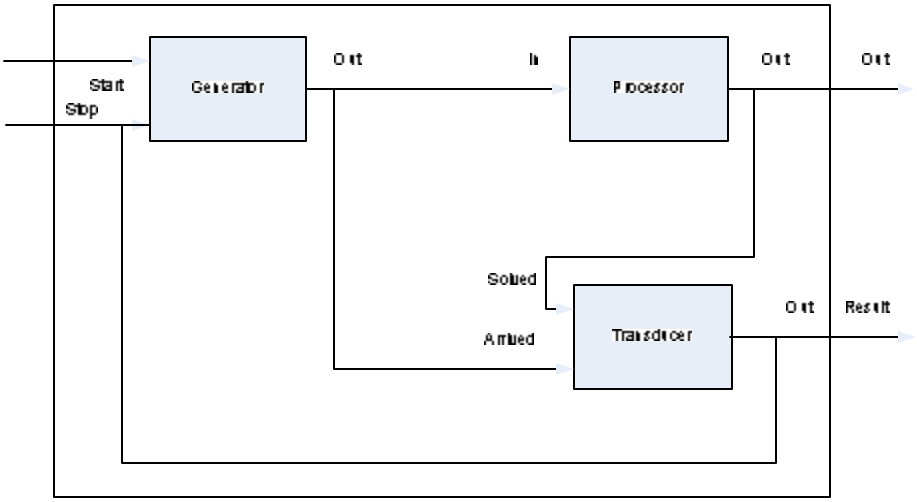


Figure 4.1 The GPT model and its internal and external connections

The following excerpts show the Generator/Processor and Transducer component groups as they defined in their respective DEVS environments. The Transducer component group is defined in DEVS C# as follows:

```
m_transducer = new Transd(observationTime,
   "transducer");
DevsThSim sim = new DevsThSim(m_transducer);
Wrapper wrap = new Wrapper(9998, 10000,
   m_transducer.Name, sim);
m_transducer.setWrapper(wrap);
```

The first lines deal with the creation of the simulator containing the transducer component. First a transducer is created, then a simulator is created to simulate the transducer's behavior. The final two lines show the integration between this component group and the Generator/Processor component group. First a wrapper is created by setting the TCP ports it will send and receive on, the source model's name and a reference to the simulator controlling the model. Then, a reference to the newly created model is set in the transducer, so that the wrapper's functions (send for instance) can be invoked by the transducer. The Generator/Processor component group is defined in CD++ as follows:

```
[top]
components : Generator@Generator Processor@CPU
Out: out

Link : out@Generator in@Processor
Link : out@Processor out

[Generator]
distribution : poisson
mean : 10

[Processor]
distribution : exponential
mean : 10
```

The top section defines the component's highest level, which contains a Generator, a CPU and an output port named *out*. This section also defines the couplings

between the two components in this group. The Generator and Processor sections of the component group's definition define the details of their respective components.

Now that we have seen how coupled models are defined in CD++ and DEVS C#, we can see why it is possible to construct a wrapper that has a reference to the simulator, allowing messages to be injected directly into it. In the DEVS C# definition, the simulator and wrapper are defined on separate lines and the simulator is passed to the wrapper's constructor so a reference can be made. In contrast, CD++ coupled models are defined using a high level script, parsed by helper classes. It is not possible to define the wrapper in this script, but rather the wrapper is created and initialized in the CD++ atomic model's constructor.

The following figure shows the GPT model as it has been created through the interfacing of DEVS C# and CD++ components:

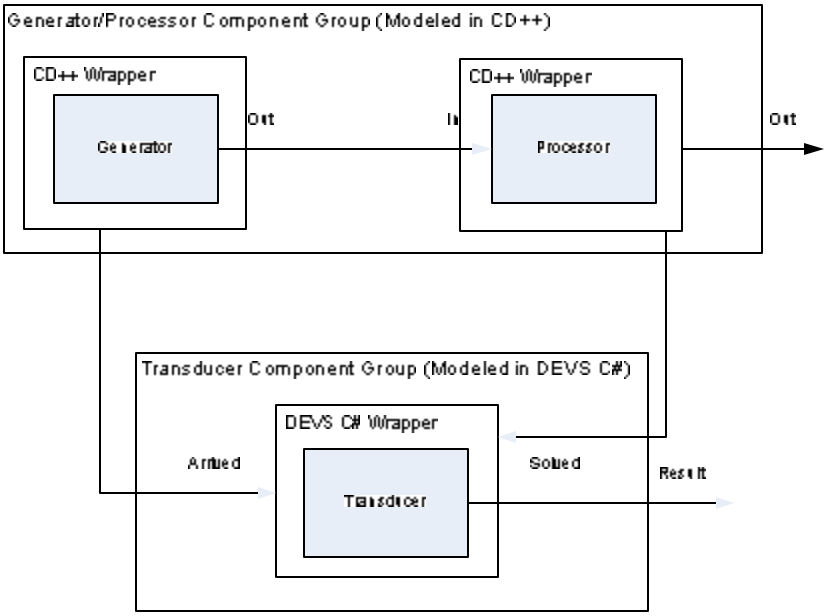


Figure 4.2 The split GPT model

The following sections will cover the steps involved in simulating the system using both CD++ and DEVS C#. Please note that for this experiment, the simulations are run in real time.

