DEVS Simulation Protocol

9

One of the hallmarks of DEVS modeling and simulation is its fundamental separation of models from the simulation engines that execute them. The alternative, which is more common in today's practice, is not to enforce such a clear separation and to indiscriminately mix constructs that relate to the model with those that relate to how it is being executed.

The separation between model and simulator leads to a layered architecture of services as illustrated in Fig. 9.1. Modeling services enable a modeler to specify a DEVS model, which is a description of a dynamic system. The simulation layer provides the ability to execute a model to get the results of simulation.

A DEVS modeler can write a DEVS model in any DEVS environment, say MS4 Me, and expect that it will be correctly simulated by a DEVS Simulator provided by that environment. Furthermore, in principle, the modeler can provide the model, as expressed in the DEVS formalism, to a friend who implements it in another environment, say ADEVS (Nutaro 2010). Now, if both environments implement the DEVS Abstract Simulator correctly, the friends are entitled to expect that the simulation results will be the same.

At this point, you have already become familiar with the Abstract DEVS Simulator, in the sense that you have worked with the methods in MS4 Me Java. As discussed in Chap. 4, these methods are in one-to-one correspondence with the DEVS characteristic functions of time advance, internal transition, external transition, confluent function, output function, and the associated sets of states, inputs, and outputs. To help grasp the concepts behind the Abstract DEVS Simulator, let's

B.P. Zeigler and H.S. Sarjoughian, *Guide to Modeling and Simulation of Systems of Systems*, Simulation Foundations, Methods and Applications,
DOI 10.1007/978-3-319-64134-8_9

consider an analogy with a calculation by a handheld calculator, or an equivalent application on your cell phone. The calculator program realizes an algorithm that specifies how it is to add, subtract, multiply, and divide—operations that are defined rigorously by arithmetic, the mathematical theory of numerical manipulations. There are many hand calculators in existence, and they are all assumed, indeed required, to correctly implement the mathematically specified operations. In the context of computing, these are abstractly specified because they don't refer to any program or computer. Nevertheless, because you were taught arithmetic, you know what to expect when you enter 2 + 2 into the calculator and would be disdainful of a device that gave 3 as an answer. In the same way that you enter an expression like 2 + 2 to a calculator and always expect 4 as the answer, you can provide a DEVS model to a DEVS simulation engine—and you should expect to get the same output no matter which engine you choose.

9.1 DEVS Simulation Protocol

As illustrated in Fig. 9.1, the tie that binds DEVS modeling and DEVS simulation services is the *DEVS Simulation Protocol* which is an extension of the DEVS Abstract Simulator in the context of networked environments. Figure 9.2 shows that the DEVS Protocol involves three types of objects, a coordinator, one or more simulators, each with an associated model (for simplicity, only one simulator and its model are shown). The coordinator has a coupled model associated with it. The DEVS Simulation Protocol specifies (1) *the interface that the model must present to the simulator* and (2) *the interface that the simulator must present to the coordinator to execute a valid DEVS simulation*.

The interface presented by the model to the simulator is determined by the Abstract DEVS Simulator.

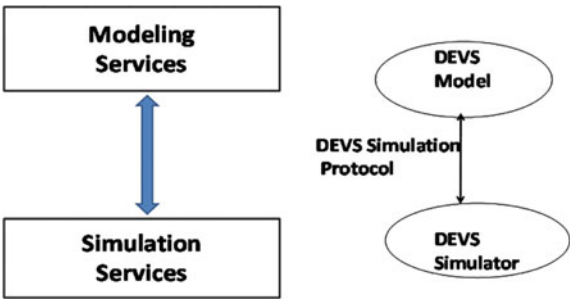

Fig. 9.1 DEVS modeling and simulation layers

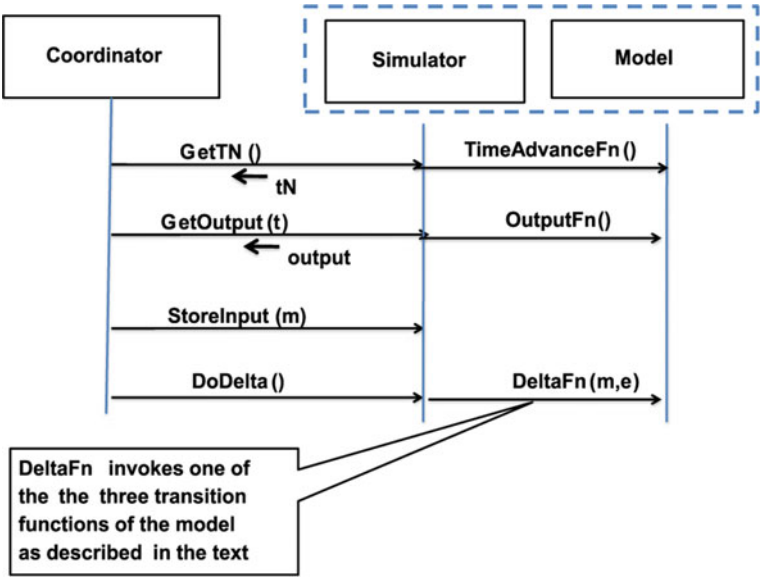

Fig. 9.2 DEVS Simulation Protocol

```
interface AbstractSimulator {
   public double TimeAdvanceFn();
   public message OutputFn();
   public void ExternalTransitionFn(message m,double
elapsedTime);
   public void InternalTransitionFn();
   public void ConfluentTransitionFn(message m,double
elapsedTime);
}
```

During an iterative cycle to be described, a simulator has to respond to operations requested by a coordinator and specified in the following interface:

```
interface DevsProtocol {
   public double OperationGetTN();
   public message OperationGetOutput(double t);
   public void OperationStoreInput(message m);
   public void OperationDoDelta();
}
```

The iterative cycle is a repetition (until some condition dictates termination) of the following steps in which the coordinator issues the operation requests in the protocol and the simulator responds by interacting with its model using the Abstract DEVS Simulator:

Step (1) OperationGetTN() requests the time of the simulator's next event—the simulator invokes its model's time advance function and adds the result to the time of last event to answer the coordinator's request.

