# Finite State Machine Application Model Libraries in Cadmium

*Amitav Shaw* *Arshpreet Singh*

Department of Systems and
Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON. K1S-5B6 Canada

**ABSTRACT**: *The DEVS formalism has been conceived in order to provide a common basis for discrete-event modeling and simulation. In regard to the class of formalisms termed as discrete-event, system models are described at an abstraction level where time base is continuous but only a finite number of events can occur in bounded time span. These events can change the state of the system. However, state of the system does not change in between these events. This is contrary to continuous models where state of the system changes continuously over time. In addition, DEVS formalism takes concepts from Discrete Event simulation to give more meaning to the model by conducting simulation analyses. In this report, we describe how DEVS formalism can be used to build a library of finite state machine using Cadmium and implement different applications of each type of finite state machine i.e. Moore Machine and Mealy Machine.*

## 1. Introduction

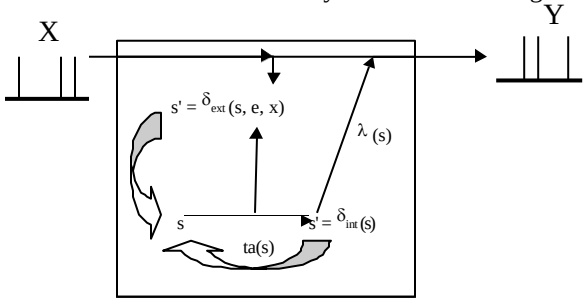
The need for modeling and simulation techniques has become increasingly important to study the complex systems of these days, in which experimentation on the actual system is not feasible or is dangerous. Discrete Event Systems Specifications (DEVS) is one such formalism that is being considered and has gained much attention in the research realm. DEVS allows describing the system behavior at two levels. At the base level, atomic model describes the autonomous behavior of the discrete event system as a sequence of deterministic transitions between sequential states as well as how it reacts to external input and how it generates output. This allows for building very complex models by connecting different atomic models in a hierarchical manner. At the higher level, a coupled model describes the system as network of such coupled components. We will show how to use DEVS to build a library of finite state machine (FSM) using cadmium as an extension of the FSM that has been implemented in CD++. In addition, we have implemented three different applications of each type of the finite state machine i.e. Moore Machine and Mealy Machine and conducted simulation analyses.

We describe an implementation where Cadmium is used for the purpose of simulating using DEVS specifications. Cadmium is a tool for Discrete-Event modeling and simulation, based on the DEVS formalism. DEVS is a discrete event paradigm that allows a hierarchical and modular description of the models. Each DEVS model can be behavioral (atomic) or structural (coupled), consisting of inputs, outputs, state variables, and functions to compute the next states and outputs. Cadmium is a cross-platform header-only library implemented in C++.

## 2. Background

DEVS is a modular and hierarchical formalism for modeling and analyzing complex systems. It provides a framework to model and simulate the discrete event systems which depend on representing the systems by a hierarchy of atomic components. This allows very complex models to be built by connecting different DEVS models either atomic or coupled models, in a hierarchical manner. DEVS provides an abstract approach of modeling by separating the modeling from simulation aspects and hence facilitating the model usability and interoperability.

The basic building block of any DEVS model is the atomic model, which can be connected to other atomic models to form what is called a coupled model. A DEVS atomic model can be informally described as in Figure 1.




Each atomic model relies on three sets and four functions. With these semantics, the formalism can specify a specific state at any point of time as well as interact with other models using I/O events which are caused by state transitions. It has an interface consisting of input (x) and output (y) ports to communicate with other models. In the absence of external events it remains in the state (s) for a lifetime

defined by time advance ta(s) function. A transition that occurs due to the consumption of time indicated by ta(s) is called an internal transition ($\delta_{int}$). The model execution results are spread through the model's output ports by activating an output function ($\lambda$). On the other hand, external transition occurs on receiving external event. An external transition function ($\delta_{ext}$) specifies how to react to those inputs based on current state (s), elapsed time since last transition (e) and external event (x).

A DEVS coupled model is composed of several atomic or coupled sub-models, as shown in Figure 2.

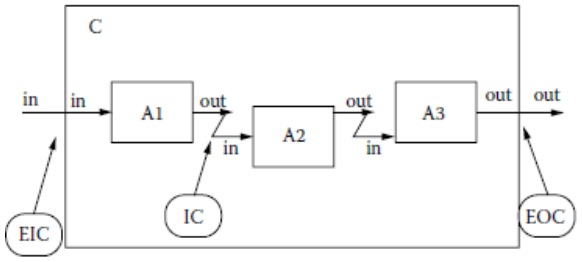



**Figure 2 Description of Coupled Model**

Coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the model interfaces. The model's coupling scheme defines the interconnectivity between models and the interface with the external world.

Figure 2 shows an example of coupled model with three sub-components A1-A3. These basic models are connected through the I/O ports presented in the figure. The models are connected to the external coupled model through EIC and EOC connectors. It is to be noted that A1-A3 are the basic models which means they can either be atomic or coupled models. The model depicted in figure 2 can be formally defined as:

C = <X, Y, D, {Md | d ∈ D}, EIC, EOC, IC, select>

Where

X = {(in, v) | in ∈ IPorts, v ∈ **R**};
Y = {(out, v) | out ∈ IPorts, v ∈ **R**};
Md = {MA1, MA2, MA3};
EIC ∈ {((Self, in), (A1, in))};
(Or EIC ∈ {((C, in), (A1, in))});
EOC ∈{((A3, out), (Self, out))};
(Or EOC ∈{((A3, out), (C, out))});
IC ∈ {((A1, out), (A2, in)); ((A2, out), (A3, in))};
Select = {A3, A1, A2}.

The model definition presented shows the specification of the three components A1-A3 and their internal/external couplings. Coupled models group several DEVS into a composite model that can be regarded due to the closure property, as a new DEVS model. The closure property guarantees that coupling of several class instances results in a model of the same class, allowing hierarchical construction.
As multiple sub-components can be scheduled for an internal transition at the same time, ambiguity could arise. If A1 executes its output/internal transition first producing an output that maps into an external event for A2 which is also scheduled for internal transition at the same time, then it is not clear which transition should the second component execute first. The select function helps to solve this ambiguity. The function defines an ordering over all the components of the coupled model so that only the first model to execute in the case of simultaneous internal events can be chosen. Hence A1 is executed before A2 thus external transition executes first.