4.1 Initialization

The generator/processor application is started and the CD++ wrappers wait for a connection from another wrapper (in this particular case from the transducer's wrapper). The transducer application is started and sockets are opened to the generator and processor wrappers. After the socket initialization is complete, the simulation can begin.

4.2 Job Generation

The generator creates a job (in this case it is an integer starting at 0 and incrementing by 1 with each job created.) The job is sent from the generator to the processor via the CD++ coupling and to the transducer via the wrapper. Upon receiving a job, the transducer adds a timestamp which will be used for calculation when completed job messages arrive. Time-stamping a message at arrival rather than prior to sending it has advantages and disadvantages. The main disadvantage is that a lost or delayed message will have a timestamp that indicates the job was generated far later than it actually was. However, in order to attach a timestamp prior to sending the message, we must have synchronized simulation clocks. This means that both simulation engines must have timers that use similar units of time.

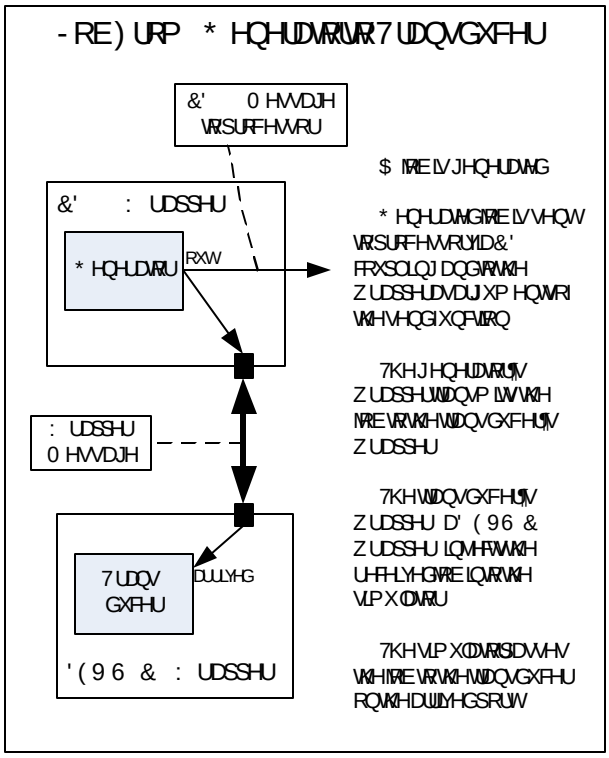


Figure 4.3 Path a job takes from the generator to the transducer through the wrappers.

4.3 Job Processing

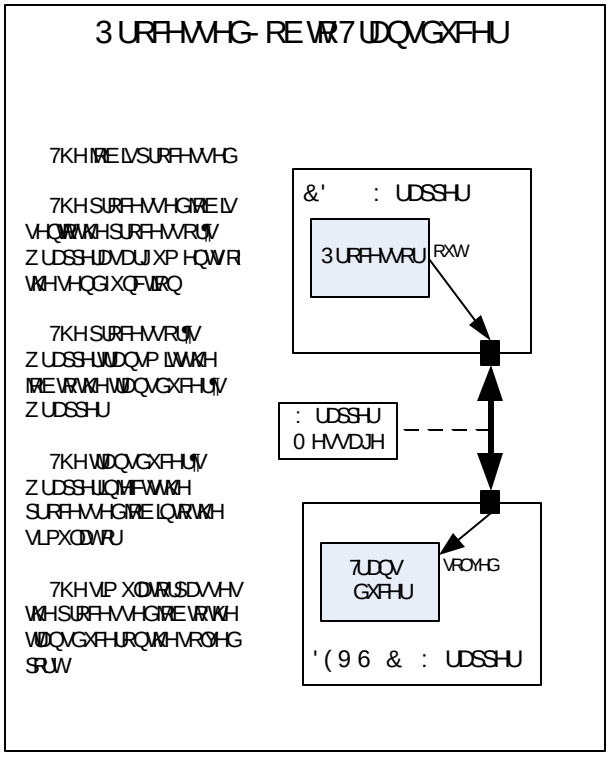


Figure 4.4. Path a processed job takes from the processor to the transducer.