Step (2) OperationGetOutput(t) provides the current simulation time to the simulator and requests its output for that time, if any, in the form of a DEVS message—the simulator determines if the model is imminent (its time of next event equals the current time) and if so invokes the model's output function to get its output message.

Step (3) OperationStoreInput(m) provides the input message, m, to the simulator, where m is a DEVS message which is a composite of messages sent to this simulator by other simulators. The coordinator gathered these messages in Step 2 and applied the coupling specification to determine which ones to package in this composite message.

Step (4) OperationDoDelta() tells the simulator to cause its model's state transition—since the simulator knows the current time from step 2, as well as any input that it has received from step 3, it can determine whether the model is to undergo an internal, external, or confluent (both external and internal) transition. This is shown as DeltaFn(m, e) in Fig. 9.2, where m is the input message and e is the elapsed time which is the difference between the current time and the model's time of last event.

It is important to note that there are many ways in which this basic protocol can be implemented. Particularly, we do not require the strict adherence to the sequential control and message exchanges that the above rendering may appear to require—so long as the resulting behavior is what the protocol specifies. We discuss some important cases later.

9.2 MS4 Me Exposition of the DEVS Simulation Protocol

The DEVS formalism has an associated well-defined concept of simulation engine to execute models and generate their behavior. A coupled model in DEVS consists of component models and a coupling specification that tells how outputs of components are routed as inputs to other components.

The basic simulation paradigm is illustrated in Fig. 9.3. It consists of a coordinator that has access to the coupled model specification as well as simulators for each of the model components, only one of which is shown for illustration.

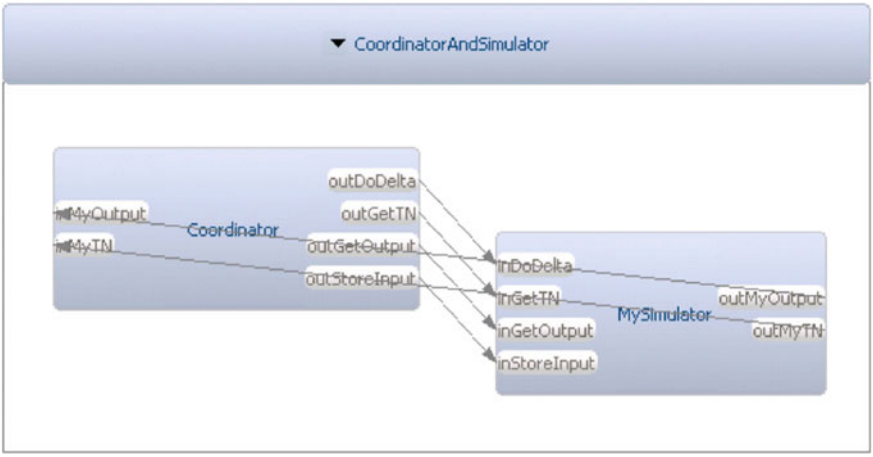

Fig. 9.3 MS4 Me simulation view of coordinator and simulator

We will use MS4 Me itself to explain the DEVS Simulation Protocol and the how the protocol works to correctly simulate DEVS models. The SES that gives rise to Fig. 9.3 is:

```
From the protocol perspective, CoordinatorAndSimulator is
made of Simulator and Coordinator!
From the protocol perspective, Coordinator sends GetTN to
Simulator!
From the protocol perspective, Coordinator sends GetOutput
to Simulator!
From the protocol perspective, Coordinator sends StoreInput
to Simulator!
From the protocol perspective, Coordinator sends DoDelta to
Simulator!
From the protocol perspective, Simulator sends MyTN to
Coordinator!
From the protocol perspective, Simulator sends MyOutput to
Coordinator!
```

Note that the interface between Coordinator and Simulator follows that given by the DEVSProtocol interface above. The coordinator performs time management and controls the message exchange among simulators in accordance with the coupled model specification. The simulators respond to commands and queries from the coordinator by referencing the specifications of their assigned models. The simulation protocol works for any model expressed in the DEVS formalism. It is an algorithm that has different realizations that allow models to be executed on a single host and on networked computers where the coordinator and component simulators are distributed among hosts.

In the following, we represent the Abstract DEVS Simulator within MS4 Me with its interfaces to both the Coordinator and its model. Note that the classes that appear are those employed in MS4 Me and may differ from the names employed in the general discussion above, e.g., MS4 Me employs MessageBag to represent the general concept of DEVS message.