In regard to fulfillment of our term project our goal is to provide a set of library to develop complex models based on multi-formalisms. In this work we will focus on implementation of finite state machine as an example of multi-formalism model definition and implementation of various applications based on finite state machine: Moore Machine and Mealy Machine. Finite State Machines (FSM) are popular for modeling systems in a variety of areas such as software design and digital logic design etc. It is formally defined as follows:
FSM = < S, X, Y, f, g >

Where

X: finite input set
Y: finite output set
S: finite state set
f: next state function
f: X * S -> S
g: output function
g: S -> Y: Moore machine
g: X * S -> Y: Mealy machine

Every Finite State Machine (FSM) is supposed to have an initial state, a next state function which defines how to obtain the next state in the system, and an output function that uses current state and inputs to generate outputs. According to the relationship between the input sets and output functions, two kinds of FSMs can be defined: Moore machines, in which outputs are independent from the inputs, and Mealy machines, in which, besides the current state variables, the current inputs are analyzed to decide the current output value.

## 3. Models Defined

Every finite state machine consists of a number of finite states with transition functions. Hence, 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 a number of those atomic models. The description of the atomic model and FSM models based on this atomic model will be discussed in this section.

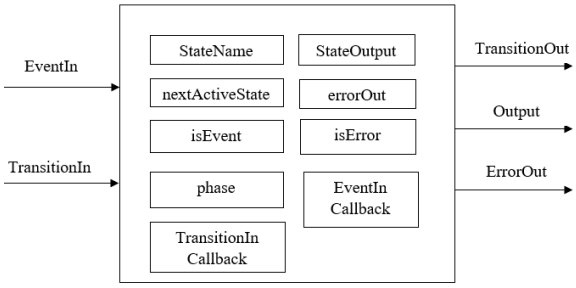



**Figure 3 Atomic Model of a Finite State**

Two atomic models were developed as per Moore and Mealy criteria. Atomic model for Mealy implementation and Atomic model for Moore implementation. In Figure 3, an atomic model is put forth to achieve building the several Finite State Machines. A unique global *stateName* is assigned to each state in a Finite State Machine. The *phase* indicates whether this state is active or passive. Only one state is active at a time in the finite state machine. The *eventIn* receives the external events. If a state receives a valid event when it is active, the *stateOutput* port sends out the Output, *transitionOut* port sends out the nextActiveState signal to all the transitionIn ports in the FSM announcing which state is active in the next step. A state becomes active if the encoded message received from transitionIn port is same as its stateName. An errorOut would be sent out from the errorOut port if an error occurs in the state. Two callbacks are provided which are implemented with the help of function pointers in the software. These callbacks are *EventIn Callback (fsm_callback_event in the code)* and *Transition Callback (fsm_callback_transition in the code).*

These callbacks are implemented by the application using the Atomic Model. The implementation of Mealy and Moore Machine depends on how these callbacks are implemented. All the state transitions depending on external event, logic for output generation and error reporting happens through these callbacks. The below code snippetThe *EventIn Callback* handles the situation when an Event arrives when in a certain state. If it is a Moore machine this event will trigger a state transition and the state's output will be given out. If it is a Mealy machine, the output will depend on the event and the output will be given out along with the state transition.

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

**Figure 4 Moore Atomic Model EventIn Callback**

Similarly, the *TransitionIn Callback* handles the situation when there is a transition of states. This function waits for the transitionIn event and compares this event with its *stateName.* If the two match this particular state becomes active and starts accepting Events.

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

**Figure 5 Moore Atomic Model TransitionIn Callback**

Both these callbacks have all the state variables as parameters which can be modified by the application logic accordingly. The only difference between Moore implementation and Mealy implementation of these callbacks in that the *StateOutput* 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 *StateOutput* need not be manipulated in these callbacks. In Mealy implementation *StateOutput* is passed to the callbacks as the outputs are decided on the basis of incoming events and decided as part of the logic of handling the events and state transitions. The EventIn and TransitionIn callbacks for Mealy Model are as follows:

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

**Figure 6 Mealy Atomic Model EventIn Callback**

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

**Figure 7 Mealy Atomic Model TransitionIn Callback**

Each state is initialized by using the constructor in defined in the Atomic model. The below code snippet is from the *MooreFSM.hpp*.

```
MooreFSM (string stateName, string stateOutput, 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.stateOutput = stateOutput;
        state.phase = phase;
        state.isEvent = isEvent;
        state.fsm_callback_event = EventInFnPtr;
        state.fsm_callback_transition=transitionInFnPtr;
  }
```

**Figure 8 Moore Atomic Model Constructor**

This constructor is called while we construct instances of the Atomic model. It can be observed that this includes the instantiation of *stateOutput* as it is a Moore implementation. An example 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 9 Moore Atomic Model/State Instantiation**

Here a state is instantiated by creating a shared_ptr from the Atomic Model Class. The fact that it is a Moore implementation, the state output value is passed during instantiation of a particular state and this is known as the output of the state. The third parameter passed into the constructor in Figure 9 is the Output for this state. Similarly, for Mealy machine the constructor is called as follows:

```
shared_ptr<dynamic::modeling::model>Idle= dynamic::translate::make_dynamic_atomic_model<MealyFSM, TIME>("Idle", "Idle", "active", true, fsm_callback_event, fsm_callback_transition);
```

**Figure 10 Mealy Atomic Model/State Instantiation**

It can be observed that the above construction of the object of the Atomic Model doesn't pass the state output value during instantiation. This is because this is a Mealy implementation and the output is determined when the event comes in and is not included during instantiation of the state.

Whenever and Event comes into the coupled model the external transition is called which shown as follows:

```
// external transition.
void external_transition(TIME e,
typename make_message_bags<input_ports>::type mbs) {
        vector<string> bag_port_eventIn;
        vector<string> bag_port_transitionIn;
        bag_port_eventIn=get_messages<typename
MooreFSM_defs::eventIn>(mbs);

bag_port_transitionIn=get_messages<typename
MooreFSM_defs::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);
} else if(!bag_port_transitionIn.empty() ){
   state.fsm_callback_transition("",bag_port_transitionIn[0],
&state.nextActiveState, &state.errorMsg, state.stateName,
&state.phase, &state.isEvent, &state.isError);
        }
  }
```

**Figure 11 External transition function in Moore model**

In the Figure 11 we can see that the event callback and transition callbacks are called with parameter as state variables which are modified by the applications. The application checks on the events and then based on whichever state is active and can react to the event, transitions to the next state and gives the output. One example is shown below for state transition, where *isEvent* to true whenever the state is active, assigns the *nextActiveState* and deactivates the state by writing "inactive" to *phase*.

```
void fsm_callback_event(string eventIn, string transitionIn, string *nextActiveState, string *errorMsg, string stateName, string *phase, bool *isEvent, bool *isError){
if(!eventIn.empty()){
      if(eventIn == "one"){
          if(stateName == "zero" && *phase == "active") {
                *nextActiveState = "one";
                *phase = "inactive";
                *isEvent = true;
```

**Figure 12 Snippet of event callback implementation**

Figure 12 is an example is from Moore example of Vending machine. Here the Output is not modified as the output here just depends on the state and not on the event.

```
else if(eventIn == "Regulator1") {
   if(stateName == "SetRegulator" && *phase == "active") {
    *nextActiveState = "Toasting";
    *stateOutput = "Set To Regulator Level 1";
    *phase = "inactive"; *isEvent = true;}
} else if(eventIn == "Regulator2") {
if(stateName == "SetRegulator" && *phase == "active") {
   *nextActiveState="Toasting";
*stateOutput = "Set To Regulator Level 2";
*phase = "inactive"; *isEvent = true;}
```

**Figure 13 Snippet of event callback implementation**

Figure 13 is an example of Mealy application for Toaster where the output is set according to the event.

