A MODEL LIBRARY FOR FINITE STATE MACHINES IN CADMIUM

Amitav Shaw
Arshpreet Singh
Gabriel Wainer

Dept. of Systems and Computer Engineering
Carleton University
Ottawa, Canada
{amitavshaw, arshpreetsingharshpr}@cmail.carleton.ca, gwainer@sce.carleton.ca

ABSTRACT

The Discrete Event System Specification (DEVS) formalism has been employed to provide a common basis for discrete-event modeling and simulation. Here, we describe a DEVS library for finite state machines built using Cadmium, a DEVS Modeling and Simulation tool, and we discuss different case studies of Moore and Mealy Machines. We build upon the concept that each state can be represented as a DEVS atomic model instance which can be coupled with other instances to form a coupled model representing a Finite State Machine.

Keywords: Finite State Machine, DEVS, Cadmium, Moore machine, Mealy machine

1 INTRODUCTION

Discrete Event Systems Specifications (DEVS) is a formalism that has gained attention in the scientific and engineering research realm (Zeigler, Praehofer and Kim 2000). DEVS allows describing the system behavior at various hierarchical levels. At the base level, atomic model describes the generic behavior of the discrete event system by encoding how it reacts to external events, makes a transition and how it generates output. This allows for building complex models by connecting atomic models in a hierarchical manner to build a network of coupled models. At the highest level, a coupled model describes the system.

The Cadmium tool (Vicino et al. 2019) allows the users defining DEVS models, and providing libraries is useful to the end users. By including libraries of other formalisms such as Finite State Machine (FSM), Petri Nets, Timed Automata, Cellular Automata etc., would allow users defining complex multicomponent model with ease. Here, we are interested in experimenting with Finite State Machines, which can be used to represent complex systems in different domains (engineering, computer science, linguistics, logic, or biology).

Here we present the definition and implementation of an FSM library in Cadmium. We discuss how to define Moore and Mealy Machines and a method to describe an atomic model which serves as a building block for FSM's based on Moore and Mealy machine design methodologies. The idea is to create a generic atomic model which comprises of state variables to manipulate the states, manage state transitions, and handle the events and outputs. Two such atomic models are built for satisfying Mealy and Moore Machine paradigm. We make it general by exposing the state variables providing callback functions for event and transition handling. These callbacks are supposed to be implemented by the applications which will be developed by creating coupled models on top of these atomic models. Using the proposed implementation, the atomic model description, the coupled model description, the instantiation of atomic models and implementations of the callbacks should be enough to build any complex finite state machine.

The rest of the paper is organized as follows. Section II presents few of related work. Section III presents the model description. Section IV presents the implementation of the model. Section V presents the applications built using the models and discusses the simulation results.

2 RELATED WORK

In (Garredu et al. 2013) the authors propose an FSM Model to DEVS Model transformation by doing Model to Model transformation using a tool known as MetaDEVS. They defined an empty atomic model with few basic state variables and generic transition functions from an FSM model. There are limitations as pointed out by the authors. Their implementation cannot generate code from a source model. Code is required to run models in tools like Cadmium. MetaDEVS converts an existing model to DEVS, and it works when there is an existing model to be reused as DEVS. It is not feasible when an FSM model needs to be created from scratch.

In (Kocı and Janoušek 2009), the authors use DEVS to communicate the messages and data between different components of an object-oriented hierarchy. The authors propose to rely on the hierarchical structure of DEVS. Although the system architecture draws inspiration from DEVS it does not build any DEVS atomic model for input places or transitions. Therefore, this implementation is not designed to be used in multi-formalism context and cannot be easily interfaced with other DEVS component.

An implementation of Petri Net in DEVS is presented (Jacques and Wainer 2002). The Petri Net model serves as a library of a different formalism. Conceptually, the FSM library and its Petri Net implementation have similarities. Both build a library using an atomic model to represent the building blocks of a Petri Net. There are two models developed in to indicate the transitions and input places, respectively. The implementation allows it to be used in a DEVS model and can be interfaced with models of other formalisms.

A previous implementation of FSM library in DEVS is presented in (Zheng and Wainer 2003) where we built a library of FSM in CD++ (Wainer 2009), using numeric codes to represent states. The transitions and event handling were done by decoding the transitions from the coupled model and calculating the next transition and output. Our implementation builds upon few of these concepts and makes a more generic model to allow building complex FSM models in Cadmium. Cadmium allows the flexibility to use user defined data type to represent the states, the events, and outputs which we take advantage of. To the best of our knowledge this work is unique in providing a finite state machine model library developed for Cadmium tool.

