Model Driven Development, Case Study: Building Controller

Ben Earle
Systems and Computer Engineering,
Carleton University,
1125 Colonel By Dr., Ottawa, ON, Canada, K1S 5B6
BenEarle@cmail.carleton.ca

Kyle Bjornson
Systems and Computer Engineering,
Carleton University,
1125 Colonel By Dr., Ottawa, ON, Canada, K1S 5B6
KyleBjornson@cmail.carleton.ca

Abstract—**Model driven development using Embedded-CDBoost has been shown in [2] to be more efficient than other implementations of embedded DEVS based executives. The purpose of this case study is to implement a more complex model utilizing the E-CDBoost library. The model controls the lights and emergency systems of a building based on room occupancy and other inputs. We reflect on the issues faced when translating a formal DEVS specification into a real-time executable model.**

Keywords—Embedded Systems, Model Driven Development, Discrete Event System Specification, DEVS, Embedded-CD boost, Building Controller

I. Introduction

The goal of this project was to design a Real-Time System (RTS) using Discrete Event System Specification (DEVS). The project was implemented on an embedded system to be used as a case study of Model Driven Development (MDD) for RTS applications. The models were designed using the DEVS formalism and implemented using the Embedded-CD Boost (E-CDBoost) library on an ARM microcontroller. The system modeled was a light and emergency evacuation controller for an intelligent building. Overall the MDD procedure was efficient and it was easy to implement the atomic models. There were some difficulties faced when creating the coupled the models, discussed in detail in the Results section. This study also found that E-CDBoost should provide more support for port drivers. This would help the modeler by abstracting away the port implementation details.

II. Background

A. *Model Driven Development*

Discrete Event Methodology for Embedded Systems (DEMES) is a MDD technique used to simplify Embedded Systems development, increasing robustness and code reusability, all while decreasing the time spent developing [1]. DEMES has a structured development cycle which can be seen in Figure 1.

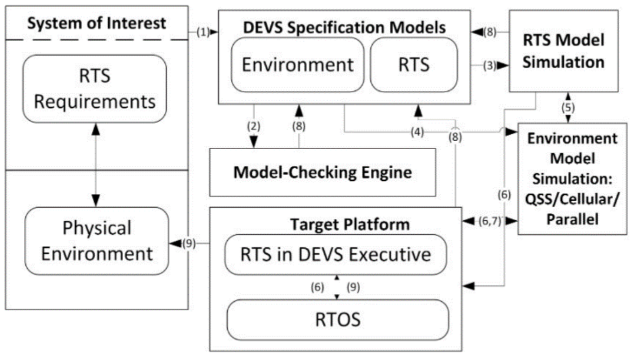

Figure 1. Discrete-Event Methodology for Embedded Systems [2].

The cycle starts by defining the relationship between the system of requirements and its physical environment (conceptual modelling). The system is then modelled using formal DEVS. The models can then be formally validated and simulated by stubbing the sensor input data. When the simulations are satisfactory the stubbed sensors can be replaced with the actual sensors on the target platform [1].

B. *Software*

The software platform is Embedded-CDBoost. It is a library written in C++ that facilitates the execution of DEVS models also defined in C++. The code can then be compiled for use on an embedded platform. The platform makes use of the C++ Boost libraries.

For ease of use, the ARM MBED libraries are used for interfacing input/output devices. MBED uses a specific devices hardware while allowing the core program to be somewhat hardware agnostic. This project utilizes MBED to control the analog to digital converter and general-purpose Input / Output (I/O) ports. These ports provide the embedded DEVS models with sensor data and accept the model's outputs.

C. *Hardware*

The target system used for this case study is a Nucleo F411RE, with an ARM Cortex M4 processor and 512kB of flash. This board is ideal for our project as it has ample storage for the E-CDBoost project, which compiles to approximately 175kB. The Nucleo board also has Arduino compatible headers with 5 analog I/O ports and 13 digital I/O ports. The Nucleo's Arduino pinouts are shown in Figure 2.

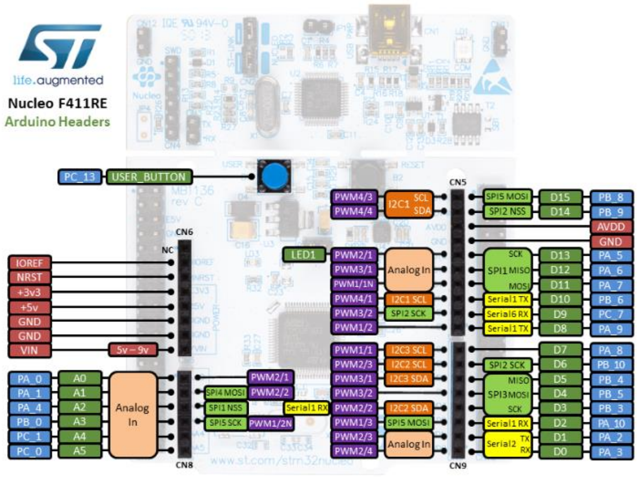

Figure 2. Nucleo Arduino pinouts [3].

All the sensors and actuators used in this project will be connected to these pins. The Nucleo board was mounted on a Seed Shield Bot, shown in Figure 3. This is to use its battery as well as its IR sensors to determine room occupancy. We are using 4 white LEDs, 2 red LEDs, and 2 green LEDs along with current limiting resistors. The LEDs represent room lighting and emergency exits. We are also using a Grove Light Sensor to detect ambient lighting in a room, a switch to represent a fire alarm, and a potentiometer to represent a thermometer. Originally, we interfaced a grove thermometer which outputs an analog voltage between 0 and 5 volts. It was challenging to alter the temperature of the sensor to test and demonstrate our model. The potentiometer output is identical to the temperature sensor but simplifies testing.

