Tram Controller Using Discreet Event Methodology for Embedded Systems

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

Vithika Joshi
Electrical and Computer Engineering,University of Ottawa, ON, Canada, K1N 6N5
vjosh031@uottawa.ca

***Abstract*— This project report illustrates the model and implementation of a Tram Controller using Discrete Event Methodology for Embedded Systems (DEMES). The model is implemented in Seed Shield Studio Bot with help of an ECDBoost simulator. The idea is to build a special purpose transportation system, like Tram, for a specific industry or institute which acquires a very large area and need to have an internal or local transportation. The passengers can request for their stops by giving the particular stop as an input from inside the Tram as well as the outside passengers waiting to go to different stations can also request to call the Tram for pick up from their respective stations. This model is mainly divided into two parts - one is to get the stop request from inside passengers and the other one is to get the stop request from outside passengers.**

Keywords—Embedded systems, Discreet Event Methodology for Embedded Systems, DEMES, Discreet Event System Specification, DEVS

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.

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, δ_{ext} is the external state transition function, δ_{int} is the internal state transition function, λ is the output function and *ta* is the time advance function. 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 δ_{int}. This transition is called an internal transition. If there is a reception of an external event, δ_{ext} determines its new state. If *ta* is infinite, then s is said to be in passive state. A DEVS coupled model is composed of several atomic models. It is defined as

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

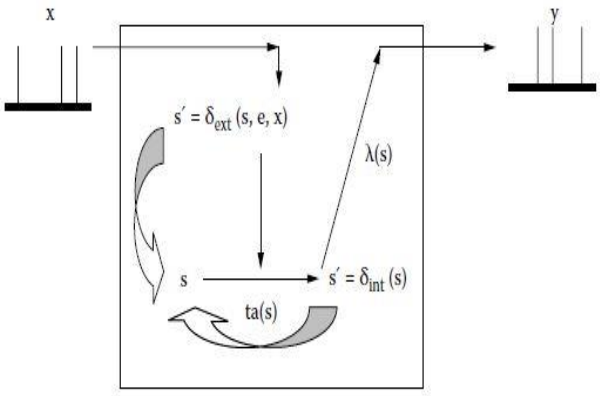

Fig. 1 DEVS atomic model.

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.

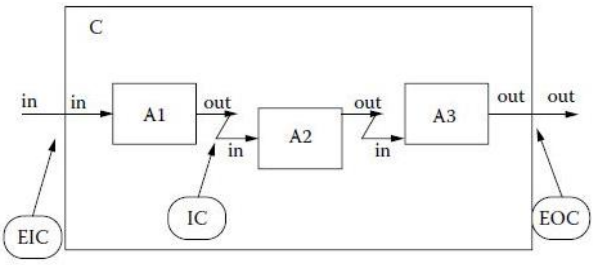

Fig. 2 DEVS atomic model.

C. DEMES

Discreet Event Modelling of Embedded Systems is an approach based on DEVS that allows models to be used throughout the development cycle. 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.

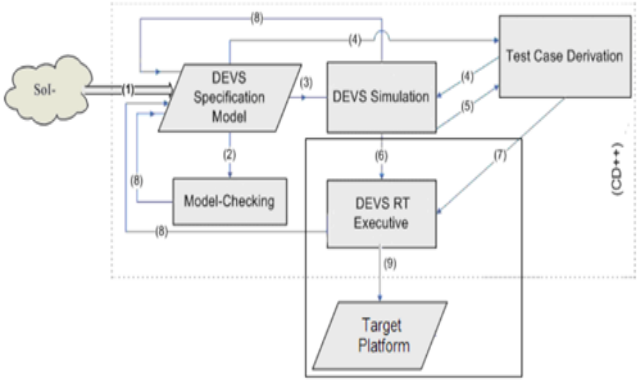

Fig. 3 DEMES Development cycle.

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.

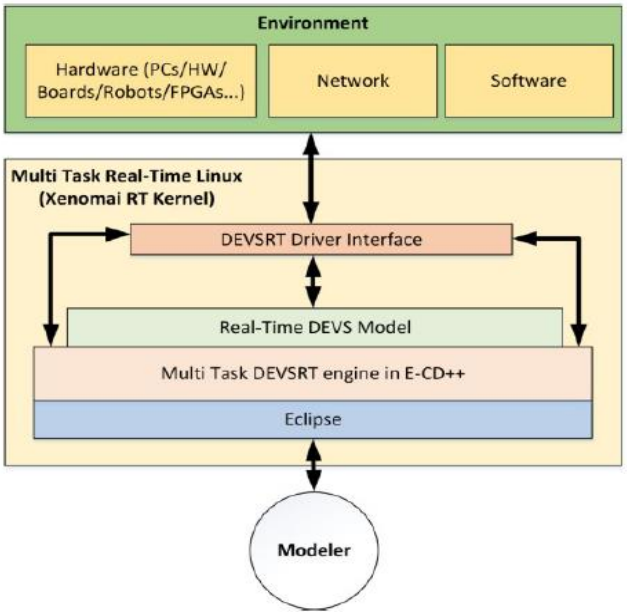

Fig. 4 ECDBoost development framework.