3 FSM LIBRARY MODELS DESCRIPTION

Every finite state machine consists of finite states with transition functions. Here, the behavior of a generic state is described as an atomic model. A Finite State Machine is created by generating a coupled DEVS model which consists of atomic models interconnected with each other. The description of the atomic model and FSM models based on this atomic model will be discussed in this section.

Two atomic models were developed as per Moore and Mealy criteria, one atomic model for Mealy implementation and another atomic model for Moore implementation. The definitions of Moore and Mealy differ in that way the output function of these finite state machines is defined. In case of Moore design of finite state machine, the output function is formally given by $G: S \rightarrow \Lambda$.

The output function of Mealy design of finite state machine is given by $G: S \times \Sigma \rightarrow \Lambda$

Here *G* is the output function, *S* is a finite set of states and *Λ* is the set of finite output alphabets. In Moore machine the state output does not depend on the event input, whereas in Mealy machine, the state output depends both on the state and the event input.

Figure 1 shows a sketch of an atomic model that serves as a building block for finite state machines. There are different input and output ports to the atomic model which are described henceforth. *EventIn* is an input port to the atomic model which takes the external event. *TransitionIn* is an input port to the atomic model which takes in the value outputted by *Out* of any other state indicating next active state. *Output* is an output port to give the output of a model. *ErrorOut* is an output port to give error messages. *Out* is an output port out of the atomic model to give the next active state.

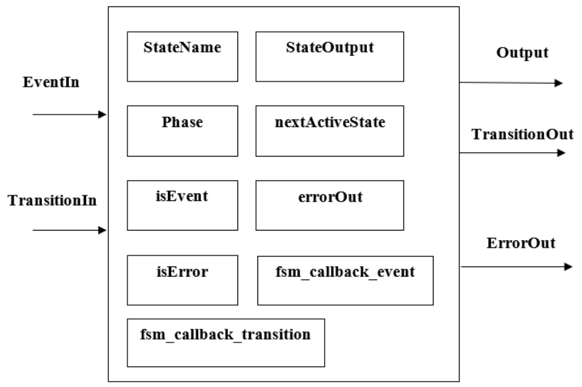

Figure 1: Atomic Model of a Finite State.

A unique global *stateName* is assigned to each state in a finite state machine. It represents the name of the state and defined as a string which uniquely identifies a state. The variable *phase* indicates whether this state is active or passive. If the input from *TransitionIn* matches the name of a state, then the state becomes active. Only one state is active at a time in the FSM. The *state* variable is used to output a state value. In case of the Moore models, this will be instantiated while creation of the state, whereas, in case of Mealy machines this variable is not instantiated and decided when an event comes. This difference is maintained in the implementation. The *nextActiveState* variable is the name of the next state to transition to. Whenever a state transitions to any other state based on a valid event input, it sends *nextActiveState* as output from *Out* to *TransitionIn* of all the connected states. A state becomes active if the message received from *TransitionIn* port is same as its *stateName*. Only one state can be active at time in a model.

The state variable *fsm_callback_event* is a function pointer to the event callback function which is implemented in the application of the FSM. It is called whenever there is an event inputted to the system. This function pointer has parameters modified whenever the callback function is called. This allows the state to be modified according to the logic of the FSM. The state variable *fsm_callback_transition* is a function pointer to transition callback function which is implemented in the application. This function pointer has parameters which are modified whenever the callback function for transition is called. The most significant role of this callback function is to turn the phase of a certain state to be active.

The state variable *isEvent* is enabled in the implementation of the *fsm_callback_event* whenever an event comes. This is checked in the output function of the atomic model before outputting the *state* and *nextActiveState* from the current state. The state variable *isError* is enabled in *fsm_callback_event* implementation if there is a spurious event. This variable is checked in the output function of the atomic model and if true, allows the error message to be outputted. The state variable e*rrorOut* contains the error message of the state which is outputted by the output function of the atomic model if *isError* is enabled.

4 FSM LIBRARY MODEL IMPLEMENTATION

The atomic models in Cadmium are programmed in C++ as a header only library. To explain the software architecture from a high level a simple class diagram is presented which shows how the concept of a state is depicted. In Figure 2, we can see that the atomic model contains all the state variables described earlier in the model description section. It also shows the methods defined according to DEVS formalism. The coupled model in the figure shows just one state instance that implements the atomic model. Any finite number of such states can be instantiated in a coupled model. The event that is inputted to the coupled model is connected to the *EventIn* port of all such states created by External Input Coupling (EIC). Each state connects its output port *Output* to different output ports of the coupled model by External Output Coupling (EOC). The different states are interconnected by Internal Coupling (IC). The output port, *Out,* of a state is connected to the input ports, *TransitionIn,* of all the states it can transition into. Conceptually, internal coupling establishes the connection between the states.