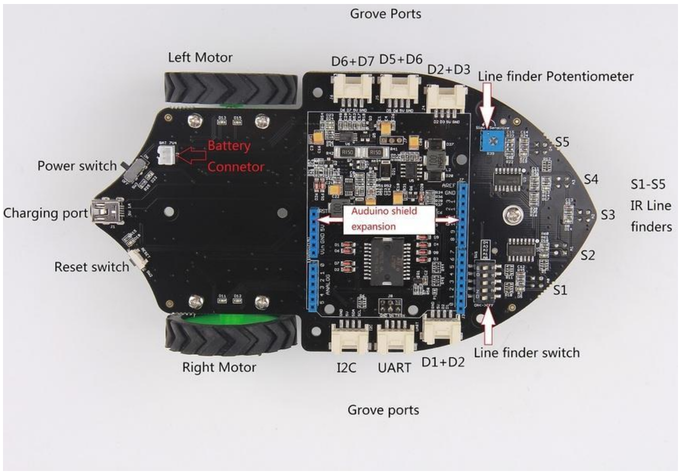

Figure 3. Seeed Shield Bot [3].

III. Models

The first step of DEMS is to identify a physical system to model. The system we are modeling for this case study is an intelligent building's light controller and emergency evacuation system. The system will turn on lights in a room depending on the ambient light and occupation. The emergency system will have a fire alarm and a heat detector by two doors. Each door has two signs, one indicating that it is a safe path to exit and the other showing that it is unsafe.

A. *Conceptual Model*

A conceptual model of the system is shown in the following the figure.

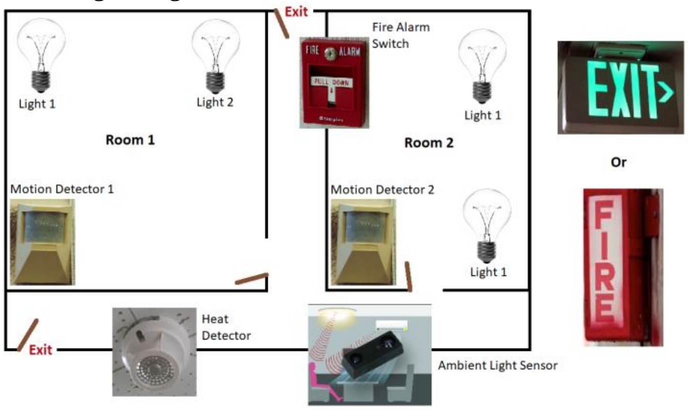

Figure 4. Conceptual Model of System.

The light controller will have two sensors: a motion detector to know if the room is occupied and an ambient light sensor to determine how much light is required in the room. If the room is brighter then we will turn the lights on to half intensity. The two motion sensors are represented as the IR Line Finders on the shield bot. Room 1 occupancy will be detected by the IR sensor on pin A0, Room 2 on A3. For simplicity we will use one ambient light sensor. However, in an actual implementation it would be preferable to have one for each room. The system will check if the ambient light is above a threshold, if it is it will consider the room to be bright. If the room is bright then only one LED will be on in an occupied room. If the room is dark, then both LEDs will be on in an occupied room.

The emergency control system will be made up of a red and green LED for each door and two emergency sensors. The first emergency sensor is a switch. The switch will be connected to a digital input pin with a pull up resister, grounding it when active. The second will be a heat sensor, raising the alarm when the temperature is above a safe threshold. The red LED will light up when the sensor near that door is alarming. The green LED will light up if the alarm is raised but not by the sensor near that door. The green LED shows that this is a safe path to exit. Additionally, when there is an alarm raised we will turn on all the lights in the room. This will allow for maximum vision in an emergency scenario.

B. *Coupled Model Block Diagram*

This section will show the structure of the coupled models. The top model is shown in Figure 5.

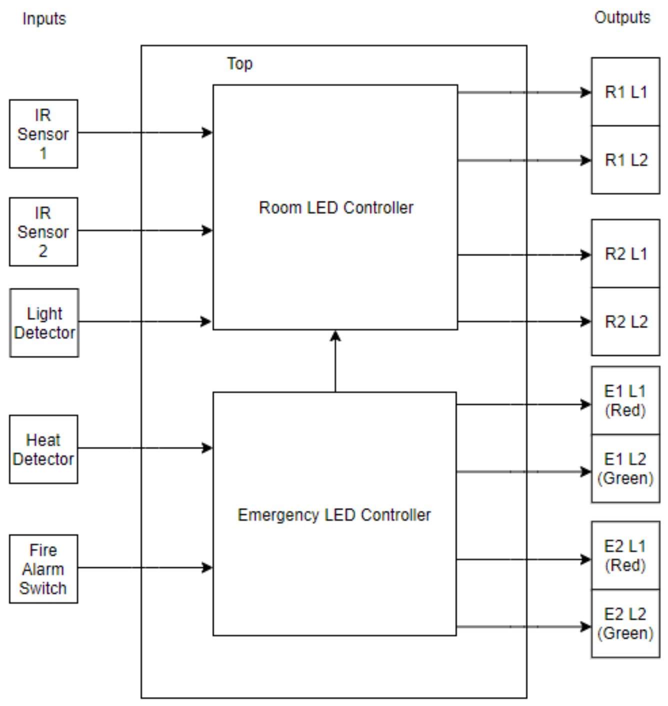

Figure 5. Top Model.

The room lighting system is the Room LED Controller, coupled model. The Emergency control system is the Emergency LED Controller. Note the input and outputs going to their corresponding subsystem as described in part A of this section. Figure 6 shows the Emergency System's atomic models.

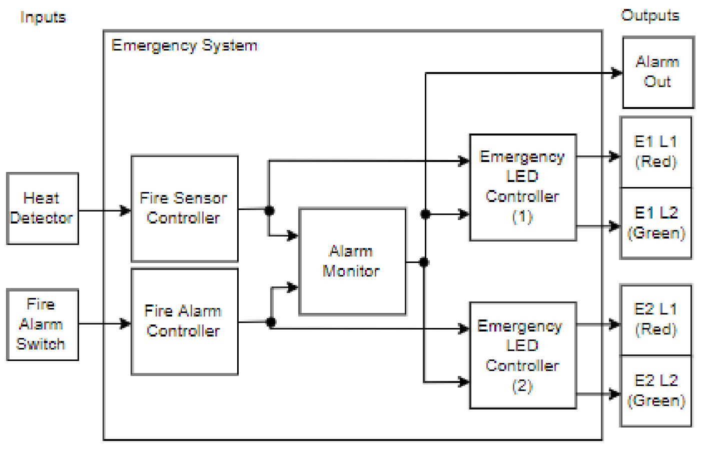

Figure 6. Emergency System

The two Sensor Controller blocks will interpret the input and determine if an alarm should be raised. The Alarm Monitor block does an OR operation on the Sensor Controller outputs, raising the alarm if any sensor is alarming. The Emergency LED Controller will turn on the red LED if it sees that its sensor is alarming. If its sensor is off but the Alarm Monitor is alarming then it will turn on the green light. If neither of the inputs are alarming the lights will both be off. The Alarm Out port goes to the Room LED Controller, shown in Figure 7.