E. Seed Shield Studio Bot

Seed Shield Studio Bot is plug and play open source, Arduino based robot. It has 5 IR reflectance sensors for line and edge following. It has 6 Grove ports and these ports enable easy attachment of more sensors and actuators. It has two durable 160:1 micro metal gearmotors and stackable shield headers.

F. Hardware Overview

Power Switch: When Power switch is off ShieldBot cannot run but the USB Charge Port can be used to charge the battery.

USB Charge Port: USB mini-B

Grove Ports: Grove ports access pins D0, D1, D2, D3, D4, D5, A4, A5.can connect Grove modules to these Grove ports.

IR Line Finder Potentiometer: Used for adjusting the Line Finders Sensitivity. Clockwise, the Sensitivity increases; Counter-clockwise, the Sensitivity decreases.

IR Line Finders: S1 to S5. Blue if non-reflective surface.

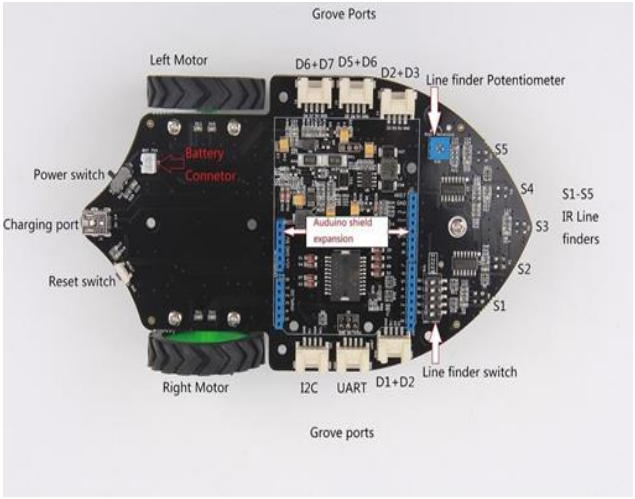

Fig. 5 Seed Shield Robot.

II. MODELS

The system modelled is a Tram Controller. It is designed to halt at various stops (Stop 1-5) according to the stop buttons pressed by the passengers. A passenger from inside the tram as well as outside the tram can choose the halting destination of the tram. This is a two-way tram; therefore, it can travel in both the directions (forward and backward).

A. Conceptual Model Description

Stop controller is an atomic model which receives the input from the digital push buttons or the infra-red sensors and gives the output to the atomic model wheel controller. Based on the received input, wheel controller gives the output to the left and right wheels of the motor. At a time, there can be only one external input either from the sensors or from the push buttons. Initially the Tram rests at station 1 and continues to do so until an external event occurs.

From station 1 it moves forward on occurrence of an external event and thereafter it may travel forward or backward according to the halting destinations of the passengers.

B. Atomic Models

STOP CONTROLLER

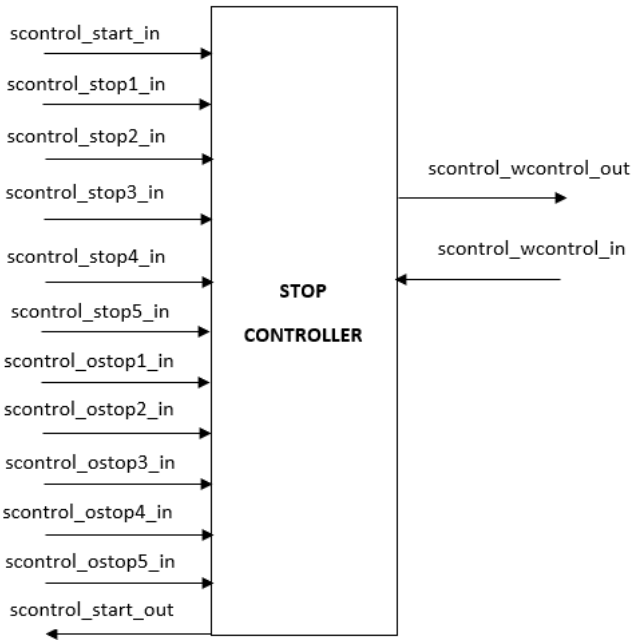

Fig.6 Atomic Model Stop Controller:

Stop Controller activates the sensors on receiving the start input on one of its input ports. The activated sensors are now ready to take the inputs. The five inputs scontrol_stop1_in, scontrol_stop2_in, scontrol_stop3_in, scontrol_stop4_in, scontrol_stop5_in represent inputs from passengers inside the tram and the inputs scontrol_ostop1_in,scontrol_ostop2_in,scontrol_ostop3_in, scontrol_ostop4_in, scontrol_ostop5_in represents the inputs from the passengers outside the tram.
The atomic model Stop Controller gives the output to the atomic model Wheel Controller. The output of the model is scontrol_wcontrol_out which represents the next stop at which the Tram is to be halted.

