Chapter 8. FSM Model Specification

Author: Soonhoi Ha and Dohyung Kim

Finite state machine (FSM) model is useful to express the behavior of a control (or reactive) system like a bending machine, a cruise control system of a car, a traffic light signal and etc. Reactive systems collect input data from buttons, sensors and timers and send output data to actuators that change external environments - a coke and coins, speed of a car and blinked walk signal of traffic light signal.

Even though FSM is simple to use, its unstructuredness and state explosion problem due to system concurrency and memory prohibit FSM model from practical representation of complex control behavior. To describe complex control modules, the extensions should support various kinds of concurrency. Another desired feature is compositionality whether the complex module can be represented as a composition of simpler modules. Then, modules can be easily reused to construct a large system. Moreover, because of their complexity, it is desired to have a static analysis method to check ambiguous behavior. Finally for fast prototyping, automatic hardware (or software) synthesis is needed from the extended model. To meet these requirements, PeaCE devises a new extension of FSM model, called fFSM Model. It extends the expression capabilities by concurrency, hierarchy, and state variable.

This chapter explains how to draw an fFSM graph in PeaCE and how to use it in the system level modeling.

NOTE: An FSM graph is not used alone but used as a sub-task in another outer graph that provides input events to the FSM graph and displays the output. The outer domain can be either DE (for simulation) or BP (for code generation).

8.1 Basic FSM Definition

State transition graph is a directed connected graph composed of states and transitions. A state is drawn as a circle and a transition is a directed arc that connects two states. Each state indicates a unique memory location that describes specific status of the system and a state with bolder line is the initial system state. And a transition has a condition that must be fulfilled to execute actions in the form of 'condition/actions' above the line and make state transition from a source state to a destination state. State transition graph also has event definitions for input, output, internal event and variable state.

Here shows a simple state transition graph (Figure 8-1).
Figure 8-1 A simple state transition graph

There are two states – ‘start’ and ‘stop’ – and ‘start’ state is the initial state. And there are two transitions which have a condition ‘toggle==1’ and no action. It means that it start at ‘start’ state and each time it gets ‘toggle’ event, it changes the current state to the other state.

To draw a simple graph, you can start with opening a new schematic and select the FSM domain and default-FSM target. And a state icon is located in library-Peace-FSM of Lib tab as shown in Figure 8-2.

Figure 8-2 Locating state blocks

You select a state block and move it to new schematic twice (Figure 8-2). And then select one of state block and fill states( we will use the term “parameter” instead of “state” in the FSM domain to avoid the confusion between the behavioral state and the block parameter) of the state block. ‘name’ is the unique name for the state and ‘init’ indicate if the state is the initial state or not with ‘TRUE’ and ‘FALSE’ value (Figure 8-3). Other states will be discussed later.
After that, draw transition lines between two states. A transition has a source state and a destination state. If you move the mouse pointer close to the state, the green circle occurs and first click makes the state as a source state. And you move the mouse pointer to the middle of a source state and a destination state and click the button again to draw smooth curve. And third click on the green circle of the destination state makes a transition line between them (Figure 8-4).

Then select a transition line and fill the condition and actions (Figure 8-5).
A transition is associated with two parameters: condition and actions. If the specified condition is met, the actions in the transition are executed and the current state is changed from the source state to the destination state.

### Parameter syntax

- **Condition**: \( C \) condition expression. All events except output events can be used in the expression.
- **Action**: \([\{\text{variable\_name}=\text{expression}\}]\)*, Variable can be any event but an input event.

If you define ‘toggle’ event as shown in Figure 8-6, it completes the design of the simple state transition graph. Finally you enroll this FSM galaxy to library block using ‘Tool-Register Galaxy’ menu. Input events defined like Figure 8-6 are shown as input ports of FSM galaxy and output events as output ports. Because you define one ‘toggle’ input event, the library block will have one input port.
In the state transition graph, there exist four kinds of events: They are input event, output event, internal event and variable state. Input and output events define the external ports of an FSM super-block. An internal event defines an event for internal communication. Variable state defines a global storage. We explain the usage of internal event and variable state later. And each event has the definition of "name", "type" and "scope".

