Chapter 7

# FINITE STATE MACHINE KERNEL IN SYSTEMC

Constructing a Finite State Machine (FSM) model in SystemC is possible with current modeling constructs of SystemC. This means that the existing SystemC can effectively provide means of constructing an FSM model. Some may argue that given a Discrete-Event simulation kernel, there is no need to add a Finite State Machine (FSM) kernel for SystemC. However, with the vision of a truly heterogeneous modeling environment in SystemC, the need for such an inclusion is arguable. Furthermore, with hierarchy in mind, the separation of an FSM kernel may result in increased simulation efficiency.

The kernel is an encapsulation of the *SC_METHOD()* processes along with several member functions to describe an FSM model. In a way it is not necessarily an alternate kernel. However, this encapsulation serves as a step towards isolating the FSM kernel completely from the execution of the DE kernel. At this moment every FSM block executes in one simulation cycle as per our definition of a period for an FSM node. This results in an untimed model of the FSM that will be extended to support timed models in further development. We envision support for timed and untimed models for relevant Models of Computation. Unfortunately, the implementation of signals using *sc_event* types makes it difficult to diverge from the Evaluate-Update semantics. We are currently investigating reconstructing *sc_signal*s such that they can be interpreted by the MoC within which they are employed. This extends the possibility of all MoCs being either timed or untimed. The revamp of the event management is still under investigation.

The Finite State Machine Model of Computation has the following properties:

- A set of states
- A start state.
- An input alphabet and
- A transition function that maps the current state to its next state.

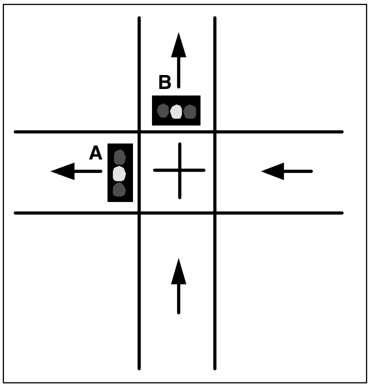


*Figure 7.1.* FSM Traffic Light Example [4]

FSMs are generally represented in the form of graphs with nodes and transitions connecting the nodes with some conditions on the transitions. Figure 7.1 shows a diagram of a two traffic light system and Figure 7.2 illustrates a Finite State Machine controller for this system.

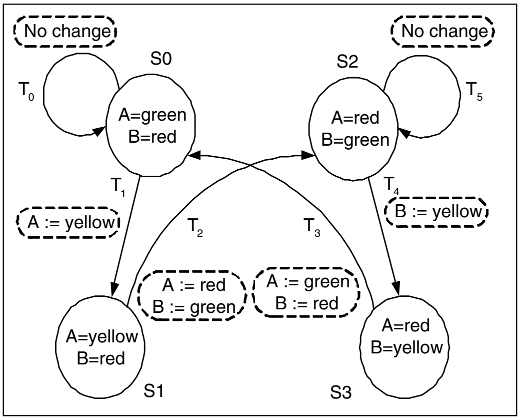


*Figure 7.2.* FSM Traffic Light Controller Example [4]

The two traffic lights are represented by A and B and the set of states contains S0, S1, S2 and S3. The transitions are represented by the

*Table 7.1.* Example of *map<...>* data structure

| Key | Value |
|---|---|
| toplevel.state.state0 | 0xf000001 |
| toplevel.state.state1 | 0xf000011 |
| toplevel.state.state2 | 0xf000101 |
| toplevel.state.state3 | 0xf001001 |
| toplevel.state.state4 | 0xf100001 |

arrows and the action associated with the transition is marked in the dotted ellipses. Suppose S0 is the initial state. Then a transition to S1 causes traffic light A to change from green to yellow and B to remain at red. This is a simple controller example, but FSMs can be extensive and large in size. We do not discuss the specifics of Moore and Mealey machines since FSMs serve as pedestals to most engineering. However, we refer the reader to [10] for additional reference and continue our discussion to the implementation details of the FSM kernel in SystemC.

# 1. Implementation Details

## 1.1 Data Structure

The FSM kernel's data structure implements a *map<...>* object from the C++ STL. A map object is simply a list of pairs consisting of a key and a value. FSM uses a *string* and a pointer to the *SC_METHOD()* process via the *sc_method_process* class as shown in Listing 7.1 as a pair entry in the *map<...>* object. For illustration purposes Table 7.1 displays the pairs inserted in the data structure. The addresses for the values are made up. The keys are of type *string* and the value is an address to an object of type *sc_method_handle*. The key field is used when searching this *map<...>* object for a particular string and if a pair entry is found with the corresponding search string, then the value is returned.