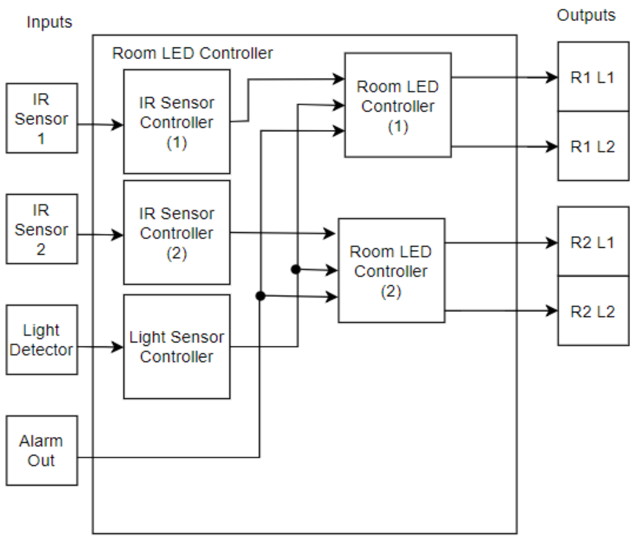

Figure 7. Room Light System.

This coupled model also has input sensor controllers. IR Sensor Controller will read a 0 when a line is present (the room is occupied) and output accordingly. Light Sensor Controller will check if the ambient light in the room is above a threshold and output if the room is dark or bright. Room LED Controller will turn both lights on if the Alarm Out port is alarming, or if the room is occupied and the light sensor says it is dark. If the room is occupied and the light sensor says it is bright then it will only turn on one light.

C. *DEVS Formal Model*

Atomic Light Sensor Controller

The Grove Light Sensor outputs an analog value. We will read this value using the onboard ADC then pass it as input to our sensor controller model. If the light is above a certain threshold then we will determine it is day. We will determine an acceptable threshold experimentally using the hardware.

sigma = ∞, **phase** = Passive;
Day = 0 (Boolean);

X = { LightSensorInput ∊ I }
Y = { LightSensorOutput ∊ {0, 1} }
S = { { phase, sigma}, Day }

```
δext (s={ Day }, e, x= LightSensorInput) {
        If (LightSensorInput > LIGHT_THRESHOLD) {
                Day = 1;
                Sigma = 0;
        } Else {
                Day = 0;
                Sigma = 0;
        }
}

δint () { Passivate; }

λ (s) { Send Day to the port out }
```

Atomic Temperature Sensor Controller

The Grove Temperature Sensor outputs an analog value. We will read this value using the onboard ADC then pass it as input to our sensor controller model. If the light is above a certain threshold then we will determine it is day. We will determine an acceptable threshold experimentally using the hardware.

```
**sigma** = ∞, **phase** = Passive;
**Day** = 0 (Boolean);

**X** = { TemperatureSensorInput ∈ I }
**Y** = { TemperatureSensorOutput ∈ {0, 1} }
**S** = { { phase, sigma}, Fire }

δext(s={TemperatureSensorInput},e,x=
TemperatureSensorInput) {
    If(TemperatureSensorInput>
    TEMPERATURE_THRESHOLD) {
        Fire = 1;
        Sigma = 0;
    } Else {
        Fire = 0;
        Sigma = 0;
    }
}
δint () { Passivate; }
λ (s) { Send Fire to the port out }
```

Atomic Digital Input Controller

This controller will be used for the fire alarm switch and the IR sensor, they will handle the input and send an update to the system when it changes.

```
**sigma** = ∞, **phase** = Passive;
**Active** = 0 (Boolean);

**X** = { Input ∈ {0, 1} }
**Y** = { Output ∈ {0, 1} }
**S** = { { phase, sigma}, Active }

δext (s={Input }, e, x={0, 1}) {
    //The digital inputs are all active low.
    If (Input == 0) {
        Active = 1
        Sigma = 0;
    } Else {
        Active = 0;
        Sigma = 0;
    }
}
δint () { Passivate; }

λ (s) { send Active to the port out }
```

Atomic Alarm Monitor

This model operates as an OR gate for its inputs. If any alarm sensor is raised then the output alarm will be raised.

```
**sigma** = ∞, **phase** = Passive;
**Alarm** = 0 (Boolean);

**X**={{AlarmSensorInput1,AlarmSensorInput2,...
AlarmSensorInputN} ∈ {0, 1}}
**Y** = { AlarmOut ∈ {0, 1} }
**S** = { { phase, sigma}, Alarm }

δext(s={Alarm},e,x={AlarmSensorInput1,
AlarmSensorInput2, ... AlarmSensorInputN}) {
    If (AlarmSensorInput1 | AlarmSensorInput2 | ...
    AlarmSensorInputN) {
        Alarm = 1;
        Sigma = 0;
    } Else {
        Alarm = 0;
        Sigma = 0;
    }
}

δint () { Passivate; }
λ (s) { Send Alarm to the AlarmOut }
```

Atomic Emergency LED Controller

This controller checks if its alarm is raised, if so then it will display the red light, depicting an unsafe path. If the system alarm is on but the sensor is not, then it will light up green to show it is a safe exit path. If neither input is high the LED is off.

```
**sigma** = ∞, **phase** = Passive;
**Alarm** = 0 (Boolean);
NONE = 0; GREEN = 1; RED = 2;
**X** = { {SensorAlarm, SystemAlarm } ∈ {0, 1}}
**Y** = {MultiColorLedOut ∈ { NONE, GREEN, RED} }
**S** = { { phase, sigma} }

δext (s={ Alarm }, e, x= { {SensorAlarm, SystemAlarm })
                                        {
    If (SensorAlarm) {
        Alarm = RED;
    } Else if (SystemAlarm) {
        Alarm = GREEN;
    } Else {
        Alarm = NONE;
    }
    Sigma = 0;
}

δint () { Passivate; }

λ (s) { Send Alarm to the MultiColorLedOut }
```