- Name: unique in the FSM domain and referred in condition and actions description
- Type: determine the event type that is one of {"PURE", "VALUE"}, default value is "PURE". "PURE" event indicates that occurrence of the event not the value of the event triggers the FSM machine. On the other hand, the value of an event is important for a "VALUE" type event.
- Scope: define the range of value that an event can have. The scope of a “PURE” event is {0,1}.

And events are defined as super-block (galaxy) parameters for the top level FSM domain as shown in Figure 8-6.

- InputNames : names of input events
- InputTypes : types of input events
- InputScopes : range of input event values
- OutputNames : names of output events
- OutputTypes : types of output events
- OutputScopes : range of output event values
- InternalNames : names of internal events
- InternalTypes : types of internal events
- InternalScopes : range of internal event values
- VariableNames : names of variable states
- VariableTypes : types of variable states
- VariableScopes : range of variable state values
- VariableInits : initial values of variable states

And the syntax of those parameters becomes

- Name definition: [\{name\}]*
- Type definition: [\{PURE\}|\{VALUE\}]*
- Scope definition: [\{0,1\}|\{min value, max value\}] *

We make the following rules to make each event name unique in FSM domain because each event must be able to be referred uniquely in conditions and actions.

- Each event name must be unique within the sub-graph. A same name can be used in multiple subgraphs.
- Each FSM galaxy name must be unique globally.
- In a child FSM, an input event name must be one of input events or internal events of the parent FSM in the hierarchical FSM graph. In a child FSM, an output event name must be one of output events or internal events of the parent FSM.

The example shows an example FSM graph and galaxy parameters located at the left bottom corner. If you look at the galaxy parameters in Figure 8-7, it helps you to understand the syntax of event descriptions.
To describe complex control modules, an FSM extension should support concurrency. Another desired feature is compositionality whether the complex module can be represented as a composition of simpler modules. Then, modules can be easily reused to construct a large system. Moreover, because of their complexity, subtle design errors are difficult to find and the equivalence check of an implementation is not easy to achieve. Therefore, it is desired to have a static analysis method to check ambiguous behavior. Finally, for fast prototyping, automatic hardware (or software) synthesis is needed from the extended model.

In this chapter, we explain how those requirements can be satisfied with a proposed FSM model. The proposed FSM model is called flexible FSM model meaning that there are more than one way of expressing concurrency. It extends the expression capabilities by concurrency, hierarchy, and state variable while it maintains the formal property of the base FSM model. Because of formality and the structured nature of fFSM model, we can apply a static analysis method to find ambiguous behavior and synthesize software/hardware automatically. Moreover, it has a special syntax to express memory in a compact form. In a formal FSM model, if a system has a memory element, the number of states explodes as the number of possible values does. We introduce a new concept of “variable states” to overcome this difficulty.
8.2.1 Concurrency

fFSM expresses concurrency as illustrated in Figure 8-8. Bold lined states indicate initial states and a dotted-line divides two concurrent FSMs. Two states A and C are concurrently active at the same time. And transitions may occur simultaneously if both inputs “a” and “c” are active. No direct transition is allowed between concurrent states. However, two concurrent states can affect each other by exchanging internal events to be explained below. This constraint is for formality and simplicity.

![Figure 8-8 Concurrency expression](image)

The example shows an fFSM example showing concurrency in PeaCE. Each concurrent FSM has its own initial state and is not connected to each other.