X = {scontrol_start_in, scontrol_stop1_in, scontrol_stop2_in, scontrol_stop3_in, scontrol_stop4_in, scontrol_stop5_in, scontrol_ostop1_in, scontrol_ostop2_in, scontrol_ostop3_in, scontrol_ostop4_in, scontrol_ostop5_in}

Y = {scontrol_wcontrol_out}

S = { IDLE, READY, WAIT, TRANSFER, STOP }

δ_{ext}:

```
If (State = WAIT)
{
 If ( Requested Stop from Outside = 1) {
    Stop = 1;
}
  Else if ( Requested Stop from Outside = 2) {
    Stop = 2;
}
  Else if ( Requested Stop from Outside = 3) {
    Stop = 3;
}
  Else if ( Requested Stop from Outside = 4) {
    Stop = 4;
}
  Else if ( Requested Stop from Outside = 5) {
    Stop = 5;
}
  Else if ( Requested Stop from Inside = 1) {
    Stop = 1;
}
  Else if ( Requested Stop from Inside = 2) {
    Stop = 2;
}
  Else if ( Requested Stop from Inside = 3){
    Stop = 3;
}
  Else if ( Requested Stop from Inside = 4){
    Stop = 4;
}
  Else if ( Requested Stop from Inside = 5){
    Stop = 5;
}
 Else
{
 Stop = 0;
}
}
```

δ_{int} () { Passivate}

λ (s) {Send Stop number to the port out}

WHEEL CONTROLLER

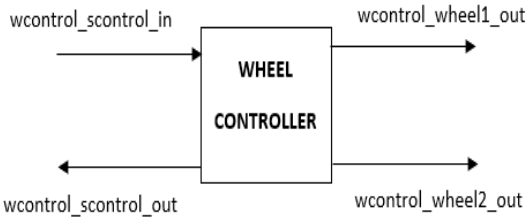

Fig.7 Atomic Model Wheel Controller

The atomic model Wheel Controller receives the input from the model Stop Controller. The input of wheel controller is the station number at which the tram is to be

halted. The wheel controller receives the input and sends the output to the wheels of the motor. The wheels rotate, and the Tram reaches its destination. The wheels are in rest position until the next input.

X = { wcontrol_scontrol_in }

Y = {wcontrol_wheel1_out, wcontrol_wheel2_out}

S = {START, WAIT, TO_FORWARD, FORWARD, TO_BACKWARD, BACKWARD }

δ_{ext}
{
If (state = WAIT)
{
  If (Request stop is greater than current stop)
    State = TO_FORWARD
  ElseIf (Request stop is less than current stop)
    State = TO_BACKWARD
}
}

δ_{int} () { Passivate }

λ (s) {Send Output to left and right wheels}

C. Coupled Model

TOP MODEL: TRAM CONTROLLER

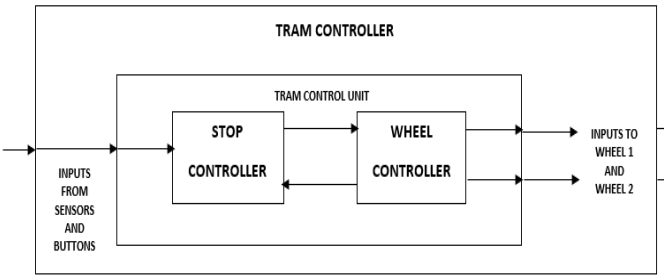

Fig.8 Top Coupled Model

X = {System_In}

Y = {System_Out1, System_Out2}

D = {Tram Control Unit}

EIC = {{Self, System_In}, {Tram Control Unit, Sensors/Buttons Input}}

EOC = {{Self, System_Out1}, {Self, System_Out2}, {Tram Control Unit, Wheel_Out1}, {Tram Control Unit, Wheel_Out2}

COUPLED MODEL: TRAM CONTROL UNIT

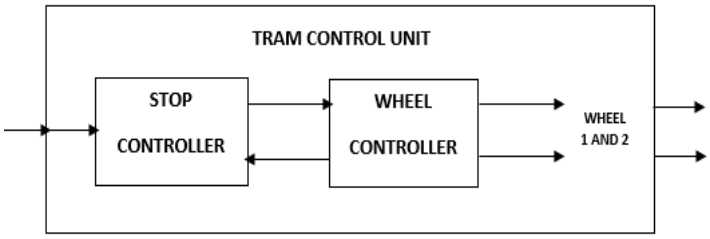

Fig.9 Top Coupled Model:

X = {Sensor/ Push button Inputs}

Y = {Output to wheels}

D = {Stop Controller, Wheel Controller}

EIC = {{Self, Sensors/Buttons Input}, {Stop Controller Sensors/Buttons Input }}

EOC = {{Self, Output to wheels}, {Wheel Controller, wheel1}, {Wheel Controller, wheel2}}

IC = {{Stop Controller, scontrol_mcontrol_out}, {Wheel Controller, wcontrol_scontrol_in}, {Wheel Controller, wcontrol_scontrol_out}, {Stop Controller, scontrol_mcontrol_in}}

SELECT = {Stop Controller, Wheel Controller}

III. IMPLEMENTATION

A. Sensors

Five Infra-red sensors labelled S1, S2, S3, S4 and S5 are used for input to the Tram. When an input is received from any of the sensors, the Tram moves to station 1,2,3,4 or 5 accordingly. The sensors ports used are A0, A1, A2, A3, D4.

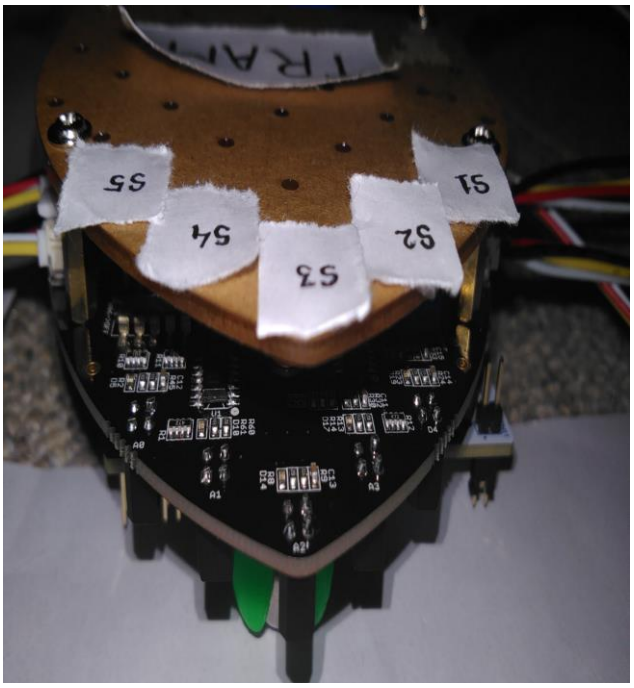

Fig.10 Infra-red sensors

B. Digital Push buttons

Digital push buttons are used to give an input to the Tram. On pressing the push button, the Tram receives the input and moves to the requested station.
Fig 12 shows the connection of 5 push buttons to the seed shield robot.