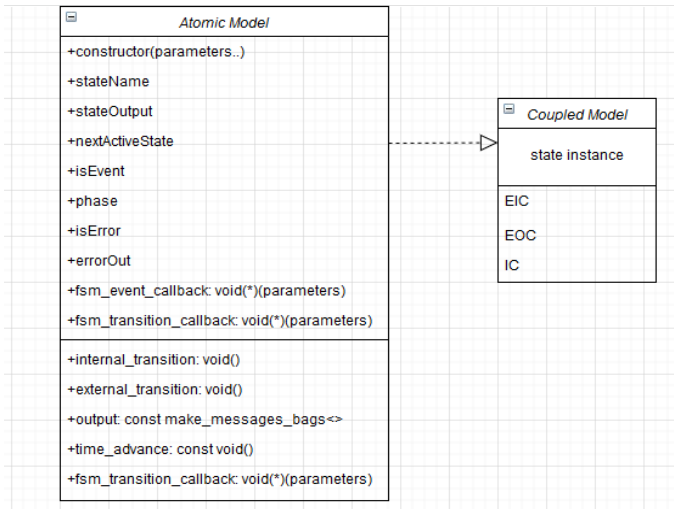
Atomic Model
+constructor(parameters..)
+stateName
+stateOutput
+nextActiveState
+isEvent
+phase
+isError
+errorOut
+fsm_event_callback: void(*)(parameters)
+fsm_transition_callback: void(*)(parameters)
+internal_transition: void()
+external_transition: void()
+output: const make_messages_bags<>
+time_advance: const void()
+fsm_transition_callback: void(*)(parameters)
Coupled Model
state instance
EIC
EOC
IC

Figure 2: Class Diagram of the software model.

4.1 Constructors

To create a state an object of the atomic model is created. To characterize a state, each state is initialized by calling the constructor defined in the atomic model. The below code snippet is from the *MooreFSM.hpp*.

```
MooreFSM (string stateName, string state, string phase, bool isEvent,
  void(*EventInFnPtr)(string eventIn, string transitionIn, string *nextActiveState,
     string *errorMsg, string stateName, string *phase, bool *isEvent, bool *isError),
       void(*TransitionInFnPtr)(string eventIn, string transitionIn,
          string *nextActiveState, string *errorMsg, string stateName, string *phase,
             bool *isEvent, bool *isError)) {
       state.stateName = stateName;
       state.state = state;
       state.phase = phase;
       state.isEvent = isEvent;
       state.fsm_callback_event = EventInFnPtr;
       state.fsm_callback_transition = transitionInFnPtr;        }
```

Figure 3: Moore Atomic Model Constructor.

The constructor given in Figure 3 is called when we construct instances of the Moore atomic model. It is to be noted that this includes the instantiation of *state*. Similarly, the constructor of the Mealy atomic model is presented below:

```
MealyFSM(string stateName, string phase, bool isEvent, void (*EventInFnPtr)
 (string eventIn, string transitionIn, string *nextActiveState,
   string *state, string *errorMsg, string stateName, string *phase,
     bool *isEvent, bool *isError), void (*TransitionInFnPtr)(string eventIn,
      string transitionIn, string *nextActiveState, string *state,
       string *errorMsg, string stateName, string *phase, bool *isEvent, bool *isError)){
       state.stateName = stateName;
       state.phase = phase;
       state.isEvent = isEvent;
       state.fsm_callback_event = EventInFnPtr;
       state.fsm_callback_transition = TransitionInFnPtr;                    }
```

Figure 4: Mealy Atomic Model Constructor

It can be observed from the Figure 4 that this constructor does not initialize the *state* as opposed to the Moore model. This is because the output of a state in Mealy implementation is not hard coded into the state but is decided based upon the event that comes in. For Mealy implementation it will be shown later how the *fsm_callback_event* decides the output of the state based on the event and the state.

A state is instantiated by creating a *shared_ptr* from the atomic model class. An example for construction of Moore state from one of the applications implemented is shown below:

```
shared_ptr<dynamic::modeling::model>TemperatureSet=dynamic::translate::make_dynamic_
  atomic_model<MooreFSM, TIME>("TempSet", "TempSet", "Enter Reference Temp.", "active",
    true, fsm_callback_event, fsm_callback_transition);
```

Figure 5: Moore State Instantiation

The fact that Figure 5 represents a constructor for Moore model, the state output value is passed during instantiation of a state and this is known as the output of the state (*TemperatureSet*). The third parameter passed into the constructor in the figure *("Enter Reference Temp.")* is the output for this state. In contrast, Mealy machine state instantiation will omit this parameter in calling of its constructor.