In the Figure it can be seen that based on the Regulator level inputted as EventIn the state output is decided.

An Example for transition callback can be seen below where it makes the state active after ensuring that the transitionIn is same as the state name.

```
void fsm_callback_transition(string eventIn, string transitionIn, string *nextActiveState, string *stateOutput, string *errorMsg, string stateName, string *phase, bool *isEvent, bool *isError){

        if(!transitionIn.empty()) {
                if (stateName == transitionIn) {
                        *phase = "active";
                        *stateOutput = "";
                        *isEvent = true;
                }
```

**Figure 14 Code Snippet of Transition callback implementation**

The output function checks for the *isEvent* and *isError* variable which are set by the application logic and gives out the state output. The code snippet is as follows:

```
// output function
   typename make_message_bags<output_ports>::type output() const {typename make_message_bags<output_ports>::type bags;
if(state.isEvent) {
    get_messages<typename MealyFSM_defs::stateOutput>(bags).push_back(state.stateOutput);
get_messages<typename MealyFSM_defs::transitionOut>(bags).push_back(state.nextActiveState); }
if(state.isError){
   get_messages<typename MealyFSM_defs::errorOut>(bags).push_back(state.errorMsg);
   }
  return bags;
}
```

**Figure 15 Code Snippet of Output function in the Atomic Model**

The state variables *isError* and *isEvent* set to false in the internal transition of the atomic model so that the Model doesn't send out outputs repeatedly in absence of any event.

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

# 4. Model Applications

Model applications were developed based on the atomic models developed as per Moore and Mealy criteria. These applications are basically DEVS coupled models and built by using several atomic models known as states. For Moore Machine applications the Moore atomic model is used while for Mealy Machine application Mealy atomic model is used. In this section we list all the applications, the definition of the model in DEVS formalism, the transitions and outputs of the states.

## A. Cold Drink Vending Machine

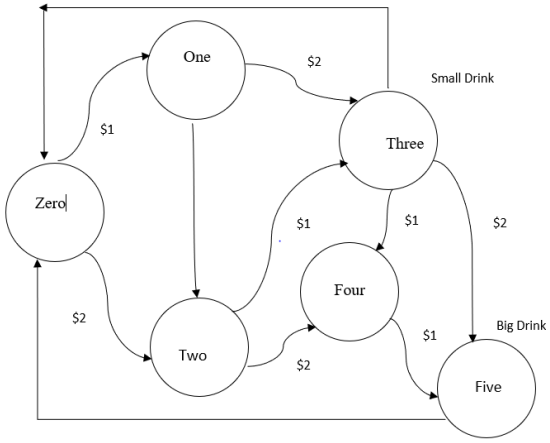



**Figure 16 State diagram of a simple Vending machine**

A simple cold drink vending machine is implemented using the Moore FSM atomic model. Figure 10 describe the links between the states. Figure 11 is a simplified couple model not showing the transition in and out. The transitions can be seen in the state diagram above.

This application accepts one-dollar coin (loonie), two - dollar coin (toonie) and "PushButton" as Events and can dispense a small cold drink bottle for $3 or a big cold drink bottle for $5. The "PushButton" after reaching the state "three" or "five" delivers a drink bottle. The names of the states are self-explanatory with the name corresponding to the amount of money inserted in the vending machine.

If an invalid event occurs, i.e. wrong denomination inserted in the machine, the machine outputs an error message.

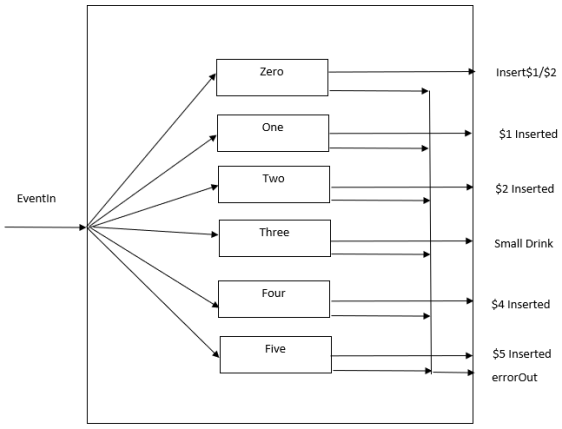



**Figure 17 DEVS coupled model describing a simple Vending machine**

The formal specifications <X, Y, D, {Mi}, {Ii}, {Zij}, SELECT > for this coupled model are defined as follows:

X = {in};
Y = {Zero, One, Two, Three, Four, Five, error};
D = {Zero, One, Two, Three, Four, Five};
I(Zero) = I(One) = I(Two) = I(Three) = I(Four) = I(Five);
Z(Zero) = Zero, Z(Zero) = One,
Z(Zero) = Two;
Z(One) = Two, Z(One) = Three, (One)=Zero;
Z(Two) = Three, Z(Two) = Four,
Z(Three) = Four, Z(Three) = Five,
Z(Three) = Zero,
Z(Four) = Five, Z(Four) = Zero,
Z(Five) = Zero;

SELECT:
Zero > One > Two > Three > Four > Five
Here Zero, One, Two, Three, Four and Five are atomic models and {One(dollar), Two(dollar), PushButton} are valid Events.
The Outputs of each state are as shown in the Figure.

