Chapter 7. Specification with DE Model

The DE model is widely used to simulate the dynamic system behavior as a function of time: for example, queueing systems, communication networks, and hardware simulations. In a Discrete Event (DE) model, a sample is associated with a time stamp indicating when it is generated. And a block processes input events in the order of arrival times. The basic scheduling strategy in the DE model is event-driven: the scheduler collects the output events from all blocks and sorts them to process one by one in the order of non-decreasing time stamp.

Since the DE domain of PeaCE is inherited from the DE domain of Ptolemy classic, both have the same simulation capabilities. Like Ptolemy PeaCE also allows a hierarchical composition of different models of computation, a DE block may encapsulate a nested block diagram with a different model of computation: for example an SPDF block diagram (for a computation task) or an FSM block diagram (for a control task). But there are a couple of differences between PeaCE and Ptolemy classic.

1. Unlike Ptolemy, PeaCE does not allow arbitrary composition of domains. The DE domain may not be nested in an SPDF and FSM domain. The DE domain should be the top-level domain as the simulation engine.
2. Unlike Ptolemy classic, the top-level DE model may intervene the internal behavior of each task. It plays the role of the simulation engine managing interaction between subgraphs (or tasks).

NOTE: The DE domain in PeaCE is used only for system simulation: It is not included in the co-design flow. Nonetheless the DE domain is useful to develop and test each task (especially a control task) since it provides a rich set of library I/O blocks for test input generation output display.

Let’s start the journey to the DE world from drawing a simple DE graph using Pre-defined Library Blocks. If you have not looked at the SPDF section, please do so before proceeding further. We omit the explanation of some terminologies in this section assuming that you are already familiar with them from the SPDF chapter.

7.1 Draw a DE graph Using Pre-defined Library Blocks.

Let us draw a simple DE graph that receives a periodic sequence of saw-tooth wave samples with exponentially distributed inter-arrival times (Poisson arrivals), samples them at a constant rate, and displays the sampled output. Figure 7-1 displays what we are now going to draw and what we obtain from the graph.

The procedure you have to perform in HAE is following:

(1) Open a new facet and name it as you like (“File”–“New”). And set the domain to “DE” and the target to “default-DE”. There is a parameter in the default-DE target to select the scheduling option: “calendar queue scheduler?”. Currently, you set it to “NO”. It means that you use the basic event-driven scheduler. The “calendar queue scheduler” uses multiple event queues to reduce the event sorting time.
(2) Bring two event generators from the block library (library/peace/DE/src): one is a Poisson block and the other
is a Clock block. The “mean” parameter of the Poisson block indicates the mean of exponentially distributed
inter-arrival times of events: we set it to “1.0”. The “magnitude” parameter indicates the constant value of the
output events. The “Clock” block produces periodic events whose period is determined by the “interval”
parameter of the block. Please refer to the document for the library blocks to understand the block function.

(3) While event generators define the timing of the events, we need to attach a functional block to define the
values of the events. A functional block manipulates the event values and usually has a constant time delay or
zero delay. Now, we bring a “Waveform” block from (library/peace/DE/src) to make saw-tooth wave samples
with the following parameters: “value = 0 1 2 3 4 5” and “periodic = YES”. In general, a “conceptual” DE-
block is modeled by a pair of a delay-type block and a functional block. A delay-type block specifies the delay
of the block and the functional block defines the functionality of the block. Considering that an event generator is
a delay-type block, pairing it with a “Waveform” block defines a source block that generates a sequence of saw-
tooth wave samples with Poisson arrival.

(4) Bring a copy of the “Sampler” block from (library/peace/DE/func/control) and connect the “input” port of the
sampler block to the output of the Waveform block and the “clock” port to the output of the Clock block.

(5) For output display, we bring a display block called “Xgraph” from (library/peace/DE/sink) and connect the
output of the sampler block to its input port. The “red” color of the Xgraph block indicates that any type of
porthole may be connected to. For now, we use the default parameter values for the Xgraph block.
Now, “run” the application by clicking the “Run” menu and setting the “When to stop” value to 30, which indicates that the scheduler stops its operation at global time 30.0 or it stops processing the events with larger time stamps than the stop value.