Fig.11 Digital Push Button

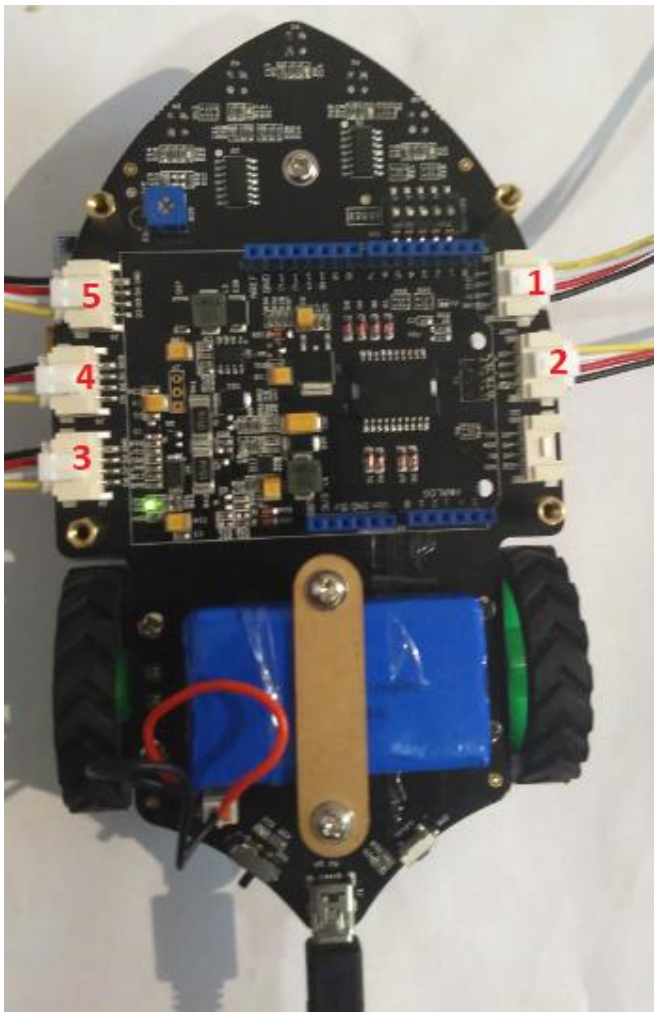

Fig.12 Digital Push Button Connections

C. Working

The Tram rests at station 1 until and unless an input through sensors or push buttons is given to it. When an input is provided to one of the infra-red sensors (A0, A1, A2, A3, D4) the tram starts moving forward. When Tram is at station 1 it cannot move backwards, in all other cases it may move in any direction. Each sensor represents a specific station; therefore, the next destination is determined by input to a particular sensor.
Similarly, there are five push buttons and each push button represent a particular station. When a button is clicked the Tram starts moving forward or backwards accordingly.
The Tram can be controlled by using a sensor or a button for both forward and backward movements. It can also be controlled by using a sensor for forward movement and push button for backward movement or vice versa. Figure 13 shows the Tram in rest position (station1). When an input is given to one of the infra-red sensors (figure 14), the tram receives the input and moves to the station3 (figure 14). It then remains at station 3 until next input is given to the Tram.

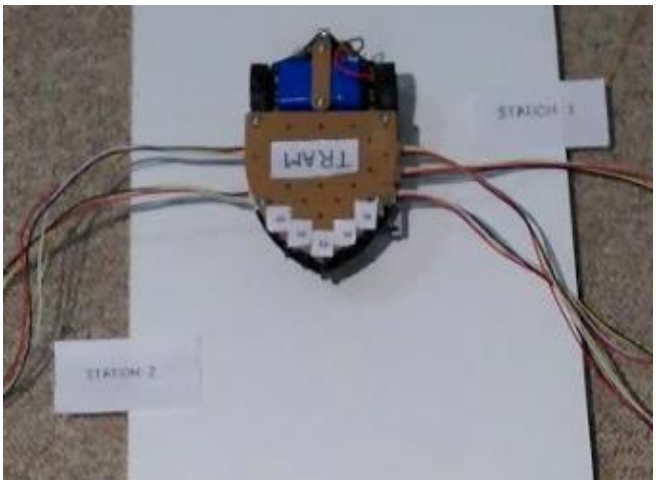

Fig.13 Initial Position of Tram

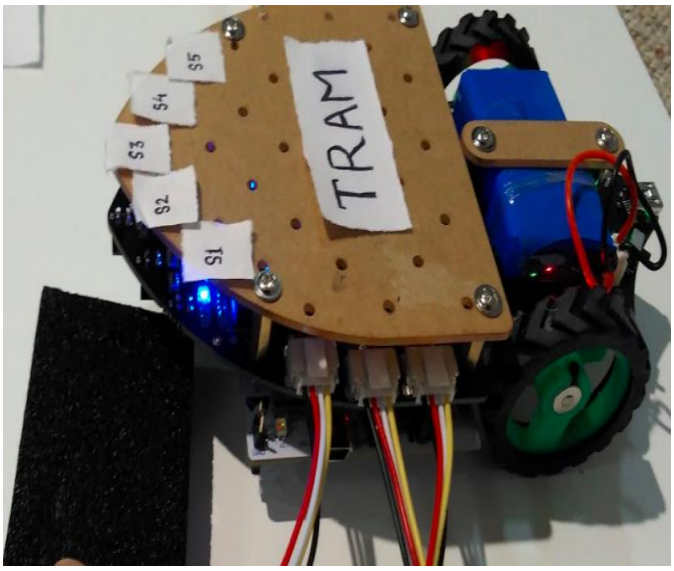

Fig.14 Input to Sensor

D. Code Snippets

This section will present code snippets of the working model. The internal and external function code snippets of the atomic models: Stop Controller and Wheel Controller are also covered in the same.

Fig.15,16,17,18 represents a code snippet from the stop_driver.hpp file. In this file various buttons (push1, push2, push3, push4, push5) are declared. Also, the wheel pins and Sensors pin declaration can be seen below.

