AGENT-BASED CONTROL OF ELEVATOR SYSTEM IN EMBEDDED DEVS

SYSC5906 - Project Report of Directed Studies

Department of Systems and Computer Engineering
Carleton University

Directed Studies Project Submission to: Prof. Babak Esfandiari

Project Report Submitted by:

Navneet Kaushal
Student Id: 101094963
Master of Engineering, Electrical and Computer Engineering,
Department of System and Computer Engineering,
Carleton University

Table of Contents

Agent-based Control of Elevator System in E-DEVS

Navneet Kaushal
Systems and Computer Engineering, Carleton University, 1125 Colonel By Dr., Ottawa, ON, Canada, K1S 5B6
navneetkaushal@cmail.carleton.ca

***Abstract*— This project report illustrates the model and implementation of an Agent-based Elevator Control System using Discrete Event Methodology for Embedded Systems. The model is implemented with the help of an ECD-Boost simulator. The aim is to combine the simulation method with agent-based models and build a library of an intelligent elevator controller.**

Keywords*— *Elevator Control, Reactive Agent, Discreet Event Methodology for Embedded Systems, Agent-based Control.

I. BACKGROUND

A. *Discreet Event System Specification (DEVS)*

DEVS is a timed event system for modelling and analyzing discreet event dynamic systems. A real system modelled using DEVS is composed of atomic and coupled models. DEVS is a methodology to specify systems whose state can change upon reception of an external input event or due to expiration of a time delay. It handles the complexity of the system by breaking the higher-level components into simpler elements.[3]

B. *DEVS Formalism*

Any real system modelled using DEVS is composed of atomic and coupled models. Atomic models are defined as

$$M = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta >$$

Where X is the set of input events, Y is the set of output events, S is the set of sequential states, δext is the external state transition function, δint is the internal state transition function, λ is the output function and ta is the time advance function as shown in Fig 1. A DEVS model stays in a state S for a time ta in absence of an external event. On expiration of ta the model outputs the value λ through a port Y and changes to a new state given by δint. This transition is called an internal transition. If there is a reception of an external event, δext determines its new state. If ta is infinite, then s is said to be in passive state.[3]

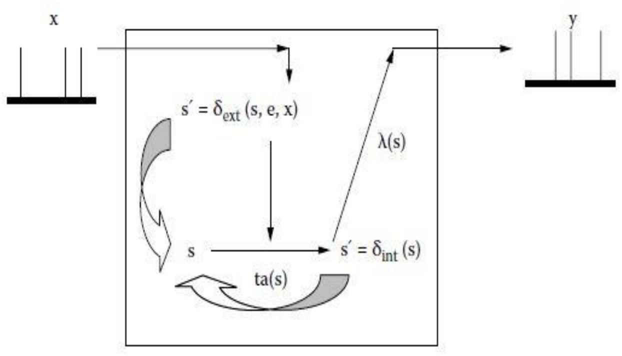

Fig. 1 DEVS atomic model. Image courtesy: [3]

A DEVS coupled model is composed of several atomic models. It is defined as:

$$N = < X, Y, D, \{M_i\}, EIC, IC, EOC, Select >$$

Where X is the set of input events, Y is the set of output events, D is the set of component names, M_i is atomic or coupled model, EIC is the set of external input couplings, EOC is the set of external output couplings, IC is the set of internal couplings and Select is the tiebreaker function. EIC, IC, and EOC are also represented in Fig 2 of a DEVS atomic model.[3]

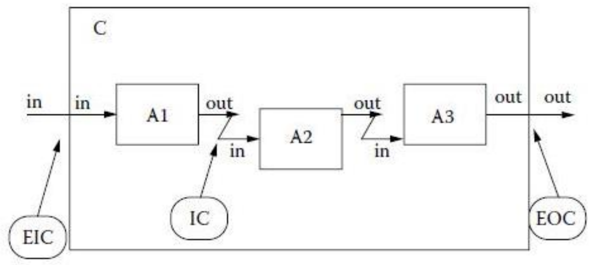

Fig. 2 DEVS atomic model. Image courtesy: [3]

C. *DEMES*

Discreet Event Modelling of Embedded Systems is an approach based on DEVS that allows models to be used throughout the development cycle. It can be seen in Fig 3. DEMES enable the development of embedded systems. Initially a System of Interest (SOI) is defined using DEVS. Once the system is defined, model checking can be done for validation of model. The model is then used to run DEVS simulations and derive test cases. The model is then incrementally moved to the target platform.[2]

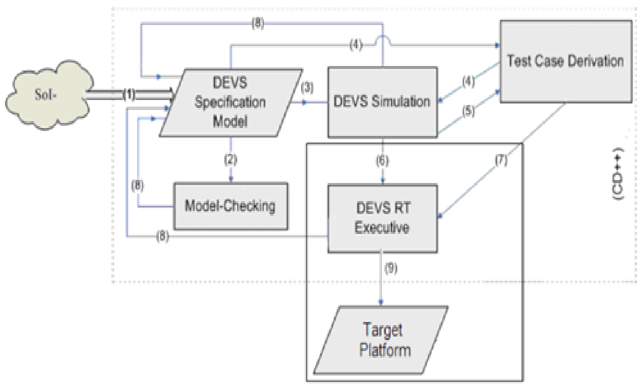

Fig. 3 DEMES Development cycle. Image courtesy: [2]

D. *ECDBoost*

It is a real time application of CDBoost. CDBoost is a simulator intended to work with DEVS models. The time advance function of CDBoost is taken by it and implemented in real time.[1]

E. Agent

Agent technology is a rapidly growing subdiscipline of computer science on the borderline of artificial intelligence and software engineering that studies the construction of intelligent systems. It is centered around the concept of an (intelligent/rational/autonomous) agent. An agent is a software entity that displays some degree of autonomy; it performs actions in its environment on behalf of its user but in a relatively independent way, taking initiatives to perform actions on its own by deliberating its options to achieve its goal(s).[4]

F. Reactive Agent

Reactive agent is an agent that perceives its environment and reacts to it in a timely fashion.

