Rapport de stage 4^{ième} année

Nicolas GAILLARD-GROLÉAS

Stage à l'Université de Carleton du 01/06/2019 au 03/08/2019

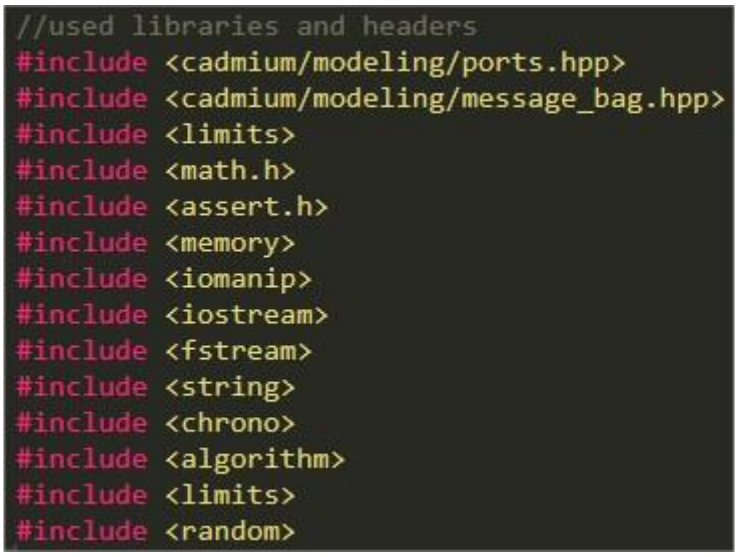
ANNEXE 1 : EXPLICATION DU CODE DÉVELOPPÉ

Cette annexe est en anglais car elle est issue d'un travail demandé par le laboratoire. Elle présente étape par étape le fonctionnement du code implémenté dans le robot.

Il est détaillé ci-après deux codes C++, le premier constitue le modèle atomique du robot et définit son comportement, le second lie les différents capteurs et moteurs au modèle atomique et permet sont fonctionnement dans l'environnement ECadmium.

The following code constitutes the atomic model presented above.

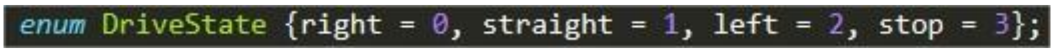
The first step for implementing the model is including the libraries we are using (Figure 2). We need to include the simulator libraries (**cadmium/modeling/ports.hpp** and **cadmium/modeling/message_bag.hpp**) and the C++ libraries we are using for our implementation.



```
//used libraries and headers
#include <cadmium/modeling/ports.hpp>
#include <cadmium/modeling/message_bag.hpp>
#include <limits>
#include <math.h>
#include <assert.h>
#include <memory>
#include <iomanip>
#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include <algorithm>
#include <limits>
#include <random>
```

Figure 1: Headers included in the atomic model definition

Then we have one of the state variables of the model. It is an enumeration containing the drive states (Figure 3). If we wanted to add more states, for example a slow speed state, we could do it here.



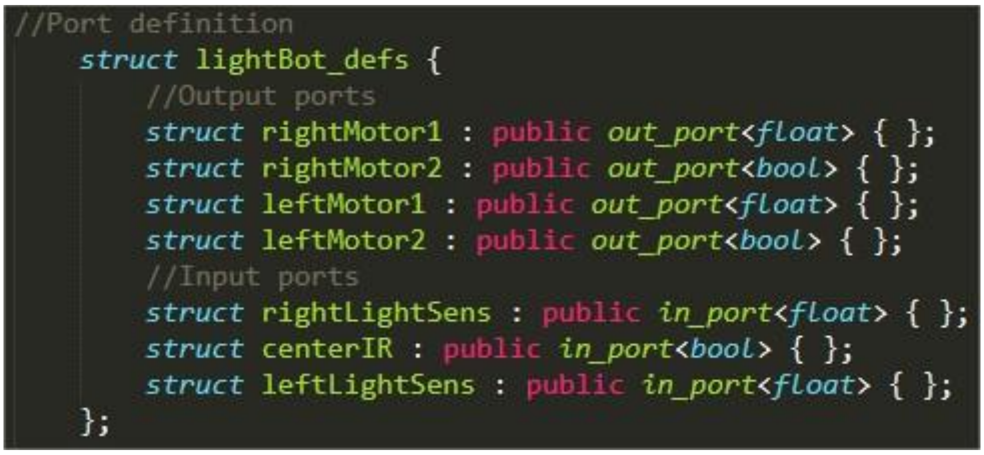
```
enum DriveState {right = 0, straight = 1, left = 2, stop = 3};
```

Figure 2: Drive states

Here we are defining the input and output ports (Figure 4). The Output ports linked to the motors, and the inputs to the three sensors we have onboard.

We can observe that we have two types of data, float and bool.

That is because the sensors on the robot are both analogic and digital. The light sensors are analogic, they return a float value between 0 and 1, whereas the infrared sensor is digital, it sends either 0 or 1 based on an adjustable threshold.