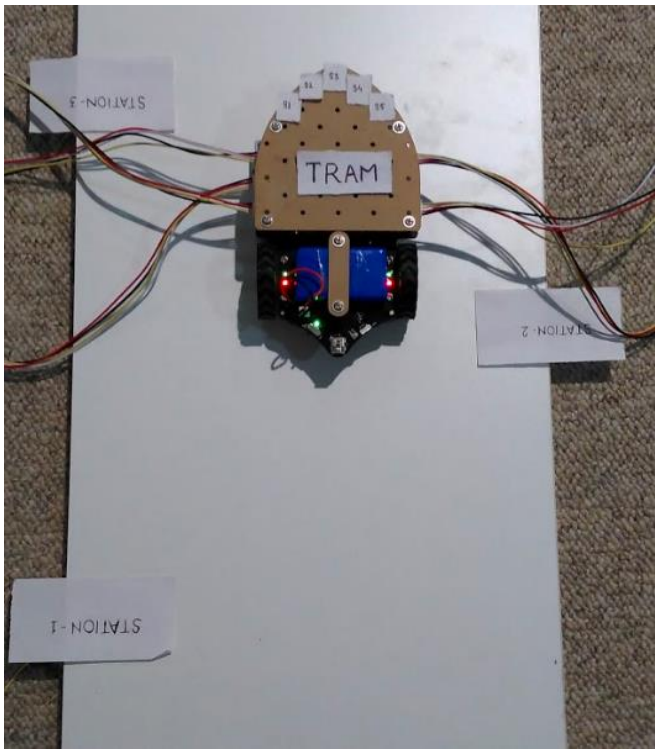

Fig.14 Tram Movement

Fig 20, 21and 22 represents internal and external function of Stop Controller model. Similarly, Fig 23 and 24 represents external and internal functions of Wheel controller model.

```
#ifndef stop_driver_HPP_
#define stop_driver_HPP_

#include "mbed.h"
#include <iostream>
#include <fstream>
#include <string>
#include "SeeedStudioShieldBot.h"
#include "stop_port.hpp"

using namespace std;
using namespace boost::simulation;
using namespace boost::simulation::pdevs;
using namespace boost::simulation::pdevs::basic_models;

DigitalIn start_btn(PC_13);
DigitalIn push1(D3);
DigitalIn push2(D2);
DigitalIn push3(D6);
DigitalIn push4(SERIAL_TX);
DigitalIn push5(D7);

SeeedStudioShieldBot bot(
    D8, D9, D11,            // Wheel 1 pins
    D12, D10, D13,          // Wheel 2 pins
    A0, A1, A2, A3, D4      // Stop Sensors' pins
);
```

Fig.15 stop_driver.hpp

```
   template<class TIME, class MSG>
bool START_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
        if(!start_btn){
            if(!isStarted){
                  isStarted = true;
                  v = 200;
            }
            else{
                  isStarted = false;
                  v = 201;
            }
            return true;
        }
        else{
            return false;
        }
}
```

Fig.16 stop_driver.hpp

```
template<class TIME, class MSG>
bool STOP1_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
      if(bot.leftSensor == 0){
            v=1;
      }
      else v=0;
      return true;
}

template<class TIME, class MSG>
bool STOP2_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
      if(bot.inLeftSensor == 0){
            v=1;
      }
      else v=0;
      return true;
}

template<class TIME, class MSG>
bool STOP3_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
      if(bot.getCentreSensor() == 0){
            v=1;
      }
      else v=0;
      return true;
}

template<class TIME, class MSG>
bool STOP4_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
      if(bot.inRightSensor == 0){
            v=1;
      }
      else v=0;
      return true;
}


template<class TIME, class MSG>
bool STOP5_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
      if(bot.rightSensor == 0){
            v=1;
      }
      else v=0;
      return true;
```

Fig.17 stop_driver.hpp

```
template<class TIME, class MSG>
bool OSTOP1_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
      v=push1;
      return true;
}
template<class TIME, class MSG>
bool OSTOP2_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
      v=push2;
      return true;
}
template<class TIME, class MSG>
bool OSTOP3_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
      v=push3;
      return true;
}
template<class TIME, class MSG>
bool OSTOP4_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
      v=push4;
      return true;
}
template<class TIME, class MSG>
bool OSTOP5_IN<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept {
      v=push5;
      return true;
}
```

Fig.18 stop_driver.hpp

```
bool WHEEL1_OUT<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept{
      if(startWheel1){
            bot.enable_left_motor();
            startWheel1 = false;
      }
      if(cmd==TOKEN){
            switch(v){
            case 12:
                  bot.left_motor(0.2);
                  break;
            case 13:
                  bot.left_motor(-0.2);
                  break;
            case 14:
                  bot.left_motor(0);
                  break;
            }}
      return true;
}

template<class TIME, class MSG>
bool WHEEL2_OUT<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept{
      if(startWheel2){
            bot.enable_right_motor();
            startWheel2 = false;
      }
      if(cmd==TOKEN){
            switch(v){
            case 12:
                  bot.right_motor(0.2);
                  break;
            case 13:
                  bot.right_motor(-0.2);
                  break;
            case 14:
                  bot.right_motor(0);
                  break;
            }}
      return true;
}

template<class TIME, class MSG>
bool LAST_STOP<TIME, MSG>::pDriver(int &cmd, int &v) const noexcept{

return true;
}
```