II. LITERATURE REVIEW

Multi DEVS allows an explicit modeling of the environment and its role in providing information and enforcing constraints for the situated agents. The formalism supports representing the environment as first-order abstraction and explicit building block of multi-agent systems. Nested models with a dynamic behavior of their own at each level, dynamic (yet explicitly defined) model interfaces, an intentional definition of interactions between models, and a combination of event processing, value couplings, and invariants to tie the different levels of nesting. Environments play an important role in multi-agent systems. They present the context agents operate in. When testing multi-agent systems by simulation, the environment and partly agents have to be modeled.[5]

Java Based Agent Modeling Environment for Simulation integrates agents within discrete event simulation. Based on DEVS it inherits its modular model design. Its concepts are aimed at facilitating the reuse of existing model libraries, they are also carefully chosen to hamper neither a possible inter-operation with other DEVS based simulation systems nor the flexible creation of test scenarios for multi-agent systems. An agent's "output" activities are decoupled from receiving external events. Both, the perception of events and the reaction directed to the environment, interact via state and the time advance function.[6]

Macro model of control for a decision system is a quite simple and efficient approach to model decision system. It describes decision and data flow in the whole system. Decisions are modeled by events and data by response on events or by information flow. Then this model can be implemented in multi-agent technology, where agents communicate with each other by messages. Messages in multi-agent system represent events in macro model. Data flow in macro model can be represented by knowledge in multi-agent system.[7]

Embedding controllers within autonomous software agents, called control agents, provides a basis for autonomous control of distributed sensor/actuator systems. These control agents are autonomous because they do not require human supervision or monitoring in order to function. However, autonomy does not imply that control agents are oblivious to humans; rather, control agents are responsive to information provided by operators as needed and react to operator demands and objectives. Information sharing among control agents will not always be enough to satisfy the control needs of all systems. Control agents that produce information that is co-useful (e.g., agent A produces information useful to agent B and vice versa) are called co-dependent.[8]

Elevator group control is a difficult domain posing a combination of challenges not seen in most multi-agent learning research. Elevator systems operate in high-dimensional continuous state spaces and in continuous time as discrete event dynamic systems. Their states are not fully observable, and they are non-stationary due to changing passenger arrival rates. The oldest relay-based automatic controllers used the principle of collective control, where cars always stop at the nearest call in their running direction. Car assignment occurs when a hall button is pressed. The car assigned is the one that minimizes a weighted sum of predicted wait time and travel time.[9]

The recent advances in Internet of Things (IoT) technologies, have enabled a plenty of devices, such as light, HVAC (heating, ventilating and air conditioning) system to be accessed by wireless network. However, elevator, working as a vertical transporting device, still stands alone from the communication infrastructure in the building and required to be more easily accessible. According to IBM’s survey of Smarter Buildings Study, the result of investigation and calculation on the 6,486 office workers in 16 U.S. cities showed that the total amount of over 92 years was wasted on waiting for the elevators in 2009. Moreover, the problem is constantly evolving and the challenges to be faced are growing with the ever-growing height of buildings. an elevator call pre-registration system – PrecaElevator for reducing passengers’ waiting time by an efficient, intuitive approach. PrecaElevator enables an ordinary elevator to perform as a smart object with the IoT technologies and location aware on passengers by integrating with the BLE-based localization framework, thus providing passengers with the possibility of zero waiting time.[10]

III. MODELS

The model that is shown as part of this directed study is Elevator Control. It is designed to move, and halt one of the two elevators system based on the button pressed from any floor of an eight-floor building. The two elevators are to be considered as two agents that communicate through each other and operate based on Agent Logic. Conceptually, it is a type of reactive agent that reacts based on some event occurred in its environment.

A. Conceptual Model Description

Since the simulation model designed is the software replica of real time elevator system, for now the inputs to be given to the system are coming from various input text files. So, the actual press buttons to start, call, or stop the elevators are written as input commands in these input text files. Similarly, the output that is supposed to be sent to the hardware (motors of elevators) is also stored in an output text file.
Elevator Control is the main model that receives the push button inputs in the form of text command from different input files and gives the output in the form of text command that would eventually control the motion of real elevators. AgentLogic file, directly associated with Elevator Control, decides the algorithm to be followed before taking the action to call the elevators. Below are the different atomic and coupled models involved in the hierarchy of Elevator Control model.

B. Input Atomic Models

All the input ports of Elevator Control are made with an input atomic model i.e. Input Event Stream. There are total nine atomic models of Input Event Stream – one is for start/stop input file (the name of model is start), and other eight are for input files of buttons from eight floors (floor0 to floor7). The main purpose of Input Event Stream models is to read the input text files and to convert them as real inputs that Elevator Controller is expecting.

START – Input Event Stream

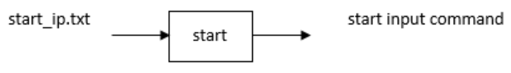

Fig. 4 Input Atomic Model: start

When the real elevator system is to be used, these models would not be needed as the hardware would send the real signal by itself through push buttons. As shown in Fig 4, this atomic model is reading start_ip.txt file and sending the input commands to the controller. The data in the file contains the input port name, the input port value, and the real time at which the data is supposed to come as an input. The start input file, start_ip.txt, is also used to stop the controller using different input value.
Fig 5 shows floor0 as an Input Event Stream atomic model that reads floor0_ip.txt file and sends the input commands to the controller. Similarly, all other input files (floor1, floor2, .., floor9) are there that are not shown individually.