Upon arrival at the processor, the job is "worked on" for a set amount of time. In reality, the processor sleeps for the amount of processing time set during initialization. Upon completion of processing, the processor forwards, via the wrapper the job message to the transducer. Upon arrival at the transducer an elapsed time for the job is calculated. This will aid in statistics calculation later.

4.4 Simulation Termination

Upon the expiration of its simulation timer, the transducer sends a message on its out port which is delivered to systems external to the simulation. A termination message is sent to the generator via the wrappers, and the DEVS C# simulation is terminated. The termination message is received by the generator's wrapper, which terminates the CD++ simulation.

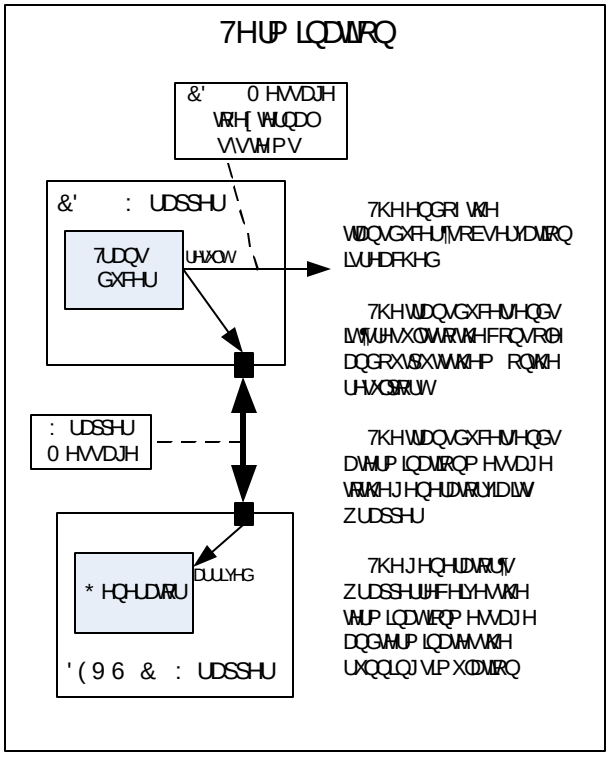


Figure 4.4. Steps in the termination of a split simulation

The GPT model was implemented in CD++ and DEVS/C# following the description presented in the previous section. The following figures show output from the two environments for a short period of simulation of the split GPT model.

```
I/00:000/Root(00) to top(01)
I/00:000/top(01) to generator(02)
I/00:000/top(01) to processor(03)
D/00:000/generator(02) / 00:00:00:000 to top(01)
D/00:000/processor(03) / ... to top(01)
D/00:000/top(01) / 00:00:00:000 to Root(00)
*/00:000/Root(00) to top(01)
*/00:000/top(01) to generator(02)
Y/00:000/generator(02) / out / 0 to top(01)
D/00:000/generator(02) / 00:00:10:000 to top(01)
X/00:000/top(01) / in / 0 to processor(03)
D/00:000/processor(03) / 00:00:10:000 to top(01)
D/00:000/top(01) / 00:00:10:000 to Root(00)
*/10:000/Root(00) to top(01)
*/10:000/top(01) to generator(02)
Y/10:000/generator(02) / out / 1 to top(01)
D/10:000/generator(02) / 00:00:10:000 to top(01)
X/10:000/top(01) / in / 1 to processor(03)
D/10:000/processor(03) / 00:00:10:000 to top(01)
D/10:000/top(01) / 00:00:10:000 to Root(00)
*/20:000/Root(00) to top(01)
*/20:000/top(01) to generator(02)
Y/20:000/generator(02) / out / 2 to top(01)
D/20:000/generator(02) / 00:00:10:000 to top(01)
X/20:000/top(01) / in / 2 to processor(03)
D/20:000/processor(03) / 00:00:10:000 to top(01)
D/20:000/top(01)/00:00:10:000 to Root(00)
```

Figure 4.4 CD++ Results of a short split simulation

The above figure shows the output generated by CD++ representing the message chatter of the Generator/Processor component group or the first 20 seconds of simulation. Please note that the time stamp has been truncated for space constraints and is usually formatted as follows: hh:mm:ss:ms. The first messages we see are of type I. These messages are initialization messages. First, the root coordinator sends an initialization message to the external model (top). The external model is responsible for distributing messages between the processor and generator and forwards the initialization message to them. The second set of messages are of type D. These are done messages in reply to the initialization messages. First the models reply to the external model, then the external model replies to the root coordinator. Along with the reply, the models send the time until their next event. The external model forwards the time until the first event to the root coordinator. In this case it is 0 seconds. The reply from the root coordinator is a * message. It is sent to imminent children (to the external model, then forwarded to the generator). The imminent children simulate and their simulators return Y and done messages. The root coordinator decides which output from a model needs to be distributed to other models and responds with x messages. This flow of events repeats until the simulation is terminated.

