Chapter 9. Task level specification model (BP Domain)

Author: Soonhoi Ha and Dohyung Kim

The SPDF model and fFSM model is used to describe the internal behavior of a computation and a control task respectively. We use another model, called Task Model, to describe the system behavior at the task level. Thus we are able to represent multi-tasking applications. In PeaCE, we can define a super block that encapsulates a block diagram with a different model of computation to make a hierarchical block diagram. At the top-level, the Task model represents a composition of multiple tasks: each task is modeled as a SPDF model or fFSM model. Since we use the Task model in this fashion, we call the domain of Task model as the BP (Backplane) domain.

NOTE: The BP domain is a code-generation domain, So a DE block (or a DE graph) can not be used in the BP domain at the current implementation. The BP domain is used mainly for HW/SW codesign of the system though you may use it only for functional simulation of the system. To use it as a codesign backplane, open a new project instead of a new facet. The next chapter will describe the overall codesign flow.

9.1 Task model in PeaCE

Tasks in the system have diverse activation conditions and port semantics that should be clearly specified in the task-level specification model at the top-level. Table 9-1 summarizes the classification of supported task types and port properties. We support three types of tasks: periodic tasks triggered by time, sporadic tasks triggered by external IO, and function tasks triggered by data. And we also define various port semantics by combining port type, data size, and data rate. For example, the data port semantic for SDF model is classified as a static-rate static-size queue and the data port semantic for FSM model as a dynamic-rate static-size queue.

<table>
<thead>
<tr>
<th>Task type</th>
<th>Periodic</th>
<th>Sporadic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Triggered by time</td>
<td>Triggered by external IO</td>
<td>Triggered by data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Port type</th>
<th>Queue</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Destructive read/non-destructive write</td>
<td>Non-destructive read/destructive write</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data size</th>
<th>Static</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size is determined at compile time</td>
<td>Size is determined at run time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data rate</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rate is determined at compile time</td>
<td>Rate is determined at run time</td>
</tr>
</tbody>
</table>
Note that the internal behavior of a task is represented as the SPDF model or the fFSM model. On the other hand, the task itself has diverse execution semantics as shown in Table 9-1. Then, the problem is how to connect the internal SDF and FSM models to the outer task-model.

We use a task wrapper approach where a task wrapper is created at the boundary of hierarchical node (or a task node) that translates the outer execution semantics to the internal execution semantics. For example, if a SPDF task is defined as a periodic task with a static-size static-rate buffer port, arrival of input data is ignored while the arrived data is stored in the port buffer. Instead, the current values stored in the input port buffers are delivered to the inside at periodic wake-up. Such translation is the role of the task wrapper and is performed in the kernel of PeaCE, being invisible to the user: Never mind how it works.

Figure 9-1 overviews the sequence of task wrapper creation using a DIVX player example. The DIVX player example consists of one control(FSM) task(control FSM) and four computation(SPDF) tasks(User interface, control FSM, AVI reader, H.263 decoder, and MP3 decoder).

Figure 9-1 (a) Task level definition without applying task wrapper (b) control ports appended by control path analysis (c) control ports appended by task types
At first the task type of each is specified by the user. The “user interface task” is a sporadic task that is triggered by the user input and the “AVI reader” task is a periodic task that is invoked periodically. The other tasks are set as function, which is the default task type so can be omitted.

The task wrapper defines port semantics for each data port of SPDF and fFSM models. In this example, data ports between the user interface task and the control FSM task are defined as dynamic-rate static-size buffer-type.

It appends control ports to the SPDF or fFSM task by analyzing the scripts in the FSM model (see Chapter 8 for the detailed explanation of the fFSM model and scripts) and also defines a port semantic for each type of control port. In Figure 9-1 (b), two control connections are created between the control task and the divx mode to exchange the control command and the status signal.

At last, it appends an additional control port associated with the task type. Through this port, the supervisory task gives commands to change the execution status of the task. We summarize the port semantics for the task wrapper in Table 9-2.

