The Architecture of GenDevs:  
Distributed Simulation in DEVSJAVA

Xiaolin Hu  
Dept of Computer Science  
Georgia State University

Updated  
Bernard P. Zeigler  
ACIMS  
January 2008

This document complements the PowerPoint Presentation, “The Architecture of GenDevs: looking under the hood of DEVSJAVA 3.0”, Bernard P. Zeigler ACIMS Updated January 2004
Contents

1 Introduction................................................................................................................. 3
  1.1 DEVS simulation -- two types of models and two types of simulation.............. 3
  1.2 Layered structure – brief introduction of each layer......................................... 4
  1.3 DEVS Interface and classes............................................................................... 5
2 GenDEVS Modeling Interface.................................................................................... 5
3 GenDEVS Simulator Interface ................................................................................... 8
4 How GenDEVS Simulator works.............................................................................. 10
  4.1 Simulate an Atomic model in fast mode........................................................... 10
  4.2 Simulate an Atomic model in real time mode.................................................. 12
  4.3 Simulate a coupled model in fast mode............................................................ 14
  4.4 Simulate a coupled model in real time mode.................................................... 17
    4.4.1 Simulate a Coupled model in real time mode in centralized way.............. 17
    4.4.2 Simulate a Coupled model in real time mode in decentralized way.......... 18
    4.4.3 DEVS Activity.......................................................................................... 21
5 Hierarchical Coupled model simulation ................................................................ 25
  5.1 Simulate a hierarchical coupled model in fast mode........................................ 26
  5.2 Simulate a hierarchical coupled model in real time mode............................... 29
    5.2.1 Centralized way....................................................................................... 30
    5.2.2 Decentralized way................................................................................... 32
6 Distributed DEVS Models and simulators............................................................... 33
1 Introduction

GenDEVS is a package of modeling and simulation classes in DEVSJAVA. It provides the capability to build discrete event models and to simulate these models. In contrast to many simulation tools, GenDEVS has a standard specification, a set of standard Interfaces and a clear and scalable structure. This specification employs the object oriented features of classes and interfaces provided by JAVA and is an implementation of the Discrete Event System Specification (DEVS) formalism. This document discusses distributed simulation of DEVS models, but in order to do, we first present the necessary groundwork. This document may be read together with the PowerPoint Presentation, “The Architecture of GenDevs: looking under the hood of DEVSJAVA 3.0”, Bernard P. Zeigler ACIMS Updated January 2004

1.1 DEVS simulation -- two types of models and two types of simulation

To understand how GenDEVS works, we need to know there are two classes of DEVS models (Atomic model and Coupled model) and two types of DEVS simulation modes (fast mode and real-time mode) in GenDEVS.

The two classes of models in GendDevs are Atomic model and Coupled model. An Atomic model is a basic component which has relatively simple functionality. It has input ports, output ports, states and internal or external transition functions. A coupled model is composed by multiple atomic models or coupled models. By coupling one model’s output port to another model’s input port, a message can flow from one model to another model. Usually, a coupled model has more complex functionality.

After building a DEVS model, we can simulate it in two simulation modes: fast mode and real-time mode. Corresponding to these two simulation modes, GenDEVS has fast mode simulators and real-time mode simulators. In fast mode simulation, the simulator interprets time as logic time so the simulation runs as fast as possible. However, in real-time simulation, time is interpreted as wall clock time, so the real-time simulator will schedule the event based on the real wall clock time.
From the above introduction, we can see that GenDEVS will have four simulation scenarios: simulating an atomic model in fast mode, simulating an atomic model in real-time mode, simulating a coupled model in fast mode and simulating a coupled model in real-time mode.

1.2 Layered structure – brief introduction of each layer

From implementation aspect of view, GenDevs uses a layered structure. We can see from figure 1, it has four layers.

![Layered structure](image)

From bottom up, the first layer is the implementation language layer. Depends on what kinds of implementation language GenDevs is using, this layer could be Java, C++ or other language (in figure 1, GenDevs is built on top of the Java language). This layer provides the basic computation and communication components such as thread, socket and so on for GenDevs to use. The second layer is the GenCol layer. This layer is the data structure layer to support DEVS modeling and simulation. It defines a set of data structures and classes such as Bag, Relation, Function and ensembleCollection and so on to be used by its up layers. The third layer is the GenDevs layer. This layer provides the DEVS modeling and simulation environment. It includes two parts. One is for DEVS modeling, the other is for DEVS simulation. With this layer, user can build DEVS models and pick up an appropriate simulator to simulate it. The top layer is the GenDevsTest layer. This layer includes user-defined models. It also includes a set of testing examples to show how to use GenDevs.
1.3 DEVS Interface and classes

As DEVS has been applied on various research and industry areas, many DEVS versions have been developed and implemented. They are implemented on different computer platforms with different program languages. They also vary in modeling level, distribution level and simulation mode such as real-time simulation and fast-mode simulation. To standardize all these implementation and facilitate future development, GenDEVS defines a set of DEVS interfaces and classes.

There are three sets of DEVS interfaces and classes: DEVS Supporting Interfaces and Classes, DEVS Modeling Interfaces and Classes, DEV Simulator Interfaces and Classes.

DEVS Supporting interfaces and Classes provide a supporting environment for DEVS implementation. It defines some basic data structures such as Content, Message, Port, etc. These classes are essential for DEVS modeling and simulation environment.

DEVS Modeling Interfaces and Classes provide the modeling environment to build DEVS models. It defines the basic functions a DEVS model needs to implement.

DEVS Simulator Interfaces and Classes provide the simulation environment in DEVS to run simulation. It defines the functions different simulators need to implement.