## B. Automatic Temperature Controller

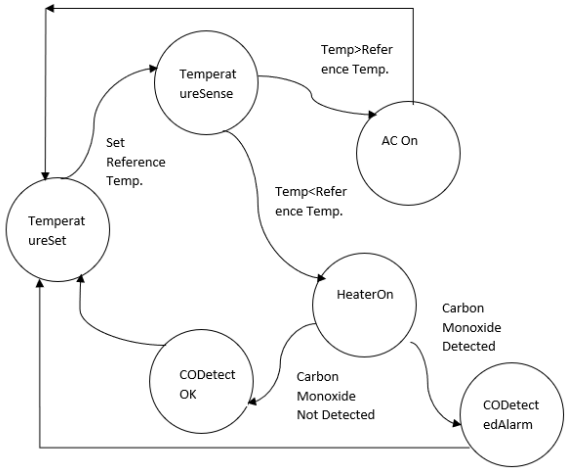



**Figure 18 State Diagram of Automatic Temperature Controller**

Automatic temperature controller application is based on Moore model and lets the user to set the reference temperature to a certain value. The sensors then sense the room temperature and feeds that to the state machine. 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 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 20ppm to 100ppm. As it is known that CO levels above 50ppm is hazardous, this state machine transitions to the CODetectAlarm state if it senses higher than normal levels. It transitions to CODetectOk if the levels are below alarming point.
A simplified coupled model diagram is given which strips off the transition In and Out for simplicity of the diagram. The transitions can be seen from state diagram and also the DEVS specification.

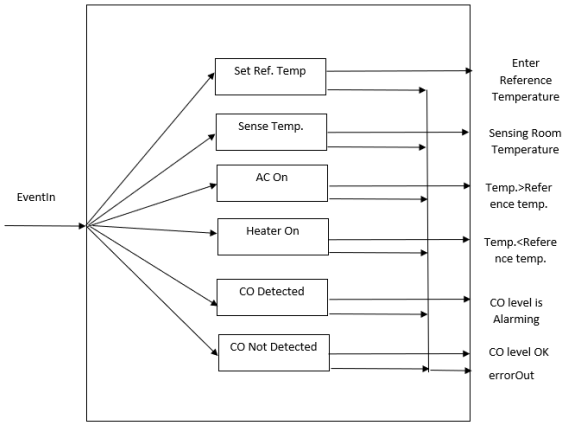



**Figure 19 Automatic Temperature Controller Coupled Model**

The formal specifications <X, Y, D, {Mi}, {Ii}, {Zij}, SELECT > for this coupled model are defined as follows:

X = {in};
Y = {Set Ref. Temp, Sensing Temp, Heater On, AC On, CO level OK, CO level Alarming};
D = {TempSet, TempSensed, HeaterOn, AcON, CODetectOK, CODetectAlarm};

I(TempSet) = I(TempSensed) = I(AcON) = I(HeaterOn) = I(CODetectOK) = I(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,

SELECT:
TempSet > TempSensed > HeaterOn > CODetectOK > CODetectAlarm > AcON

Here TempSet, TempSensed, HeaterOn, AcON, CODetectOK and CODetectAlarm are atomic models and {SetTemperature, a numerical value of Sensed temperature} are valid Events.

### C. Online Shopping Payment

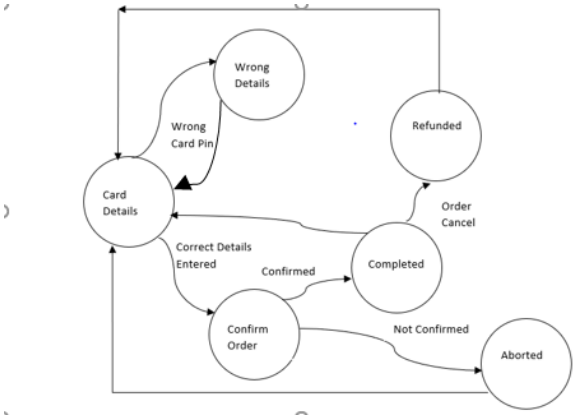



**Figure 20 State Diagram of Online Payment System**

This application is based on Moore model and is a simple implementation of an online payment portal. After the shopping cart is finalized the shopping website takes the user to a payment portal where it asks for the payment mode and details. In this application it has been assumed that the card is the choice of payment mode and the only detail that the user needs to fill is the card pin number to proceed with the payment. In the initial state after the card details (card pin) is entered this state checks if the pin is valid by checking against a preset value. If it passes the check it goes to the state where it asks the user to confirm the payment otherwise it goes to the state where it outputs Wrong Details and returns back to initial state. If the user confirms the order, the next state is completed. The user may also cancel the order after payment in which case it goes to Refunded state. If the user does not confirm the payment/order, the state transitions to Aborted and returns to initial state.

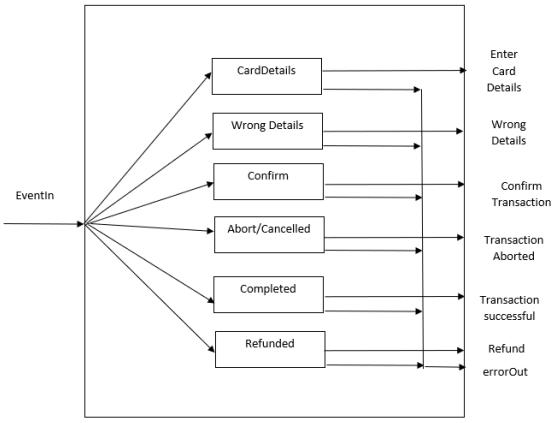



**Figure 21 Online Payment System Coupled Model**

The formal specifications <X, Y, D, {Mi}, {Ii}, {Zij}, SELECT > for this coupled model are defined as follows:

X = {in};
Y = {EnterCardDetails, WrongDetailsEntered, ConfirmTransaction, TransactionAborted, Transaction completed, Refunded};
D = {CardDetails, WrongDetails, ConfirmOrder, Completed, Aborted, Refunded};

I(CardDetails) = I(ConfirmOrder) = I(Completed) = I(Aborted),
I(Completed) = I(Refunded),
I(CardDetails)= (WrongDetails);

Z(CardDetails) = ConfirmOrder, Z(CardDetails) = WrongDetails, Z(ConfirmOrder)=Completed, Z(ConfirmOrder)=Aborted, Z(Completed) = Refunded, Z(Refunded)=CardDetails,
(Aborted)= CardDetails;
Z(WrongDetails) = CardDetails, Z(Completed) = CardDetails,

SELECT:
CardDetails>WrongDetails>ConfirmOrder>
Completed > Aborted > Refunded

Here EnterCardDetails, WrongDetailsEntered, ConfirmTransaction, TransactionAborted, Transaction completed and Refunded are atomic models and

{Numeric value for Card pin, Confirm, Not Confirmed, Cancel } are valid Events.