Now, we believe that you understand how to draw and execute a DE graph using pre-defined blocks. It is fairly straightforward except that it is your responsibility to find out suitable blocks from the block library. If there is no suitable block for your purpose, you have to define your own. This is the topic of the next section.

7.2 DE Block Definition

Let’s make “PWM” (Pulse-Width-Modulation) block which will output as many samples as the input value. We define a parameter called “interval” to determine the time interval between output samples. Make sure that the output samples associated with a given input event are produced before receiving the next input event, by setting the “interval” parameter smaller than the clock period of the Sampler divided by the maximum input value. And, insert this block between the Add block and the Xgraph block in Figure 7-1. Since there is no “PWM” block in the block library, it has to be created.

To begin with, we will create the PWM block using the “copy-and-modify” approach. Since the PWM block has one input and one output port, we copy the Ramp block and modify it: “cp SPEACE/src/domains/de/stars/DERamp.pl ~your_name/work/DEPWM.pl”. Figure 7-2 shows the definitions of the Ramp block and the PWM block. The following sections are modified.

1. name: Sure, you have to set the name of the block.
2. author, location, desc, version: These are optional.
3. input, output: You may change the names. The type of the “input” port is set to “int” meaning that the block receives integer samples from this port.
4. state or defstate: Define two float states, “interval” and “magnitude” to specify the time interval between output samples and the sample magnitude respectively.
5. constructor, setup: This block needs no special constructor and setup methods.
6. go: This section defines the main function body of the block. The PWM block first reads a current sample from the “input” port by calling the “get()” method: input.get(), and assigns the sample value to a local variable, v. If this value is negative, we generate an error message and halt the simulation. The get() method returns the most recent sample value. Note that the Ramp block does not read the input sample since the input sample is used only to trigger the block execution.

The time stamp of the incoming event is set to the “arrivalTime” variable before block execution. A DE block has an internal variable, “completionTime” to record the time stamp of the current output sample. This block is designed to produce $2+$(input value) output samples including two samples of value 0.0 at the start and the end of the output sequence. The time interval between two output samples is determined by the “interval” parameter. An output sample is produced by using “put(time_stamp)” method whose argument is nothing but the time stamp of the output event: output.put(completionTime).
After you define a new block, insert the block into PeaCE by “importing” the block by clicking “Tool – Import Star” in HAE. Then, the block is compiled and dynamically linked to the PeaCE kernel and a new icon is created in HAE. Figure 7-3 shows how the newly created block is inserted into the block diagram and what are the results of the graph execution.

Figure 7–2 Definitions of (a) the Ramp block and (b) the newly created PWM block
7.2.1 Delay-type and Functional Blocks

A DE block can be viewed as an event-processor; it receives events from the outside, processes them, and generates output events after some latency. In a DE block, generating output values and computing time stamps are separable tasks. For the purpose of modularity, therefore, some DE blocks, so-called delay-type blocks, are dedicated to time management, adding a non-zero processing delay to the output timestamp, while other blocks, so-called functional blocks, omit time-management and produce output events with the same time stamp as the input events. They, however, do manipulate the value of the samples. Such separation is not obligatory but recommended. The PWM block described above is not a delay-type star since it generates an output event with the same time stamp with the input event. But, it is not purely functional either since it manipulates the time stamps of the output samples.

An example of a delay-type block is the “Delay” block that adds a constant delay to the time stamp of an event. The body of the block definition includes the following:

```
constructor { delayType = TRUE; }
go {
    completionTime += double(delay);
    Particle& pp = input.get();
    output.put( completionTime) = pp;
}
```

If a block is delay-type, you are requested to say so in the “constructor” section by setting “delayType = TRUE”. This information is used by the scheduler to detect whether there is a delay-free loop or not. If there is a delay-free loop, the simulation may go into an infinite loop! Another thing you should note is “Particle” object in the go() body. Since the Delay block does not manipulate the sample value but only the time stamp, it just copies the input sample to the output sample. An input sample is carried through a “Particle” object in PeaCE (or in Ptolemy classic). Method call of input.get() returns the Particle object. If you type-cast a Particle object, you can get the
value of the sample as shown in the body of the PWM block: \texttt{int v = input.get();} When a Particle object is passed to the output port, you should use "=" operator rather than "<<" operator: \texttt{output.put(completionTime) = pp;}