The *FSMReceiver* class once again derives from the *baseReceiver* class. The *baseReceiver* holds the type of the receiver that is derived from it. Besides the *fsmlist* private data member, there is an *id* and a string type variable called *currentState*. This *currentState* variable preserves the current state that the simulation has reached for the FSM. This may not seem necessary in a pure FSM model. However, in heterogeneous models, a particular state in the FSM may resume another MoC and return back to the FSM requiring the preservation of the last state that it had executed. The member functions of class *FSMReceiver* are standard functions used to insert elements into the *fsmlist* and retrieve a particu-

*Listing 7.1.* class *FSMReceiver*

```
class FSMReceiver : public baseReceiver {
  private:
    map<string, sc_method_handle> * fsmlist;
    string id;
    string currentState;

  public:
    FSMReceiver(const string & s);
    ~FSMReceiver();

    void insert(const string &s, sc_method_handle h);
    sc_method_handle find(const string &s);
    bool myid(const string &s);

    void setState(const string & s);
    string & getState();

    void fsm_execute();
    sc_method_handle register_fsm_method( const char* name,
                  SC_ENTRY_FUNC entry_fn,
                  sc_module* module );
};
```

*Table 7.2.* Some Member functions of class *FSMReceiver*

| Member Function | Purpose |
|---|---|
| insert(...) | Inserts a pair into *fsmlist* |
| find(...) | Returns a pointer to the FSM process if the string associated with the name of the process is found |
| setState(...) | Set the *currentState* with the *string* argument that is passed |
| getState() | Returns the *currentState* |
| fsm_execute() | Execute the FSM model |
| register_fsm_method(...) | Called from a C macro that registers a *sc_method_process* object as an FSM process |

lar *sc_method_handle* by supplying a *string*. The *register_fsm_method(...)* function is invoked by the C macro defined by *SC_FSM_METHOD(...)*. Listing 7.2 shows the module construction for *SC_FSM_METHOD(...)*. Table 7.2 lists some of the important member functions of class *FSMReceiver* and their use.

The *fsm_execute()* member function is responsible for initiating the execution of the FSM model. The simulation begins at the initial state, which is set by the modeler. The modeler can do this by using the *setState(...)* member function to designate one of the states as an initial state. To schedule an FSM process to execute, the *setState(...)* member function is employed. A schedule for an FSM process means changing the *currentState* variable to reflect the name of the next process to

*Listing 7.2.* Macros used to register FSM processes

```
//SC_FSM_METHOD...
#define SC_FSM_METHOD(func, mod)
    fsm_declare_method_process( func ## _handle,
                                #func,
                                SC_CURRENT_USER_MODULE,
                                func, mod )
//fsm_declare_method_process
#define fsm_declare_method_process(handle, name, module_tag,
    func, mod )
sc_method_handle handle;
{
    SC_DECL_HELPER_STRUCT( module_tag, func );
    handle = mod->register_fsm_method( name,
                  SC_MAKE_FUNC_PTR( module_tag, func ), this );
                  sc_module::sensitive << handle;
    sc_module::sensitive_pos << handle;
    sc_module::sensitive_neg << handle;
}
```

execute. This function sets the *currentState* to the argument that is passed into that function and with the next execution of the FSM; the process with that *string* name is executed. Every time *fsm_execute()* runs, the *currentState* of the FSM model is retrieved and a search is done on the data structure. The *sc_method_process* pointer is returned if an entry is found and then executed. The key entries in Table 7.1 are *sc_method_process* object names. The naming convention preserves SystemC naming conventions by adding a dot between module names. This naming convention is discussed in Chapter 5 with an example. For the FSM kernel the *FSMReceiver* is the most integral class. The remainder of the classes implemented to support the FSM kernel are shown in Listing 7.3. The *FSMkernel* class is responsible for allowing multiple FSM models to simulate together.

Channels and ports specific for the FSM MoC are included with the declarations shown in Listing 7.4. Since there is no specific communication functionality for the FSM MoC, the *FSMport* and *FSMchannel* classes inherit from *sc_moc_port* and *sc_moc_channel* respectively. They exhibit the same behavior as their base classes. The source listing for the base classes is shown in Chapter 4.

## 2. Example of Traffic Light Controller Model using FSM Kernel in SystemC

To further illustrate our FSM kernel we present an FSM traffic light controller example. Figure 7.2 describes the state diagram of this simple example. Listing 7.5 shows the *SC_MODULE(state)* definition along with its respective entry functions. The entry functions are *state0*, *Listing 7.3.* FSMkernel and FSMnode class definition

```
class FSMkernel {

  private:
    vector<FSMReceiver*> * fsms;

  public:
    FSMkernel();
    ~FSMkernel();
    void insert(FSMReceiver * f);
    FSMReceiver* find_fsm(const string & id);
    void fsm_crunch();
};

class FSMnode {

  private:
    sc_method_handle handle;
    string name;

  public:
    FSMnode();
    ~FSMnode();
    void set(const string &s, sc_method_handle h);

};
```