9.2.1 Interface Objects

The objects exchanged between coordinator and simulators carry the relevant event times and the DEVS messages to be exchanged among them. They are defined as follows:

```
A DoubleEnt has value!
The range of DoubleEnt's value is double!

A NamedMessage has myName and myMessage!
The range of NamedMessage's myName is String!
The range of NamedMessage's myMessage is MessageBag!
```

9.2.2 Input and Output Ports

These objects are placed on the input and output ports by the operations invoked by the DEVS Simulation Protocol. In other words, the ports and types are defined to correspond to the operations in the interface of the DEVS Protocol.

For the Simulator, the port definitions are:

```
accepts input on GetTN!
accepts input on GetOutput with type DoubleEnt!
accepts input on StoreInput with type MessageBag!
accepts input on DoDelta!
generates output on MyTN with type DoubleEnt!
generates output on MyOutput with type NamedMessage!
```

These input and output ports are shown in Fig. 9.4.

For the Coordinator, the port definitions are:

```
accepts input on MyTN with type DoubleEnt!
accepts input on MyOutput with type NamedMessage!
generates output on GetTN!
generates output on GetOutput with type DoubleEnt!
generates output on StoreInput with type MessageBag!
generates output on DoDelta!
```

These input and output ports are shown in Fig. 9.5.

Note that the input and output ports of the simulator and coordinator match each other so that they can exchange data in a manner consistent with Fig. 9.3.

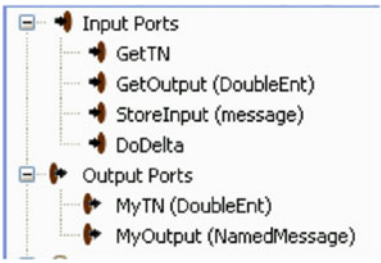

Fig. 9.4 Input and output ports of simulator

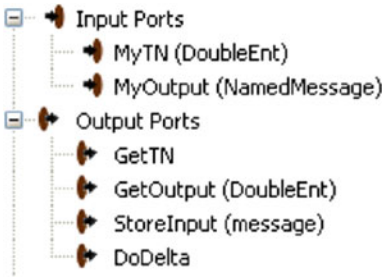

Fig. 9.5 Input and output ports of coordinator

9.2.3 FDDEVS Specifications

The interaction between coordinator and simulator that carries out the iterative cycle described earlier can be outlined in the FDDEVS natural language in the following texts:

For the Coordinator, the FDDEVS specification is:

```
to start hold in sendGetTN for time 0!
after sendGetTN output GetTN!
output event for sendGetTN

from sendGetTN go to waitForAllTN!

passivate in waitForAllTN!

when in waitForAllTN and receive MyTN go to sendGetOutput!

hold in sendGetOutput for time 0!
after sendGetOutput output GetOutput!

from sendGetOutput go to waitForAllOutput!

passivate in waitForAllOutput!
when in waitForAllOutput and receive MyOutput go to
sendStoreInput!

hold in sendStoreInput for time 1!
after sendStoreInput output StoreInput!

from sendStoreInput go to sendDoDelta!

hold in sendDoDelta for time 1!
after sendDoDelta output DoDelta!
from sendDoDelta go to sendGetTN!
```

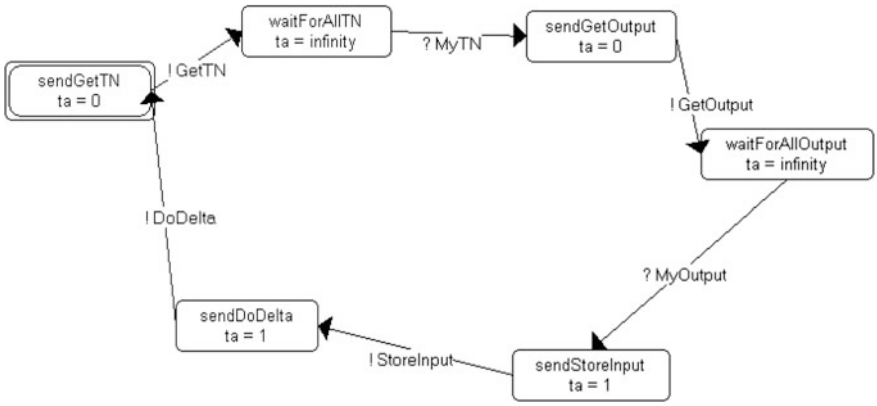

Fig. 9.6 Coordinator's state diagram

The state diagram generated from this text is shown in Fig. 9.6.

For the Simulator, the FDDEVS specification is:

```
to start passivate in waitForGetTN!
when in waitForGetTN and receive
GetTN go to sendMyTN!

hold in sendMyTN for time 0!
after sendMyTN output MyTN!
from sendMyTN go to waitForGetOutput!

passivate in waitForGetOutput!
when in waitForGetOutput and receive GetOutput go to
sendMyOutput!

hold in sendMyOutput for time 0!
after sendMyOutput output MyOutput!
from sendMyOutput go to waitForStoreInput!

passivate in waitForStoreInput!
when in waitForStoreInput and receive StoreInput go to
waitForMyDoDelta!

passivate in waitForMyDoDelta!
when in waitForMyDoDelta and receive DoDelta go to
waitForGetTN!
```

The state diagram generated from this text is shown in Fig. 9.7.

To create the simulation models, these specifications are filled in with tagged blocks as illustrated in Appendices A and B.