Atomic Room LED Controller

This model will determine the intensity of the lights in the room based on occupancy and ambient light readings. One LED on will indicate half brightness, while two LEDs on is full brightness.

sigma = ∞, **phase** = Passive;
OFF = 0; HALF = 1; FULL = 2;
RoomOut = OFF;
X = { {IRSensor, AmbientLightSensor, SystemAlarm } ∈ {0, 1}}
Y = {LEDOut ∈ { OFF, HALF, FULL} }
S = { { phase, sigma} }

```
δext (s={ Alarm }, e, x= {{IRSensor, AmbientLightSensor, SystemAlarm }){
        If (SystemAlarm) {
                RoomOut = FULL;
        } Else If (IRSensorIn) {
                If (AmbientLightSensor) {
                        RoomOut = HALF
                } Else {
                        RoomOut = FULL;
                }
        } Else {
                RoomOut = OFF;
        }
        Sigma = 0;
}

δint () { Passivate; }

λ (s) { Send RoomOut to the LEDOut }
```

Coupled Top Model

The block diagram for this coupled model is shown in Figure 5. This links the two coupled models and the I/O ports to their controllers.

```
X = {
        HeatDetector ∈ float,
        FireAlarmSwitch ∈ bool,
        IRSensor1 ∈ bool,
        IRSensor2 ∈ bool,
        LightDetector ∈ float
};
Y = {
        R1_L1,
        R1_L1,
        R2_L1,
        R2_L2,
        E1_L1,
        E1_L2,
        E2_L1,
        E2_L2
};
D = {emergencyLedController, roomLedController};
EIC = {
        (Top.HeatDetector,
        emergencyLedController.HeatDetector),

        (Top.FireAlarmSwitch,
        emergencyLedController.FireAlarmSwitch),

        (Top.IRSensor1,
        roomLedControler.IRSensor1),

        (Top.IRSensor2,
        roomLedControler.IRSensor2),

        (Top.LightDetector,
        roomLedControler.LightDetector)
};
EOC = {
        (roomLedControler.R1_L1,        Top.R1_L1),
        (roomLedControler.R1_L2,        Top.R1_L2),
        (roomLedControler.R2_L1,        Top.R2_L1),
        (roomLedControler.R2_L2,        Top.R2_L2),
        (emergencySystem.E1_L1,         Top.E1_L1),
        (emergencySystem.E1_L2,         Top.E1_L2),
        (emergencySystem.E2_L1,         Top.E2_L1),
        (emergencySystem.E2_L2,         Top.E2_L2)
};
IC = {
        (emergencySystem.AlarmOut,
        roomLedController. AlarmOut)
 };
SELECT = {
        ({emergencyLedController, roomLedController}
) = emergencyLedController;
};
```

Coupled Emergency Controler

The block diagram for this coupled model is shown in Figure 6. Note that the fireAlarmController is an instances of the Digital Input model. fireSensorController is an instance of the Temperature Sensor Controller.

```
X = {HeatDetector ∈ float, FireAlarmSwitch ∈ bool};
Y = {AlarmOut, E1_L1, E1_L2, E2_L1, E2_L2};
D = {
        fireSensorController,
        fireAlarmController,
        alarmMonitor,
        emergencyLedController_1,
        emergencyLedController_2
};
EIC = {
        (EmergencySystem.HeatDetector,
        fireSensorController.TemperatureSensorInput),

        (EmergencySystem.FireAlarmSwitch,
        fireAlarmController.Input),
};
EOC = {
        (alarmMonitor.AlarmOut,
        EmergencySystem.AlarmOut),

        (emergencyLedController_1.LEDOut [0],
        EmergencySystem.E1_L1),
        (emergencyLedController_1.LEDOut [1],
        EmergencySystem.E1_L2),

        (emergencyLedController_2.LEDOut [0],
        EmergencySystem.E2_L1),

        (emergencyLedController_2.LEDOut [1],
        EmergencySystem.E2_L2)
};
IC = {
        (fireSensorController.TemperatureSensorOutput,
        alarmMonitor.AlarmSensorInput1),

        (fireAlarmController.Output,
        alarmMonitor.AlarmSensorInput2),

        (alarmMonitor.AlarmOut,
        emergencyLedController_1.SystemAlarm),

        (alarmMonitor.AlarmOut,
        emergencyLedController_2.SystemAlarm),

        (fireSensorController.TemperatureSensorOutput,
        emergencyLedController_1.SensorAlarm),

        (fireAlarmController.Output,
        emergencyLedController_2.SensorAlarm)
};
SELECT = {
        ({fireSensorController, fireAlarmController,
        alarmMonitor, emergencyLedController_1,
        emergencyLedController_2}) =
        emergencyLedController_1;

        ({fireSensorController, fireAlarmController,
        alarmMonitor, emergencyLedController_2}) =
        emergencyLedController _2;

        ({fireSensorController, fireAlarmController,
        alarmMonitor}) = alarmMonitor;

        ({fireSensorController, fireAlarmController}) =
        fireSensorController;
};
```