4.2 Callbacks

Two function callbacks, *fsm_callback_event* and *fsm_callback_transition* passed as parameters to the constructor of each state define how inputs are managed and how one state transitions to another state. These callbacks are implemented in the applications which will be using these atomic models for developing an FSM. The implementation of these callbacks differs depending on if it is a Mealy implementation or a Moore implementation of the FSM. *fsm_callback_event* is responsible to handle the external events inputted to a state, whereas, *fsm_callback_transition* manages the transition input to a particular state. The state transitions depending on external events, logic for output generation and error reporting are defined in these callbacks. The code snippet below shows the *fsm_callback_event* declaration for Mealy model.

```
void (*fsm_callback_event) (string eventIn, string transitionIn, string
       *nextActiveState, string *state, string *errorMsg, string stateName, string
              *phase, bool *isEvent, bool *isError);
```

Figure 6: Mealy Model EventIn Callback declaration

In Figure 6, it can be observed that the state variables are passed as parameters. It is to be noted here that all the parameters passed as pointer may be subject to modification in the implementation of the callback. In Mealy model implementation the *state* is passed as a parameter to populate this variable according to the event and the active state. Similarly, the code snippet below shows the declaration of *fsm_callback_event* of a Moore FSM implementation.

```
void (*fsm_callback_event) (string eventIn, string transitionIn, string
       *nextActiveState, string *errorMsg, string stateName, string *phase, bool
              *isEvent, bool *isError);
```

Figure 7: Moore Model EventIn Callback

In Figure 7, certain state variables which need to be modified in the function are passed as pointers. Here, the *state* is not passed as a parameter as the output is not modified in the event callback but hardcoded during state instantiation and does not depend on the event.

To demonstrate how *fsm_callback_event* and *fsm_callback_transition* are constructed, a simple FSM is presented. This FSM presents itself as an example for illustrating how the logic for managing the input events and state transitions are encoded. There are three state in the FSM described in Figure 8, *StateA, StateB* and *StateC*. It has only two valid events *'X'* and *'Y'*. If an event *'X'* arrives in the FSM and if the *StateA* is active, it transitions to StateB, otherwise if the *StateB* is active, it transitions to *StateC*. If an event *'Y'* arrives only *StateB* reacts to this event if it is active and transitions to *StateC*. In case of invalid inputs, the FSM outputs error message and resets to the initial state which is *stateA*.

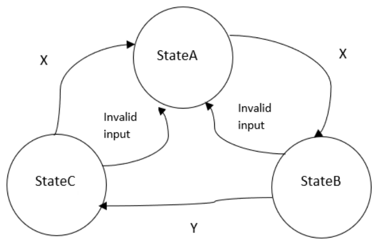

Figure 8: A simple Finite State Machine

The implementation of *fsm_callback_event* follows a structure which can serve as a guideline towards building any application. The *fsm_callback_event* definition below shows how certain state variables are modified to maintain the state machine.

```
fsm_callback_event (parameters: eventIn, transitionIn, pointer to nextActiveState,
pointer to   errorMsg, stateName, pointer to phase, pointer to isEvent, pointer to
isError)
  if (eventIn == X) then
     if (stateName == "StateA" && pointer to phase == "active") then
                pointer to nextActiveState = "StateB"
          pointer to state = "Output of StateA"
          pointer to phase = "inactive"
          pointer to isEvent = true
     else if (stateName=="StateC" && pointer to phase=="active") then
          pointer to nextActiveState = "StateA"
          pointer to state = "Output of StateC"
          pointer to phase = "inactive"
          pointer to isEvent = true
else if (eventIn == Y) then
     if(stateName== "StateB" && pointer to phase=="active")then

                pointer to nextActiveState =   "StateC"
          pointer to state = "Output of StateB"
          pointer to phase = "inactive"
          pointer to isEvent = true
  else if (eventIn!= X && eventIn!= Y) then
          pointer to nextActiveState = "StateA"
          pointer to errorMsg = "Invalid Input"
          pointer to isError = true
          pointer to isEvent = true
        if (stateName!= "stateA") then
            pointer to phase = "inactive"
```

Figure 9: Mealy event callback pseudo code

Figure 10 presents a pseudo code to show how the *fsm_callback_event* would manage the events. The pseudo code shows that for each event there may be one or more conditional block corresponding to each state that may react to the particular event. For each state whose *phase* is “active”, it can be seen that the *nextActiveState* is made to point to the next state based on a particular event*, phase* is made “inactive” and *isEvent* is made “true” to indicate an event arrived into the system. In case of Mealy implementation *state* is populated with the output of the state, whereas in Moore implementation, *state* is not modified in the *fsm_callback_event* function and the step of assignment to *state* is omitted from the function. In the case when an invalid event arrives (neither *‘X’* nor *‘Y*’), *isError* is assigned “true” and *errorOut* is populated with the error message of the state. This pseudo code presents the intuition behind handling of the events without knowing any detail about the FSM. More conditional checks may be necessary depending on the application. However, the basic structure will follow the algorithm presented in the pseudo code.

```
fsm_callback_transition (eventIn, transitionIn, pointer to nextActiveState, pointer to
state, stateName, pointer to phase, pointer to isEvent, pointer to isError)
   if (!transitionIn.empty()) then
      if (stateName == transitionIn) then
         pointer to phase = "active";
         pointer to isEvent = true;
```