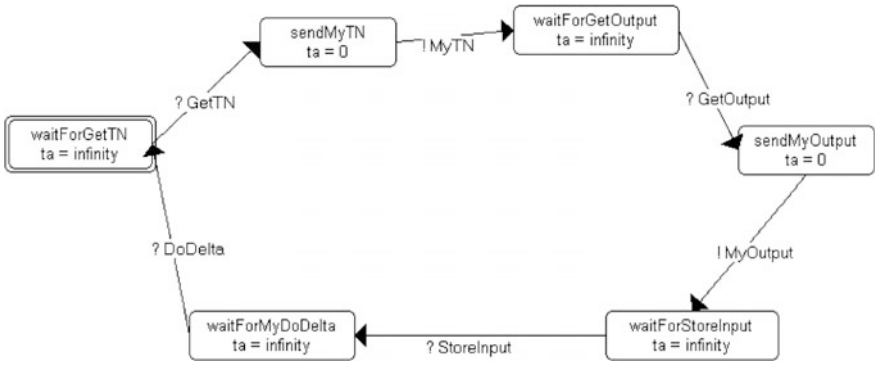

Fig. 9.7 Simulator's state diagram

Exercise

Use the sequence design interface tool to develop a sequence diagram that describes the DEVS Simulation Protocol and generates SES and FDDEVS descriptions similar to those shown above.

9.3 Distributed Simulation Implementations of the DEVS Protocol

The DEVS Simulation Protocol is an abstract specification of how a distributed simulation should proceed to correctly generate the behavior of a DEVS coupled model. As emphasized before, this means that there can be many different implementations of the same specification. In the following discussion, we discuss three such implementations and illustrate them with formulations using MS4 Me. As before, in each implementation, the components of the coupled model are assigned to simulators in one-to-one fashion and the coupled model is assigned to the coordinator. However, the implementations differ in the degree to which the coordinator is involved in the routing of messages and management of time.

The implementations considered are:

1. **Standard DEVS Protocol**—the basic formulation in which the coordinator uses the coupling information supplied by its coupled model to distribute messages to the simulators.
2. **Peer Message Exchanging Implementation**—modifies the basic formulation by partitioning the coupled model coupling information according to its components and distributing these segments to the respective simulators. This allows the simulators to each exchange DEVS messages without intervention of the coordinator. There is an extensive literature on parallel and distributed simulation that extends this basic implementation (see, e.g., Zeigler et al. 2000 and Nutaro 2010.)

3. **Real-Time Message Exchanging Implementation**—takes the peer message exchanging implementation one step further by letting the simulators decide on when to execute their next events. This can work when the simulation proceeds in real time. This obviates the further coordination that is required when executed in logical time (see Gholami and Sarjoughian 2012 for an in-depth discussion of DEVS real-time simulation).

9.3.1 Standard DEVS Protocol

As expected, the coordinator and simulator definitions in Sect. 9.2 are employed in the standard formulation. Furthermore, we capture the centrality of the coordinator in controlling the simulators by plugging the coordinator and simulator into a larger SES using a suitable multi-aspect as follows:

```
From the protocol perspective, DEVSDistributedSim is made
of Coordinator and Simulators!
From the multiSim perspective, Simulators are made of more
than one Simulator!
Simulator can be id in index!
From the protocol perspective, Coordinator sends GetTN to
all Simulator!
From the protocol perspective, Coordinator sends GetOutput
to all Simulator!
From the protocol perspective, Coordinator sends StoreInput
to all Simulator!
From the protocol perspective, Coordinator sends DoDelta to
all Simulator!

From the protocol perspective, all Simulator sends MyTN to
Coordinator!
From the protocol perspective, all Simulator sends MyOutput
to Coordinator!
```

Note the use of all-to-one and one-to-all coupling specification as discussed in Chap. 6. Such coupling is illustrated in Fig. 9.8.

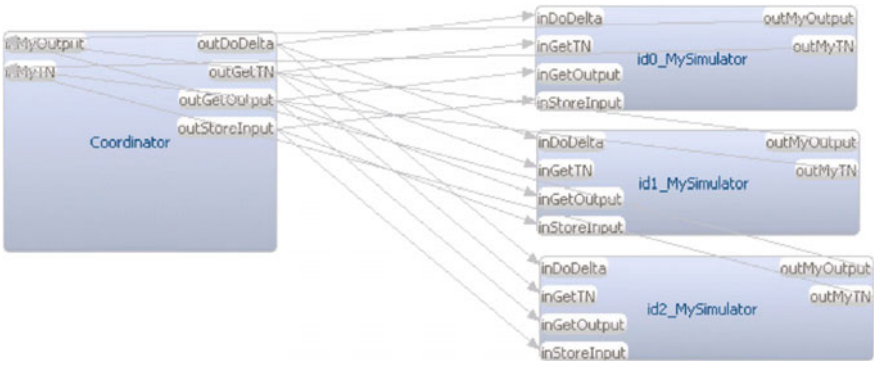

Fig. 9.8 Standard DEVS Protocol with multi-aspect coupling

9.3.2 Peer Message Exchanging Implementation

As described above, the Peer Message Exchanging DEVS Protocol modifies the basic formulation by distributing relevant segments of coupled model coupling to the respective simulators. The following FDDEVS specifications of the CoordinatorPeer and SimulatorPeer models illustrate how this allows the simulators to exchange DEVS messages without intervention of the coordinator.