*Listing 7.4.* Ports and Channels for FSM MoC

```
template <class T>
  class FSMport : public sc_moc_port<T> {};
template <class T>
  class FSMchannel : public sc_moc_channel<T> {};
```

*state1*, *state2* and *state3* representing the states S0, S1, S2, and S3 respectively. Each of these entry functions are bound to an *SC_METHOD()* process via the *SC_FSM_METHOD()* macro. Registration of the entry functions as FSM processes is performed via this macro. The constructor of *SC_MODULE(...)* remains the same as existing SystemC syntax with the use of *SC_CTOR(...)*. Notice the initial state of the FSM is set within the constructor with *fsm_model→setState("toplevel.state.state0")*. We preserve the naming conventions of SystemC to target the FSM process for execution. However, this requires knowledge of the encapsulating process as well since the naming convention of SystemC concatenates the names by taking the module name, adding a dot character at the end, followed by appending the entry function name. The hierarchy of the module is preserved by preceding with the name of the toplevel module name as shown by *toplevel.state.state0*.

Two instances of *light* are present where *A* represents traffic light A and *B* represents traffic light B. The colors are enumerated by *enum*

*Listing 7.5.* Module Definition of 'state' in Traffic Light Controller Model

```
SC_MODULE(state) {

  light A;
  light B;
  int random;

  void state0() {
    random = rand();
    cout << "--------------------------------------------------------"
         << endl;
    cout << "S0 -- Random value = " << random << endl;
    A = GREEN;
    B = RED;
    printLight(A, B);
    if (random % 2 == 0)
      fsm_model->setState("toplevel.state.state1");
  };

  void state1() {
    random = rand();
    cout << "--------------------------------------------------------"
         << endl;
    cout << "S1 -- Random value = " << random << endl;
    A = YELLOW;
    B = RED;
    printLight(A, B);
    if (random % 2 == 0)
      fsm_model->setState("toplevel.state.state2");
  };

  void state2() {
    random = rand();
    cout << "--------------------------------------------------------"
         << endl;
    cout << "S2 -- Random value = " << random << endl;
    A = RED;
    B = GREEN;
    printLight(A, B);
    if (random % 2 == 0)
      fsm_model->setState("toplevel.state.state3");
  };

  void state3() {
    random = rand();
    cout << "--------------------------------------------------------"
         << endl;
    cout << "S3 -- Random value = " << random << endl;
    A = RED;
    B = YELLOW;
    printLight(A, B);
    if (random % 2 == 0)
      fsm_model->setState("toplevel.state.state0");
  };

  SC_CTOR(state) {
    fsm_model->setState("toplevel.state.state0");
    SC_FSM_METHOD(state0, fsm_model);
    SC_FSM_METHOD(state1, fsm_model);
    SC_FSM_METHOD(state2, fsm_model);
    SC_FSM_METHOD(state3, fsm_model);
  };
};
```

*Listing 7.6.* Module Definition of *top* in Traffic Light Controller Model

```
1  SC_MODULE(top) {
2
3    state *s1;
4    void entry() {
5
6      while(true) {
7        fsm_trigger();
8        wait();
9      }
10   };
11
12   SC_CTOR(top) {
13     s1 = new state("state");
14     SC_THREAD(entry) {
15     };
16   };
17 };
18
19
20 int main() {
21   fsm_model = new FSMReceiver("fsm1");
22   fsm_kernel.insert(fsm_model);
23
24   top tp("toplevel");
25   sc_start(-1);
26
27   return 0;
28 }
```

*light RED=1, YELLOW= 2, GREEN=3;*. The values for the traffic lights are set followed by a execution of a global function *printLight(...)* that displays status of the lights. Full source is not presented, but is available at our website [36]. The next state is set by using the *setState(...)* function call, which describes the transition presented in Figure 7.2. However, since C++ is a sequential programming language, implementing non-determinism for transitions $T_0$ and $T_5$ requires the use of randomization. A simple policy where if the randomly generated number is not zero then the transitions $T_0$ or $T_5$ are traversed depending on the current state of the FSM is implemented.

The *top* module is a regular *SC_THREAD()* process with an infinite loop and a single suspension statement. This is to allow the FSM to run infinitely, as expected behavior of a traffic light controller. The model progresses after every cycle due to the *wait(...)* statement. Similar to the SDF MoC implementation, a call to *fsm_trigger(...)* is mandatory to indicate the execution of the FSM kernel. Listing 7.6 shows the module definition for *top* along with definition of *sc_main(...)*.

A global object of type *FSMkernel* holds the *fsm_model* that is to execute. The simulation starts using the *sc_start(...)* function call [Listing 7.6, Line 25].