Figure 10: transition callback pseudo code

The *fsm_callback_transition* callback function handles the input from the *TransitionIn* input port in a state. In this function the *TransitionIn* is compared with its *stateName.* If the two match this state becomes active and starts accepting external events. This is a basic structure of a *fsm_callback_transition.* Depending on application, it may have more conditional checks. If a state triggers a transition without any event (*i.e.* timer expired), it may contain conditional blocks like in *fsm_callback_event* to set the *nextActiveState, isEvent* and *phase.*

Both these callbacks have all the state variables as parameters which can be used by the application logic accordingly. The only difference between Moore implementation and Mealy implementation of these callbacks in that the *State* is not passed as a parameter in case of Moore Machine whereas it is included in case of Mealy Atomic model. This is because the events do not decide the state output in case of Moore Machine and the *State* need not be manipulated in these callbacks. However, in Mealy implementation *State* is passed to the callbacks as the outputs are decided based on incoming events and decided as part of the logic of handling the events and state transitions.

4.3 Transition functions implementation

Whenever an event comes into the coupled model the external transition is called which as follows:

```
void external_transition(TIME e, typename make_message_bags<input_ports>::type mbs) {
    vector<string> bag_port_eventIn, bag_port_transitionIn;
    bag_port_eventIn=get_messages<typename MooreFSM:eventIn>(mbs);
    bag_port_transitionIn=get_messages<typename MooreFSM:transitionIn>(mbs);
    if(!bag_port_eventIn.empty()){
      state.fsm_callback_event(bag_port_eventIn[0],"",&state.nextActiveState,
         &state.errorMsg,state.stateName,&state.phase,state.isEvent,&state.isError);
    if(!bag_port_transitionIn.empty() )
       state.fsm_callback_transition("",bag_port_transitionIn[0], &state.nextActive
        State,&state.errorMsg,state.stateName,&state.phase,&state.isEvent);      }
```

Figure 11. External transition function in Moore model

In the Figure 11 it can be observed that *fsm_callback_event* or *fsm_callback_event* is called based on whether it is an event or a transition, respectively. In this implementation only one of the callbacks is called in the external transition function.

The output function is as follows:

```
typename make_message_bags<output_ports>::type output() const {typename
       make_message_bags<output_ports>::type bags;
if(state.isEvent) {
  get_messages<typenameMealyFSM:state>(bags).push_back(state.state);
  get_messages<typename MealyFSM:Out>(bags).push_back(state.nextActiveState);}
if(state.isError)
    get_messages<typename MealyFSM:errorOut>(bags).push_back(state.errorMsg); }
```

Figure 12. Output function of the Atomic Model

The output function described in Figure 12 checks for the *isEvent* and *isError* variable which are set by the application logic and gives out the state output or error output, respectively. These variables are set to true only in the callback functions described in *B*.

4.4 Internal transition function implementation

If the state variables *isEvent* and *isError* are not assigned false, the output function would generate output after the time advance expires every time.

```
    void internal_transition() {
       state.isEvent = false;
       state.isError = false;   }
```

Figure 13. Internal transition function

As shown in Figure 13, *isEvent* and *isError* are set to false in the internal transition of the atomic model so that the model doesn't send out outputs repeatedly after expiry of time advance in absence of any event.

5 CASE STUDIES

In this section we discuss several case studies showing the definition of the model in DEVS, the transitions and outputs of the states.

5.1 Automatic Temperature Controller

The automatic temperature controller model in Figure 14 is built as a Moore model, following a design presented in (Zungeru et al. 2018). This application first lets the user to set the reference temperature to a certain value. The sensors then sense the room temperature. If the temperature sensed is less than the reference temperature, the heater is turned on; otherwise the AC is turned on. If the heater is turned on it also checks for Carbon Monoxide (CO) levels. It raises and alarm to see if the CO level is above the allowable limit. The CO level here is generated as a random number between 20 to 100ppm; as CO levels above 50ppm are hazardous, the FSM transitions to the state *CODetectAlarm*, or to *CODetectOk* if the levels are below alarming point.