### D. Online Library Portal

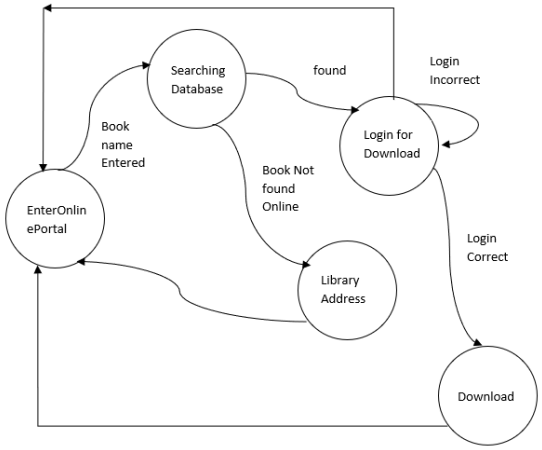



**Figure 22 Online Library Portal Coupled Model**

In the application for a simple Online Library portal the Mealy model is used. The user enters the Online portal in the University Website and enters the Title of the book/journal. It then transitions to "Searching" state. If the item is found in the predetermined database, the portal prompts for Login Details. The correct Login is a predetermined Password. Password fed to the system is checked against this and on a successful match it goes to Download state and gives out a link. If the password does not match the predetermined password, the portal prompts the user to type the correct password. On the other hand, if the book is not found in the database, the portal gives out the address of the Library Building to check for further query of the item. It can be noted that the output of the states now depends on the event and not just the states.

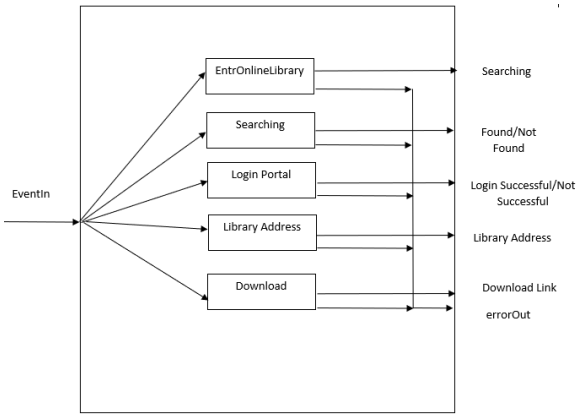



**Figure 23 Online Library Portal Coupled Model**

The formal specifications <X, Y, D, {Mi}, {Ii}, {Zij}, SELECT > for this coupled model are defined as follows:

X = {in};
Y = {Searching, Found/Not Found, Login Successful/NotSuccessful, Library Address, Download Link};
D = {EnterOnlineLibrary, Searching, LoginPortal, LibraryAddress, Download};

I(EnterOnlineLibrary) = I(Searching) = I(LoginPortal) = I(Download) = I(LibraryAddress)

Z(EnterOnlineLibrary) = Searching;
Z(Searching) = LoginPortal, Z(Searching) = LibraryAddress;
Z(LoginPortal) = LoginPortal, Z(LoginPortal) = Download

Z(Download)=EnterOnlineLibrary, Z(LibraryAddress) = EnterOnlineLibrary;

SELECT:
EnterOnlineLibrary > Searching > LoginPortal > Download > LibraryAddress

Here EnterOnlineLibrary, Searching, LoginPortal, LibraryAddress and Download are atomic models and {Search Book, Book Name, Login password} are valid Events.

### E. Elevator System

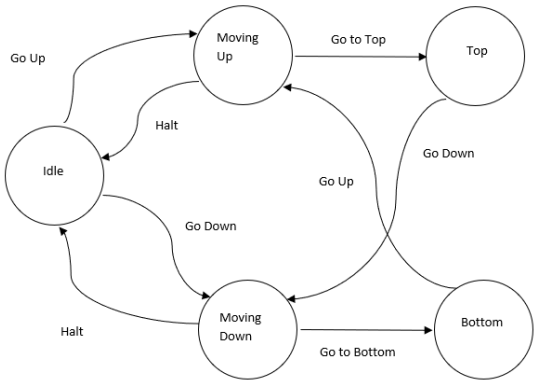



**Figure 24 State Diagram for Elevator System.**

A simple Elevator FSM is developed using Mealy Model. A simplified adaptation of the movement of elevator is captured in this application. There will be five states: Idle, Moving Up, Moving Down, At Top, At Bottom. Based on the event and on what state it is in the elevator moves up, down or halts. If the elevator is at Idle state it can either output MovingUp or MovingDown based on the event. Once it starts moving up it can go to the Top or halt based on the

event. Similarly, if it starts moving down it can either go all the way to the Bottom or halt based on the event. It can also be observed that once it reaches Top or Bottom it accepts only events to go down or up respectively.

A simplified coupled model is given in Figure 25.

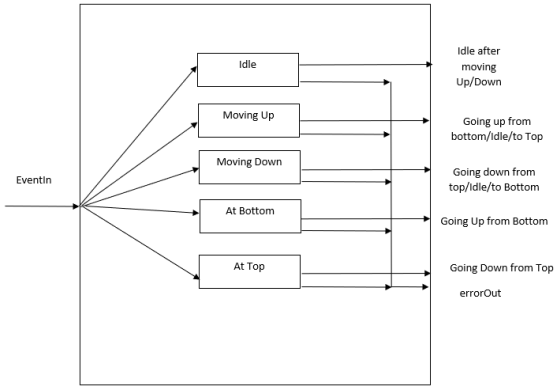



**Figure 25 Elevator System Coupled Model**

The formal specifications <X, Y, D, {Mi}, {Ii}, {Zij}, SELECT > for this coupled model are defined as follows:

X = {in};
Y={Going Up from Idle/Bottom, Going Down from Top/Idle, Going to Top, Going To Bottom, Halt};
D = {Idle, MovingUp, MovingDown, Top, Bottom};

I(Idle) = I(MovingUp) = I(Top) = I(MovingDown) = I(Bottom) = Idle;

Z(Idle) = MovingUp;
Z(MovingUp) = Idle, Z(MovingUp) = Top;
Z(MovingDown) = Idle, Z(MovingDown) = Bottom
Z(Top)=MovingDown, Z(Bottom) = MovingUp;

SELECT:
Idle > MovingUp > MovingDown > Top > Bottom

Here Idle, MovingUp, MovingDown, Top and Bottom are atomic models and {Go up, Go down, halt} are valid Events.

## F. Toaster System

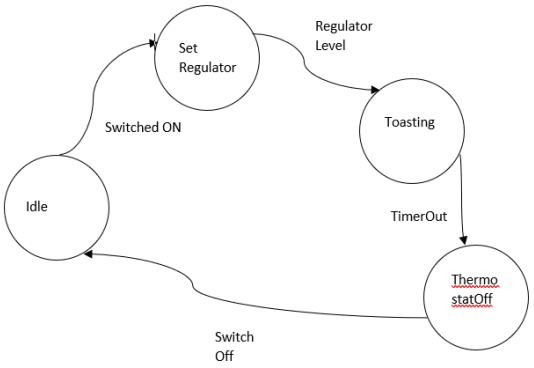



**Figure 26 State Diagram for Toaster System.**

Finite state machine for the operation of a simple toaster has been designed in this application based on Mealy Model. There are four states: Idle, Set Regulator, Toasting and Thermostat switch off. The toaster is signaled to switch on when idle. Then the regulator is set to a certain level in the next state. The output of the "Set Regulator" state is dependent on the level of Regulator event. For example, level 1 outputs regulator level 1, and so forth. This state transitions into toasting state and waits for the timer to go out. This event is the timer out event. After the timer goes off the Thermostat switch turns off. After this state if user decides to turn of the device, it switches off the toaster and that takes it back to idle state.