![Figure 8-9 Concurrent FSMs](image)

8.2.2 Hierarchy

Hierarchy means that a state can have another FSM inside. When the outer state is active, the internal FSM becomes active and initial state is ready to execute. And as soon as the outer state is inactive, the internal FSM also becomes inactive. We avoid the use of inter-hierarchy transitions, unlike Statechart. Figure 8-10 shows an example fFSM graph that shows hierarchy.
If the current state is A and input event “a” occurs, the next state becomes state B and internal FSM becomes active, which makes the state C active. When input event “b” occurs, the internal FSM exits to go to state A. But it has an ambiguity problem. Suppose that the current states are B and C and input events “b” and “c” occur simultaneously. Which transition should be executed first? There are a number of methods to determine the priority between transitions. We assign an outer transition higher priority in PeaCE.
To draw a hierarchical FSM, first you design a FSM subgraph like Figure 8-11 and save it. And at the parent FSM, fill ‘internal FSM’ state in the state block as the file name which you just created as shown in Figure 8-12. And if you want to see the internal FSM of a state, just double-click the mouse button on the state in which you are interested.

### 8.2.3 Internal event

Internal event is used for communication between concurrent states and/or hierarchical states. Because it may cause ambiguity, we adopt the *delta-delay* model. All input events exist during delta-delay phase only. If there is any new internal event, it becomes valid at the next phase after delta-delay and all previous events set to be invalid.

![Figure 8-12 Parent FSM](image)

Figure 8-12 Parent FSM

**Figure 8-13 Interval event expression**

![Figure 8-13 Interval event expression](image)

Figure 8-13 shows how we can avoid ambiguity by applying the delta-delay model of execution. If a current state is *start* and input event “a” and “e” occur, an internal event “b” becomes valid and the current state is changed to state *A*. We cannot determine which transition is executed if the current state is *A* and events “e” and “b” are active. But we assume that event “e” is deactivated after delta-delay and the next state will become *B*.

If you want to use an internal event between concurrent FSMs, first you define an internal event in the galaxy parameter. And then if you want to send the signal to other FSM, you can write an action script like ‘internal_event = value’. And the receiving FSM uses the internal event in the condition definition. Let’s see an
example (Figure 8-14). We define ‘tostart’ internal event as PURE type. And at ‘reset’ state, the ‘tostart’ internal event is sent to ‘stop’ state, which makes a state transition to ‘start’ state.

If you want to use an internal event across hierarchical FSMs, you define an internal event at the parent FSM. And at the child FSM, you define the internal event either as the input event if you want to receive the signal from the parent, or as output event if you want to send the signal to the parent. To avoid ambiguous transition, you would better to have only one writer for the internal event.

In the example, ‘stop’ state has a child FSM (Figure 8-15). And the parent FSM defines ‘tostart’ internal event. And the child FSM defines ‘tostart’ output event to send the signal to the parent FSM. So at the ‘stop’ state, if the FSM receives ‘reset’ event, the parent FSM makes a state transition to ‘start’ state by the ‘tostart’ internal event.
8.2.4 Variable state

A variable state provides a memory to FSM model. It is considered as a local variable and expressed as a concurrent FSM. The usage of a variable state is similar to an internal event. While an internal event disappears after delta-delay, the variable state always exists. A variable state is formally equivalent to a concurrent FSM except that it can be examined and updated by other concurrent FSMs. If it is used as a condition, it is called a state reference. If it is used as an output, it implies a state transition.
Figure 8-16 implements a time-out FSM using a variable state. An input event “time” is an external clock and an output signal ‘timeout’ is the result output which indicates that actual timeout occurs. If an outer block wants to set a timeout, it sends “start” signal to the timeout FSM with a required timeout value. After the timeout FSM gets the input, it sets the variable state “remain” as the value of the input event “start”, changes the state to “wait”. It decreases the “remain” state at each “time” input until the “remain” value becomes zero. Then a transition from “wait” to “init” occurs and a “timeout” signal is supplied to the outer block. If we specify this example by using previous FSM models, we must create different FSMs with different time-out values.