By carefully defining these DEVS interface and classes, we have a clear structure for the DEVS environment. This makes it possible to implement a subset of the whole system and define a just-as-needed DEVS environment based on different application scenarios. For example, if we are interested in the real time application such as real-time data management, we only need to implement the part corresponding to real time simulation interface. Similarly, if an application needs only atomic model such as some embedded programs which have sole functionality, only the part corresponding to Atomic interface is needed to be implemented.

2 GenDEVS Modeling Interface

Figure 2 shows the structure of DEVS modeling Interface and structure. In this figure, each circles represents a DEVS interface (a Java interface if using Java language). Each rectangle represents a class. The dashed arrow from a rectangle to a circles means that class implements the interface.
Figure 3 shows in detail each interface and the relationship between them. As we can see from the figure, each interface defines a subgroup of related functions. An interface can extend several parent interfaces. By extending several interfaces, a child interface will inherit the attributes and functions from its parents. It can also add its own functions and attributes so it will have its own personality.

Among these interfaces, basicDevs Interface defines the basic functions a DEVS model needs to implement such as deltext(), deltint(), out(), ta() and so on. IODevs interface defines the functions which are needed for a non-DEVS object to communicate with DEVS models. These functions including addInport(), addOutport(), makeContent() and messageOnPort(). A non-DEVS object can implement these functions so it can receive input from DEVS models and send output to DEVS models. IOBasicDevs interface extends IODevs interface and basicDevs interface. It provides a common basis for Atomic models and Coupled models. Combine IOBasicDevs with AtomicDevs, we get AtomicInterface which defines the functions an Atomic model need to implement. Combine IOBasicDevs with CoupledDevs, we get Coupled which defines the functions a coupled model need to implement.
CoupledDevs interface defines the functions which are only used in DEVS coupled models. AtomicDevs interface defines the functions only used in DEVS atomic models. From figure 2, we know that an atomic model implements the Atomic interface, and a coupled model implements the Coupled interface. These are the two basic models in DEVS and both of them are subclass of devs.

![Figure 3: DEVS Modeling Interface Detail](image-url)
3 GenDEVS Simulator Interface

Figure 4 shows the structure of DEVS simulator Interface and structure. Similar to figure 2, here each circles represents a simulator interface (a Java interface if using Java language). Each rectangle represents a class. The dashed arrow from a rectangle to a circles means that class implements the interface.

Figure 5 shows in detail each interface and the relationship between them. As we can see from the figure, each simulator interface defines a subgroup of related functions which is needed for that simulator. A simulator interface can extend several other simulator interfaces so it will inherit the attributes and functions of those simulators. It can also add its owns functions and attributes so it will have its own personality.

The basic interface for DEVS simulator is the CoreSimulator Interface. Under the CoreSimulator Interface, two classes of simulators have been defined. One is for fast-mode simulation, such as AtomicSimualtor interface, CoupledSimulator interface, CoupledCoordinator interface and Coordinator interface. Another is for real-time
simulation, such as atomicRTSimulator interface, CoupledRTSimulator interface, RTCoupledCoordinator interface, RTCentralCoord interface and RTCoordinator interface. Compared to fast mode simulation interfaces, RT add in Runnable Java interface and interpret time as real wall clock time. This means a real time simulator interface (except RTCoupledCoordinator interface) has its own thread and time, so it can control the execution of DEVS models in real time.

In these interfaces, coupledSimulator interface extends coresimulator interface and also couplingProtocolInterface, ActivityProtocolInterface and Hierparent interface. Among them couplingProtocolInterface defines the functions needed to handle message interaction between models; ActivityProtocolInterface defines the functions needed to handle DEVS activity; Hierparent defines the functions needed to handle hierarchical coupled model simulation.

![Figure 5: DEVS Simulator Interface Detail](image)

A coupledCoordinator Interface extends from coordinator interface and also from couplingProtocolInterface and Hierparent interface. This interface is useful when we simulate a hierarchical coupled model. By extends coordinator class, a coupledCoordinator class will have a similar functionality as a coordinator. Meanwhile,
by implementing couplingProtocolInterface and Hieparent interface, it can act as a coupledSimulator to simulate its coupled model.

4 How GenDEVS Simulator works

4.1 Simulate an Atomic model in fast mode

In GenDEVS, the simplest simulation scenario is to simulate an Atomic model in fast mode. To simulate an atomic model in fast mode, an AtomicSimulator is needed. Figure 6 shows how the AtomicSimulator works.

![Diagram of AtomicSimulator](image)

Figure 6: AtomicSimulator
In AtomicSimulator’s construction function, the atomic model is assigned to the AtomicSimulator. In the initialize() function, the AtomicSimulator initializes the atomic model and set tL = 0 and tN = myModel.ta(). Below is the pseudo-code of these two functions.

```java
public atomicSimulator(IOBasicDevs atomic){
    myModel = atomic;
}

public synchronized void initialize(){ // for non real time usage, assume the time begins at 0
    myModel.initialize();
    tL = 0.0;
    tN = myModel.ta();
}
```

After the initialization, as we can see from the figure, the AtomicSimulator has a while loop. In each loop (simulation cycle), it executes the nextTN() to get the atomic model's next tN, computeInputOutput() to ask the atomic model to generate output, Deltfunc() to execute the atomic model's transition function. Below is the code showing how it works.

```java
public void simulate(int numIter)
{
    int i=1;
    tN = nextTN();  // get tN
    while( (tN < DevsInterface.INFINITY) && (i<=numIter) ) {
        computeInputOutput(tN);
        DeltFunc(tN);
        tL = tN;
        tN = nextTN();
        i++;
    }
}
```

```java
public void computeInputOutput(double t){
    if(equalTN(t)) {
        output = myModel.Out();
    }
    else{
        output = new message(); // empty message
    }
}
```

```java
public synchronized void wrapDeltfunc(double t,MessageInterface x){
    if(x == null){
        System.out.println("ERROR RECEIVED NULL INPUT " + myModel.toString());
        return;
    }
    if (x.isEmpty() && !equalTN(t)) {
        return;
    }
```
else if(!x.isEmpty()) && equalTN(t)) {
    double e = t - tL;
    myModel.deltcon(e,x);
} else if(equalTN(t)) {
    myModel.deltint();
} else if(!x.isEmpty()) {
    double e = t - tL;
    myModel.deltext(e,x);
} 

From the code, we can see that the computeInputOutput() will check if model’s tN equals current time t. If yes, it will call atomic model’s out() function. In DeltFunc(), the simulator will check the input message x, the current time t and the atomic model’s tN and decide to execute atomic model’s deltcon() or deltint() or deltext() or nothing.