CoordinatorPeer

```
to start hold in sendGetTN for time 0!
after sendGetTN output GetTN!
from sendGetTN go to waitForAllTN!

passivate in waitForAllTN!
when in waitForAllTN and receive MyTN go to sendSendOutput!

hold in sendSendOutput for time 0!
after sendSendOutput output SendOutput!
from sendSendOutput go to waitForAllDone!

passivate in waitForAllDone!
when in waitForAllDone and receive MyDone go to
sendDoDelta!

hold in sendDoDelta for time 1!
after sendDoDelta output DoDelta!
from sendDoDelta go to sendGetTN!
```

SimulatorPeer

```
to start passivate in waitForGetTN!
when in waitForGetTN and receive GetTN go to sendMyTN!

hold in sendMyTN for time 0!
after sendMyTN output MyTN!
from sendMyTN go to waitForGetSendOutput!

passivate in waitForGetSendOutput!
when in waitForGetSendOutput and receive SendOutput go to
sendMyOutput!

hold in sendMyOutput for time 0!
after sendMyOutput output MyOutput!
from sendMyOutput go to waitForStoreInput!

passivate in waitForStoreInput!
when in waitForStoreInput and receive StoreInput go to
sendMyDone!

hold in sendMyDone for time 0!
after sendMyDone output MyDone!
from sendMyDone go to waitForMyDoDelta!

passivate in waitForMyDoDelta!
when in waitForMyDoDelta and receive DoDelta go to
waitForGetTN!
```

A multi-aspect SES to couple the coordinator with simulators is and given by:

```
From the protocolPeer perspective, DEVSPeerDistributedSim
is made of CoordinatorPeer and SimulatorPeers!
From the sims perspective, SimulatorPeers are made of more
than one SimulatorPeer!
SimulatorPeer can be id in index!

From the protocolPeer perspective, CoordinatorPeer sends
GetTN to all SimulatorPeer!
From the protocolPeer perspective, CoordinatorPeer sends
SendOutput to all SimulatorPeer!

From the protocolPeer perspective, CoordinatorPeer sends
DoDelta to all SimulatorPeer!
From the protocolPeer perspective, all SimulatorPeer sends
MyTN to CoordinatorPeer!
From the protocolPeer perspective, all SimulatorPeer sends
MyDone to CoordinatorPeer!

From the protocolPeer perspective, all SimulatorPeer sends
outMyOutput to all SimulatorPeer as inStoreInput!
```

A pruning for three Simulators is illustrated in Fig. 9.7.

Note that in Fig. 9.9 the simulators exchange DEVS messages with each other directly without going through the coordinator.

Exercise

Prune the SES to select each of the alternative decompositions representing example implementations of the DEVS Protocol. Run the resulting models in the Simulation Viewer.

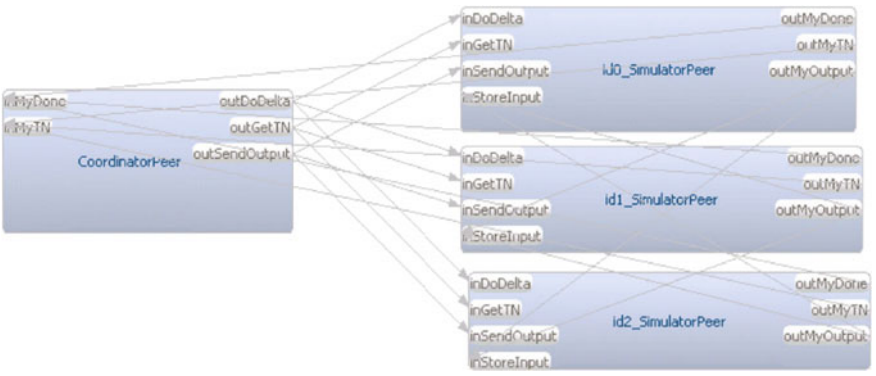

Fig. 9.9 Peer message exchanging implementation with multi-aspect coupling

Exercise

Use the sequence design interface tool to develop a sequence diagram that describes the Peer Message Exchange implementation of DEVS Simulation Protocol and generates SES and FDDEVS descriptions similar to those shown above.

Exercise

Provide tagged blocks for the SimulatorPeer and CoordinatorPeer FDDEVS specifications to implement the Peer Message Exchanging Implementation of the DEVS Simulation Protocol.

9.3.3 Real-Time Message Exchanging Implementation

As described earlier, the Real-Time Message Exchanging Implementation takes the peer message exchanging implementation one step further by letting the simulators decide on when to execute their next events to occur in real time. The following FDDEVS specifications of the CoordinatorRTPeer and SimulatorRTPeer models illustrate how this allows the simulators to determine their own time of next event in addition to exchanging DEVS messages without intervention of the coordinator.