Figure 8-17 shows the timeout example using variable states. First we define ‘remain’ variable state to store remaining time that has a scope from 0 to 10 and is initialized by value ‘0’. And if the upper FSM sends ‘timeset’ internal event, the lower FSM receives the event and stores it in the ‘remain’ variable state. And each time the lower FSM gets ‘time’ input event, it decreases the ‘remain’ variable state until the ‘remain’ variable state become 0. If the ‘remain’ variable state becomes 0, then it sends ‘timeout’ internal event to the upper FSM.

8.3 FSM Simulation

FSM model cannot be executed alone but is simulated with DE domain. After registering an FSM galaxy, you use the FSM block in the DE domain to make it a hierarchical block diagram. The DE blocks provides input events to the FSM block and receives the output samples. To understand how it works, open the ReflexGame demo in
PeaCE as illustrated as Figure 8-18. Note that this demo can also be found in Ptolemy class but with different specification method.

Run the demo, then you will see a nice user interface come up with three buttons inside. The TclScript block at the input of the FSM graph reads a Tcl script (specified as a block parameter) to construct such a nice user interface. If you click a button, the block generates an input to the FSM block. The Synchronize block generates clock events to the FSM. The TclScript block at the output port displays the result. It is a neat game to measure your reaction speed!

If you look inside the game_FSM block, you will see a concurrent and hierarchical fFSM graph. Even though the internal behavior of the game_FSM block is rather complicated, it can be easily represented in fFSM model using concurrency, hierarchy, and variable states. The target of the FSM graph is 'default-FSM' that is a simulation target with no settable parameter.

Figure 8-19 shows top-level fFSM graph specification of the reflex game demo. As shown in the left table, you can define input events, output events, internal events and variable states. The upper fFSM graph of the right schematic controls ON/OFF state of the game and the lower fFSM graph implements time-out fFSM using a variable state. Two fFSM graphs in Figure 8-19 are simultaneously active by concurrency.

In the GameOn state, an hierarchical fFSM graph is defined to handle interactions of the game which is illustrated in Figure 8-20. Events used in the child fFSM graph can be inherited from the parent fFSM graph or be newly defined for its own usage. When they are inherited, you write only event names of the parent fFSM graph. Otherwise, you should define all properties of the event (Ex. 'rand' variable state). The upper fFSM graph handles interactions and the lower fFSM graph generates random time delay to test reaction speed.
Figure 8-19 Top level fFSM graph specification of reflex game demo

Figure 8-20 Child fFSM graph specification in the GameOn state
8.4 Talking with Computation Module

While an FSM graph specifies the control behavior of a system, we use SPDF subgraphs or DE blocks to specify the computation module of the system. For example, in Figure 8-21, Ramp block in the DE domain models the computation module of the system. The Ramp block generates a sequence of increasing numbers to make a counter. The FSM graph receives the user input and commands the Ramp block to reset the counter value or to stop counting. So interaction mechanism between the FSM graph and the computation module is needed. This section explains how to make an FSM graph talk with computation module. For simplicity, we assume that a DE block represents a computation module. Note that the DE computation block encapsulates an SPDF graph inside to model a complicated computation module of the system.

![Figure 8-21 Simple Counter Demo](image)

Interaction between a DE block(function module) and an FSM graph(control module) is divided into two categories: one is **synchronous interaction** and the other is **asynchronous interaction**.

In a synchronous interaction, a control module and a computation module communicate with each other by exchanging data samples through an arc between two modules. The computation module may send a sample to set a flag of a control register and the controller becomes active on the arrival of the flag. Or, the control module may send a data sample that activates the computation module to process the sample. The Simple Counter Demo does not possess this kind of interaction.