FLOOR0 – Input Event Stream

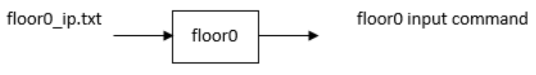

Fig. 5 Input Atomic Model: floor0

C. Atomic Models

ELEVATOR CONTROLLER

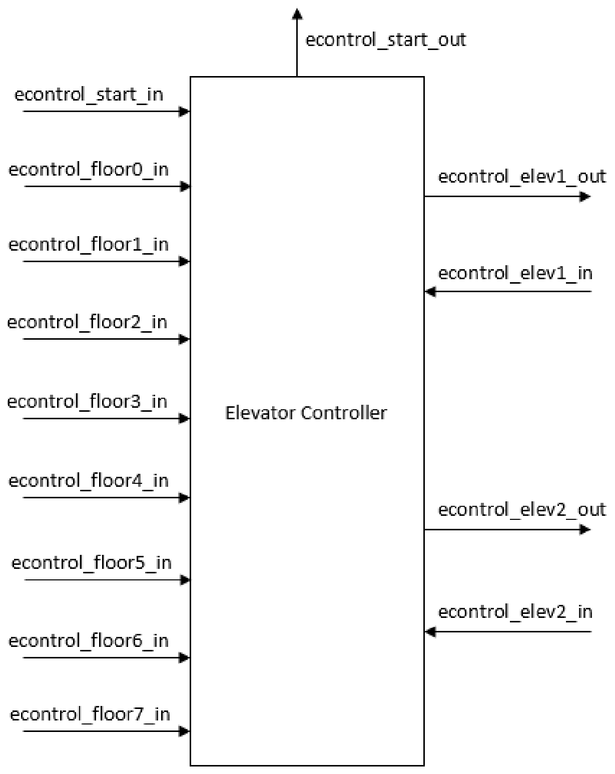

Fig 6. Atomic Model Elevator Controller

Elevator Controller activates other input ports on receiving the input from start Input Event Stream model that is also one of its ports. The activated ports are now ready to take inputs. Without getting appropriate start input (econtrol_start_in) the elevator would not be able to receive any other inputs. The eight other inputs (econtrol_floor0_in, econtrol_floor1_in, etc.) are representing inputs from requests from different floors of the building. The atomic model Elevator Controller gives the output to the atomic models Elevator1 and Elevator2. The outputs of the model are econtrol_elev1_out, econtrol_elev2_out, and econtrol_start_out that represents the input signals to operate Elevator1, Elevator2 and to stop the elevator respectively.

X = { econtrol_start_in, econtrol_floor0_in, econtrol_floor1_in, econtrol_floor2_in, econtrol_floor3_in, econtrol_floor4_in, econtrol_floor5_in, econtrol_floor6_in, econtrol_floor7_in, econtrol_elev1_in, econtrol_elev2_in }

Y = { econtrol_elev1_out, econtrol_elev2_out, econtrol_start_out }

S = { WAIT_DATA, IDLE, PREP_RX, TX_DATA, PREP_STOP }

$\delta_{ext}()$:

```
If ( State = IDLE & Input = start )
{
    State = WAIT_DATA
}
Else If ( Input = stop )
{
    State = PREP_STOP
}
If ( State = WAIT_DATA )
{
    If ( Floor 0 button pressed ) {
        floorNum = 0;
    }
    Else If ( Floor 1 button pressed ) {
        floorNum = 1;
    }
    Else If ( Floor 2 button pressed ) {
        floorNum = 2;
    }
    Else If ( Floor 3 button pressed ) {
        floorNum = 3;
    }
    Else If ( Floor 4 button pressed ) {
        floorNum = 4;
    }
    Else If ( Floor 5 button pressed ) {
        floorNum = 5;
    }
    Else If ( Floor 6 button pressed ) {
        floorNum = 6;
    }
    Else If ( Floor 7 button pressed ) {
        floorNum = 7;
    }
}
```

$\delta_{int}()$: { Passivate; }

λ():

//*Algorithm to decide which elevator to call based on AgentLogic*

{ Send Floor number & Token (which elevator to call) command to the output port }

ELEVATOR1

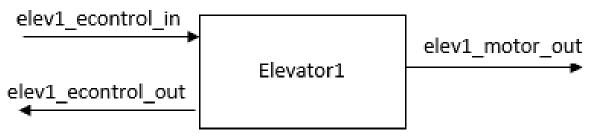

Fig. 7 Atomic Model: Elevator1

Elevator1 receives input from the Elevator Controller through elev1_econtrol_in and gives output command to its motor through elev1_motor_out as shown in Fig 7. This command is stored in a form of message in ElevatorOutput.txt file. And further when running the real elevator, this same command would be sent to the motor of the elevator1.

X = { elev1_econtrol_in }

Y = { elev1_ motor_out, elev1_econtrol_out }

S = { PREP_START, WAIT_DATA, PREP_MOVE_UP, PREP_MOVE_DOWN, MOVE_UP, MOVE_DOWN, PREP_STOP, IDLE }

$\delta_{ext}()$:

```
If ( State = WAIT_DATA )
{
    If ( floorNum > currentFloor_elev1 ) {
        State = PREP_MOVE_UP;
    }
    Else If ( floorNum < currentFloor_elev1 ) {
        State = PREP_MOVE_DOWN;
    }
    If ( floorNum = currentFloor_elev1 ) {
        //Msg: elevator is already at the floor
    }
}
```

$\delta_{int}()$:

```
If ( State = PREP_MOVE_UP ){
    State = MOVE_UP;
}
Else If ( State = PREP_MOVE_DOWN ){
    State = MOVE_DOWN;
}
Else If ( State = MOVE_UP ){
    currentFloor_elev1++ ;
    If ( floorNum > currentFloor_elev1 ) {
        State = PREP_MOVE_UP;
    }
}
Else If ( State = MOVE_DOWN ){
    currentFloor_elev1-- ;
    If ( floorNum < currentFloor_elev1 ) {
        State = PREP_MOVE_DOWN;
    }
}
{ Passivate; }
```

λ():

```
If ( State = PREP_MOVE_UP ){
    Send command to elevator1 to move up.
}
Else If ( State = PREP_MOVE_DOWN ){
    Send command to elevator1 to move down.
}
Else If ( State = PREP_STOP or State =  MOVE_UP or
State = MOVE_DOWN ){
    Send command to elevator1 to stop.
}
```

ELEVATOR2

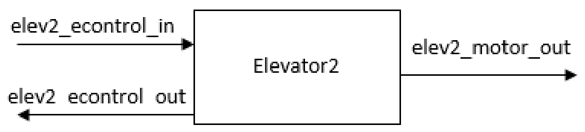

Fig. 8 Atomic Model: Elevator2

Like previous atomic model, Elevator2 also receives input from the Elevator Controller through its port elev2_econtrol_in and gives output command to its motor through elev2_motor_out as shown in Fig 8. This command is stored in a form of message in the same output file ElevatorOutput.txt. And further when running the real elevator, this command would be sent to the motor of the elevator2.