### 7.2.2 Blocks With Multiple Inputs

In the DE model of computation, a block is \textit{Runnable} (ready for execution), if any input port has a new event, and that event has the smallest time stamp of any pending event in the system. When the block is executed, it may need to know which input or inputs contain the events that triggered the firing. It can be done by examining the \texttt{dataNew} flag associated with an input port since the scheduler sets the flag when it delivers a new event to the block. The \texttt{dataNew} flag of the port is reset to zero after the input sample is read by \texttt{get()} method. To see how this is done, consider the Sampler block of Figure 7-1.

```plaintext
defstar{
    name {Sampler} domain {DE}
    desc { }
    input { name {input} type {anytype } }
    input { name {clock} type {anytype } }
    output { name {output} type {=input} }
    constructor {
        input.triggers();
        input.before(clock);
    }
    go {
        if (clock.dataNew ) {
            completionTime = arrivalTime;
            output.put(completionTime) = input%0;
            clock.dataNew = FALSE;
        }
    }
}
```

The Sampler block has two input portholes: input, and clock. When an event arrives at the \textit{input} port, it does nothing and simply returns. But when an event arrives at the clock port, it routes the most recently arrived input sample to the output port. In the \textit{go} section, it checks whether a new input event has arrived at the clock input by examining the \texttt{clock.dataNew} flag. Note that the most recently arrived sample is accessed by using "\%" operator instead of \texttt{get()} method: \texttt{input\%0} instead of \texttt{input.get()}. If we use "\%" operator to access the sample, the \texttt{dataNew} flag is not reset to zero.

You can notice that there are two lines you have never seen before in the constructor section. In fact, these two lines are optional. They are needed to help the scheduler schedule the events with the same time stamp. In case there is more than one event with the same time stamp, the scheduler should decide which event to process first. Method call \texttt{input.before(clock)} indicates that the scheduler should give a higher priority to the event to the input port than the event to the clock port. If this method is not called, the behavior is not easily predicted when simultaneous events arrive at the block. On the other hand, \texttt{triggers()} method indicates which output port is triggered instantaneously by which input port. By default, all outputs are triggered, or produce events, when an event arrives at any input. Therefore \texttt{input.triggers()} method indicates that an event to the input port does not produce any zero-delay output. By default, \texttt{clock.triggers(output)} is assumed. Why do we need these complicated things? It helps you to make debugging easier. See section 7.3.1 to understand the need of these methods.
Now we summarize the methods and the operators associated with the ports in the DE domain in Table 7-1. The first four methods are associated with input ports and the last one with output ports.

Table 7-1 Methods associated with ports in the DE domain

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>get()</td>
<td>Read the newly arrived input sample, or Particle. The dataNew flag is reset to zero.</td>
<td>int v = input.get();</td>
</tr>
<tr>
<td>%0</td>
<td>Read the most recently arrived input sample, or Particle.</td>
<td>Particle&amp; pp = input%0;</td>
</tr>
<tr>
<td>triggers()</td>
<td>Indicate which output ports will produce events with the same time stamp as the input event.</td>
<td>input.triggers(out1, out2); input.triggers();</td>
</tr>
<tr>
<td>before()</td>
<td>Determine the event processing order between simultaneous events that arrive at multiple input ports.</td>
<td>input.before(clock);</td>
</tr>
<tr>
<td>put(double time)</td>
<td>Send the output sample with the given time stamp.</td>
<td>output.put(completionTime) &lt;&lt; v; output.put(x) = pp;</td>
</tr>
</tbody>
</table>

7.2.3 Event Generators

Some DE blocks are event generators that do not consume any event, and hence cannot be triggered by input events. In the beginning of the simulation, these blocks are triggered by the scheduler. All such blocks are derived from the base class RepeatStar in the PeaCE (or Ptolemy classic) kernel. In a RepeatStar, a special hidden pair of input and output ports are created and connected together. This allows the block to schedule itself to execute at any desired future time(s), by emitting events with appropriate time stamps on the feedback loop port. The hidden ports are identical to normal ports, except that they are not visible in the graphical user interface. Let’s look at the body of the Clock definition below.
The Clock block is derived from “RepeatStar” that has a hidden pair of feedbackIn and feedbackOut ports. To schedule itself, we use the refireAtTime() method that puts an event to its feedbackOut port. RepeatStar can also be used to emulate time-driven execution of the block: the block is not triggered by the arrival of input events, but by the arrival of feedback events only.