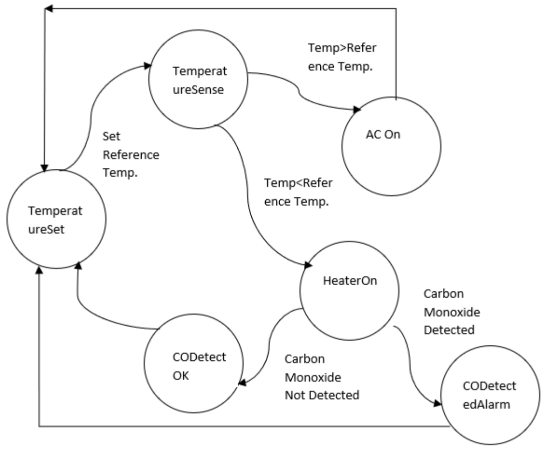

Figure 14: State Diagram of Automatic Temperature Controller

The coupled model specification is as follows:

X = {SetTemperature, a numerical value of Sensed temperature}
D = {TempSet, TempSensed, HeaterOn, AcON, CODetectOK, CODetectAlarm};
Z(TempSet) = TempSet, Z(TempSet) = TempSensed,
Z(TempSensed)=HeaterOn, Z(TempSensed) = AcON;
Z(HeaterOn) = CODetectOK, Z(HeaterOn) = CODetectAlarm, (CODetectOK)= TempSet;
Z(CODetectAlarm) = TempSet, Z(AcON) = TempSet

A simplified coupled model diagram is depicted in Figure 15 which strips off the transition in and out for simplicity of the diagram. The transitions can be seen from state diagram and the DEVS specification, with *TempSet, TempSensed, HeaterOn, AcON, CODetectOK* and *CODetectAlarm* atomic models.

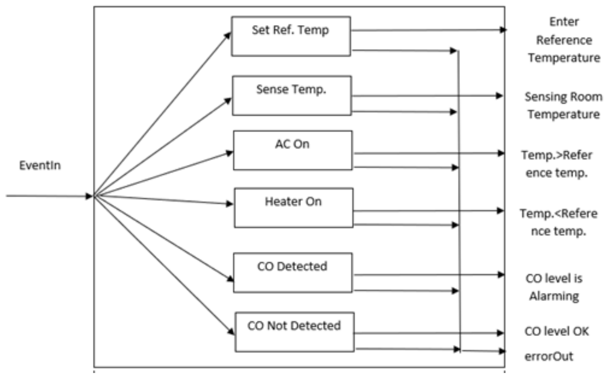

Figure 15: Automatic Temperature Controller Coupled Model

To evaluate the models, we use an *input_reader* model, whose output port is connected to the *EventIn* input port of the models. Following, we show a sample test case for the model:

Inputs for eventIn

```
00:00:10 setTemperature
00:00:20 70
00:00:30 setTemperature
00:00:40 70
00:00:50 setTemperature
00:01:00 90
00:01:10 setTemperature
00:01:20 150
00:01:30 setTemperature
00:01:40 50
```

Initially, we generate a temperature of 70 (F), while the reference temperature is set at 80 F. Hence, if the sensed temperature is below this reference temperature it is supposed to switch on heater otherwise it would have switched on the Air Conditioner. Following we show a simulation trace for the FSM model.

```
out: {70} generated by model input_reader
23:000 - [MooreFSM:state: {Sensing Room Temp}, MooreFSM:Out:{HeaterOn}] model:TempSense
26:000 - [MooreFSM:state: {Heater is ON}, MooreFSM:Out: {CODetectAlarm}] model: HeaterOn
```

Figure 16: Simulation log snippet for Automatic Temperature controller model

The highlighted simulation logs in the Figure 16 indicate that the on sensing 70 F the state transition from "Sensing Room Temp." to "HeaterOn" state and the heater is turned on. In this application, turning the heater on is associated with CO detection activity in a closed environment. Figure 17 indicates the one instance where the random number may have been below 50, this application transitions from "HeaterOn" state to "CODetectOk" state as can be observed from the highlighted simulation log.

```
59:000 [MooreFSM:state: {Heater is ON}, MooreFSM:Out: {CODetectOK}], model: HeaterOn
```

Figure 17: Highlighted log to suggest normal CO level

Following, we can see the results obtained when the temperature goes above 80F; in that case, the Air Conditioning is turned on, as seen in Figure 18 where the sensed temperature is 90 F.

```
out: {90}, model: input_reader
[MooreFSM:Out: {AirCondOn}], model:TempSense
01:06:000 - [MooreFSM:state: {AC ON}, MooreFSM:Out: {TempSet}], model:AirCondOn
```

Figure 18: Highlighted log indicating higher temperature

The FSM also puts a restriction on the temperature sensed. If it is either above 120 F or less than -40 F, it sends an error message to indicate the sensor is producing invalid data. This can be seen in the highlighted simulation log in Figure 19, where we can see an error message when there is an input of 150 F to the FSM indicating that the sensor read may be faulty.

```
out: {150}, model:input_reader
01:36:000 MooreFSM:errorOut: {Invalid Temperature sensed}], model:  TempSense
```

Figure 19: Highlighted logs indicating faulty reading

5.2 Online Library Portal

The case study presented in this section is used to represent an Online Library portal as described in Figure 20. In this case, we use the Mealy FSM model is used to build model.