### Table 9-2 Port semantic definitions for the task wrapper

<table>
<thead>
<tr>
<th>Port type</th>
<th>Port semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data port of SDF model</td>
<td>Static-rate static-size queue</td>
</tr>
<tr>
<td>Data port of FSM model</td>
<td>Dynamic-rate static-size buffer</td>
</tr>
<tr>
<td>Scheduling control</td>
<td>Static-rate static-size queue</td>
</tr>
<tr>
<td>Parameter delivery control</td>
<td>Static-rate static-size buffer</td>
</tr>
<tr>
<td>Exception handling control</td>
<td>Dynamic-rate static-size queue</td>
</tr>
<tr>
<td>Control for periodic task</td>
<td>Static-rate static-size queue</td>
</tr>
<tr>
<td>Control for sporadic task</td>
<td>Static-rate static-size queue</td>
</tr>
</tbody>
</table>

Sometimes we meet a need to specify a dynamic-rate variable-size (DRVS) port type in the SPDF task. In the MP3 player task, for example, the required data size of one encoded frame cannot be determined until the MP3 decoder task starts decoding and detects it at runtime. In Figure 9-2, the sample rate x is determined at runtime by the MP3 decoder itself. The Task model in PeaCE allows dynamic-size ports to specify this case.

![Figure 9-2 Dynamic-rate variable-size port type](image)

NOTE: the port semantics and task semantics only define the behavior not the implementation of the task-model.

### 9.2 How to design a task-model in PeaCE

This chapter tries to explain the basic notions of task-model by using a simple example. After reading this chapter, you should understand how to define a functional task and a periodic task in the task-model, how to define a task-model using pre-defined tasks and what meaning the target parameters in the task-model have.
As explained in the previous section, task-model supports three types of tasks: functional task, periodic task and sporadic task. Functional task is invoked by data, periodic task by time and sporadic task by interrupt. It is simple to define a functional task. After you describe the algorithm in the CGC domain, you can make it as a functional task by setting the target of the graph “Task-Model” as shown in Figure 9-3. You can place an Xgraph from library/Peace/SPDF/Sink, a super-block input port from library/Peace/Port and make a connection between them. Then you register the task as Galaxy to use it in task-model.

To define a task as a periodic task, what you have to do is only defining a new galaxy parameter named “period” as INT type and setting a positive period value. However, if you set the value of the “period” parameters to –1, the task is handled as a functional task again. Let’s make another task as shown in Figure 9-4; It is a periodic task with a Ramp block (library/Peace/SPDF/Src) and a galaxy output port (library/Peace/Port). And the “period” parameter is set to 1, which indicates that the task is invoked periodically at every one-time unit.

Then we are ready to design first example in BP domain (Figure 9-5) for the task-model which is composed of two tasks. One task with Ramp block sends an increasing integer value periodically (Figure 9-4) and the other task with Xgraph block displays the value when each data arrives (Figure 9-3). You move two tasks from saved library to the schematic and make a connection between them.
Figure 9-4 Defining a period task

Figure 9-5 Defining a BP graph with two tasks
To use the Task-model, you should select the BP (backplane) domain and the default-BP target. The default-BP target is in charge of generating codes, compiling the codes and running the executable for functional simulation. In this chapter, we explain default-BP target for functional simulation and other targets in BP domain will be explained in the PeaCE design flow chapter and later chapters in more details. The default-BP target supports the following target parameters:

<table>
<thead>
<tr>
<th>Target parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display?</td>
<td>show a header file which has the data structure of all component tasks</td>
</tr>
<tr>
<td>compile?</td>
<td>compile the generated codes</td>
</tr>
<tr>
<td>run?</td>
<td>run the compiled executable</td>
</tr>
<tr>
<td>debug?</td>
<td>run the compiled executable with gdb debugger</td>
</tr>
<tr>
<td>profile?</td>
<td>show the execution profile of each task</td>
</tr>
<tr>
<td>profile block?</td>
<td>show the execution profile of each block in tasks</td>
</tr>
<tr>
<td>compile options</td>
<td>if it is specified, all compile options in tasks are ignored</td>
</tr>
<tr>
<td>link options</td>
<td>if it is specified, all link options in tasks are ignored</td>
</tr>
</tbody>
</table>

Now run the application. Then you will obtain the codes for each task, a header file with data structure of all tasks, a simulation engine for functional simulation and makefile (Table 9-3). And they are compiled together and executed as shown in Figure 9-6.