The following figure shows the DEVS C# output for the transducer component group over the same 20 second period. Information for each messages is formatted to take 2 lines. The first line shows the time of the event, the name of the model and the function triggered (internal, external or confluent). The second line shows the previous state, the port name and value on the port and the new state. For this example the states are blank since the transducer has only one non-passive state and it is unnamed. Following termination, the transducer displays its results, showing the end time of the simulation, the number of arrived and solved jobs, the total and average time advance and the processor's throughput.

```
0 transducer's ext:
         -- {portAriv:0} -->
10 transducer's ext:
         -- {portSolv:0} -->
10 transducer's ext:
         -- {portAriv:1} -->
20 transducer's ext:
         -- {portSolv:1} -->
20 transducer's ext:
         -- {portAriv:2} -->

End Time     : 20
Jobs Arrived : 3
Jobs Solved  : 2
Total TA     : 20
Average TA   : 10
Throughput   : 0.1
```

Figure 4.5 DEVS C# results for the split simulation

5. Conclusion

We have presented the results of an experiment on the interoperation between two existing DEVS environments (namely, CD++ and DEVS/C#), in an effort within the DEVS Standardization study group. The DEVS formalism defines a theory for discrete-events systems specification, which permits building formal models using a hierarchical and modular approach. DEVS formal nature showed to be useful for easy reuse of models that have been validated.

Although this interface between CD++ and DEVS C# is done through explicit coupling via a set of interconnected wrappers, it serves as a proof of concept. At present, we are working on the definition of a central coordinator to provide synchronization between simulations using SOA services, and adding managed coupling, rather than explicit wrapper coupling. By having the simulator sending the messages to a different peer rather than a specific wrapper, the model programming becomes easier for the user. This is one of the main strenghts of the approach: the DEVS simulation protocol is the same indepdendent of the way the models are expressed. The

DEVS coordinator (for CD++ or DEVS-C# will be used and interfaced to simulators for the two groups).

This work provides the basis for future discussion on the standardization effort, by providing an actual experimental result on sharing of DEVS models developed by different teams using different DEVS simulation engines, permitting discussing the basic issues involved in this effort. It provides a framework to conduct experiments, and to address the main issues of our standardization effort, namely, in which way a DEVS simulation engine can be standardized to provide simulation services to multiple modeling environments, which would later influence a higher level definition for a standard modeling mechanism to share modeling information at the level of the DEVS model.

REFERENCES

[1] IEEE standard for Modeling and Simulation (M&S) High Level Architecture (HLA) Framework and Rules. IEEE Std. 1516-2000, 2000.

[2] G. Wainer, B. Zeigler, H. Sarjoughian, J. Nutaro: "DEVS Standardization Study Group Terms of Reference", Simulation Interoperability Standards Organization, 2004.

[3] "SISO Product Development Activity." URL: http://www.sisostds.org/stdsdev/index.cfm. Last accessed: Nov. 1, 2005.

[4] B. Zeigler, H. Praehofer, T.G. Kim: "Theory of modeling and simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.", Academic Press, San Diego, CA, second edition, 2000.

[5] H. Sarjoughian, B. Zeigler: "DEVS and HLA: Complimentary Paradigms For M&S?", Transactions of the SCS Vol. 17, pp. 187-197, 2000.

[6] Y. Kim, T.G. Kim: "A Heterogeneous Simulation Framework Based on the DEVS BUS and the High Level Architecture", Proceedings of the Winter Simulation Conference, Washington, DC. 1998.

[7] SISO SIW DEVS Study Group. URL: http://www.sce.carleton.ca/faculty/wainer/standard/ Last Accessed: November 5, 2005

[8] J. Nutaro: "A middleware based standard for DEVS simulator interoperability", Proceedings of SISO SIW 2004. Arlington, VA. 04F-SIW-114. 2004.

[9] G. Wainer: "CD++: a toolkit to define discrete-event models ", Software, Practice and Experience, Vol. 32, pp. 1261-130, November 2002.

[10] H. Sarjoughian, B. Zeigler: "Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models", URL: http://acims.arizona.edu/SOFTWARE/devsjava_licensed/CBMSManuscript.zip. Last accessed: December 28, 2005.

[11] Kim, D., S. J. Buckley, and B. P. Zeigler.1999. Distributed supply chain simulation in a DEVS/CORBA execution environment. In *Proceedings of the 1999 Winter Simulation Conference.* Phoenix, AZ. 1999.

[12] Troccoli, A.; Wainer, G. "Implementing Parallel CD++". Proceedings of the Annual Simulation Symposium. Orlando, FL. 2003.

[13] C. Zhang. "Integrating existing DEVS simulations with the HLA". M.A.Sc. Thesis (Supervisor: T. Pearce). Carleton University. 2004.