Using asynchronous interactions, a control module plays the role of a supervisory module to manage the state of a computation module. We define three states of a computation module by its current activity: **active**, **suspend** and **stop** as illustrated at Figure 8-22, where a computation module is represented as an SPDF graph. If there is no synchronous input interaction from the FSM module to a computation module, the computation module goes into the active state from the start by default. When the control module enters into a certain state, it may want to suspend the computation module and resume it later. When the computation module goes into the suspend state, it stops its execution and just discards the incoming samples from the outside environment.
Another situation of asynchronous interaction occurs when the control module wants to change parameters of computation module asynchronously. Suppose a computation module plays an encoded audio. If the user wants to lower the volume, that action is delivered to the control module and finally to the computation module by asynchronously changing the "gain" parameter of the internal dataflow node that amplifies the sound samples as shown in Figure 8-23.

Though there are many possible methods to describe asynchronous interactions, we choose to use a "script" language inside a state node in an FSM. For asynchronous interaction, we use a script language inside a state node in an FSM. Each FSM state has the ‘script’ parameter. If the ‘script’ parameter is defined, the script is executed after the state is entered.

To use a script in the state, you must specify a node name of a destination computation module first, which make possible to indicate a specific node in a script. In DE domain, every block except wormholes has the ‘nodename’ state as shown in the left of Figure 8-24.
Figure 8-24 Using a script in the state

But if the computation block is a wormhole that contains a CGC graph, you must add a ‘nodename’ state as string type to the super-block(or Galaxy). Then you can specify the block name.

After specifying the unique name ‘ramp’, you can define the script in the state. Table 8-1 shows the syntax of the scripts. The first three manage the states of the computation modules and the last script describes the asynchronous parameter updates of the named block. The difference between “suspend” and “stop” command is that the suspended block resumes its action starting from where it was suspended while the stopped block resumes from the beginning when resumed.

Table 8-1 Syntax of script language

<table>
<thead>
<tr>
<th>Scripts</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run n_name</td>
<td>Resume the n_name block</td>
</tr>
<tr>
<td>Suspend n_name</td>
<td>Suspend the n_name block</td>
</tr>
<tr>
<td>stop n_name</td>
<td>Stop the n_name block.</td>
</tr>
<tr>
<td>Set n_name parameter value</td>
<td>Update the parameter with value in the n_name block</td>
</tr>
</tbody>
</table>

Figure 8-25 shows an example of script language usage. In the ‘start’ state, ‘run ramp’ is specified. And in the ‘stop’ state, ‘stop ramp’ is used. When each ‘toggle’ input data is arrived, a state transition is occurred and each script of destination state is called. So we can change the run time status of the ramp block in the DE domain. In
run status, the Ramp block increases the counter value. In stop status, the Ramp block initializes the counter value and is stopped. You can use ‘suspend’ script as the same way.

![Figure 8-25 Script language usage example](image)

In the start state, there is a hierarchical FSM, called ‘resetFSM’, as shown in Figure 8-26.

![Figure 8-26 resetFSM](image)

The reset state has ‘set ramp value 0’ script which means that it change ‘value’ state of ‘ramp’ node to ‘0’. So each time FSM receives ‘reset’ input data at ‘start’ state, FSM send the signal to reset the counter value.
8.4.1 Simple Counter example

In this section, let’s make the Simple counter demo as shown in Figure 8-21. Upper DE blocks models a increasing counter and the TclScript block creates a user interface with ‘toggle’ and ‘reset’ buttons to generate input events to the FSM block. The FSM block gets two inputs and supervises the execution of Ramp block.

(1) First you open a new facet and name it as you like. Set the domain to DE and the target to “default-DE”.