X = { elev2_econtrol_in }

Y = { elev2_ motor_out, elev2_econtrol_out }

S = { PREP_START, WAIT_DATA, PREP_MOVE_UP, PREP_MOVE_DOWN, MOVE_UP, MOVE_DOWN, PREP_STOP, IDLE }

$\delta_{ext}()$:

```
If ( State = WAIT_DATA )
{
        If ( floorNum > currentFloor_elev2 ) {
              State = PREP_MOVE_UP;
        }
       Else If ( floorNum < currentFloor_elev2 ) {
              State = PREP_MOVE_DOWN;
        }
        If ( floorNum = currentFloor_elev2 ) {
              //Msg: elevator is already at the floor
        }
}
```

$\delta_{int}()$:

```
If ( State = PREP_MOVE_UP ){
        State = MOVE_UP;
}
Else If ( State = PREP_MOVE_DOWN ){
        State = MOVE_DOWN;
}
Else If ( State = MOVE_UP ){
        currentFloor_elev2++ ;
        If ( floorNum > currentFloor_elev2 ) {
              State = PREP_MOVE_UP;
        }
}
Else If ( State = MOVE_DOWN ){
        currentFloor_elev2-- ;
        If ( floorNum < currentFloor_elev2 ) {
              State = PREP_MOVE_DOWN;
        }
}
{ Passivate; }
```

$\lambda()$:

```
If ( State = PREP_MOVE_UP ){
        Send command to elevator2 to move up.
}
Else If ( State = PREP_MOVE_DOWN ){
        Send command to elevator2 to move down.
}
Else If ( State = PREP_STOP or State =  MOVE_UP or
State = MOVE_DOWN ){
        Send command to elevator2 to stop.
}
```

D. Coupled Model

TOP MODEL: ELEVATOR CONTROL

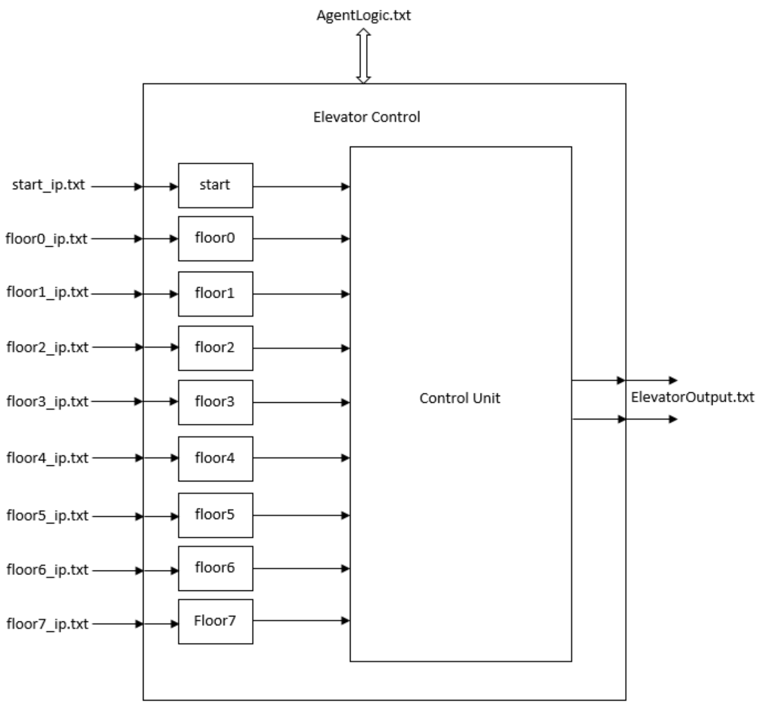

Fig. 9 Top Coupled Model: Elevator Control

The model shown in Fig 9 represents the complete top model of our elevator system i.e. Elevator Control (ElevControl).

X = { start_ip, floor0_ip, floor1_ip, floor2_ip, floor3_ip, floor4_ip, floor5_ip, floor6_ip, floor7_ip }

Y = { ElevatorOutput }

D = { start, floor0, floor1, floor2, floor3, floor4, floor5, floor6, floor7, ControlUnit }

EIC = { {{Start Input File, Self}, {Self, Start Input Atomic Model}}, {{Floor Input Files, Self}, {Self, Floor Input Atomic Models}}}

EOC = { {{elev1_output from ControlUnit, Self }, {Self, OutputFile}}, {{elev2_output from ControlUnit, Self }, {Self, OutputFile}} }

IC = { {start, ControlUnit}, {floor atomic models, ControlUnit} }

SELECT = { {start}, { floor0, floor1, floor2, floor3, floor4, floor5, floor6, floor7}, {ControlUnit} }

COUPLED MODEL: CONTROL UNIT

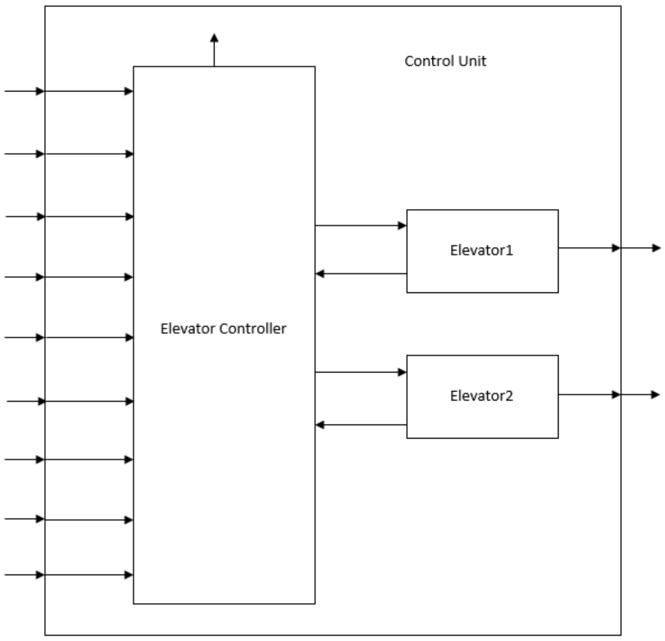

Fig. 10 Coupled Model: ControlUnit

The coupled model ControlUnit consists of three atomic models – Elevator Controller (econtrol), Elevator1 (elev1), and Elevator2 (elev2). This model is an internal coupled model of the top model ElevModel.