Fig.19 wheels_driver.hpp

```
/**
 * @brief internal function.
 */
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;
        }
}
```

Fig.20 Internal Function StopController.hpp

```
void external(const std::vector<MSG>& mb, const TIME& t) noexcept {
MSG msg = mb.back();

        if (msg.port == portName[scontrol_start_in]){
                if(_state == IDLE && msg.val == START_PROC){
                        _state = PREP_RX;
                        _next = scRxPrepTime;
                }
                else if (msg.val == STOP_PROC) {
                        _state = PREP_STOP;
                        _next = Time::Zero;
                }
        }

        else if (msg.port == portName[scontrol_stop1_in]){
                if(_state == WAIT_DATA) {
                        stop1 = static_cast<int>(msg.val);
                        if(stop1 == 1){
                        stopNum = 1;
                        _state = TX_DATA;
                        _next = scTxTime;
                        }
                }
        }
        else if (msg.port == portName[scontrol_stop2_in]){
                if(_state == WAIT_DATA) {
                        stop2 = static_cast<int>(msg.val);
                        if(stop2 == 1){
                        stopNum = 2;
                        _state = TX_DATA;
                        _next = scTxTime;
                        }
                }
        }
        else if (msg.port == portName[scontrol_stop3_in]){
                if(_state == WAIT_DATA) {
                        stop3 = static_cast<int>(msg.val);
                        if(stop3 == 1){
                        stopNum = 3;
                        _state = TX_DATA;
                        _next = scTxTime;
                        }

                }
        }
        else if (msg.port == portName[scontrol_stop4_in]){
                if(_state == WAIT_DATA) {
                        stop4 = static_cast<int>(msg.val);
                        if(stop4 == 1){
                        stopNum = 4;
                        _state = TX_DATA;
                        _next = scTxTime;
                        }
                }
```

Fig.21 External Function StopController.hpp

```
else if (msg.port == portName[scontrol_stop5_in]){
                if(_state == WAIT_DATA) {
                        stop5 = static_cast<int>(msg.val);
                        if(stop5 == 1){
                        stopNum = 5;
                        _state = TX_DATA;
                        _next = scTxTime;
                        }
                }
        }
        else if (msg.port == portName[scontrol_ostop1_in]){
                if(_state == WAIT_DATA) {
                        ostop1 = static_cast<int>(msg.val);
                        if(ostop1 == 1){
                        stopNum = 6;
                        _state = TX_DATA;
                        _next = scTxTime;
                        }
                }
        }
        else if (msg.port == portName[scontrol_ostop2_in]){
                if(_state == WAIT_DATA) {
                        ostop2 = static_cast<int>(msg.val);
                        if(ostop2 == 1){
                        stopNum = 7;
                        _state = TX_DATA;
                        _next = scTxTime;
                        }
                }
        }
        else if (msg.port == portName[scontrol_ostop3_in]){
                if(_state == WAIT_DATA) {
                        ostop3 = static_cast<int>(msg.val);
                        if(ostop3 == 1){
                        stopNum = 8;
                        _state = TX_DATA;
                        _next = scTxTime;
                        }

                }
        }
        else if (msg.port == portName[scontrol_ostop4_in]){
                if(_state == WAIT_DATA) {
                        ostop4 = static_cast<int>(msg.val);
                        if(ostop4 == 1){
                        stopNum = 9;
                        _state = TX_DATA;
                        _next = scTxTime;
                        }
                }
        }
```

Fig.22 External Function StopController.hpp

```
/**
 * @brief internal function.
 */
void internal() noexcept {
  switch (_state){
                case IDLE:
                        _state = WAIT_DATA;
                        _next = infinity;
                        break;

                case MOVE_FWD:
                        stn=stn+1;
                        if(stn< scontrol_input){
                                _state = PREP_MOVE_FWD;
                                _next = startPrepTime;
                        }
                        else{
                                _state = WAIT_DATA;
                                _next = infinity;
                        }
                        break;

                case PREP_MOVE_FWD:
                        _state = MOVE_FWD;
                        _next = startTime;
                        break;

                case MOVE_BWD:
                        stn=stn-1;
                        if(stn > scontrol_input){
                                _state = PREP_MOVE_BWD;
                                _next = startPrepTime;
                        }
                        else{
                        _state = WAIT_DATA;
                        _next = infinity;
                        }
                        break;

                case PREP_MOVE_BWD:
                        _state = MOVE_BWD;
                        _next = startTime;
                        break;

                case PREP_STOP:
                        _state = WAIT_DATA;
                        _next = infinity;
                        break;

                default:
                        break;
  }
}
```