4.2 Simulate an Atomic model in real time mode

In AtomicSimulator, all the time is logic time. So if current time is t and next event time tN is t+Δt, the simulator will jump Δt directly to tN and schedule the next event. However, in real-time simulation mode, time is real wall clock time. For the above situation, the real time simulator will actually wait Δt. It won’t schedule the next event until the actually system time is tN.

AtomicRTSimulator is the simulator to simulate an Atomic model in real time mode. The figure below shows how it works. What is not show here is the initialize() function which are similar to the non real-time simulator: to initialize the atomic model.
As you can see from AtomicRTSimulator's construction function below, an AtomicRTSimulator is actually an independent thread. So it can get its own time and sleep some time to wait for the next event.

```java
class AtomicRTSimulator extends AtomicSimulator {
    super(atomic);
    myThread = new Thread(this);
    numIter = 0;

    public AtomicRTSimulator(IOBasicDevs atomic) {
        super(atomic);
        myThread = new Thread(this);
        numIter = 0;
    }

    public void run() {
        tL = timeInMillis();
        tN = tL + myModel.ta() * 1000;
        int iter = 0;
        while ((tN < DevsInterface.INFINITY) && (iter < numIter)) {
            while (timeInMillis() < getTN() - 10) {
                timeToSleep = (long)(getTN() - timeInMillis());
                if (timeToSleep >= 0) {
                    try {
                        myThread.sleep(timeToSleep);
                    } catch (Exception e) { continue; }
                }
                computeInputOutput(this.getTN());
                sendMessages();
                wrapDeltfunc(this.getTN());
                while (!pauseFlag) { }  // busy waiting
                tL = timeInMillis();
                tN = tL + myModel.ta() * 1000;
                iter ++;
            } // end of while
        }
    }
}
```

From the run() function of this thread, we can see that the basic loop (simulation cycle) of the AtomicRTSimulator is almost the same as AtomicSimulator except that in AtomicRTSimulator, the thread will actually sleep for some time if next tN is greater than current time. Note that here sendMessage() is actually an empty function because there is no message interaction if we only simulate one Atomic model.

In AtomicRTSimulator, we can also start an injectThread to inject an input to the atomic model at a specific time (refer to figure 7). When the injectThread starts, it will first set a pauseFlag to true. Then it will call AtomicRTSimulator’s wrapDeltfunc() and then set pauseFlag to false. In AtomicRTSimulator, each cycle it will check if the
pauseFlag is true or not. If pauseFlag is false, the AtomicRTSimulator will wait there until the pauseFlag becomes true. By this way, the pauseFlag acts as a semaphore to synchronize these two threads.

4.3 Simulate a coupled model in fast mode

In GenDevs, an atomic model is a basic component. By coupling several atomic models together, we can have a coupled model. So, a coupled model has several components. These components might be Atomic model or be Coupled model (hierarchical construction). In a coupled model, all the components are coupled together by linking one component’s output port to another component’s input port.

First let’s consider a relatively simple kind of Coupled model. In this kind of Coupled model, all components are atomic models. Because no component is coupled model, the coupled model is actually a “one-level” coupled model.

To simulate this kind of coupled model in fast model, we need a coordinator to control the whole simulation cycle. For each Atomic model, there is a CoupledSimulator to take care of it. The figure below shows the relationship between coordinator, coupledSimulator, the coupled model and its Atomic model components.

Figure 8: Simulating "one-level" coupled model in fast mode

As we can see from figure 8, for each atomic model, there is a CoupledSimulator corresponding to it. The coordinator is the main thread. It will tell all the
CoupledSimulators to initialize, send nextTN, computeInputOutput, sendMessages and wrapDeltfunc.

Note that if model Atomic1 want to send a message to Atomic2, CoupledSimulator1 will find the message destination (CoupledSimulator2) based on its ModToSim data and coupling information (described below). Then it will call CoupledSimulator2’s putMessages() and directly put the message to CoupledSimulator2. So the message doesn't go through the coordinator.

Below is the sequence diagram showing how the simulation works.