Coupled Room LED Controler

The block diagram for this coupled model is shown in Figure 7. Note that the irSensorControllers are just instances of the digital Input model.

```
X = {
        IRSensor1 ∈ bool,
        IRSensor2 ∈ bool,
        LightDetector ∈ float,
        AlarmOut ∈ bool
};
Y = {R1_L1, R1_L1, R2_L1, R2_L2};
D = {
        irSensorController_1,
        irSensorController_2,
        lightSensorController,
        roomLEDController_1,
        roomLEDController_2
};
EIC = {
        (RoomLedControler.IRSensor1,
        irSensorController_1.Input),

        (RoomLedControler.IRSensor2,
        irSensorController_2.Input),

        (RoomLedControler.LightDetector,
        lightSensorController.LightSensorInput),

        (RoomLedControler.AlarmOut ,
        roomLEDController_1.SystemAlarm),

        (RoomLedControler.AlarmOut ,
        roomLEDController_2.SystemAlarm)
};
EOC = {
        (roomLEDController_1.MultiColorLedOut[0],
        RoomLedControler.R1_L1),

        (roomLEDController_1.MultiColorLedOut[1],
        RoomLedControler.R1_L2),

        (roomLEDController_2.MultiColorLedOut[0],
         RoomLedControler.R2_L1),

        (roomLEDController_2.MultiColorLedOut[1],
        RoomLedControler.R2_L2),
};
IC = {
        (lightSensorController.LightSensorOutput,
        roomLEDController_1.AmbientLightSensor),

        (lightSensorController.LightSensorOutput,
        roomLEDController_2.AmbientLightSensor),

        (irSensorController_1.Output,
        roomLEDController_1.IRSensor),

        (irSensorController_2.Output,
        roomLEDController_2.IRSensor)
};
SELECT = {
        ({irSensorController_1, irSensorController_2,
        lightSensorController, roomLEDController_1,
        roomLEDController_2}) =
        roomLEDController_1;

        ({irSensorController_1, irSensorController_2,
        lightSensorController, roomLEDController_2})
        = roomLEDController_2;

        ({irSensorController_1, irSensorController_2,
        lightSensorController}) = irSensorController_1;

        ({irSensorController_2, lightSensorController})
        = irSensorController_2;
};
```

IV. Results and Implimentation

This section discusses the details of implementation.

A. *Circuit Design*

The Seeed Shield Bot platform has 5 infrared sensors that on board that were utilized to model detecting room occupancy. These sensors are connected to the A0, A1, A2, A3 and D4 headers on the Seeed Shield and can be disabled with on board dip switches. For this study, dip switches 1 and 4 were enabled to use the IR sensors on A0 and A3, while switches 2, 3, and 5 are turned off.

Some simple external circuits were also needed. The room LEDs are driven through ports D6, D7, D8, and D11 skipping D9 and D10 as these are used to enable the motors on the Seeed Sheild Bot. The Emergency LED circuit is similar but utilizes ports D2, D3, D4 and D5, this time skipping D0 and D1 as these are used for USART debugging of the Nucleo board. This circuit also uses Red and Green LEDs as opposed to the white LEDs of the Room. The circuits are shown in Figure 8 and 9 below.

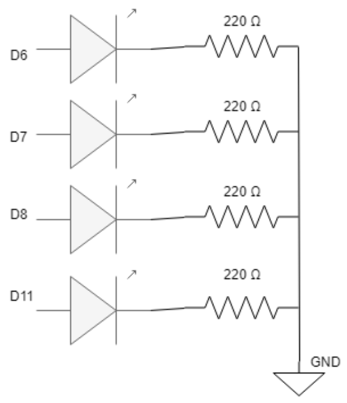

Figure 8. Room LED Controller Circuit.

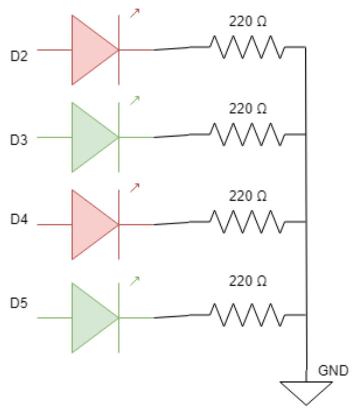

Figure 9. Emergency LED Controller Circuit.

The fire alarm is represented with a simple switch. Pin D12 is used as an active low input and is setup with an internal pull up resistor. Closing the switch grounds the port and activates the alarm.

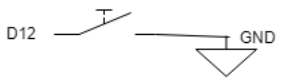

Figure 10. Fire Alarm Circuit.

The temperature sensor was modelled for this case study as a 10K Ohm potentiometer as mentioned previously.

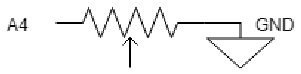

Figure 11. Temperature Sensor Circuit.

The Grove light sensor was interfaced with port A5 which outputs an analog value between 0 and 5V. The Analog to Digital converter on the Nucleo board enables reading this pin and determining the light intensity on the sensor.

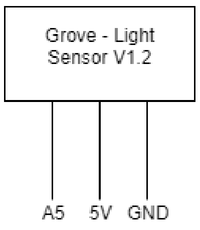

Figure 12. Ambient Light Sensor Circuit.

B. *Circuits on Hardware*

This section will show the implementation of the circuit described in part A. Figure 13 below shows the connections to the Nucleo board as described in the previous section. The Nucleo board mounts to the underside of the Seeed Shield and passes its pin connections to the top. The dip switches are also shown in their correct positions, along with the two Infrared sensors that are being utilized.

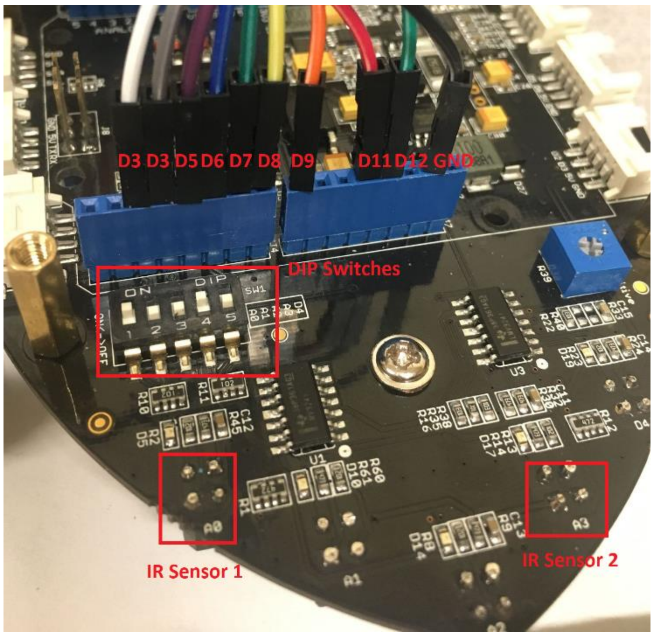

Figure 13. Front of Seed Shield.