![Figure 9-6 Results after running the application of Figure 9-5](image)

Table 9-3 Generated files from Figure 9-5

<table>
<thead>
<tr>
<th>Generated codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task10.c, test_sync12.c</td>
<td>task codes</td>
</tr>
<tr>
<td>taskAddressMap.h</td>
<td>header file with task data structures</td>
</tr>
<tr>
<td>makefile_test_BP_0</td>
<td>Makefile</td>
</tr>
<tr>
<td>task.h, sem.h, sem.c, taskSynthesisAPI.c</td>
<td>simulation engine for functional simulation</td>
</tr>
</tbody>
</table>
If you set both “profile?” and “profile block?” flags to YES and execute the task-model, then “log.c, log.h” files are appended to the simulation engine. Then you will get the profile results on the execution times of each task and blocks, as shown in Figure X. It can be used to analyze the complexity of the algorithm and optimize the algorithm.

Figure 9-7 Profiling results from Figure 9-5

9.3 Sporadic task in the Task-model (EXPERIMENTAL)

Sporadic task is an aperiodic task which is triggered by interrupt. For example, a user input from the user interface and a call request from network generates such an interrupt. So a sporadic task should wait for occurrence of an interrupt. Although interrupt and wait routines are usually implemented using OS primitives, how to model such interrupt and wait routines in function simulation without operating system is not clear. So it is now at the experimental stage supporting sporadic tasks in functional simulation.

To define a sporadic task, you should write an interface SPDF block based on the pre-defined form to support sporadic execution as shown in Figure 9-8. Blocking/releasing mechanism is implemented by semaphore OS APIs as shown in Table 9-4. In the initCode section, it creates a thread which is blocked on a Linux system call to receive data from external IO. In the main body of the block (go section), “wait interrupt” OS API is called, which releases the task after the thread receives data from the system call. Then it sends the data to the output port. At the wrapup section, it generates the wrap-up code to detach the thread.

Table 9-4 OS APIs implemented in PeaCE for functional simulation of sporadic tasks

<table>
<thead>
<tr>
<th>Names</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem_t</td>
<td>semaphore variable</td>
</tr>
<tr>
<td>sem_init(sem_t *sem, int initial_number)</td>
<td>initialize a semaphore with an initial number</td>
</tr>
<tr>
<td>sem_v(sem_t *sem)</td>
<td>release a semaphore</td>
</tr>
<tr>
<td>sem_p(sem_t *sem)</td>
<td>acquire a semaphore</td>
</tr>
</tbody>
</table>
Once you write an interface block which supports sporadic execution as the source block of a task, it is handled as a sporadic task.

9.4 SPDF extension in Task-model

PeaCE provides two SPDF extensions in the Task-model. One extension is defining a variable-size dynamic-rate port to enable a block to consume or produce variable-size and dynamic-rate data. The SPDF model assumes that the number of samples to be consumed is already known at compile-time or at best before execution of the block at run-time (like dynamic construct). But there are cases that we do not know how many samples are needed beforehand in some media applications such as MP3 player. Therefore we need a mechanism of blocking read in the Task model. The variable-size dynamic rate port implements the blocking read mechanism in PeaCE. This type of port is allowed only at the boundary of the task.
The other extension is exception handling. An SPDF task may need to exit during execution if a certain exit condition is met. An error generated during the execution can be an exit condition. If such an exit condition is met, it may need to signal the exception to the top-level BP domain.

In this section, we explain these two extensions with simple examples.