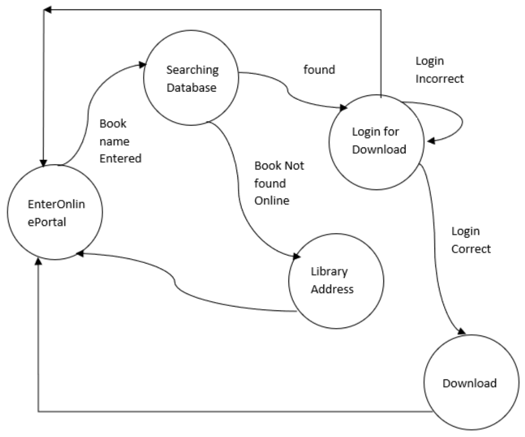

Figure 20: State Diagram of Online Library Portal

As we can see, users enter an Online portal in a University website, and then they enter the title of the book/journal requested. The FSM represents the search of the book in the database ("Searching" state). If the item is found, the portal prompts for Login Details. The Login needs a correct Password. The Password entered is checked, and if there is a successful match, it gives out a link ("Download" state). If the password does not match with the one in the system, the portal prompts the user to retype the correct password. If the book is not found, the portal gives out the address of the Library Building to check for further query of the item. As we can see, the outputs of the states now depend on the event, and not just the states.

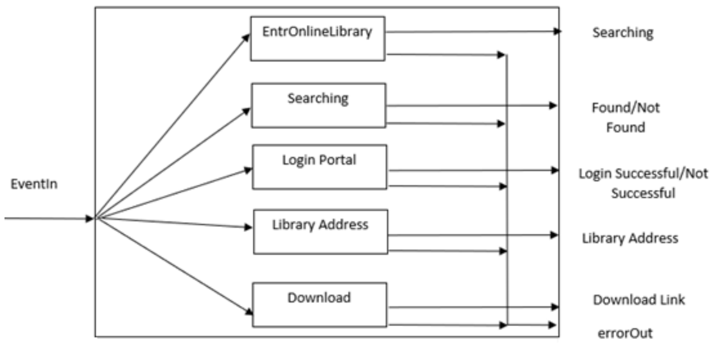

Figure 21. Online Library Portal Coupled Model

A simplified coupled model diagram is depicted in Figure 21 (which omits the transition in and out for simplicity of the diagram), shows the mapping of the FSM into a DEVS coupled model.

This coupled model, which represents the Mealy machine depicted in Figure 20, can be formally specified as follows:

X = {Search Book, Book Name, Login password};
D = {EnterOnlineLibrary, Searching, LoginPortal, LibraryAddress, Download};
Z(EnterOnlineLibrary) = Searching;
Z(Searching) = LoginPortal, Z(Searching) = LibraryAddress;
Z(LoginPortal) = LoginPortal, Z(LoginPortal) = Download
Z(Download)=EnterOnlineLibrary, Z(LibraryAddress) = EnterOnlineLibrary.

where *EnterOnlineLibrary, Searching, LoginPortal, LibraryAddress* and *Download* are instances of the atomic model.

To study the model, we can run different test cases using the Cadmium *input_reader* described in the previous section. Let us consider the following test scenario:

Inputs for eventIn

```
00:00:10 SearchBook
00:00:20 ParticlePhysics
00:00:40 MaCoDrum
00:00:50 SearchBook
00:01:00 Cosmos
00:01:10 MaCDum
00:01:20 MaCoDrum
00:01:30 Gibberish
00:01:40 SearchBook
00:01:50 AlgorithmBook
```

In the online library portal, the password set is "MaCoDrum". Whenever this event comes after the queried book or journal is found in the database it gives out a Download link. This application accepts few names of the book/topics. "ParticlePhysics" is one of them as it is assumed that this book is present in the database.

```
out: {ParticlePhysics}] generated by input_reader
23:000 - [MealyFSM:state: {Enter Login Details to Access.}, MealyFSM:Out: {Login},
out: {MaCoDrum}], model:input_reader
43:000 - [MealyFSM:state:{Download Link}, MealyFSM:Out:{Download}], model:  Login
```