Figure 9: Sequence diagram of simulating "one-level" coupled model in fast mode

As you can see from the sequence diagram, the coordinator actually controls the whole simulation process. The pseudo-code of the coordinator looks like this:

```java
public void simulate(int num_iter)
{
    int i=1;
    tN = simulators.AskAll("nextTN");
    while( (tN < DevsInterface.INFINITY) && (i<=num_iter) ) {
        simulators.tellAll("computeInputOutput");
        simulators.tellAll("sendMessages");
        simulators.tellAll("DeltFunc");
        tN = simulators.AskAll("nextTN");
        i++;
    }
}
```
From figure 9 we can also see that in coordinator’s construction phase, for each atomic model, it will add a CoupledSimulator. Then it will call the setModToSim() and informCoupling() to download the modelToSim data and coupling information to all the CoupledSimulators. Here the modelToSim data stores the mapping information between each model and its corresponding simulator. The coupling information records the mapping information between one model’s output port and another model’s input port so we know where the message will flow to. By having these two data table (the coordinator download it to the simulators), a simulator can send its model’s output message directly to another simulator. Specifically, in CoupledSimulator’s sendMessages(), if there are output messages generated by its model, the CoupledSimulator will first find the message’s destination model based on the coupling information. Then it will find the corresponding simulator of that model based on the modelToSim data. Finally it will call that simulator’s putMessage() to put the message to that simulator. Below we show the sendMessages() and putMessages() functions.

```java
public void sendMessages() {
    if (!activityDue) {
        returnResultFromActivity(myActivity.computeResult());
        activityDue = false;
    }
    MessageInterface o = getOutput();
    if (o != null && !o.isEmpty()) {
        Relation r = convertMsg((Message) getOutput()); // assume computeInputOutput done first
        Iterator rit = r.iterator();
        while (rit.hasNext()) {
            Pair p = (Pair) rit.next();
            Object ds = p.getKey();
            if (modelToSim.get(ds) instanceof CoupledSimulatorInterface) {
                CoupledSimulatorInterface sim = (CoupledSimulatorInterface) modelToSim.get(ds);
                sim.putMessages(co);
            } else if (modelToSim.get(ds) instanceof CoupledCoordinatorInterface) {
                CoupledCoordinatorInterface sim = (CoupledCoordinatorInterface) modelToSim.get(ds);
                sim.putMessages(co);
            } else {
                // this is an internal output coupling
                CoupledCoordinatorInterface cci = getParent();
                CoordinatorInterface ci = getRootParent();
                if (cci != null) myParent.putMyMessages(co);
                else if (ci != null) myRootParent.putMyMessages(co);
            }
        }
    }
}
```
public void putMessages(ContentInterface c){
    input.add(c);
}

4.4 Simulate a coupled model in real time mode

Again, here we only consider the "one-level" coupled model which only includes atomic model components. For hierarchical coupled model, please refer to later sections.

There are two ways to simulate a coupled model in real time: centralized way and decentralized way. In the centralized way, the whole simulation cycles are centrally controlled. There is a RTCentralCoord which is the only thread in the system to centrally drive the simulation. However, in the decentralized way, each atomic model is driven by its own simulator (the coupledRTSimulator). There is no central control and each coupledRTSimulator has its own thread. The RTCoordinator is also a thread. But its only role is to decide when to start or stop the simulation.

4.4.1 Simulate a Coupled model in real time mode in centralized way

To simulate a “one-level” coupled model in real time mode in centralized way, we need a RTCentralCoord. And for each atomic model, there is a CoupledSimulator. Figure 10 shows the relationship between the RTCentralCoord, CoupledSimulator, the coupled model and its atomic model components. As we can see from the figure, the structure here is almost the same as that when simulating a coupled model in fast mode. What is different is that the RTCentralCoord has a sleep() function.
From Figure 10, we can see that the RTCentralCoord is the main thread. It controls all the simulators to initialize, to computeInputOutput, to sendMessage and to wrapDeltfunc. In each simulation cycle, it will check the nextTN and then sleep there until the current time reaches nextTN. By this way, it synchronizes to the real wall clock time.

Similar to fast mode simulation, even the whole simulation is controlled by the RTCentralCoord, the message passing between each CoupledSimulators doesn’t go through the RTCentralCoord. Because each CoupledSimulator has the modelToSim data and coupling information, it can put its output message directly to the destination simulator. This is the same as simulating a coupled model in fast mode.

### 4.4.2 Simulate a Coupled model in real time mode in decentralized way

To simulate a “one-level” coupled model in real time mode in decentralized way, we need a Rtcoordinator. And for each atomic model, there is a CoupledRTSimulator. Figure 11 shows the relationship between the RTCoordinator, CoupledRTSimulator, the coupled model and its atomic model components. As we can see from the figure, the structure is similar to that when simulating a coupled model in centralized way. However, There is a big difference between these two ways. Here the RTCoordinator no longer controls the simulation cycle. Instead, each CoupledRTSimulator is an independent thread and it has its own simulation cycle.
In figure 11, we have four threads here instead of only one main thread. Among these four threads, one thread is the RTCoordinator, which initializes all the simulators and then tell them to start the simulation or stop the simulation. Other three threads are CoupledRTSimulators. Once a CoupledRTSimulator is started, it doesn’t need to interact with the RTCoordinator anymore. It has its own thread and will keep running until the whole simulation stops. So here all the “real” work is actually done by the CoupledRTSimulator. It controls its own simulation cycle and also is responsible to communicate with other simulators and synchronize with the real wall clock time.

The figure below shows how a CoupledRTSimulator works.
In each cycle of CoupledRTSimulator's main thread, the simulator will compare the current time to the next TN. If current time is less than nextTN-10 (here 10 is an arbitrary time tolerance number) and nextTN is not INFINITY, the simulator will start a simTimer thread and then wait there. If next TN equals INFINITY, the simulator will also wait there (execute the wait() function).

There are two situations can make the simulator goes forward again. One situation is that the simulator gets an external event from other simulator. This means another model sends an input message to this model so this model need to execute its external transition function. The other situation is that the time has elapsed in the simTimer thread. This means the current time has reached the nextTN so the model need to execute its internal transition function.

When the simulator get a notify() from other simulators or from the simTimer thread, it will go forward. First it will check if the inputReady flag is true. If it is true, this means there are external event come in from other models. So the simulator need to respond to them right away. Otherwise (there is no external input), the simulator will recheck the current time and might restart another simTimer thread. This is because in non-real time OS, the simTimer thread might finish the sleep() faster than it should be.