(2) Draw the block diagram using the library blocks. You can find the Clock block, the TclScript, and the Ramp block from “/library/peace/DE/Src”, the TkShowValue block from “library/peace/DE/Sink”. And then you set the ‘nodename’ state of the Ramp block as ‘ramp’ for asynchronous interaction with the FSM graph. And make as below the ‘button.tcl’ file which is a tcl script file that creates two buttons. The file is set to the ‘tcl_file’ state of the TclScript block.

```tcl
set s $ptkControlPanel.SimpleCounter
if {![winfo exists $s]} {
    toplevel $s
    wm title $s "FSM Demo"
    wm iconname $s "FSM Demo"
} else {
    wm deiconify $s
    raise $s
}
set buttons $s.buttons
catch {destroy $buttons}
frame $buttons -relief groove -bd 3
pack $buttons -padx 3 -pady 3 -expand 1 -fill both
label $buttons.label -text "You can push:"
pack $buttons.label -side top
button $buttons.toggle -text toggle -bd 4 -padx 3 -pady 3 
    -command "setOutput_$starID 1 1"
button $buttons.reset -text reset -bd 4 -padx 3 -pady 3
    -command "setOutput_$starID 2 1"
pack $buttons.toggle $buttons.reset -side left
    -expand 1 -fill both -padx 3 -pady 3
tkwait visibility $buttons
```

(3) Create the FSM subgraph as explained above. Figure 8-27 shows the resultant FSM graph and the state parameters. Note that the “start” state nests an internal FSM graph for “reset” event handling.
Figure 8-27 Parent FSM and related parameters

The internal FSM and the associated state parameters are displayed in Figure 8-28.
If you run the design, you can see the two buttons and the run panel that contains a display box of increasing numbers (Figure 8-29). You can change the execution by pushing buttons.
8.4.2 Add a SPDF super-block

Now we add a SPDF super-block as another computation module in the system. You will see that if you want to change the state value of CGC galaxy, you must do a few more things than specifying the “nodename” state of the block.

![Figure 8-30 Adding a buffer super-block](image)

We add a new buffer super-block (a CGC graph inside) which buffers the counter value, multiplies a gain value and sends the result data to display as shown in Figure 8-30. And we suppose that we want to change the gain value.

1. First we add ‘nodename’ state to the super-block.

2. Because we cannot directly access the gain value of the block, we must make a path from the super-block to the block. So we add a ‘globalstate’ state as a string array. The state indicates which states are used for the parameter update. And we add a super-block parameter that has the same type of the gain value. We name it as ‘gainvalue’ and the type is float.

3. Next we set the ‘globalstate’ of the super-block and the ‘gain’ state of the Gain block to ‘gainvalue’. Left Figure shows the super-block parameters and right Figure the block parameter (Figure 8-31).
And then if you change scripts of FSM state as follows, you can see that the gain value for multiplication is changed each time you push 'toggle' button (Figure 8-32).

Figure 8-32 Changing scripts of a FSM state
8.5 Code Generation

This section shows how to synthesize a C code or a VHDL code from the FSM graph. To generate a C code ‘FSMCGCTarget’ should be taken: It generates C code from the fFSM graph, compiles, and executes it. For VHDL code generation ‘FSMVHDLTarget’ or ‘FSMSimVHDLTarget’ should be taken. The latter generates a VHDL code from the fFSM graph and interface codes for Synopsis VHDL simulator. The target parameters for each target are summarized in the sub-sections.

Note: current version of PeaCE only support C code generation in ‘FSMCGCTarget’ and VHDL code generation in ‘FSMVHDLTarget’. But these two targets cannot be cosimulated with other targets. Instead, current version of PeaCE supports another C code generation in ‘FSMTaskTarget’ that is simulated with BP domain. It will be explained in Section 9.

8.5.1 FSMCGCTarget

The target parameters of ‘FSMCGCTarget’ are listed below with brief explanation.