### 9.4.1 Variable-size dynamic-rate port

To use a variable-size dynamic-rate port in a block, you should define the port as MESSAGE type and set some properties at setup function as shown in Figure 9-9. Put “VARIABLE” as the argument of `setMessageName` method, which indicates that this port is a variable-size dynamic-rate port. The argument of `setMessageSize` method indicates the maximum buffer size of the port.

```plaintext
input {
    name {input}
    type {MESSAGE}
}

setup {
    input. setMessageName("VARIABLE");
    input. setMessageSize(128);  // ring buffer size
}
```

**Figure 9-9 Definition of a variable-size dynamic rate port**

After defining the port, you can use the following four OS API functions (Table 9-5) to access the port.

<table>
<thead>
<tr>
<th>OS API functions</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>available($ref(port_name))</td>
<td>wait data arrival and then return available data size</td>
</tr>
<tr>
<td>available_nonblock($ref(port_name))</td>
<td>return available data size immediately</td>
</tr>
<tr>
<td>read_port($ref(port_name), void *data, int size)</td>
<td>wait data with the ‘size’ and copy the data from the buffer to ‘data’ variable</td>
</tr>
<tr>
<td>write_port($ref(port_name), void *data, int size)</td>
<td>write data with the ‘size’ from ‘data’ variable to the buffer</td>
</tr>
</tbody>
</table>

**Table 9-5 OS APIs to access a variable-size dynamic rate port**

Figure 9-10 shows a simple example of variable-size dynamic-rate port.

(1) Open a new facet and set the domain to BP and the target to default-BP.

(2) Reuse two SPDF tasks of Figure 9-5. Instantiate two periodic tasks with the same period (we set the period as 1) to generate Ramp inputs and a display task as a functional task.
(3) Define two new tasks of which each contains only one block inside for simplicity. STPacketize block reads two values and packetizes them into an AddPacket structure as shown in Figure 9-11 (a). Then AddVariable block defines a variable-size dynamic-rate port although it reads two values from the port statically and sends the summation of values (Figure 9-11 b). Then sink task displays the value.

(4) Run the application.

Figure 9-10 An example of using a variable size dynamic rate port
hinclude { <math.h>, "Error.h" } 
cinclude { "Message.h" } 

setup {
    output.setMessageName("VARIABLE");
    output.setMessageSize(2*sizeof(float));
}

codeblock(mainBlock) {
    float value;
    read_port($ref(data), (char *)&value, 4);
    $ref(output) = (double)value;
    read_port($ref(data), (char *)&value, 4);
    $ref(output) += (double)value;
}
go { 
    addCode(mainBlock);
}

hinclude { <math.h>, "Error.h" } 
cinclude { "Message.h" } 

setup {
    output.setMessageName("AddPacket");
    output.setMessageSize(2*sizeof(float));
}

codeblock(mainStruct) {
    typedef struct AddPacket {
        float inA, inB;
    } AddPacket;
}

input {
    name { dataA } 
    type { float } 
}

input {
    name { dataB } 
    type { float } 
}

output {
    name { output } 
    type { message } 
}

codeblock(mainBlock) {
    $ref(output).inA=$ref(dataA);
    $ref(output).inB=$ref(dataB);
}

initCode { 
    addGlobal(mainStruct, "struct AddPacket");
}
go { 
    addCode(mainBlock);
}

Figure 9-11 Definitions of (a) CGCSTPacketize block and (b) CGCAddVariable block

Figure 9-12 Execution result
9.4.2 Exception handling

To produce an exception in a block, you can use exit_application() OS API in the main body of the block as shown in Figure 9-13. The template code reads data from the input port and sends data to the network using the write system call. During the execution, if the size of sent data is not equal to the requested size, it means that there is a problem in the network. In that case, it calls the exit_application() function and the task which has the block is terminated with an exception signal.

Figure 9-14 shows an fFSM example that has a concurrent FSM to handle the exception condition. The exception signal generated from the “exit-application” API is delivered to the control FSM task. In the control FSM task, the signal can be handled by “get” script shown below.