A simple DEVS model is presented in the Figure 27. The transitions have been omitted in this diagram to simplify the diagram. The transitions can be seen from the formal definition and the state diagram.

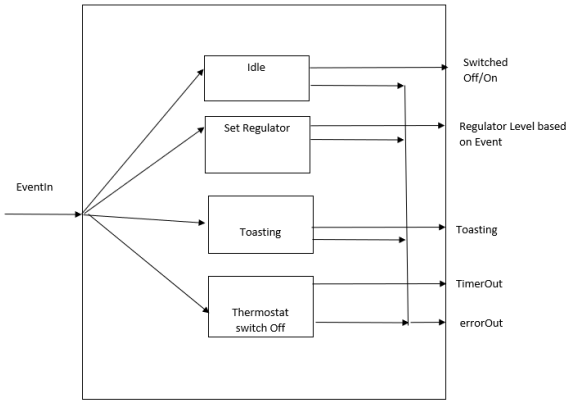



**Figure 27 Toaster System Coupled Model**

The formal specifications <X, Y, D, {Mi}, {Ii}, {Zij}, SELECT > for this coupled model are defined as follows:

X = {in};
Y = {Switched On, Regulator level, Toasting, Timer out, Switched Off};
D = {Idle, SetRegulator, Toasting, ThermostatOff};

I(Idle) = I(SetRegulator) = I(Toasting) = I(ThermostatOff) = Idle;

Z(Idle) = SetRegulator;
Z(SetRegulator) = Toasting, Z(Toasting) = ThermostatOff;
Z(ThermostatOff) = Idle;

SELECT:
Idle > SetRegulator > Toasting > ThermostatOff

Here Idle, SetRegulator, Toasting and ThermostatOff are atomic models and {Switch On, SetRegulator level1/level2/level3, SwitchOff} are valid Events.

## 5. Simulation Results of the Atomic Models and the Applications

Test cases for Moore and Mealy Atomic Models have been developed to test the working of these models. To test the Atomic Models an application implementation has been picked up for testing the results. For example, for Moore Atomic Model testing the Vending Machine callbacks have been used and similarly for Mealy Atomic Model testing, Elevator callbacks have been used.

### A. Test Results for Moore Atomic Model

For the Moore Atomic Model, the inputs have been provided for EventIn and TransitionIn. Based on the Vending Machine application the output message and the state outputs are as follows:

**Inputs for eventIn**
00:00:10 one
00:00:30 Gibberish

**Inputs for TransitionIn**
00:00:20 Idle

We can see from the logs below that for Event "One" the Atomic Model sends out "one" in the transitionOut which is the accepted behavior.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char,std::char_traits<char>, std::allocator<char> > >::**out: {one}] generated by model input_reader_EventIn***
*00:00:13:000*
*[MooreFSM_defs::stateOutput:{InsertLoonieOrToonie}, MooreFSM_defs::**transitionOut:{one}**, MooreFSM_defs::errorOut: {}] generated by model **MooreFSM1***

For the case when transitionIn comes in the correct state becomes active as can be observed from the logs below.

***00:00:20:000***
*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char,std::char_traits<char>, std::allocator<char> > >::**out: {zero}] generated by model input_reader_TransitionIn***

***00:00:20:000***
*State for model MooreFSM1 is **State: zero & Output: InsertLoonieOrToonie & errorMsg: & Phase: active***

For event when a wrong event "Gibberish" comes in the Model throws out the error message. ("Try Again .." in this case)

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char,std::char_traits<char>, std::allocator<char> > >::out: **{Gibberish}] generated by model input_reader_EventIn***

*00:00:33:000*
*[MooreFSM_defs::stateOutput: {InsertLoonieOrToonie}, MooreFSM_defs::transitionOut: {zero}, MooreFSM_defs::errorOut: **{Try Again...}** generated by model MooreFSM1*

### B. Test Results for Mealy Atomic Model

For the Mealy Atomic Model, the inputs have been provided for EventIn and TransitionIn. Based on the Elevator System application the output message and the state outputs are as follows:

**Inputs for eventIn**
00:00:10 GoUp
00:00:30 Gibberish

**Inputs for TransitionIn**
00:00:20 Idle
It can be seen that for the Event "GoUp" transitionOut correctly sends out the next transition state in the Elevator application.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::out: **{GoUp}] generated by model input_reader_EventIn***
*00:00:13:000*
*[MealyFSM_defs::stateOutput: {Going Up..}, MealyFSM_defs::transitionOut: {**MovingUp**}, MealyFSM_defs::errorOut: {}] generated by model **MealyFSM1***

For the transitionIn, the correct state becomes active:
***00:00:20:000***
*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::out: {Idle}] generated by model input_reader_TransitionIn*

***00:00:20:000***
*State for model MealyFSM1 is State: Idle & Output: & errorMsg: & Phase: active*

For wrong EventIn “Gibberish in the Model throws out the error message. (Invalid Input for this case)

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::out**: {Gibberish}]** *generated by model input_reader_EventIn*
*00:00:33:000*
*[MealyFSM_defs::stateOutput: {Not a Valid Input}, MealyFSM_defs::transitionOut: {Idle}, MealyFSM_defs::errorOut: **{Invalid Input}]** generated by model MealyFSM1*

### C. Analyzing Simulation results for Vending Machine

The below events were used to generate the results:
**Inputs for eventIn**
00:00:10 one
00:00:20 two
00:00:30 PushButton
00:00:40 one
00:00:50 one
00:01:00 one
00:01:10 two
00:01:20 PushButton
00:01:30 Four

Firstly, the transitions for these events are as expected. The transitions of the states due to a particular event can be checked to see if the next state is as per the design.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char,std::char_traits<char>, std::allocator<char> > >::**out: {two}] generated by model input_reader***
*00:00:23:000*
*[MooreFSM_defs::stateOutput:{LoonieInserted}, MooreFSM_defs**::transitionOut:{three}**, MooreFSM_defs::errorOut: {}]* ***generated by model one***
*00:00:26:000*
*[MooreFSM_defs::**stateOutput:{SmallDrink}**, MooreFSM_defs::transitionOut: {}, MooreFSM_defs::errorOut: {}] generated by **model three***

The next transition state if Event one comes in the state “one” is “three” and this dispenses the Small Drink if “PushButton” is the next Event. After dispensing the drink, it goes back to state “zero” to accept new order.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {PushButton}] generated by model input_reader***
*00:00:33:000*
***[MooreFSM_defs::stateOutput: {SmallDrink}, MooreFSM_defs::transitionOut: {zero}, MooreFSM_defs::errorOut: {}] generated by model three***
*00:00:36:000*

If invalid event occurs it gives out error message and goes to “zero” state. “Four” is one such invalid event as this model accepts only “one” and “two” (dollar).