So if there is external event or the nextTN is reached, the simulator will gets out of that loop. After that it will first sleep for a short period of time (time granule) to wait for other external inputs or internal time event which happened at almost the same time as the first one. Then it checks the Elapsed flag. If Elapsed is true, this means the next TN has been reached (there might also be external inputs). So the simulator call the computeInputOutput(), SendMessage() and WrapDeltfunc(). If Elapsed is false, this means the nextTN hasn’t been reached but there are external inputs. In this case, the simulator set the externalEventTime to currentTime and call the WrapDeltfunc().

After all this, the simulator will reset inputReady to false, Elapsed to false, and set tL and tN. Then it starts another simulation cycle.

Below is the actually code of the main CoupledRTSimulator thread.

```java
public void run(){
    setTN();
    int iter = 0;
    while( iter < numIter ) {
```
while(timeInMillis() < getTN() - 10){
    timeToSleep = (long)(getTN() - timeInMillis());
    if (timeToSleep < DevsInterface.INFINITY){
        new simTimer(this,timeToSleep);
        Elapsed = false;
    }
    waitforNextEvent(); // call the wait()
    if (inputReady) break;
} // out of the while loop because of getting an external input or time elapsed
try{Thread.sleep(100);} // time granule --- wait for other input
    catch (Exception e){}
if(Elapsed){ // time elapsed
    computeInputOutput(getTN());
    sendMessages();
    wrapDeltfunc(getTN());
}
else if(inputReady){ // get external input
    double externalEventTime = timeInMillis();
    if (externalEventTime > getTN()) externalEventTime = getTN();
    wrapDeltfunc(externalEventTime);
}
    inputReady = false;
    Elapsed = false;
    tL = timeInMillis();
    tN = tL + myModel.ta()*1000;
    iter ++;
}

In figure 12, besides the main thread, there are two other threads. One is the simTimer thread which is started by the main thread. This thread sleeps for a period of time and then set the Elapsed flag and call notify() to notify the simulator that the time has elapsed. Another kind of threads are other simulator threads or Activity threads. Whenever they send messages to the atomic model, these threads will call the coupledRTSimulator's putMessage(), set the inputReady and call notify() to notify the simulator that there are external inputs come in.

4.4.3 DEVS Activity

When simulating a coupled model in real time mode, another important concept is DEVS Activity. Basically, an Activity is a thread which has been wrapped into DEVS domain. Each Activity belongs to an atomic model. That atomic model can decide when to start or kill the Activity. An Activity may or may not return result to the atomic model. If an activity return result to the atomic model, the result will go through the CoupledRTSimulator and being put on a reserved input port (the "outputFromActivity"
port) as an external event. Note that in coupled model real time simulation, each atomic
model has a reserved input port "outputFromActivity".

Within the Activity thread, user can essentially do whatever he wants to and then
return the result to the DEVS atomic model. This is especially useful in real time systems.
In real time systems, it’s very common for the system to interact with the real world.
Typically, this kind of real world interaction is time unpredictable. The same task may
takes one millisecond in one case, several minutes in another case. This is why a DEVS
model needs to start a specific Activity thread to deal with this kind of unpredictable or
computation intensive tasks. As you can see from the below figure, a DEVS Activity acts
as a bridge between DEVS models and the real world.

Some typical Activities: Read sensor data, Interact with hardware, Read/write disk
files, Send data through network, act a socket proxy and so on.

Note that as DEVS Activity is an independent thread, it doesn’t work with simulating
a coupled model in real time mode in centralized way.

We’ll discuss, in a semi-informal manner, the concepts underlying the activity
implementation in DEVS. Let A denote a set of activities, where each activity has a
completion time and a result, properties that can be represented as mappings:

completionTime:A → R^+
result:A → V

where R^+ denotes the positive reals and V is a set of possible results. (To be more
realistic both representations should be non-deterministic to allow for the unpredictability
of execution.) Now consider a DEVS atomic model with state set, S and a mapping
which represents an assignment of activities to states. We allow the mapping to be partial so that not all states need be assigned an activity. For a state, s, that has an activity, f(s), there are two times, ta(s), the time advance of the DEVS, and the completion time, completion(f(s)) of the activity. We interpret the DEVS time advance as a time out for the activity – the activity must complete within that interval; if it doesn’t it will be terminated. We can represent this situation by defining a resulting time advance, tar(s) = min(ta(s), τ(f(s))). If the activity completes first, i.e., tar(s) = τ(f(s)) then the model’s external transition function δ_ext() receives the result(a) on the port “outputFromActivity”, i.e., an input of the form (“outputFromActivity”, result (a)). Otherwise, the internal transition function, δ_int() executes and kills the activity. (Since we are dealing with real time, we take the confluent case as unlikely to occur.)

**Example of Activity Operation**

Here’s an example containing a model genWActivity and a user-defined activity class, trialActivity.

*Atomic Model with Activity*

```
AtomicModel with Activity

genWActivity(String name, double interArrivalTime)…
```

```java
public void initialize() {
    currentActivity = new trialActivity();
    holdIn("active", interArrivalTime, currentActivity);
    //can initiate an activity using the holdIn primitive initially or in the transition functions of model
}
```

```java
public void deltext(double e, message x) {
    Continue(e);
    if (phaseIs("active") && somethingOnPort(x, "outputFromActivity")){
```

System.out.println(x.getValOnPort("outputFromActivity",0));
//output from activity comes in on port “outputFromActivity”
System.exit(3);
}

public void deltint() {
currentActivity.kill(); //kill the current activity before going further
}

Derived Activity Class

class trialActivity extends activity…

public trialActivity(double completionTime){ //completionTime is the life time of the activity thread
    super("trialActivity",completionTime);
}

public void run(){ // define run method of activity thread
    try {
        sleep((long) completionTime *1000);
    } catch (InterruptedException e) {return;}
    sim.returnResultFromActivity(new entity("result"));
}

coupledRTSimulator:

//simulator starts activity, collects its result, and puts it on port “outputFromActivity”
public void startActivity(ActivityInterface a){
a.setSimulator(this);
a.start();
}

public void returnResultFromActivity(EntityInterface result) {
    content c = new content("outputFromActivity",(entity)result);
    putMessages(c);
}

public synchronized void putMessages(ContentInterface c){
    if(c == null) return;
    System.out.flush();
input.add(c);
inputReady = true;
notify(); //interrupts the simulation cycle to force it to respond to the message is on its model’s input port
}

Executing the model:
As mentioned this can only be done within the scope of a real-time coordinator. For example,

```java
public static void main(String[] args){
    genWActivity gen = new genWActivity();
digraph dig = new digraph("dig ");
dig.add(gen);
    genDevs.simulation.realTime.RTcoordinator r =
        new genDevs.simulation.realTime.RTcoordinator(dum);
r.initialize();
r.simulate(2,20000.0);//run for 20 seconds
}
```

5 Hierarchical Coupled model simulation
In the last section, we described how to simulate a “one-level” coupled model in fast mode and real time mode. In this section, we will describe how to simulate a hierarchical coupled mode in fast mode and real time mode.

In a hierarchical coupled model, at lease one component of the model is a coupled model. As a coupled model has input ports and output ports, we can couple a coupled model with other models (coupled model or atomic model) and get a hierarchical coupled model. If the component of a hierarchical coupled model is still a hierarchical coupled model, we have a multiple-level hierarchical coupled model. The figure below shows a multiple-level hierarchical view of a model.
5.1 Simulate a hierarchical coupled model in fast mode

To simulate a hierarchical coupled model in fast mode, we need a coordinator to control and drive the whole simulation. For each Atomic model component, there is a CoupledSimulator corresponding to it. For each Coupled model component, there is a CoupledCoordinator corresponding to it. The figure below shows the relationship between coordinator, coupledCoordinator, coupledSimulator and the hierarchical coupled model (including its coupled model components and atomic model components).

As you can see from figure 15, similar to simulate a “one-level” coupled model, the coordinator is the main thread. It controls the whole simulation. The coupledSimulator works the same way as when simulating a “one-level” coupled model. For each coupled component, there is a CoupledCoordinator. The CoupledCoordinator combines the functionality of a CoupledSimulator and a coordinator. To its brothers (CoupledSimulator3 and CoupledSimulator4 in figure 15), it works as a
CoupledSimulator. They will send messages and call each other’s putMessage() function. To its children (CoupledSimulator1 and CoupledSimulator2 in figure 15), it works as a coordinator. So in figure 15, if CoupledCoordinator gets external input from CoupledSimulator3 or CoupledSimulator4, it will call its sendDownMessage() to send the message down to its children (CoupledSimulator1) based on the internalModelTosim data structure explained below. On the other hand, if Atomic2 generates output, CoupledSimulator2 will call CoupledCoordinator’s putMyMessage() to put the message to CoupledCoordinator’s output port. Then CoupledCoordinator will put the message to CoupledSimulator3’s input port.

We can easily see that simulating a “one-level” coupled model in fast mode is a special case of simulating a hierarchical coupled model in fast mode.

Figure 16 shows the sequence diagram of simulating a hierarchical coupled model in fast mode.

From figure 16, we can see that in the construction phase, similar to simulate the “one-level” coupled model, the coordinator will download the modelToSim data and coupling information to its sub level simulators (Either CoupledSimulator or CoupledCoordinator). Meanwhile, each CoupledCoordinator will also download its local coupled model’s modelToSim data and coupling information to its sub level simulators (Either CoupledSimulator or CoupledCoordinator). This will keep going until reach the bottom level. For each CoupledCoordinator, there is another data structure internalModelTosim. This data structure stores the mapping information between its child models and their corresponding simulators. This is useful when a CoupledCoordinator gets an input and need to pass the message to its children. (Please see sendDownMessages() code).
Figure 16: Sequence diagram of simulating a hierarchical coupled model in fast mode

Below we shows the code of CoupledCoordinator’s `sendMessage()`, `sendDownMessages()`, `putMessages()` and `putMyMessages()`.

```java
public void sendMessage() { //extend so they send message to its parent also
    MessageInterface o = getOutput();
    if( o != null && !o.isEmpty()) {
        Relation r = convertMsg((message) getOutput()); //assume computeInputOutput done first
        Iterator rit = r.iterator();
        while (rit.hasNext()) {
            AtomicModel2.2 coordinator coupledCoordinator2 coupledSimulator2.1 AtomicModel2.1 coupledSimulator1 AtomicModel1 coupledSimulator1 start construction addsimulator setmyModel addsimulator addsimulator setmyModel setModToSim informCoupling setModToSim informCoupling setModToSim informCoupling finish construction initialize initialize & get tN initialize initialize & get tN initialize initialize & get tN next tN next tN next tN next tN return smallest tN 
        }
    }
}
```