7.3 Discrete Event Simulation

Discrete event simulation is a well-known technique to simulate the system’s timing behavior. In the DE domain, events are irregularly spaced in time and system responses are generally very dynamic, all scheduling actions are performed at run-time. At run-time, the DE scheduler processes the events in chronological order until simulated time reaches the global "stop time."

The current DE domain has two schedulers that maintain a global event queue where particles currently in the system are sorted in accordance with their time stamps; the earliest event in simulated time being at the head of the queue. They are the default scheduler and the “calendar queue” scheduler. The default DE scheduler uses a single sorted list with linear searching. You may use the “calendar queue” scheduler by setting the “calendar queue scheduler?” parameter of the default DE target to YES. As the number of unprocessed events increases, the performance gain of using the calendar queue scheduler gets larger.

The basic action of the scheduler is to fetch an event from the head of the global event queue and deliver to the destination block. One thing to note is that the scheduler searches the event queue to find out whether there are any simultaneous events at the other input port of the same block, and fetches those events. Thus, for each firing, a block can consume all simultaneous events for its input portholes. After a block is executed it may generate some output events on its output ports. These events are put into the global event queue. Then the scheduler fetches another event and repeats its action until the given stopping condition is met.
7.3.1 Simultaneous events

You have to pay special attention to simultaneous events in any discrete event simulation. Any DE simulator has its own mechanism to decide which event to process first among the simultaneous events. A simplest approach is to select one randomly. Then, the system behavior becomes non-deterministic and the programmer has hard time to predict the system behavior. In general, we cannot avoid such non-determinism. But, DE simulators usually do something to reduce the probability of such problem as much as possible.

Open the “sampler” demo in Peace/Demo/De/Basic as shown in Figure 7-4. The Ramp block is a functional block with zero latency. At time 0, two Clock blocks generate events to the Ramp block and the Sampler block respectively. Which event do you thing should be processed first? If the Ramp block is executed first, the Sampler block will receive two input events, one from the input port and the other from the clock port and produce the output sample that is nothing but the received input sample. But, if the Sampler block is executed first, the Sampler block misses the simultaneous event from the input port.

PeaCE solution, or Ptolemy solution, to this problem is to sort the events in the topological order among simultaneous events. The event to the Ramp block is given a higher probability than the event to the Sampler clock port because the Ramp block is topologically higher than the Sampler block. We set the topological level of all delay-type blocks to zero. On the other hand, some simulators use “delta” time concept for the functional block meaning that the non-zero but infinitesimal delay is added to the time stamp. So, the output sample from the Ramp block is assigned a higher time stamp than the event at the clock port of the Sampler block.

If functional blocks form a loop, topological sorting is impossible. To prevent this from happening, cut the loop setting the delay parameter of any arc on the loop. In the Dataflow model, this delay element indicates the sample delay. However, in the DE model, this delay element is just to set the topological level of the destination block to zero.
If a zero-delay event-path forms a loop, we call it a delay-free loop. While a delay-free loop in the SPDF domain results in a deadlock of the system, a delay-free loop in the DE domain potentially causes unbounded computation. Therefore, it is advisable to detect the possible delay-free loop at compile-time, while it may be a false alarm. In PeaCE, we detect the possibility of a delay-free loop by examining the “triggering” path specified in the Block definitions. If a zero-delay triggering path is found, a warning is signaled at compile time. The programmer may ignore such warning at his/her responsibility.

7.4 Mixed Simulation Capability

The key feature of Ptolemy is to mix heterogeneous models of computation in a hierarchical fashion. Since PeaCE is extended from Ptolemy, it has the same capability. Differently from Ptolemy that allows arbitrary mixture of models, however, we restrict the usage of the DE model as the top-level model for system-level design. If you use a super block of a different domain in a DE domain, the super block is called a Wormhole in Ptolemy’s terminology. In this section, we will explain how to mix the DE simulation with SPDF simulation and analog simulation. See the next chapter for the mixture of the DE model and the FSM model.