The following figure shows the rest of the connections to the Nucleo board. The A4/5 pins connect to the onboard analog to digital converter to accept analog input signals.

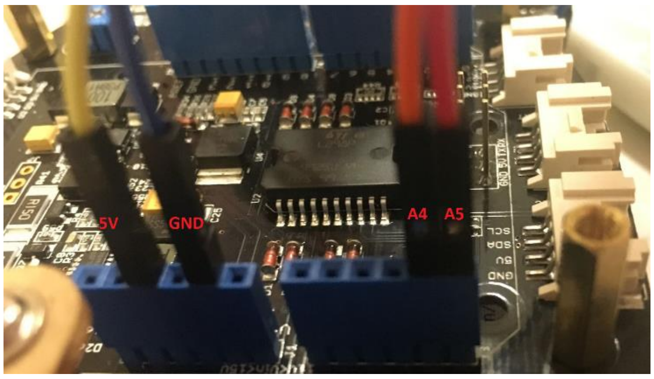

Figure 14. Back of Seed Shield.

Figure 14 shows the LEDS representing the emergency and room lighting.

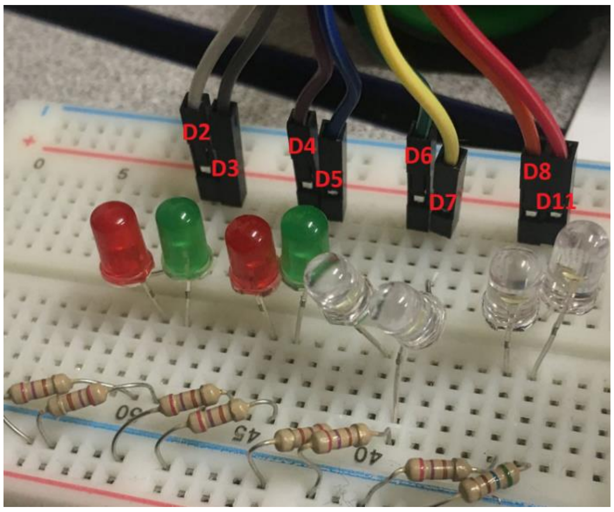

Figure 15. LED Circuitry.

Figure 16 shows the switch and potentiometer connections.

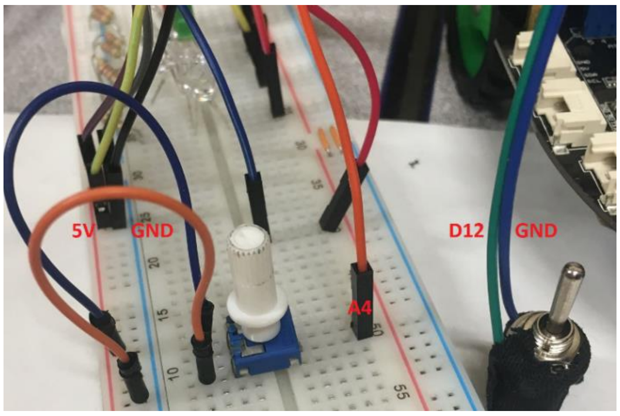

Figure 16. Alarm Sensor Circuitry.

Figure 17 shows the Grove light sensor. It is meant to be used in conjunction with a Grove port, but here we have just probed the three wires we need to connect the sensor. It takes 5V, Ground, and port A5 as an analog input. Note the pin connected to the white wire is labeled as NC for Not Connected.

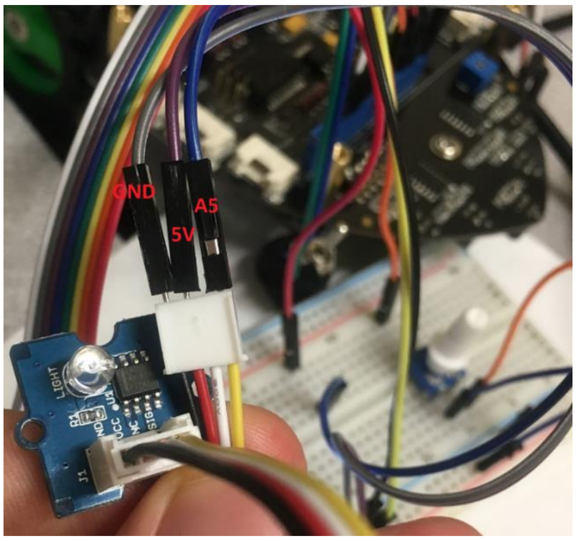

Figure 17. Grove Light Sensor.

C. *Software*

This section will introduce code snippets from the code base. The full project can be found in [5].

Figure 18 shows the pin declarations on the Nucleo STM32F4 using the MBED library. This exists in 'drivers.hpp' and is also where the input controller models, as defined formally above, are implemented. That is, the Temperature Sensor Controller, Light Sensor Controller, and the Digital Input Controller. As a note, 'mot1En' and 'mot2En' are the enable pins for the motors. We must pull these pins low if we want to use the D10-D13 for something other than driving the motors.

```
DigitalIn leftSensor(A0);
DigitalIn rightSensor(A3);

DigitalIn fireAlarm(D12);

AnalogIn ambientLightSensor(A5);
AnalogIn temperatureSensor(A4);

DigitalOut mot1En(D9);
DigitalOut mot2En(D10);

DigitalOut room1Led1(D6);
DigitalOut room1Led2(D7);
DigitalOut room2Led1(D8);
DigitalOut room2Led2(D11);

DigitalOut emerg1Red(D2);
DigitalOut emerg1Green(D3);
DigitalOut emerg2Red(D4);
DigitalOut emerg2Green(D5);
```

Figure 18. Pin Declarations using MBED in 'drivers.hpp'.

For example, the implementation of the Light Sensor Controller is shown in Figure 19 below.

```
template<class TIME, class MSG>
bool AMBIENT_LIGHT_IN<TIME, MSG>::pDriver(Value &v)
const noexcept {
    if(ambientLightSensor.read() >
AMBIENT_LIGHT_THRESHOLD)
        v = 1;
    else
        v = 0;
        return true;
}
```

Figure 19. Light Sensor Controller implemented in ‘drivers.hpp’.