Figure 22. Highlighted simulation log showing successful search and login

If a wrong password is entered, the state remains at "Login" state and prompts for entering the correct password. This is indicated in the simulation logs provided in the Figure 23.

```
out:{MaCDum}, model:input_reader
01:13:000  -[MealyFSM:state:  {Enter  Correct  Password},  out:{MaCoDrum}],  model:
input_reader
01:23:000 - MealyFSM:state: {Download Link}, MealyFSM:Out: {Download}], model:Login
```

Figure 23. Highlighted simulation log showing wrong password entered

As it can be seen in the Figure 23, when we enter the incorrect password "MaCDum" instead of the correct one which is "MaCoDrum", we are requested to retry, and once the correct password is entered it transitions to "Download" state.

6 CONCLUSION

We presented a method to build Moore and Mealy in DEVS, defining basic components as atomic models and building applications based on these models. The idea was to implement FSM libraries for Cadmium, providing one of the components of several DEVS multi-formalism. This implementation provides a framework and method to define DEVS model for finite state machines and aims to provide an insight into building any complex application using the libraries built. With Cadmium any data type can be used to define the state variables and message passing between states.

Cadmium provides a clean and efficient platform to simulate DEVS model. By observing the design pattern and language features used it is quite efficient. For example, it uses the modern C++ features such as *shared_ptr* which ensures no memory leak. It creates Abstract Atomic Model which can be used in several applications. This design facilitated the task of building FSM library on top of it and provided a platform for a clean design.

As the implementation of FSM presented here follows DEVS formalism, it lends itself to be easily interfaced with other DEVS models easily and provides a seamless interaction with each other in a simulation. Further enhancements can be made to the model by including a queue model to store the events and manage each event to maintain proper states. This will ensure that no event is ignored, and all events will be processed in due course of time. This will make the model faster and asynchronous in terms of arrival of events and processing them.

REFERENCES

Zeigler, B.P., H. Praehofer, and T. G. Kim 2000. *Theory of modeling and simulation*. Academic press.

D. Vicino, L. Belolli, C. Ruiz-Martin, G. Wainer. 2019 “Building DEVS Models with the Cadmium Tool”. In *Proceedings of the 2019 Winter Simulation Conference*. National Harbor. MD.

Garredu, S., Vittori, E., Santucci, J., & Bisgambiglia, P. 2013. “From State-Transition Models to DEVS Models: Improving DEVS external interoperability using MetaDEVS”. In *Proceedings of International. Conference on Simulation and Modeling Methodologies, Technologies and Applications*. Reykjavik. Iceland.

Kocı, R., & Janoušek, V. 2009. “Simulation based design of control systems using DEVS and Petri Nets”. *In Proceedings of International Conference on Computer Aided Systems Theory*. Las Palmas, Spain.

C. Jacques, G. Wainer. 2002. “Using the CD++ DEVS toolkit to develop Petri Nets”. In *Proceedings of the 2002 SCS Summer Computer Simulation Conference*. San Diego, CA. USA.

T. Zheng, G. Wainer. 2003 “Implementing finite state machines using the CD++ toolkit”. In *Proceedings of the 2003 SCS Summer Computer Simulation Conference*. Montreal, QC. Canada.

G. Wainer 2009. *Discrete-Event Modeling and Simulation: a Practitioner’s approach*. CRC Press.

Zungeru, A. M., Mangwala, M., Chuma, J., Gaebolae, B., & Basutli, B. 2018. Design and simulation of an automatic room heater control system. Heliyon, 4(6), e00655.

AUTHOR BIOGRAPHIES

AMITAV SHAW is a Masters of Engineering student at the Department of Systems and Computer Engineering at Carleton University. His email address is amitavshaw@cmail.carleton.ca.

ARSHPREET SINGH is a Masters of Engineering student at the Department of Systems and Computer Engineering at Carleton University. His email address is arshpreetsingharshpr@cmail.carleton.ca.

GABRIEL A. WAINER, FSCS, SMIEEE, is a Full Professor and Associate Chair for Graduate Studies at Carleton University (Ottawa, Canada). Prof. Wainer is a member of the Board of Directors of the SCS. He is one of the founders of the Symposium on Theory of Modeling and Simulation, SIMUTools and the symposium on Simulation of Architecture and Urban Design (SimAUD). Prof. Wainer is the Special Issues Editor of SIMULATION, a member of the Editorial Board of IEEE Computing in Science and Engineering, Wireless Networks (Elsevier), Journal of Defense Modeling and Simulation (SCS). He is the head of the Advanced Real-Time Simulation lab, located at Carleton University's Centre for advanced Simulation and Visualization (V-Sim). He is a Fellow of SCS. His email address is gwainer@sce.carleton.ca.