X = { start input, {floor inputs} }

Y = { elev1_out, elev2_out }

D = { econtrol, elev1, elev2}

EIC = { {{ start, Self }, {Self, econtrol}}, {{ floors, Self }, {Self, econtrol}} }

EOC = { {{elev1, Self }, {Self, ElevControl}}, {{elev2, Self}, {Self, ElevControl}} }

IC = { {econtrol, elev1}, {econtrol, elev2}, {elev1, econtrol}, {elev2, econtrol} }

SELECT = { econtrol, elev1, elev2 }

IV. IMPLEMENTATION

A. *Input Files*

There are total nine input files all together. Out of which one is to start or stop the elevator system, and other eight input files are to send the commands from different floors. All these input files are in place of real push buttons for the simulator, and so the content of the files also include the real time to operate. Nine input atomic models are created to read the input text files and convert it to the input signals needed for the simulation models. These input port models are created through Input Event Stream model that is located at *.vendor/input_event_stream.hpp*. The input file to start/stop the elevator system that is used is – start_ip.txt.

start_ip.txt

```
00:00:00:001:000 start 200
00:00:52:001:000 start 201
```

The format of the input file is <Time> <Port> <Value>

wherein time follows hour:min:sec:msec:micsec pattern. When the value is 200, it means the signal is sent to Elevator Controller to activate the other ports and elevator. Without giving **start 200**, the other inputs will not be activated, and system will not operate if after giving floor inputs to the system.

To stop the controller, again the same command is to be sent but with value **201**. The time associated is to inform at what real time after the execution, the elevator system should be started.

The input files floor0_ip.txt, floor1_ip.txt, floor2_ip.txt, floor3_ip.txt, floor4_ip.txt, floor5_ip.txt, floor6_ip.txt, floor7_ip.txt are used for floor0, floor1, floor2, floor3, floor4, floor5, floor6, floor7 respectively.

floor0_ip.txt

```
00:00:22:001:000 floor0 1
```

The format of this file is also same <Time> <Port> <Value>

Here, when the value is **1**, that means the elevator button associated to that floor is pressed and one of the elevators is supposed to go to that floor to receive the passenger. And the time suggests the real time at which the controller should receive this input.

B. *Output File*

The output file that is generated after successful execution of the simulation is ElevatorOutput.txt. Basically, it stores the action (up/down/stop) command with the real time stamp at which it would be sending the command to the hardware.

ElevatorOutput.txt

```
00:00:02:061:032
Requested Input: 5
Port: elevator2
Command(12=Up, 13=Down, 14=Stop): 12
```

The format in which the output message is to be displayed is written in *.input/eMessage.hpp* file. This output file shows the real time output data that contains the real time, the input value requested, which elevator is called, and what command value is given to that elevator port. The above output data is observed for instance, at time 00:00:02:061:032 for input value 5, elevator2 (Port) is used and it is asked to move up (Command = 12). When requested input is 201, that means the elevator would be stopped.

C. *Logic File*

AgentLogic.txt is a logic file that decides what algorithm is to be followed by the agent to call the elevators. It is also called as **rule-action** file because in this file, Reactive Agent function is described in form of rule-action.

The environment states of this system are: requested floor (floorNum), current floor of elevator1 (curr_flr1), and current floor of elevator2 (curr_flr2). Based on these values, the rule-action file is written. Basically, the rule-action has to decide what action the agent has to take depending on the different environment states, that's what makes it a reactive agent. The agent will decide which elevator is to be called depending upon the rule-action file. To reduce the complexity of the algorithm, the environment states (floorNum, curr_flr1, and curr_flr2) is filtered into two variables – d1 and d2 where d1 is the distance between requested floor (floorNum) and current floor of elevator1 (curr_flr1) and d2 is the distance between floorNum and curr_flr2.

$$d1 = |\,\text{floorNum} - \text{curr\_flr1}\,|$$

$$d2 = |\,\text{floorNum} - \text{curr\_flr2}\,|$$

There are three different rules –

(d1 > d2) => when elevator2 is closer to the requested floor,
(d1 < d2) => when elevator1 is nearer to the requested floor,
(d1 = d2) =>.when both elevators are at same distance.

During these rules which elevator to call is described as action of the agent, so there would be two actions e1 (calling elevator1), and e2 (calling elevator2). Different possible sets of rule-action that can be formed:

a) {(d1 > d2) : e2, (d1 < d2) : e1, (d1 = d2) : e2}
b) {(d1 > d2) : e1, (d1 < d2) : e2, (d1 = d2) : e2}
c) {(d1 > d2) : e2, (d1 < d2) : e1, (d1 = d2) : e1}
d) {(d1 > d2) : e1, (d1 < d2) : e2, (d1 = d2) : e1}

Rule-action a	Nearer elevator is to be called. And in case of conflict, elevator2 would be preferred.
Rule-action b	Farther elevator is to be called. And in case of conflict, elevator2 would be preferred.
Rule-action c	Nearer elevator is to be called. And in case of conflict, elevator1 would be preferred.
Rule-action d	Farther elevator is to be called. And in case of conflict, elevator1 would be preferred.

Table I. Rule-Action table of Agent Logic

The way to read or edit the AgentLogic.file –
For instance,

```
d1 > d2 : e2
d1 < d2 : e1
d1 = d2 : e2
```

$\updownarrow$ $\updownarrow$

Rules : Actions

So, depending upon the perceived environment states, a particular rule would be true and corresponding action would be performed by the agent. By changing the actions in text file as mentioned in set of rule-action, corresponding changes can be observed that is described in Table I.

D. *Code Snippets*

This section presents code snippets of the working model. All the relevant and essential part of code from different files would be presented.