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {Four}] generated by model input_reader***
*00:01:33:000*
*[MooreFSM_defs::stateOutput: {InsertLoonieOrToonie}, MooreFSM_defs::transitionOut: {zero}, MooreFSM_defs::**errorOut: {Try Again...}] generated by model zero***

### D. Analyzing the Simulation results for Automatic Temperature Control

**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

From the Events inputted above it can be observed that the temperature sensed is 70. In this application 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.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {70}] generated by model input_reader***
*00:00:23:000*
*[MooreFSM_defs::stateOutput:* ***{Sensing Room Temp.}, MooreFSM_defs::transitionOut: {HeaterOn}, MooreFSM_defs::errorOut: {}] generated by model TempSense***
*00:00:26:000*
*[MooreFSM_defs::stateOutput:* ***{Heater Turned ON}**, MooreFSM_defs::transitionOut: {CODetectAlarm}, MooreFSM_defs::errorOut: {}] generated by model HeaterOn*

In this application, turning the heater on is associated with Carbon Monoxide detection in a closed environment. This Carbon Monoxide level is a randomly generated number between 20-100ppm. For a level less than 50 the state transitions to CODetectOK otherwise it transitions to CODetectAlarm. One instance where the random number may have been below 50, this application transitions to CODetectOk from HeaterOn state.

*00:00:59:000*
*[MooreFSM_defs::stateOutput:* ***{Heater Turned ON}, MooreFSM_defs::transitionOut: {CODetectOK}, MooreFSM_defs::errorOut: {}] generated by model HeaterOn***

For Event indicating temperature above 80F the Air-Condition is turned on.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {90}] generated by model input_reader***
*00:01:03:000*
*[MooreFSM_defs::stateOutput: {Sensing Room Temp.}, MooreFSM_defs**::transitionOut: {AirCondOn}, MooreFSM_defs::errorOut: {}] generated by model TempSense***
*00:01:06:000*
*[MooreFSM_defs::stateOutput: **{AC ON}, MooreFSM_defs::transitionOut: {TempSet}, MooreFSM_defs::errorOut: {}] generated by model AirCondOn***

This application also puts a restriction on the temperature sensed. If the temperature sensed is either more that 120F or less than -40F it sends an error message to indicate the sensor is producing Invalid data.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > **>::out: {150}] generated by model input_reader***
*00:01:36:000*
*[MooreFSM_defs::stateOutput: {Sensing Room Temp.}, MooreFSM_defs::transitionOut: {TempSet}, MooreFSM_defs::errorOut: **{Invalid Temperature sensed}] generated by model TempSense***

### E. Analyzing the Simulation results for Online Payment FSM

**Inputs for eventIn**
00:00:10 109
00:00:20 Confirmed
00:00:40 Cancel
00:00:50 109
00:01:00 NotConfirmed
00:01:10 108
00:01:30 Gibberish

In the application for Online Payment System, the password is hard coded as 109, so the first event will let the initial state to transition to the next state which will ask for confirmation of the order.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {109}] generated by model input_reader***
*00:00:13:000*
*[MooreFSM_defs::stateOutput: {Enter Card Details}, MooreFSM_defs**::transitionOut: {ConfirmDetails}**, MooreFSM_defs::errorOut: {}] generated by model CardDetails*
*00:00:16:000*
***[MooreFSM_defs::stateOutput: {Confirm your details}, MooreFSM_defs::transitionOut: {}, MooreFSM_defs::errorOut: {}] generated by model ConfirmDetails***

On the other hand, if the eventIn doesn't match the set password, the CardDetails state transitions to Wrong Details state.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {108}] generated by model input_reader***
*00:01:13:000*
*[MooreFSM_defs::stateOutput: {Enter Card Details}, MooreFSM_defs::transitionOut: {**WrongDetails}, MooreFSM_defs::errorOut: {Invalid Card Detail}] generated by model CardDetails***

If the event "Confirm" comes, the transaction is successful otherwise it goes to Aborted state.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > **>::out: {Confirmed}] generated by model input_reader***
*00:00:23:000*
*[MooreFSM_defs::stateOutput: {Confirm your details}, MooreFSM_defs**::transitionOut: {Completed}**, MooreFSM_defs::errorOut: {}] generated by model ConfirmDetails*
*00:00:26:000*
*[MooreFSM_defs**::stateOutput: {Transaction Completed}**, MooreFSM_defs::transitionOut: {}, MooreFSM_defs::errorOut: {}] **generated by model Completed***

When the user enters "NotConfirmed":

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {NotConfirmed}] generated by model input_reader***
*00:01:03:000*
*[MooreFSM_defs::stateOutput: {Confirm your details}, MooreFSM_defs**::transitionOut: {Aborted}**, MooreFSM_defs::errorOut: {}] **generated by model ConfirmDetails***

User can also place a cancel request after transaction completion. In this case the state transition from "Completed" to "Refunded". Thereafter it goes back to the initial state "CardDetails".

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {Cancel}] generated by model input_reader***
*00:00:43:000*
*[MooreFSM_defs::stateOutput: {Transaction Completed}, MooreFSM_defs**::transitionOut: {Refunded}, MooreFSM_defs::errorOut: {}] generated by model Completed***
*00:00:46:000*
*[MooreFSM_defs::stateOutput: **{Transaction Refunded}, MooreFSM_defs::transitionOut: {CardDetails}, MooreFSM_defs::errorOut: {}] generated by model Refunded***

If any non-numeric value is inputted at "CardDetails" state it transitions to "WrongDetails" state and throws out an error message.