CoordinatorRTPeer

```
to start hold in sendStart for time 0!

after sendStart output StartUp!
from sendStart go to sendStop!

hold in sendStop for time 100!
after sendStop output Stop!
from sendStop go to passive!
passivate in passive!
```

SimulatorRTPeer

```
to start passivate in waitForStart!
when in waitForStart and receive StartUp go to
sendMyOutput!
hold in sendMyOutput for time "modelTimeAdvance"!
after sendMyOutput output MyOutput!
from sendMyOutput to to sendMyOutput!
when in sendMyOutput and receive Stop go to waitForStart!
when in sendMyOutput and receive StoreInput go to
sendMyOutput!
```

A multi-aspect SES to couple the coordinator with simulators is given by:

```
From the realTimePeer perspective, DEVSDistributedSim is
made of CoordinatorRTPeer and SimulatorRTPeers!

From the rtsims perspective, SimulatorRTPeers are made of
more than one SimulatorRTPeer!

SimulatorRTPeer can be id in index!

From the realTimePeer perspective, CoordinatorRTPeer sends
Start to all SimulatorRTPeer!

From the realTimePeer perspective, CoordinatorRTPeer sends
Stop to all SimulatorRTPeer!
From the realTimePeer perspective, all SimulatorRTPeer
sends outMyOutput to all SimulatorRTPeer as inStoreInput!
```

Pruning for three Simulators is illustrated in Fig. 9.10.

Note that in Fig. 9.10, the simulators schedule their own next outputs and exchange DEVS messages with each other directly without going through the coordinator. A simulator invokes its model's time advance function to set the time at which to output a message (which can be infinity) and does this immediately after an internal, external, or confluent event. The role of the coordinator is reduced to stopping and starting a simulation run.

Exercise

Use the sequence design interface tool to develop a sequence diagram that describes the Real-Time Peer Message Exchange Implementation of the DEVS Simulation Protocol and generates SES and FDDEVS descriptions similar to those shown above.

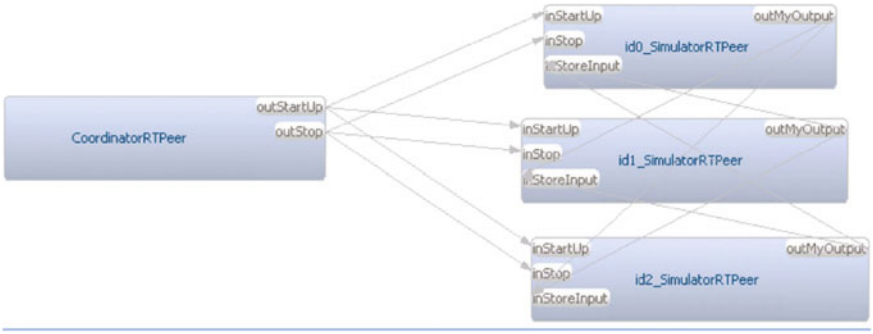

Fig. 9.10 Peer message exchanging implementation with multi-aspect coupling

Exercise

Provide tagged blocks for the SimulatorPeer and CoordinatorPeer FDDEVS specifications to implement the Real-time Peer Message Exchanging Implementation of the DEVS Simulation Protocol.

9.4 DEVS Protocol as a Standard for Simulation Interoperability

This book focuses on the DEVS modeling environment more than on the relation of DEVS to the wider world of simulation. However, one important question is the extent to which DEVS plays together with other simulation approaches. In the following, we study a typical Event-Scheduling simulator to understand how DEVS and non-DEVS simulators can be federated within the same distributed simulation.

9.4.1 DEVS Protocol with Event-Scheduling Simulator

An Event-Scheduling simulator typically can be described as follows:

- It maintains an event list ordered by time of next event.
- It has an operation, GetTimeOfImminentEvent (), which returns the time of the event at the top of the list, i.e., the smallest of all times of next event (call it tN).
- It has an operation, GetNRemoveImminentEvent(t), which stores the time, t, as the current time; then, if the current time equals tN, it also executes the code of the event at the top of the list (the imminent event) and as an effect of this code it may generate output and new events, as well as canceling already scheduled events; the output is returned as a result of the operation, and the events are inserted into the right places in the event list (these times of next event cannot be earlier than the current time).
- It has an operation, AddEvent(m, t), which treats the message, m, as an external input arriving at current time whose code is executed and may result in new events inserted into the right places in the event list.

We are interested in how to interoperate such an Event-Scheduling simulator with other models (DEVS and non-DEVS). To do so, we assign a DEVS Simulator to the event simulator which interacts with it and a DEVS Coordinator as illustrated in Fig. 9.11. In the following, we assume that the Event-Scheduling simulator accepts input and produces output in the form of DEVS messages. If it doesn't do this, then the DEVS Simulator has to be enhanced to make this translation. We return to this issue later.

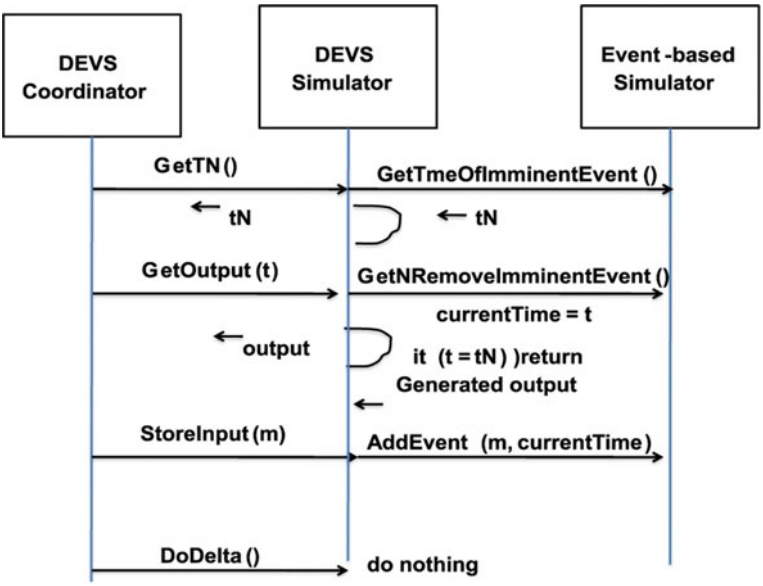

Fig. 9.11 DEVS Simulation Protocol applied to an event-based simulator

As shown in Fig. 9.11, the DEVS Simulator translates the DEVS Protocol operations sent to it by the DEVS Coordinator into operations that it invokes on the Event-Scheduling simulator. The operations GetTN, GetOutput, and StoreInput are translated into GetTimeOfImminentEvent, GetNRemoveImminentEvent, and AddEvent, respectively. The DoDelta operation is not passed on to the Event-Scheduling simulator since the latter has already executed its event code earlier.