MakeFile

This make-file differs to the one that requires to run the real time hardware, elevator in this case.

```
#This file generates the executable for the project i.e.
ElevatorControlAgent.exe
CPP     = g++
BIN     = ElevatorControlAgent
OBJECTS = main.o ./internal/eTime.o
CFLAGS=-std=c++11
INCLUDE_PATHS = -I. -I./internal -I./vendor -I./pdevslib
INCLUDE_USER_PATHS = -I./user_models
all: $(OBJECTS) $(CPP) -g -o $(BIN) $(OBJECTS)
main.o: main.cpp $(CPP) -g -c $(CFLAGS) $(INCLUDE_PATHS)
$(INCLUDE_USER_PATHS) main.cpp -o main.o
internal/eTime.o: $(CPP) -g -c $(CFLAGS) $(INCLUDE_PATHS)
$(INCLUDE_USER_PATHS) ./internal/eTime.cpp -o internal/eTime.o
clean:
rm -f $(BIN) *.o *~
-for d in internal; do (cd $$d; rm -f *.o *~ ); done
```

main.cpp

```
// Atomic models definition
auto econtrol = make_atomic_ptr<ElevatorController<Time,
Message>>();
auto elev1 = make_atomic_ptr<Elevator1<Time, Message>>();
auto elev2 = make_atomic_ptr<Elevator2<Time, Message>>();

//Coupled model definition
shared_ptr<flattened_coupled<Time, Message>> ControlUnit(new
flattened_coupled<Time, Message>(
      {econtrol,elev1,elev2},
      {econtrol},
      {{econtrol,elev1}, {econtrol,elev2}},
      {elev1, elev2}
));

//Top I/O port definition
shared_ptr<istringstream> pointer_iss1{ new istringstream{} };
pointer_iss1->str(readInput("input/start_ip.txt"));
auto start = make_atomic_ptr<input_event_stream<Time, Message,
Time, Message>, shared_ptr<istringstream>, Time>(pointer_iss1,
Time(0), [](const string& s, Time& t_next, Message& m_next)-
>void{ //parsing function
      m_next.clear();
      istringstream ss;
      ss.str(s);
      ss >> t_next;
      ss >> m_next.port;
      ss >> m_next.val.val;
      string thrash;
      ss >> thrash;
      if ( 0 != thrash.size()) throw exception();});
```

```
shared_ptr<istringstream> pointer_iss2{ new istringstream{} };
pointer_iss2->str(readInput("input/floor0_ip.txt"));
auto floor0 = make_atomic_ptr<input_event_stream<Time, Message,
Time, Message>, shared_ptr<istringstream>, Time>(pointer_iss1,
Time(0), [](const string& s, Time& t_next, Message& m_next)-
>void{ //parsing function
      m_next.clear();
      istringstream ss;
      ss.str(s);
      ss >> t_next;
      ss >> m_next.port;
      ss >> m_next.val.val;
      string thrash;
      ss >> thrash;
      if ( 0 != thrash.size()) throw exception();});
```

Similarly, other input atomic models floor1, .., floor7 are created.

```
//Top model definition
shared_ptr<flattened_coupled<Time, Message>> ElevControl (new
flattened_coupled<Time, Message>{
     {start, floor0, floor1, floor2, floor3, floor4, floor5,
floor6, floor7, ControlUnit},
     {},
     {{start, ControlUnit}, {floor0, ControlUnit}, {floor1,
ControlUnit}, {floor2, ControlUnit}, {floor3, ControlUnit},
{floor4, ControlUnit}, {floor5, ControlUnit}, {floor6,
ControlUnit}, {floor7, ControlUnit}},
     {ControlUnit} });

//Output File and Runner
Time initial_time = Time();
ofstream out_data("ElevatorOutput.txt");
runner<Time, Message> root(ElevControl, initial_time, out_data,
[](ostream& os, Message m){ os << m;});
Time end_time = Time(80,0,0,0);
end_time = root.runUntil(end_time);
```

ElevatorControl.hpp

```
void internal() noexcept {
   switch (_state){
      case PREP_STOP:
         _state = IDLE;
         _next = infinity;
         break;
      case PREP_RX:
      case TX_DATA:
         _state = WAIT_DATA;
         _next = infinity;
         break;
      case IDLE:
         _next = infinity;
   }
}

void external(const std::vector<MSG>& mb, const TIME& t)
noexcept {
   MSG msg = mb.back();
   if (msg.port == portName[econtrol_start_in]){
      if(_state == IDLE && msg.val.val == START_PROC){
         _state = PREP_RX;
         _next = scRxPrepTime;
      }
      else if (msg.val.val == STOP_PROC) {
         _state = PREP_STOP;
         _next = TIME::Zero;
      }
   }
   else if (msg.port == portName[econtrol_floor0_in]){
      if(_state == WAIT_DATA) {
         floor0 = static_cast<int>(msg.val.val);
         if(floor0 == 1){
             floorNum = 0;
             _state = TX_DATA;
             _next = scTxTime;
         }
      }
   }
```

Similarly, other logics for floor1, .., floor6 are written.

```
   else if (msg.port == portName[econtrol_floor7_in]){
      if(_state == WAIT_DATA) {
         floor0 = static_cast<int>(msg.val.val);
         if(floor0 == 7){
             floorNum = 7;
             _state = TX_DATA;
             _next = scTxTime;
         }
      }
   }

}
```