Pair p = (Pair)rit.next();
content co = (content)p.getValue();
Object ds = p.getKey();
if(modelToSim.get(ds) instanceof CoupledSimulatorInterface) {
    CoupledSimulatorInterface sim = (CoupledSimulatorInterface)modelToSim.get(ds);
    sim.putMessages(co);
} else if(modelToSim.get(ds) instanceof CoupledCoordinatorInterface) {
    CoupledCoordinatorInterface sim = (CoupledCoordinatorInterface)modelToSim.get(ds);
    sim.putMessages(co);
} else { // this is an internal output coupling
    CoupledCoordinatorInterface cci = getParent();
    CoordinatorInterface ci = getRootParent();
    if(cci != null) myParent.putMyMessages(co);
    else if(ci != null) myRootParent.putMyMessages(co);
}
}
}

public void sendDownMessages() {
if(!input.isEmpty()) {
    Relation r = convertInput(input);
    Iterator rit = r.iterator();
    while (rit.hasNext()) {
        Pair p = (Pair)rit.next();
        Object ds = p.getKey();
        content co = (content)p.getValue();
        if(internalModelTosim.get(ds) instanceof CoupledSimulatorInterface) {
            CoupledSimulatorInterface sim = (CoupledSimulatorInterface)internalModelTosim.get(ds);
            sim.putMessages(co);
        } else if(internalModelTosim.get(ds) instanceof CoupledCoordinatorInterface) {
            CoupledCoordinatorInterface sim = (CoupledCoordinatorInterface)internalModelTosim.get(ds);
            sim.putMessages(co);
        }
    }
}

public void putMessages(ContentInterface c) {
    input.add(c);
}

public void putMyMessages(ContentInterface c) {
    output.add(c);
}

5.2 Simulate a hierarchical coupled model in real time mode

Similar to simulate a “one-level” coupled model, three are two ways to simulate a hierarchical coupled model in real time mode: centralized way and decentralized way. In
the centralized way, the whole simulation cycles are centrally controlled. There is a RTCentralCoord which is the only thread in the system to centrally drive the simulation. However, in the decentralized way, each atomic model is driven by its own simulator (the coupledRTSimulator). There is no central control and each coupledRTSimulator has its own thread. The RTCordinator is also a thread. But its only role is to decide when to start or stop the simulation.

5.2.1 Centralized way

To simulating a hierarchical coupled model in real time mode in centralized way is very similar to simulating it in fast mode. The only difference is that we replace the coordinator with the RTCentralCoord. Similar to the coordinator, this RTCentralCoord is to control the whole simulation. For each atomic model component, there is CoupledSimulator. For each coupled model component, there is a CoupledCoordinator. Figure 17 shows the relationship between RTCentralCoord, coupledCoordinator, coupledSimulator and the hierarchical coupled model (including its coupled model components and atomic model components).

![Figure 17: Simulating hierarchical coupled model in real time mode in centralized way](image-url)
From figure 17 we can see that the RTCentralCoord is the main thread. It tells all the simulators to initialize, to computeInputOutput, to sendMessage and to wrapDeltfunc. In each simulation cycle, it will check the nextTN and then sleep there until the current time reaches nextTN. By this way, it synchronizes to the real wall clock time.

We can easily see that simulating a “one-level” coupled model in real time mode in centralized way is a special case of simulating a hierarchical coupled model in real time mode in centralized way.

The figure below shows the sequence diagram of simulating a hierarchical coupled model in real time mode in centralized way.

Figure 18: Sequence diagram of simulating a hierarchical coupled model in fast mode
5.2.2 Decentralized way

Simulating a hierarchical coupled model in real time mode in decentralized way is similar to simulating a “one-level” coupled model in real time mode in decentralized way. We need a RTCoordinator to start and stop the simulation. For each atomic model component, there is a CoupledRTSimulator corresponding to it. For each coupled model component, there is a RTCoupledCoordinator corresponding to it. The figure below shows the relationship between RTCoordinator, CoupledRTSimulator, RTCoupledCoordinator and the hierarchical coupled model (including its coupled model components and atomic model components).

![Diagram of simulating hierarchical coupled model in real time mode in decentralized way](image)

Figure 19: Simulating hierarchical coupled model in real time mode in decentralized way

From figure 19, we can see that the RTCoordinator no longer control each simulation cycle. Its role is just to initializes the system and to decide when to start and stop the simulation. Each Atomic model is driven by its own CoupledRTSimulator which is an independent thread (please refer to figure 12 to see how it works). For each coupled component, there is a RTCoupledCoordinator. Note that RTCoupledCoordinator is not a thread. It acts as a message relay.

As a relay, the RTCoupledCoordinator redirect its incoming messages to the appropriate destinations based on its ModToSim data and coupling information. The input messages might come from its brother simulator (by calling its putMessage()) or from its
child simulators (by calling its putMyMessage()). Below is the code of putMessage() and putMyMessage().

```java
public void putMessages(ContentInterface c) {
    input.add(c);
    sendDownMessages();
    input = new message();
}

public void putMyMessages(ContentInterface c) {
    output.add(c);
    sendMessages();
    output = new message();
}
```

So in figure 19, if Atomic2 sends a message to Atomic3, CoupledRTSimulator2 will call RTCoupledCoordinator’s putMyMessage() and put the message to RTCoupledCoordinator’s output port. Then message will be redirected right away by the RTCoupledCoordinator to CoupledRTSimulator3 (by executing sendMessages()). If Atomic4 needs to send a message to Atomic1, CoupledRTSimulator4 will call RTCoupledCoordinator’s putMessage() and put the message to RTCoupledCoordinator’s input port. Then message will be redirected right away by the RTCoupledCoordinator to CoupledRTSimulator1 (by executing sendDownMessages()).

We can see that simulating a “one-level” coupled model in real time mode in decentralized way is a special case of simulating a hierarchical coupled model in real time mode in decentralized way. Because every atomic model has its own CoupledRTSimulator thread, DEVS activity works here.

## 6 Distributed DEVS Models and simulators

In the previous sections, we have described how to simulate all kinds of DEVS models in fast mode and real time model. So far, all these simulations (including DEVS models and simulators) stay in one computer. In this section, we will discuss how to simulate DEVS models which have been distributed on several computers or even embedded chips (such as TINI chip).