<table>
<thead>
<tr>
<th>Script definition</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>get task_name exit internal_event_name</td>
<td>Catch an exception from task_name task and set internal_event_name event active</td>
</tr>
</tbody>
</table>

```c
codeblock (goCode) {
    int write_size, data_size;

    read_port($ref(input), $ starSymbol (buf ), 8);
    data_size = *( int *)($ starSymbol (buf )+4);
    read_port($ref(input), $ starSymbol (buf )+8, data_size);

    write_size = write($ref( sId), $ starSymbol (buf ), data_size+8);
    if (write_size!=data_size+8) {
        exit_application(); // error condition
        return;
    }
}
```

Figure 9-13 Code template to exit the task at an error condition
Define task group to control multiple tasks simultaneously

Sometime you may want to control multiple tasks simultaneously in a single state transition in the control task. For example, you may want to change the operation mode of the system, which would terminate the active tasks at the same time. Then you need to define a task group. To define a task group, you create a new schematic for the task group and set the domain to “BP” and target to “parent”. And place the group of tasks on the schematic. Finally, define the “nodename” galaxy parameter as STRING as shown in Figure 9-15. Figure 9-15 is composed of two tasks shown at the bottom. Then, we register it as a new galaxy using “register galaxy” command. The schematic at the right side shows the registered task group. Then the fFSM can control the task group using the “nodename” of the task group.
9.6 Task-model design example

Now you are ready to design a neat example having two task groups and a control FSM in the BP domain. The control FSM task will select one task group between two task groups periodically. Each task group is composed of two tasks. One task has a Ramp block and the other task has a BPXMGraph. Two task groups have different “increment” parameter values in Ramp block. The control FSM task delivers the current value of one Ramp block to that of the other Ramp block when it switches the task group.

(1) We create a source task of the first task group. It consists of a ramp block, a type conversion (float to int), and a galaxy port. We define a period for periodic execution, a nodename to get control signals from FSM, and value1 and globalstate to exchange the current value to the other source task as galaxy parameters. Remind that these are needed for asynchronous interaction between the control fFSM graph and the SPDF graph (see Chapter 8 for detailed explanation).
Then, we set the step state of ramp block as ‘4’ and the value state as ‘value1’. The source blocks of two task groups will have different step values, so we can notice the different slopes of graph.

Let’s define the sink task of the first task group. It is composed of a type conversion block and a BPXMgraph block as shown below.

By registering those two tasks as galaxys in the menu, we make the first task group and add the nodename parameter ‘src1’ to be controlled by the control FSM.
Similarly make the second task group. The second source task is made to have the following galaxy parameters and the step value as ‘1’ in the ramp block as shown below.

![Image of Task Model](image1)

By registering the second source task as a galaxy in the menu and getting the sink task from the library, we make the second task group and add the nodename parameter as ‘src2’ to be controlled by the control FSM.

![Image of Task Model](image2)

Now make another source task that will generate a periodic event to the control FSM. Set the period to ‘5’ and the level of the const block to ‘1’ as shown below. The task sends the value ‘1’.

![Image of Task Model](image3)
The next step is to create the control FSM graph as shown below. Note that ‘processorId’ to be ‘0’ in FSMTaskTarget.

The control FSM gets a trigger signal and activates one task group selectively. When the ‘src1’ state become active, script ‘{suspend src1} {deliver ramp1 value1 ramp2 value2} {run src2}’ of ‘src1’ block is executed, which suspends the task group ‘src1’, delivers the value of ‘value1’ state of the ‘ramp1’ task to ‘value2’ state of the ‘ramp2’ task and resume the ‘src2’ task group. In turn, when the ‘src2’ state becomes active, script ‘{suspend src2} {deliver ramp2 value2 ramp1 value1} {run src1}’ of ‘src2’ block is executed, which suspends the task group ‘src2’, delivers the value of ‘value2’ state of the ‘ramp2’ task to ‘value1’ state of the ‘ramp1’ task, and resume the ‘src1’ task group. We can make the following schematic by putting them altogether.
When we execute the application ‘BP_FSM_0’ at $HOME/PEACE_SYSTEMS/CGC, we can acquire the following X graphs.