Elevator1.hpp

```
void internal() noexcept {
     switch (_state){
          case IDLE:
               _state = WAIT_DATA;
               _next = infinity;
               break;
          case MOVE_UP:
               curr_flr1=curr_flr1+1;
               if(curr_flr1< econtrol_input){
               _state = PREP_MOVE_UP;
               _next = startPrepTime;
               }
               else{
               _state = WAIT_DATA;
               _next = infinity;
               }
               break;
          case PREP_MOVE_UP:
               state = MOVE_UP;
               _next = startTime;
               break;
         case MOVE_DOWN:
               curr_flr1=curr_flr1-1;
               if(curr_flr1 > econtrol_input){
               _state = PREP_MOVE_DOWN;
               _next = startPrepTime;
               }
               else{
               _state = WAIT_DATA;
               next = infinity;
               }
               break;
          case PREP_MOVE_DOWN:
               _state = MOVE_DOWN;
               _next = startTime;
               break;
          case PREP_STOP:
               _state = WAIT_DATA;
               _next = infinity;
               break;
          default:
               break;
     }
}

void external(const std::vector<MSG>& mb, const TIME& t)
noexcept {
     MSG msg = mb.back();
     if(_state == WAIT_DATA ) {
          econtrol_input = static_cast<int>(msg.val.cmd);
          if (econtrol_input - curr_flr1 > 0) {
               _state = PREP_MOVE_UP;
               _next = startPrepTime;
          }
          else if (econtrol_input - curr_flr1 < 0) {
               _state = PREP_MOVE_DOWN;
               _next = startPrepTime;
          }
     }
     if(msg.val.cmd == STOP_PROC){
          _state = PREP_STOP;
          _next = TIME::Zero;
     }
}
```

The logic and functionality of **Elevator2.hpp** is same as Elevator1.hpp, just it is created for elevator2.

V. SIMULATION RESULTS

To discuss the simulation results of Agent-based Elevator Control model and evaluate its performance systematically, a case study based on some test inputs has been done and its output is observed. In the beginning, both the elevators are at floor0. And it waits for start input to begin.

Input	Output
00:00:00:001:000 start 200	//Elevator System is On. Ready to take inputs.

00:00:02:001:000
floor5 1
	`00:00:02:061:032`
	`Requested Input: 5`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 12`
	
	`00:00:03:061:032`
	`Requested Input: 5`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 14`
	
	`00:00:03:081:032`
	`Requested Input: 5`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 12`
	
	`00:00:04:081:032`
	`Requested Input: 5`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 14`
	
	`00:00:04:101:032`
	`Requested Input: 5`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 12`
	
	`00:00:05:101:032`
	`Requested Input: 5`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 14`
	
	`00:00:05:121:032`
	`Requested Input: 5`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 12`
	
	`00:00:06:121:032`
	`Requested Input: 5`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 14`
	
	`00:00:06:141:032`
	`Requested Input: 5`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 12`
	
	`00:00:07:141:032`
	`Requested Input: 5`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 14`

`00:00:18:000:000`
`floor2 1`
	`00:00:18:060:032`
	`Requested Input: 2`
	`Port: elevator1`
	`Command(12=Up, 13=Down, 14=Stop): 12`
	
	`00:00:19:060:032`
	`Requested Input: 2`
	`Port: elevator1`
	`Command(12=Up, 13=Down, 14=Stop): 14`
	
	`00:00:19:080:032`
	`Requested Input: 2`
	`Port: elevator1`
	`Command(12=Up, 13=Down, 14=Stop): 12`
	
	`00:00:20:080:032`
	`Requested Input: 2`
	`Port: elevator1`
	`Command(12=Up, 13=Down, 14=Stop): 14`

00:00:22:000:000
floor0 1
	`00:00:22:060:032`
	`Requested Input: 0`
	`Port: elevator1`
	`Command(12=Up, 13=Down, 14=Stop): 13`
	
	`00:00:23:060:032`
	`Requested Input: 0`
	`Port: elevator1`
	`Command(12=Up, 13=Down, 14=Stop): 14`
	
	`00:00:23:080:032`
	`Requested Input: 0`
	`Port: elevator1`
	`Command(12=Up, 13=Down, 14=Stop): 13`
	
	`00:00:24:080:032`
	`Requested Input: 0`
	`Port: elevator1`
	`Command(12=Up, 13=Down, 14=Stop): 14`

00:00:28:000:000
floor7 1
	`00:00:28:060:032`
	`Requested Input: 7`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 12`
	
	`00:00:29:060:032`
	`Requested Input: 7`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 14`
	
	`00:00:29:080:032`
	`Requested Input: 7`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 12`
	
	`00:00:30:080:032`
	`Requested Input: 7`
	`Port: elevator2`
	`Command(12=Up, 13=Down, 14=Stop): 14`

00:00:32:000:000 floor0 1	No action. //Msg: Elevator1 is already at the floor.

00:00:52:001:000
start 201
	`00:00:52:001:032`
	`Requested Input: 201`
	`Port: elevator1`
	`Command(12=Up, 13=Down, 14=Stop): 14`

Table II. Simulation Result – Input and Output

Before describing the result and observation, stepwise testing strategy is to be explained. Below are the execution steps –

1. First of all, some input files are given some meaningful values:

start_ip.txt	00:00:00:001:000 start 200 00:00:52:001:000 start 201
floor0_ip.txt	00:00:22:000:000 floor0 1 00:00:32:000:000 floor0 1
floor2_ip.txt	00:00:18:000:000 floor2 1
floor5_ip.txt	00:00:02:001:000 floor5 1
floor7_ip.txt	00:00:28:000:000 floor7 1

These input files would be executed at their respective real time values as shown in Table II.

2. The AgentLogic.txt file would be given a set of rule-action. Below rule-action set is taken in this case:

```
d1 > d2 : e2
d1 < d2 : e1
d1 = d2 : e2
```

This is rule-action 'a' which follows the logic as- Nearer elevator is to be called. And in case of conflict, elevator2 would be preferred.

3. In Eclipse tool, after doing these changes in the text files, the project needs to be cleaned once before the execution. (Right click on the project and select "Clean Project"). If running it very first time, the project needs to be built once so that its MakeFile generates the executable- ElevatorControlAgent.exe. Now select this file every time when you need to execute the simulation, click on Run.

After successful execution, an output file ElevatorOutput.txt would be generated. It contains the data as shown in Table II. The output is presented to the corresponding input. At 00:00:00:001:000, the system receives start input as 200 that activates other inputs to operate. At 00:00:02:001:000, input request from floor5 is received. Initially, both the elevators are at floor0 and request has come from floor5. This is the case of d1 = d2, so the agent would call elevator2. At 00:00:02:061:032, Elevator2 motor will receive Up (12) command. At 00:00:03:061:032, it reaches to floor1 and stops for a negligible time of 20 msec*, at 00:00:03:081:032 it again receives the command to move up. These commands would be continuous until it reaches the requested floor i.e. floor5 at 00:00:07:141:032. At this floor, it would stop, and controller would be in idle state till next input event.