*std::allocator<char> > >::**out: {Gibberish}] generated by model input_reader***

*MooreFSM_defs::transitionOut:* ***{WrongDetails}, MooreFSM_defs::errorOut: {Invalid Card Detail}] generated by model CardDetails***

### F. Analyzing the Simulation results for Online Library Portal

**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.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::out:* ***{ParticlePhysics}] generated by model input_reader***
*00:00:23:000*
*[MealyFSM_defs::stateOutput:* ***{Enter Login Details to Access.}, MealyFSM_defs::transitionOut: {Login},*** *MealyFSM_defs::errorOut: {}] generated by model Searching*
*00:00:40:000*
*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::***out: {MaCoDrum}] generated by model input_reader***
*00:00:43:000*
*[MealyFSM_defs::***stateOutput: {Download Link}, MealyFSM_defs::transitionOut: {Download}, MealyFSM_defs::errorOut: {}] generated by model Login***

If a wrong password is entered, the state remains at "Login" and prompts for the correct password. Once the correct password is entered it transitions to "Download" state. Here the incorrect password is "MaCDum" instead of the correct one which is "MaCoDrum"

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::***out: {MaCDum}] generated by model input_reader***
*00:01:13:000*
*[MealyFSM_defs::stateOutput:* ***{Enter Correct Password},*** *MealyFSM_defs::transitionOut: {}, MealyFSM_defs::errorOut: {}] generated by model Login*
*00:01:20:000*
*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::***out: {MaCoDrum}] generated by model input_reader***
*00:01:23:000*
*[MealyFSM_defs::stateOutput:* ***{Download Link}, MealyFSM_defs::transitionOut: {Download}, MealyFSM_defs::errorOut: {}] generated by model Login***

### G. Analyzing the Simulation results for Elevator System

**Inputs for eventIn**
00:00:10 GoUp
00:00:20 GoToTop
00:00:40 GoDown
00:00:50 GoToBottom
00:01:00 GoUp
00:01:10 Halt
00:01:20 Error

In the application for Elevator System FSM, the states respond to user inputs in form of events. The "GoUp" event can be analogous to the button for a higher floor. The elevator starts moving up after this event until "halt" event makes the elevator transition to "Idle" state.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::***out: {GoUp}] generated by model input_reader***
*00:00:13:000*
*[MealyFSM_defs::stateOutput:* ***{Going Up..},*** *MealyFSM_defs::transitionOut:* ***{MovingUp}****, MealyFSM_defs::errorOut: {}] generated by model Idle*
*00:00:16:000*
*[MealyFSM_defs::stateOutput: {}, MealyFSM_defs::transitionOut: {},* ***MealyFSM_defs::errorOut: {}] generated by model MovingUp***

Similarly, "GoDown" event can be a button press for a lower floor and the elevator start moving down until "halt" event makes the elevator transition to "Idle" state.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::***out: {GoDown}] generated by model input_reader***
*00:00:43:000*
*[MealyFSM_defs::stateOutput: {Going Down from Top..}, MealyFSM_defs::transitionOut:* ***{MovingDown}, MealyFSM_defs::errorOut: {}] generated by model Top***

"GoToTop" and "GoToBottom" are analogous to pressing the buttons for the extreme ends of the building. These states can only be reached once the state transitions from "Idle" to "MovingUp" or "MovingDown" respectively.

*std::allocator<char> > >::***out: {GoToTop}] generated by model input_reader***
*00:00:23:000*
*[MealyFSM_defs::***stateOutput: {Going To Top..},*** *MealyFSM_defs::***transitionOut: {Top},*** *MealyFSM_defs::errorOut:* ***{}] generated by model MovingUp***

*std::allocator<char> > >::***out: {GoToBottom}] generated by model input_reader***

*00:00:53:000*
*[MealyFSM_defs::**stateOutput: {Going To Bottom..}**, MealyFSM_defs::**transitionOut: {Bottom}**, MealyFSM_defs::errorOut: {}] **generated by model MovingDown***

"Halt" event is simply when the elevator stops or user presses stop button at any floor. After "halt" event the elevator reaches the "Idle" state.

*std::allocator<char> > >::**out: {Halt}] generated by model input_reader***
*00:01:13:000*
*[MealyFSM_defs::**stateOutput: {Halting After Moving Up}**, **MealyFSM_defs::transitionOut: {Idle}**, **MealyFSM_defs::errorOut: {}] generated by model MovingUp***
*00:01:16:000*
*[MealyFSM_defs::stateOutput: {}, MealyFSM_defs::transitionOut: {}, MealyFSM_defs::errorOut: **{}] generated by model Idle***

### H. Analyzing the Simulation results for Toaster Device

**Inputs for eventIn**
00:00:10 On
00:00:20 Regulator1
00:00:40 TimerOut
00:00:50 SwitchOff
00:01:00 On
00:01:10 Regulator2
00:01:20 TimerOut
00:01:30 Regulator4

The application to simulate Toaster FSM is a takes few inputs to start the transitions between the states. Here, the regulator level can be set according to the input by the user. The output of the state "Set Regulator" depends on what input has been provided.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {Regulator1}] generated by model input_reader***
*00:00:23:000*
*[MealyFSM_defs::**stateOutput: {Set To Regulator Level 1}**, MealyFSM_defs::transitionOut: {Toasting}, MealyFSM_defs::errorOut: {}] **generated by model SetRegulator***

After the Regulator is set it transitions to "Toasting" state which will toast the food item till timer expires. After the timer-out the state transitions from Toasting to Thermostat switch off which basically ejects the food item.
*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {TimerOut}] generated by model input_reader***
*00:00:43:000*
*[MealyFSM_defs::**stateOutput: {Toasting done}**, MealyFSM_defs::**transitionOut: {ThermostatOff}**, MealyFSM_defs::errorOut: **{}] generated by model Toasting***

After the Thermostat switching off, the user may completely turn off the device. This transition brings the device back to "Idle" state.

*[cadmium::basic_models::pdevs::iestream_input_defs<std::__cxx 11::basic_string<char, std::char_traits<char>, std::allocator<char> > >::**out: {SwitchOff}] generated by model input_reader***
*00:00:53:000*
*[MealyFSM_defs::**stateOutput: {Switching Off the Toaster}**, MealyFSM_defs::**transitionOut: {Idle}***

## 6. Conclusion

This Project dealt with creating Moore/Mealy Atomic Models and building applications based on these models. The idea was to implement and extend the FSM Library previously written in CD++. Previously only numeric data types could be used to represent states. With Cadmium any data type can be used to define the state variables and message passing between states. Cadmium allows to define user defined data structure and use them in the applications. Further enhancements can be made to the model by including a queue model to store the events and handle each element 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.

Cadmium provides a clean and efficient platform to simulate DEVS model. The design is simple yet elegant to deal with building any complex DEVS Model. By observing the design pattern and language features used it can be seen that 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.

## References


[1] Tao Zheng; Gabriel A. Wainer: Implementing Finite State Machines Using the CD++ toolkit

[2] Gabriel A. Wainer: Discrete-event modeling and simulation: a practitioner's approach

[3] Adamu Murtala Zungeru; Mmoloki Mangwala; Joseph Chuma; Baboloki Gaebolae; Bokamoso Basutli: Design and simulation of an automatic room heater control system