7.4.1 Mixed Simulation With SPDF Model

Mixed simulation with SPDF model can be easily achieved by using a SPDF super-block (or Galaxy) in the DE domain. Figure 7-5 shows a simple “worm” demo that contains a SPDF super-block that adds an input sample and a Gaussian random value to produce an output. You can find this demo in the demo directory of PeaCE. In this simple example, the super-block contains only two SPDF blocks: one is Add and the other is IIDGaussian. If you look-inside this super block and check the domain, you can find that it is set to “CGC” and its target is set to “worm-CGC”.

Since the CGC domain follows the SPDF model, the inside graph is a SPDF graph. At the setup stage of the application graph, the super block creates a scheduler object and schedules the inside graph: the schedule of the inside graph becomes IIDGaussian – ADD. When the super block receives an input event from the Poisson block,
the inside graph is executed once. The output of the super-block is now delivered to the destination block with the same time stamp as the input sample. In other words, the super-block is regarded as a functional block. You have to append a delay block if you want to model the execution delay of the super-block.

In case a super-block of SPDF model has only one input sample, the operation is simple. But if a super-block has more than two inputs, you need to be cautious. Figure 7-6 shows such an example: “adder” example. The “sdf_adder” super-block contains only a single Add block that is executable only when both inputs have new samples. On the other hand, the super-block is fired from the DE scheduler when a new input event arrives at any input. There is a semantic mismatch between the outside DE model and the inside SPDF model. To resolve this mismatch, the wormhole (super-block) performs some interface logic. The default interface logic is to queue the input samples at each port until the wormhole has new events at all input ports. When all input ports have new events, the wormhole deliver one input sample per port to the inside SPDF graph. In this example, one input port receives input samples with period equal to 3 while the other with period equal to 2. Then, the input samples with period 2 are queued up.

![Figure 7-6 “adder” demo](image)

Since such data synchronization action is not evident from the programmer, we recommend you to use explicit synchronization block at the input ports to a wormhole with more than two input ports. In Figure 7-7, a Sampler block is added between the Ramp block and the “sdf_adder” super-block. The input samples with period 2 are sampled by the samples with period 3. Then, some input samples with period 2 are overwritten by the subsequent samples before being sampled. For instance, a sampled produced at time 4 is missing in the result.
In fact, mixed simulation with SPDF model is a distributed simulation between PeaCE kernel and the C process created by the CGC domain. Therefore, the target of the wormhole should be set to “worm-CGC” since this target will add the required socket interface code to the generated C code.

### 7.4.2 Mixed simulation with Octave Analog Simulator

In this section, we show how PeaCE can achieve mixed simulation with an analog simulator, called Octave. Octave has similar capability to MatLab for numerical computation but it is a free-ware under GPL license. Since Octave allows interactive execution by interpreting the command line queries, it is simple to use Octave program in the DE model. Figure 7-8 shows a simple example of mixed simulation.
There is a DE block, called OctaveODESolver, that is used to compose the queries and send them to Octave. At the setup stage, PeaCE forks Octave as a child process and kills it when the simulation ends. The OctaveODESolver block writes the queries to the standard input, which are delivered to the Octave via unix pipe mechanism. The OctaveODESolver waits until it receives the outputs from the Octave via pipe.

If you click the OctaveODESolver block, you can see the parameter window as shown in Figure 7-9. There are two states you have to fill up for composing the query. In the “query” state, you write down the first order differential equation and variable initialization. The initial values of state elements are put into the “initValue” state that is float-array type. The example of Figure 7-9 shows a simple RC circuit response. The Clock and WaveForm blocks generate a square wave of amplitude 5. The OctaveODESolver block specifies the first order differential equation of the RC circuit. The result is displayed with an X graph.

![Figure 7-9 Parameters of OctaveODESolver block for simple RC circuit](image)

Using the same graph, we can obtain a different result by changing the state values. Figure 7-10 shows a different set of states to solve van der Pol equation. This demo can be found in the demo directory.
Figure 7–10 Parameters of OctaveODESolver block for van der Pol equation

For more detailed discussion on the query format, please refer to the Octave manual that is available on-line. There are some limitations of mixed simulation with Octave. Currently, mixed simulation is performed as a distributed simulation that takes significant portion of time for inter-process communication. Second, the current version of PeaCE supports a subset of Octave queries: queries for differential equation. It is kind of experimental capability of PeaCE. If needs arise, more complete support will be provided.