- host (default “”): host name which compiles and executes the code
- directory (default “$HOME/PTOLEMY_SYSTEMS/FSM”): directory where the file locates
- file (default “”): name of the file to save the generated code
- printStruct? (default “YES”): print the fFSM structure text file if “YES”
- display? (default “YES”): display the generated C code if “YES”
- compile? (default “YES”): compile the generated C code if “YES”
- run? (default “YES”): run the executable after compilation if “YES”
- compileCommand (default “g++”): indicate the compiler command PeaCE uses
- compileOptions (default “”): indicate the compiler options
- linkOptions : (default “”): indicate the link options during compilation

Code b) shows the C code structure generated from an FSM subgraph. Each concurrent state makes a ‘switch() { case }’ phase, each transition makes an ‘if then else’ sentence, and each action in transition constructs a C sentence. We distinguish normal transitions and internal transitions, which makes more efficient execution. Because of delta-delay execution semantics, an input event is only valid during the first phase and any internal event will be valid from the second phase. So first we check normal transitions and if there exists any internal event, we iterate to check internal transitions until there is no more events as code a).

read input events;
loop init code;
process normal transitions of fFSMs;
transition wrapup code;
while(if there is any internal event) {
    process internal transitions of fFSMs;
    transition wrapup code;
}
send output events
a) fFSM main loop code
switch (for each concurrent FSM) {
    case for each state:
        if (for each condition) {
            do_actions;
        } else if (another condition) {
            ...
        } else {
            execute internal fFSM; // if exists
        }
    case ...
}

b) Each FSM code

8.5.2 FSMVHDLTarget

- host (default “”) : host name which compiles and executes the code
- directory (default “$HOME/PTOLEMY_SYSTEMS/FSM”) : directory where the file locates
- file (default “”) : name of the file to save the generated code
- printStruct? (default “YES”) : print the fFSM structure text file if “YES”
- display? (default “YES”) : display the generated C code if “YES”
- clockPeriod(ns) (default 20) : indicate the default clock period for VHDL simulation
- Environ (default “/usr/tools/synopsis”) : the location at which synopsis tools is installed
- Architecture (default “sparcOS5”) : the host architecture on which the tools are executed
- Analyze (default “YES”) : indicate whether analyze process will be executed or not

8.5.3 FSMSimVHDLTarget

- host (default “”) : host name which compiles and executes the code
- directory (default “$HOME/PTOLEMY_SYSTEMS/FSM”) : directory where the file locates
- file (default “”) : name of the file to save the generated code
- printStruct? (default “YES”) : print the fFSM structure text file if “YES”
- display? (default “YES”) : display the generated C file if “YES”
- clockPeriod(ns) (default 20) : indicate the default clock period for VHDL simulation
- Environ (default “/usr/tools/synopsis”) : the location at which synopsis tools is installed
- Architecture (default “sparcOS5”) : the host architecture on which the tools are executed
- Analyze (default “YES”) : indicate whether analyze process will be executed or not
- Simulation (default “YES”) : run Synopsis VHDL Simulator if “YES”
Figure 8-33 shows the main fFSM block diagram. Because main fFSM block consists of the network of concurrent FSM blocks, it has the definition of input and output events for external interface. And there exist concurrent fFSMs and internal event logic between fFSMs. Below code shows the basic VHDL structure generated from an FSM.

```vhdl
{FSM name}_SEQ_LOGIC:
  process(clk, reset)
  begin
    if ( reset='1' ) then
      current state <= initial state;
    elseif rising_edge(clk) then
      ## if the FSM has a parent FSM ##
      if (parent FSM's current state=parent state) then
        current state <= next state;
      else
        current state <= idle;
      endif;
      ## if the FSM does not have a parent FSM ###
      current state <= next state;
    endif;
  end process {FSM name}_SEQ_LOGIC;

{FSM name}_COM_LOGIC:
  process(current state, [{Input event}], ...)
  begin
    case (current state) is
      -- for each state
      when {state name} =>
        -- for each transition
        if ({condition}) then
```
next state <= next state name;
-- for each action
output event <= expression;
....
elseif .. other conditions
endif
when {state name} => other state
when others =>
current state <= initial state;
end process {FSM name}_COM_LOGIC;