Whenever a port driver is defined, it must be instantiated with a polling period. This is done for the ambient light sensor in ‘sensors_port.hpp’. The code snippet can be found in Figure 20 below, showing that the port is set to be polled every 200ms.

```
template<class TIME, class MSG>
class AMBIENT_LIGHT_IN : public port<TIME, MSG>
{

public:
    /**
     * @brief ambient_light_in constructor.
     *
     * @param n Name assigned to the port.
     * @param polling Polling period associated
with the port.
     */
     explicit AMBIENT_LIGHT_IN(const std::string &n
= "ambient_light_in", const TIME &polling =
TIME(0,0,0,200)) noexcept : port<TIME,
MSG>(n,polling) {}
     void print()  noexcept {}
     bool pDriver(Value &v) const noexcept;

};
```

Figure 20. Light Sensor port instantiation in ‘sensors_port.hpp’.

The output drivers are also defined in ‘drivers.hpp’. That is, the method of interpreting a DEVS message and setting the physical ports accordingly. For example, Figure 21 shows the driver for the first room.

```
template<class TIME, class MSG>
bool ROOM1_OUT<TIME, MSG>::pDriver(Value& v) const
noexcept{

        if (v == 0) {
                room1Led1 = 0;
                room1Led2 = 0;
        } else if (v == 1) {
                room1Led1 = 0;
                room1Led2 = 1;
        } else {
                room1Led1 = 1;
                room1Led2 = 1;
        }

        return true;
}
```

Figure 21. Room Output Driver implemented in ‘drivers.hpp’.

The output driver must also be instantiated. In this case for the room driver it is instantiated in ‘actuators_port.hpp’.

```
template<class TIME, class MSG>
class ROOM1_OUT : public port<TIME, MSG>
{

public:
    /**
     * @brief room1 constructor.
     *
     * @param n name assigned to the port
     */
     explicit ROOM1_OUT(const std::string &n =
"room_out1") noexcept : port<TIME, MSG>(n) {}
     void print() noexcept{}
     bool pDriver(Value &v) const noexcept;
};
```

Figure 22. Room Output Driver instantiation in ‘actuators_port.hpp’.

Now moving away from the port drivers, the DEVS models are implemented in ‘AlarmMonitor.hpp’, ‘AlarmOutControl-ler.hpp’, and ‘RoomController.hpp’. For example, the following figures show the DEVS functions defined for the RoomController atomic model.

Figure 23 below shows the simple internal transition function.

```
void internal() noexcept {
        _state = IDLE;
        _next = infinity;
}
```

Figure 23. RoomController internal transition function in ‘RoomController.hpp’.

Figure 24 shows the output function of the RoomController. Outputs depend on the current state.

```
std::vector<MSG> out() const noexcept {
    if(_state == IDLE) _outputMessage =
    MSG(portName[rctrl_room_out], 0);
    if(_state == HALF) _outputMessage =
    MSG(portName[rctrl_room_out], 1);
    if(_state == FULL) _outputMessage =
    MSG(portName[rctrl_room_out], 2);
    return std::vector<MSG>{_outputMessage};
    }
```

Figure 24. RoomControler output function implemented in ‘RoomController.hpp’.

Finally, Figure 25 shows the implementation of the external transition function. It receives an input and checks which port it was received on. It then saves the corresponding value of the message being received.

Next a state change is decided based on the current value of the two state variables: ‘isOccupied’ and ‘isBright’. The time advance is set to 0 seconds which triggers the output function, then internal transition function which passivates the model.

```
void external(const std::vector<MSG>& mb, const
TIME& t) noexcept {

        MSG msg = mb.back();

        _next = Time::Zero;
        _state = IDLE;

    if (msg.port == portName[rctrl_light_in]) {
        isOccupied = (msg.val == 1);
    } else if (msg.port ==
    portName[rctrl_ambient_light_in]) {
        isBright = (msg.val == 1);
    }

    if(isOccupied){
        if (isBright){
                _state = HALF;
        } else {
                _state = FULL;
                }
        } else {
                _state = IDLE;
        }
    }
```

Figure 25. RoomControler external transition function implemented in 'RoomController.hpp'.

To create a coupled model using ECDBoost the atomic models must be instantiated and coupled together. The code snippet in Figure 26 shows first the instantiation and then the creation of the 'ControlUnit'. This uses a constructor that takes a list of all models, external input couplings, internal couplings, and external output couplings in that order.

It is worth noting some of the comments in the snippet below showing additional couplings. The implementation of the model directly from the formal specification introduced problems that could not be circumvented. This will be discussed in the next section.

```
auto rctrl1 = make_atomic_ptr<RoomController1<Time,
Message>>();
auto rctrl2 = make_atomic_ptr<RoomController2<Time,
Message>>();
auto actrl1 =
make_atomic_ptr<AlarmOutController1<Time,
Message>>();
auto actrl2 =
make_atomic_ptr<AlarmOutController2<Time,
Message>>();

//Coupled model definition
shared_ptr<flattened_coupled<Time, Message>>
ControlUnit(new flattened_coupled<Time, Message> {
        {rctrl1, rctrl2, actrl1, actrl2},
        {rctrl1, rctrl2, actrl1, actrl2},
        //{rctrl1, rctrl2, amon, actrl1, actrl2},
        //{rctrl1, rctrl2, amon, actrl1, actrl2},
        {
        //      {amon,actrl1},
        //      {amon,actrl2}
        },
        {rctrl1, rctrl2, actrl1, actrl2}
});
```

Figure 26. Coupled Model Definition in 'main.cpp'.

Figure 27 shows the port definitions that connect the physical port drivers to the DEVS model. First, all the required ports are instantiated.

```
// Input ports
auto light_left = make_port_ptr<LIGHT_IN_LEFT<Time,
Message>>();
auto light_right =
make_port_ptr<LIGHT_IN_RIGHT<Time, Message>>();
auto ambient_light =
make_port_ptr<AMBIENT_LIGHT_IN<Time, Message>>();
auto fire_alarm = make_port_ptr<FIRE_ALARM<Time,
Message>>();
auto temperature =
make_port_ptr<TEMPERATURE_IN<Time, Message>>();

// Output ports
auto room1 = make_port_ptr<ROOM1_OUT<Time,
Message>>();
auto room2 = make_port_ptr<ROOM2_OUT<Time,
Message>>();
auto emerg1 = make_port_ptr<EMERGENCY1_OUT<Time,
Message>>();
auto emerg2 = make_port_ptr<EMERGENCY2_OUT<Time,
Message>>();
```

Figure 27. Port definitions in 'main.cpp'.