Exercise

Develop a DEVS atomic model to implement an Event-Scheduling simulator. Use the approach of first developing an FDDEVS model and then enhance it using the process supported by MS4 Me. Hint: First define an event pair that pairs an event name with a time, e.g.,

```
An EventPair has myName and myTime!
The range of EventPair's myName is
String!
The range of EventPair's myTime is DoubleEnt!
```

Then, use a list to store and manage the event pairs appropriately. To represent how events cause outputs and schedule/cancel other events, define a method that interprets strings as instructions for generating outputs and manipulating the event list.

9.4.2 Lessons for Simulation Interoperability

From the operation of the Event-Scheduling simulator within the DEVS Protocol, we learn that there are two facets to interoperation of distributed simulators in general:

1. *Data exchange compatibility*—federates in a distributed simulation need to understand each other's messages. In the example, we assumed that the Event-Scheduling simulator understood DEVS messages and allowed that, more generally, the DEVS Simulator would have to translate between DEVS messages and a non-DEVS format. The general problem involves syntactic, semantic, and pragmatic agreements as explained in other publications (see Zeigler and Hammonds 2007; Himmelspach and Uhrmacher 2007; Kim et al. 2006; Seo and Zeigler 2012).
2. *Time management compatibility*—a correct simulation requires that all federates adhere to the same global time and their transitions and message exchanges are timed accordingly. One major feature of a DEVS-based approach is that the DEVS Simulation Protocol provides a means to enforce these timing requirements that is based on the DEVS framework, a sound theory of simulation (see Nutaro 2010 and Al-Zoubi and Wainer 2009 for a comparison with other approaches).

Developing models, simulations, and systems using MS4 Me enables you to work within a firm foundation of theory and concepts. DEVS's well-defined message and transition structures, with their well-defined semantics, give you assurance that your artifacts will stand their own ground when interfaced with non-DEVS artifacts. In subsequent chapters, we discuss how the DEVS Simulation Protocol is implemented in Data Distribution and Service-Oriented Computing middleware.

9.5 Summary

This chapter discussed the fundamental separation of models from the simulation engines that execute them intrinsic to the DEVS framework. This leads to a layered architecture of modeling and simulation services that provides the basis for

simulating DEVS coupled models that are created in a DEVS modeling environment such as MS4 Me. We used MS4 Me itself to describe the operation of the DEVS Simulation Protocol in terms of its interface requirements. These require DEVS-based agreements between a component model and its simulator, and between the simulator and the coordinator that handles the time advance and message exchange within the coupled model. We showed how different implementations can satisfy the protocol using multi-aspects and uniform coupling patterns, which also illustrated the application of modeling concepts introduced earlier in the book. In addition, there was a discussion of how a typical event-based simulator can be simulated with the DEVS Protocol and casts light on the requirements for interoperability among DEVS and non-DEVS simulators.

A Extracts from Simulator.dnl

```
use tN with type double!
use tL with type double!
use t with type double!
use myInput with type MessageBag!
use myModel with type AtomicModelImpl!

a DoubleEnt has value!
the range of DoubleEnt's value is double!
a NamedMessage has myName and myMessage!
the range of NamedMessage's myName is String!
the range of NamedMessage's myMessage is MessageBag!

accepts input on GetTN!
accepts input on GetOutput with type DoubleEnt!
accepts input on StoreInput with type MessageBag!
accepts input on DoDelta!
generates output on MyTN with type DoubleEnt!
generates output on MyOutput with type NamedMessage!

Initialize variables

<%
myModel = new AtomicModelImpl("MyModel");
myModel.initialize();
tL = 0;
tN = tL + myModel.getTimeAdvance();
t = 0;
%> !
```

```
to start passivate in waitForGetTN!
when in waitForGetTN and receive GetTN go to sendMyTN!

external event for waitForGetTN with GetTN
<%
//no processing needed, just make the transition to send
the
time of next event, tN
%>!

hold in sendMyTN for time 0!
after sendMyTN output MyTN!
output event for sendMyTN

<%
//send tN out on port outMyTN
output.add(outMyTN,new DoubleEnt(tN));
%>!

from sendMyTN go to waitForGetOutput!
passivate in waitForGetOutput!
when in waitForGetOutput and receive
GetOutput go to
sendMyOutput!

external event for waitForGetOutput with GetOutput
<%
//get the value of the current time from the input port
GetOutput
$t = messageList.get(0).getData().getValue();$
%>!

hold in sendMyOutput for time 0!
after sendMyOutput output MyOutput!
output event for sendMyOutput
<%
//if myModel is imminent (has its tN equal to t), get
myModel's output and
//send it out on port MyOutput along with myName to
identify the source
NamedMessage sm = new
NamedMessage(getName(),computeOutput(t));
output.add(outMyOutput,sm);
%>!

from sendMyOutput go to waitForStoreInput!
passivate in waitForStoreInput!
when in waitForStoreInput and receive StoreInput go to
waitForMyDoDelta!

external event for waitForStoreInput with StoreInput
```

```
<%
//look through all messages in the incoming Bag
//if there is a message for me
//store the input message on port StoreInput in myInput
MessageBag bag = messageList.get(0).getData();
myInput = getMyMessage(bag);
%>!

passivate in waitForMyDoDelta!
when in waitForMyDoDelta and receive DoDelta go to
waitForGetTN!

external event for waitForMyDoDelta with DoDelta
<%
//execute myModel's transition: check whether this is a
confluent, internal, //or external event and apply the
designated transition function
doDelta();

%>!
```