When we say distributed DEVS models, we mean the components (Devs atomic model or Devs coupled model) of a DEVS coupled model have been distributed on several computers. However, the coupling (input-ports output-ports connection) between
these components keeps the same way, though they are actually happens across the network. For example, if we have a coupled model composed from Atomic models Atomic1, Atomic2 and Atomic3, we can distribute (put) Atomic1 on computer1, Atomic2 on computer2 and Atomic3 on computer3. The coupling between these atomic models keeps the same as when we put these models in one computer.

To simulate a distributed DEVS model, we need a set of DEVS simulators. Figure 20 shows how these distributed simulators relate to the non-distributed simulator. The simulators above the dash line are for non-distributed simulation. The simulators under the dash line are for distributed simulation. From the figure we can see that the coordServer is a child of RTcoordinator. The RTcoordServer is a child of the coordServer. SimulatorProxy and clientSimulator are children of coupledSimulator. RTsimulatorProxy is child of simulatorProxy. ClientHieSimualtor is child of CoupledCoordinator. To simulate distributed DEVS model in fast mode, we need coordServer, SimulatorProxy and clientSimulator (or ClientHieSimualtor). To simulate distributed DEVS model in real time mode, we need RTcoordServer, RTSimulatorProxy and clientSimulator (or ClientHieSimualtor).

![Figure 20: Simulators for distributed simulation](image)

To simulate a distributed DEVS model, for each computer which holds a DEVS model (atomic model or coupled model), we need a clientSimulator (for atomic model) or clientHieSimulator (for coupled model) on that computer. This clientSimulator is responsible to simulate/drive its local DEVS model as well as to communicate and synchronize with other simulators. Besides these clientSimulators, we also need a coordServer somewhere to control the whole simulation and to synchronize all the
clientSimulators. This coordServer can stay on a special computer or it will stay on the same computer as one of the clientSimulators. For each clientSimulator, the coordServer will create a simulatorProxy. This simulatorProxy is responsible to communicate with its corresponding clientSimulator.

As an example in fast mode simulation, the figure below shows the relationship between coordServer, SimulatorProxy, clientSimulator, ClientHieSimulator and the distributed coupled model.

![Figure 21: Simulating a distributed model](image)

In figure 21, the whole coupled model is composed by coupled model Coupled1, atomic model Atomic3 and Atomic4. Coupled1 is composed by atomic model Atomic1 and Atomic2. In this example, model Coupled1, Atomic3 and Atomic4 are distributed in three different computers. On each of these computers, there is a clientSimulator (clientHieSimulator for Coupled1). The coordServer stays on another computer. For each clientSimulator, the coordServer has a simulatorProxy corresponding to it.

From the figure, we can also see that the coordServer, each simulatorProxy and each clientSimulator (or clientHieSimulator) has its own thread.

In distributed DEVS model simulation, the top level coupling information is kept by the coordServer and downloaded to each simulatorProxy. Note that each clientSimulator
(or clientHieSimulator) doesn’t have the top level coupling information. But each clientHieSimulator knows its local coupling information. For example, clientHieSimulator1 knows its local model Coupled1’s coupling information and will download them to the CoupledSimulator1 and CoupledSimulator2. This means if Atomic1 need to send a message to Atomic2, then CoupledSimulator1 will put the message directly to CoupledSimulator2. However, if Coupled1 need to send a message to Atomic3, the clientHieSimulator1 will first send the message over network to simulatorProxy1. Then simulatorProxy1 will based on the coupling information put the message to simulatorProxy2. And then simulatorProxy2 will send the message over network to clientSimulator2.

Again the coordServer is the central controller. It controls the whole simulation cycle. Below is the psude code of the coordServer.

```java
public void simulate() {
    tL = 0;
broadcast("initialize:" + tL);
broadcast("nextTN"); //broadcast request for nextTN
    int i = 1;
    while ( (tN < INFINITY) && (i <= numIter) ) {
        broadcast("continue");
broadcast("computeInputOutput:" + tN);
broadcast("DeltFunc");
tL = tN;
broadcast("nextTN");
i++;
    }
broadcast("terminate");
}
```

To simulate a distributed DEVS model in real time mode is very similar to fast mode. The only difference is that we replace coordServer with RTcoordServer and replace simulatorProxy to RTsimulatorProxy. RTsimulatorProxy is almost the same as simulatorProxy. RTcoordServer is also very similar to coordServer except that it adds the sleep function to synchronize to the real wall clock time. So this real time simulation is actually a centralized way because there is only one time which is kept by the RTcoordServer. Further work need to be done to make it decentralized (including the timer and the message passing).
Below is a sequence diagram showing how the distributed DEVS model real time simulation works.

Figure 22: simulating distributed Devs model in real time mode
From the sequence diagram, we can see that in the construction phase, there are a lot of interactions between simulatorProxys and clientSimulators to set up the network connection.

First, when the coordServer starts, it will create a ServerSocket and then wait for client’s connection. Meanwhile, each clientSimulator will first try to connect to the server once it gets started. Whenever the coordServer gets a connection from a clientSimulator, it will create a simulatorProxy for it. This simulatorProxy will be fully responsible to communicate to the corresponding clientSimulator later. After the coordServer gets all the connection, it will set up the modelToSim data and download the coupling information to each simulatorProxy. Below is the psude code of the coordServer to describe this sequence.

```java
public void run() {
    // Establish ServerSocket for listening
    ss = new ServerSocket(iServerPort); // Create server socket
    // Loop for listening and processing client calls
    setRegisterCount(myCoupled.getComponents().size());
    while(numConnected < myCoupled.getComponents().size()){
        s = ss.accept(); // Listen for client call.
        numConnected++;
        simulatorProxy proxy = new simulatorProxy(s,this);
    } // end of while( true )
    waitForAllSimRegistered();
    setSimulators();
    informCoupling();
    simulate(); // finish from construction and start simulation
}
```