**This is the* `startPrepTime = TIME(00,00,00,20);` *variable in Elevator1.hpp and Elevator2.hpp and can be minimized accordingly.*

Next input event occurs at 00:00:18:000:000, this call is from floor2. Again, AgentLogic would see its rule-action set and decide which elevator to call. d1 is 2 (|2-0|) and d2 is 3 (|2-5|). This is the case of d1 < d2, so elevator1 would be called. And elevator1 would reach floor2 at 00:00:20:080:032. Similar operations would happen when inputs from floor0 and floor7 are received.

At 00:00:32:000:000, input from floor0 is received. However, elevator1 is already at floor0 so no action or elevator movement is required. Message "`Elev 1 is already at the same floor. Door opens.`" is displayed. These user-friendly messages of different stepwise execution can be seen in console. Below is the log from console for this case study:

```
Nucleo Board - Embedded CD-Boost

Creating atomic models Econtrol, elev1, elev2 ...
elev1 CREATED
elev2 CREATED
Creating Coupled model - CU ...
Creating input atomic models ...
Creating Main Coupled model - Elevator Control ...
Creating runner ...
Calling run endtime...
Elevator Start button is pressed.
Both elevators are at floor0.
Elevator call button is pressed from floor5
..Calling elevator2..
current floor of Elevator2 is: 1
current floor of Elevator2 is: 2
current floor of Elevator2 is: 3
current floor of Elevator2 is: 4
Elevator 2 stopped at floor 5. Door opens.
current floor of Elevator2 is: 5
Elevator call button is pressed from floor2
..Calling elevator1..
current floor of elevator1 is: 1
Elevator 1 stopped at floor 2. Door opens.
current floor of elevator1 is: 2
Elevator call button is pressed from floor0
..Calling elevator1..
current floor of elevator1 is: 1
Elevator 1 stopped at floor 0. Door opens.
current floor of elevator1 is: 0
Elevator call button is pressed from floor7
..Calling elevator2..
current floor of Elevator2 is: 6
Elevator 2 stopped at floor 7. Door opens.
current floor of Elevator2 is: 7
Elevator call button is pressed from floor0
..Calling elevator1..
Elev 1 is already at the same floor. Door opens.
Elevator Stop button is pressed.
Elevator controller stopped.
Application executed successfully.
```

VI. TESTING STARTEGY

The model and its outputs are observed by changing various input values. By default, time and values are 0 in other input files that are not active. To activate any of the floor input files, value needs to be changed to 1. The real time value should be changed to any valid value and input events should follow a sequence. The system is tested with various test strategies by changing input file values and/or its sequence. The operation of start/stop is also tested, the elevator inputs are processed only after start (start 200) is active and till the time stop (start 201) is requested. All rule-action sets of Agent are also tested.

VII. ASSUMPTION AND LIMITATION

It is assumed that beginning points of both the elevators are floor0. And they would start from floor0 whenever the elevator system is activated. While assigning real time value to input file, it should be taken care that this value is not in between the execution time of any other operation. The system only accepts any input when it is in idle state. This is said to be the limitation of this model.

VIII. CONCLUSION

Agent-based control of elevator system has been implemented, moreover, simulation of this model has been successful. The application is found to be accurately operative under various testing strategies. The output file data and the log in console are matched and coming as expected in all testing scenarios. It can be concluded that the agent that is used to control the elevators is a reactive agent as it perceives the environment state values and reacts to it accordingly.

IX. FUTURE WORK

As part of its future scope, the execution can also be performed with the real time elevators or robots. Communication between two different elevators, being the constraint, it could not be included in the current implementation. The mechanism of message transfer or any communication link is essential between two hardware elevators to control them through an Agent.

X. ACKNOWLEDGEMENTS

I would like to thank Prof. Gabriel Wainer for helpful suggestions and encouragement. I'm grateful to Cristina R Martin too who assisted me with fixing the simulator. It took a lot of time to fix the library, help from Juan Lanuza is also appreciative in this regard.

REFERENCES

[1] Gabriel Wainer. "*DEVS modelling and simulation for development of embedded systems*." Proceedings of the 2015 Winter Simulation Conference. IEEE Press, 2015.

[2] Gabriel Wainer, and Rodrigo Castro. "*DEMES: a Discrete-Event methodology for Modeling and simulation of Embedded Systems*." Modeling and Simulation Magazine 2 (2011)

[3] Gabriel Wainer. "*Discrete-event modeling and simulation: a practitioner's approach*." CRC press, 2009.

[4] J.J. Ch. Meyer. "*Agent Technology*" from "*Encyclopedia of Information Science and Technology*,". 2009.

[5] Alexander Steiniger, Frank Kruger, and Adelinde M. Uhrmacher. "*Modeling Agents and Their Environment In Multi-Level-DEVS*." Proceedings of the 2012 Winter Simulation Conference.

[6] A.M. Uhrmacher, B. Schattenberg. "*Agents in Discrete Event Simulation*." Department of Computer Science Institute of Artificial Intelligence.

[7] Baltazár Frankovič, Viktor Oravec. "*Design of the Agent-based Intelligent Control System*. " Acta Polytechnica Hungarica Vol. 2, No. 2, 2005.

[8] David H. Scheidt. "*Intelligent Agent-Based Control.* "

Johns Hopkins Apl Technical Digest, Volume 23, Number 4 (2002)

[9] Robert H. Crites, Andrew G. Barto. "*Elevator Group Control Using Multiple Reinforcement Learning Agents*." Machine Learning, 33, 235–262 (1998)

[10] Ge Hangli, Takeo Hamada, Takahiro Sumitomo, Noboru Koshizuka. "*PrecaElevator : Towards Zero-Waiting Time on Calling Elevator By Utilizing Context Aware Platform in Smart Building*." 7th Global Conference on Consumer Electronics. IEEE. 2018.