B Extracts from Coordinator.dnl

```
use tN with type double!
use tL with type double!
//use t with type double!
use simulatorOutput with type HashSet!
use simulatorInput with type MessageBag!
use myModel with type CoupledModelImpl!

accepts input on MyTN with type DoubleEnt!
accepts input on MyOutput with type NamedMessage!
generates output on GetTN !
generates output on GetOutput with type DoubleEnt!
generates output on StoreInput with type MessageBag!
generates output on DoDelta!

Initialize variables

<%
myModel = new CoupledModelImpl("MyCoupledModel");
myModel.initialize();
tL = 0;
tN =0;
//t = 0;
%> !
to start hold in sendGetTN for time 0!
after sendGetTN output GetTN!
output event for sendGetTN
<%
//none needed
```

```
%>!
from sendGetTN go to waitForAllTN!

passivate in waitForAllTN!
when in waitForAllTN and receive MyTN go to sendGetOutput!

external event for waitForAllTN with MyTN
<%
tN = Double.MAX_VALUE;
     // get the time of next event from each Simulator
     // assume they all come in together
               // ensembleBag times =
x.valuesOnPort("inMyOutput");
          for (int i = 0;i<messageList.size();i++){
          double t =
          messageList.get(i).getData().getValue();
          // get their minimum
          if (t < tN)
          tN = t;
          }
%>!

hold in sendGetOutput for time 0!
after sendGetOutput output GetOutput!

output event for sendGetOutput

<%
//send the time of next event on port outGetOutput
//to enable simulator to check if it is imminent
//and respond with its output if it is
output.add(outGetOutput,new DoubleEnt(tN));
%>!

from sendGetOutput go to waitForAllOutput!
passivate in waitForAllOutput!
when in waitForAllOutput and receive MyOutput go to
sendStoreInput!

external event for waitForAllOutput with MyOutput
<%
//get the output from each Simulator
//assume they all come in together
  for (int i = 0;i<messageList.size();i++){
     NamedMessage simout = messageList.get(i).getData();
//store each message with the simulator
                         simulatorOutput.add(simout);
                    }
//then apply the coupling to get the messages to be sent to
each simulator
                         simulatorInput =
```

```
ApplyCoupling(simulatorOutput);
%>!
hold in sendStoreInput for time 1!
after sendStoreInput output StoreInput!
output event for sendStoreInput
<%
//send each simulator the collected inputs
output.add(outStoreInput, simulatorInput);
%>!
from sendStoreInput go to sendDoDelta!
hold in sendDoDelta for time 1!
after sendDoDelta output DoDelta!
from sendDoDelta go to sendGetTN!
```

References

Al-Zoubi, K., & Wainer, G. (2009). Performing distributed simulation with RESTful web-services approach. In *Proceedings of Winter Simulation Conference*, Austin, TX (pp. 1323–1334).

Gholami, S., & Sarjoughian, H. S. (2012). Real-time network-on-chip simulation modeling. In G. Riley, F. Quaglia, & J. Himmelspach (Eds.), *SIMUTOOLS, Fifth International Conference on Simulation Tools And Techniques*, 19th–23rd March 2012. Italy, Desenzano del Garda: ACM.

Himmelspach, J., & Uhrmacher, A. M. (2007). Plug'n simulate. In *Proceedings of the 40th Annual Simulation Symposium (ANSS'07)*, Norfolk, VA, March 2007 (pp. 137–143).

Kim, J.-H., Hong, S.-Y., & Kim, T. G. (2006). Design and implementation of simulators interoperation layer for DEVS simulator. In *Proceedings of M&S-MTSA'06*, Ottawa, July 2006 (pp. 195–199).

Nutaro, J. (2010). Building simulation software: Theory. In *Algorithms, and Applications*. New York: Wiley.

Seo, C., & Zeigler, B. P. (2012). Simulation model standardization through web services: Interoperation and federation on the DEVS/SOA platform. In *DEVS Integrative M&S Symposium Proceedings of the Spring Simulation Conference*, Orlando, FL, March 2012.

Zeigler, B. P., Kim, T. G., & Praehofer, H. (2000). *Theory of modeling and simulation: Integrating discrete-event and continuous complex dynamic systems* (2nd ed.). Boston: Academic Press.

Zeigler, B. P., & Hammonds, P. (2007). *Modeling & simulation-based data engineering: Introducing pragmatics into ontologies for net-centric information exchange*. Boston: Academic Press, 448 p.