Next the root model is created, shown in Figure 28. This model has defines the links between the ControlUnit coupled model and the systems I/O port drivers.

```
Time initial_time{Time::currentTime()};
erunner<Time, Message> root {ControlUnit,
        {//External Input Coupling
                {light_left,rctrl1},
                {ambient_light,rctrl1},
                {light_right,rctrl2},
                {ambient_light,rctrl2},
                fire_alarm,actrl2},
                {fire_alarm,actrl1},
                {temperature,actrl1},
                {temperature,actrl2},
        },
        {//External Output Coupling
                {room1,rctrl1},
                {room2,rctrl2},
                {emerg1,actrl1},
                {emerg2,actrl2},
        }
};
```

Figure 28. Coupled Model Port definitions in 'main.cpp'.

D. *Implementation Limitiations and Tool Recomendations*

As mentioned in previous sections, there were some limitations encountered during the implementation phase that required alterations of the models. The first problem we encountered was instantiating multiple instances of one atomic model. In this version of ECD-Boost all the models subscribed to receive inputs will listen to all inputs and check if input's port matches the hardcoded port name for that individual model. If two models are instantiated they will have the same port names and there will be no way to differentiate which message is for which model. A possible solution to this would be using the observer pattern where each output port has a list of input port listeners subscribed to get its messages. This way if many instances of a model exist it would be possible to separate the messages.

Although internal couplings work in the case study discussed in [2], we were unable to have a successful internal message transfer. After building the model as shown in section III and doing extensive testing it was clear that Alarm Monitor never got an input messages. The internal couplings could be more intuitive if the previous recommendation were incorporated. Since we were unsuccessful in creating internal couplings we modified the model to bypass the Alarm Monitor since it was the only model that needed them. We took the sensor data directly from the opposing sensor and used it as the 'SystemAlarm' input for the two 'emergency-LedControllers'. Furthermore, we removed the connection between the Emergency LED Controller and the Room LED Controller.

Another feature that would streamline the development process is premade I/O port drivers. There are only 4 types of port drivers that are needed: digital input, digital output, analogue input, and analogue output. The tool should include a class for each of these with the pin number as an operand in the constructor. This would abstract away the port drivers and the need for the modeller to interface with their pins directly. The port drivers would interpret or send the DEVS message and set the pin accordingly. Then the modeller would need only instantiate the port drivers in their main.cpp file (shown in in Figure 27). This would save the modeller from writing the code shown in Figure 18 to Figure 22 inclusive. In our case study we spent much more time working with MBED and the port driver then we spent implementing the DEVS models in C++. Implementing this feature would allow E-CDBoost to better meet the goals of DEMS.

Another recommendation for the tool is creating an inter-device communication protocol that passes DEVS messages between that multiple embedded boards. In this case study we were very close to running out of available ports on the Arduino header. If a modeller wanted to create a more complex system with this setup they would need more then one board. This would also facilitate controlling distributed systems, where each node operates as its own coupled model. Most ARM based microcontrollers support UART and SPI communications which would be ideal for passing such messages. Furthermore, there are cheep Arduino compatible wireless transceivers such as the NRF24L01, which could be used for short range wireless systems.

E. *Annotation of Included Video*

This section describes the video included in the project files. Each comment includes a timestamp from the included video.

00:05 – IRSensor1 is triggered turning the lights on in Room1

00:20 – Ambient light sensor moved in and out of light to show one of the LEDs turning on and off while the room is occupied.

00:35 – IRSensor2 is triggered turning the lights on in Room2.

00:38 – Both IR sensors triggered showing all lights on.

00:40 – Ambient light sensor moved to show half of the lights turning off in a bright environment.

01:00 – Temperature sensor (Potentiometer) turned to indicate a fire alarm being triggered.

01:05 – The red LED representing the exit with the temperature sensor is triggered. This indicates that this exit is not safe. The green LED is triggered for the alternate exit as that exit is deemed safe.

01:16 – The fire alarm switch is flipped representing a fire alarm being triggered at that exit. The red LED representing the exit with the switch is triggered. This indicates that this exit is not safe. The green LED is triggered for the other exit as now that exit is deemed safe.

01:25 – If both alarms have triggered, neither exit is deemed safe and both red LEDs are enabled.

01:30 – And of course the room LEDs still operate if the alarm is enabled…

01:45 – If both alarms are disabled, all the emergency LEDs turn off.

F. *Conclusion*

Overall, we were successful in implementing the atomic models and interfacing our project with the sensors as described in the conceptual model. We had some difficulties with internal coupling of our coupled models and instantiating multiple instances of our atomic models. We proposed solutions to these issues and some additional features in section D. We are interested in MDD and would like to investigate upgrading the tool to better facilitate embedded systems development.

References

[1] D. Niyonkuru, G.A. Wainer, A Kernal for Embedded Systems Development and Simulation using the Boost Library

[2] D. Niyonkuru and G. Wainer, "Discrete Event Methodology for Embedded Systems." Computing in Science & Engineering vol.17, no.5, pp.52-63, Sept-Oct. 2015

[3] St.com. (2018). STM32 MCU Nucleo - STMicroelectronics. [online] Available at: https://www.st.com/en/evaluation-tools/stm32-mcu- nucleo.html?querycriteria=productId=LN1847 [Accessed 1 Dec. 2018].

[4] Wiki.seeedstudio.com. (2018). Shield Bot V1.1. [online] Available at: http://wiki.seeedstudio.com/Shield_Bot_V1.1/ [Accessed 1 Dec. 2018].

[5] K. Bjornson, B. Earle, Buidling Controller Git Repository. Avalilable at: https://github.com/KyleBjornson/SYSC5104-TermProject