Fig.23 Internal Function WheelController.hpp

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

    MSG msg = mb.back();
    if (msg.port == portName[wcontrol_scontrol_in]) {
                if(_state == WAIT_DATA ) {
                      scontrol_input = static_cast<int>(msg.val);
                      if (scontrol_input - stn > 0) {
                            _state = PREP_MOVE_FWD;
                            _next = startPrepTime;

                      }
                      else if (scontrol_input - stn < 0) {
                            _state = PREP_MOVE_BWD;
                            _next = startPrepTime;

                      }
                        else {
                            //print invalid
                      }
                }
                if(msg.val == STOP_PROC){
                      _state = PREP_STOP;
                      _next = Time::Zero;
                }
    }

}
```

Fig.24 External Function WheelController.hpp

```
// Atomic models definition
printf("Creating atomic models ... \n");
auto scontrol = make_atomic_ptr<StopController<Time, Message>>();
auto wcontrol = make_atomic_ptr<WheelController<Time, Message>>();

//Coupled model definition
printf("Creating Coupled models ... \n");
shared_ptr<flattened_coupled<Time, Message>>
ControlUnit( new flattened_coupled<Time, Message>
{{scontrol,wcontrol}, {scontrol}, {{scontrol,wcontrol}}, {wcontrol}});
```

Fig.25 Atomic and Couple Model definition in main.cpp

```
//Top I/O port definition
      printf("Defining top I/O ports ... \n");

      // Input ports
      auto start = make_port_ptr<START_IN<Time, Message>>();
      auto stop1 = make_port_ptr<STOP1_IN<Time, Message>>();
      auto stop2 = make_port_ptr<STOP2_IN<Time, Message>>();
      auto stop3 = make_port_ptr<STOP3_IN<Time, Message>>();
      auto stop4 = make_port_ptr<STOP4_IN<Time, Message>>();
      auto stop5 = make_port_ptr<STOP5_IN<Time, Message>>();
      auto ostop1 = make_port_ptr<OSTOP1_IN<Time, Message>>();
      auto ostop2 = make_port_ptr<OSTOP2_IN<Time, Message>>();
      auto ostop3 = make_port_ptr<OSTOP3_IN<Time, Message>>();
      auto ostop4 = make_port_ptr<OSTOP4_IN<Time, Message>>();
      auto ostop5 = make_port_ptr<OSTOP5_IN<Time, Message>>();

   // Output ports
      auto wheel1 = make_port_ptr<WHEEL1_OUT<Time, Message>>();
      auto wheel2 = make_port_ptr<WHEEL2_OUT<Time, Message>>();
```

Fig.26 Input and Output Port definitions in main.cpp

```
      // Execution parameter definition
      printf("Preparing runner \n");
      Time initial_time{Time::currentTime()};
      erunner<Time, Message> root{ControlUnit, {{start,scontrol},
                  {stop1,scontrol},{stop2,scontrol}, {stop3,scontrol},
                  {stop4,scontrol},{stop5,scontrol},{ostop1,scontrol},
                  {ostop2,scontrol}, {ostop3,scontrol},{ostop4,scontrol},
                  {ostop5,scontrol}} , {{wheel1,wcontrol},{wheel2,wcontrol}}};
      Time end_time{Time(0,30,0,0)};

      printf(("Starting simulation until time: seconds " + end_time.asString() +
"\n").c_str());
      end_time = root.runUntil(end_time);


}
```

Fig.27 Execution parameter definition in main.cpp

Fig 25, 26, and 27 shows the code snippet from the main.cpp file where atomic and couple models are defined along with the input and output ports.

E. Limitations and Challenges

Initially, the idea of the project was to build it into three phases –

1. Stop requests from inside passengers
2. Stop requests from outside passengers waiting at different station
3. Multiple and consecutive requests

But due to the limitations of DEVS modeling the third phase of the project is out of scope of this project. As DEVS can handle only one external input at a given time, therefore, simultaneous multiple station requests could not be implemented. It was also difficult to give two consecutive requests as input while first is still under process.

The first phase of this project, which is Stop requests from inside passengers, is successfully implemented and properly tested. On the other hand, the second phase (Stop requests from outside passengers waiting at different station) could not be implemented for all the stations as it causes more complexity.

Taking forward to this working DEVS implemented model, instead of Push Buttons, the wireless buttons can also be used which will not deviate the motion of the bot. Some wireless extendable port can be integrated with the existing microcontroller to get it enabled.

F. Conclusion

We have successfully implemented the atomic models of the Tram control system. The Tram is working correctly on receiving external inputs from the Sensors as well as from the digital push button. Due to complex behavior of the system there were problems in implementing the coupled models. Moreover, as of now only one push button for external inputs has been implemented and tested.

REFERENCES

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

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

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

[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] D.Melean. "*Maze Solving Robot Using Discreet Event Methodology for Embedded Systems*".

[6] 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].