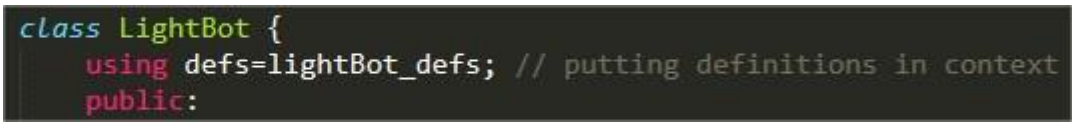


```
//Port definition
    struct lightBot_defs {
        //Output ports
        struct rightMotor1 : public out_port<float> { };
        struct rightMotor2 : public out_port<bool> { };
        struct leftMotor1 : public out_port<float> { };
        struct leftMotor2 : public out_port<bool> { };
        //Input ports
        struct rightLightSens : public in_port<float> { };
        struct centerIR : public in_port<bool> { };
        struct leftLightSens : public in_port<float> { };
    };
```

Figure 3: Input and output ports

We can now start building the main class of this model, LightBot. The following code will be inside this class.

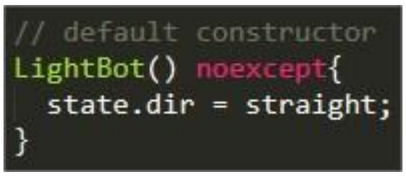
The first line of this class allows the use of the previously defined ports inside the class, with the shorter name « defs » (Figure 5).



```
class LightBot {
    using defs=lightBot_defs; // putting definitions in context
    public:
```

Figure 4: Start of the LightBot class

This is the default constructor (Figure 6), which means the original state of the robot will be to go straight. We could also change it to be the stop state.



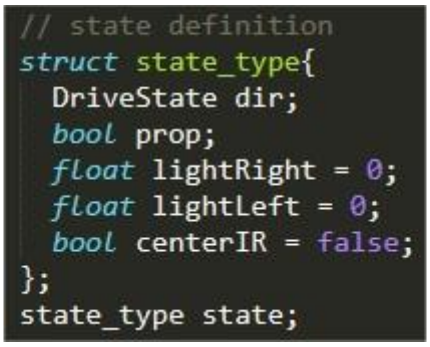
```
// default constructor
LightBot() noexcept{
  state.dir = straight;
}
```

Figure 5: Default constructor

Now this is where the DEVS formalism is implemented:

First we have the state definition (Figure 7), where we can find the different drive states of the robot contained in « DriveState dir» and also the prop variable.

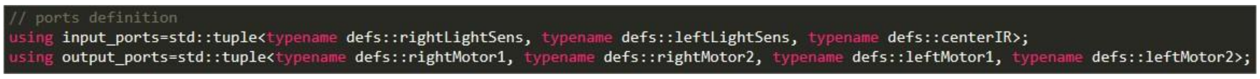
But we also have states for the sensors that store the previous values, making sure there are always correct values for the external transition to compute. In the event the sensor data isn’t pulled at the same time, it will use the previous value contained in the corresponding state.



```
// state definition
struct state_type{
  DriveState dir;
  bool prop;
  float lightRight = 0;
  float lightLeft = 0;
  bool centerIR = false;
};
state_type state;
```

Figure 6 : State definition

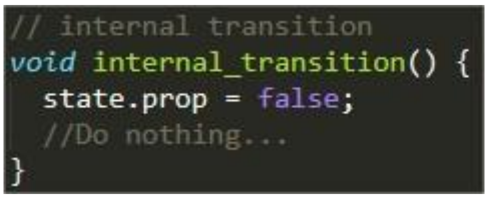
Then we have inputs and outputs. We already defined those earlier, but now we are telling the model what are the X and Y vectors (Figure 8).



```
// ports definition
using input_ports=std::tuple<typename defs::rightLightSens, typename defs::leftLightSens, typename defs::centerIR>;
using output_ports=std::tuple<typename defs::rightMotor1, typename defs::rightMotor2, typename defs::leftMotor1, typename defs::leftMotor2>;
```

Figure 7: X and Y vectors

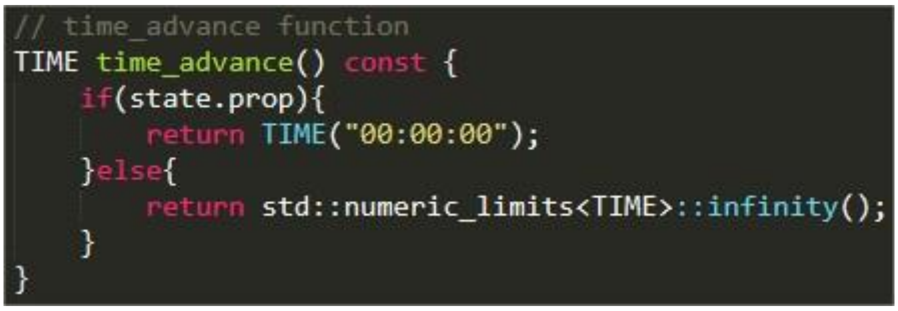
The internal transition function’s purpose is to change the state variable « prop » to false. By doing so, it allows the outputs to be sent to the motors (Figure 9).



```
// internal transition
void internal_transition() {
  state.prop = false;
  //Do nothing...
}
```

Figure 8: Internal transition function

Associated to the internal transition, there is the time advance function (Figure 10), which decides when the internal will be executed. Here, as explained previously, we have no delay if prop = T, and infinite delay else.



```
// time_advance function
TIME time_advance() const {
    if(state.prop){
        return TIME("00:00:00");
    }else{
        return std::numeric_limits<TIME>::infinity();
    }
}
```

Figure 9: Time advance function

The external transition function (Figure 11) however is where most of the behaviour of the model is defined.

It stores the values coming from the light sensors in two local variables lightRight and lightLeft, and updates the centerIR.

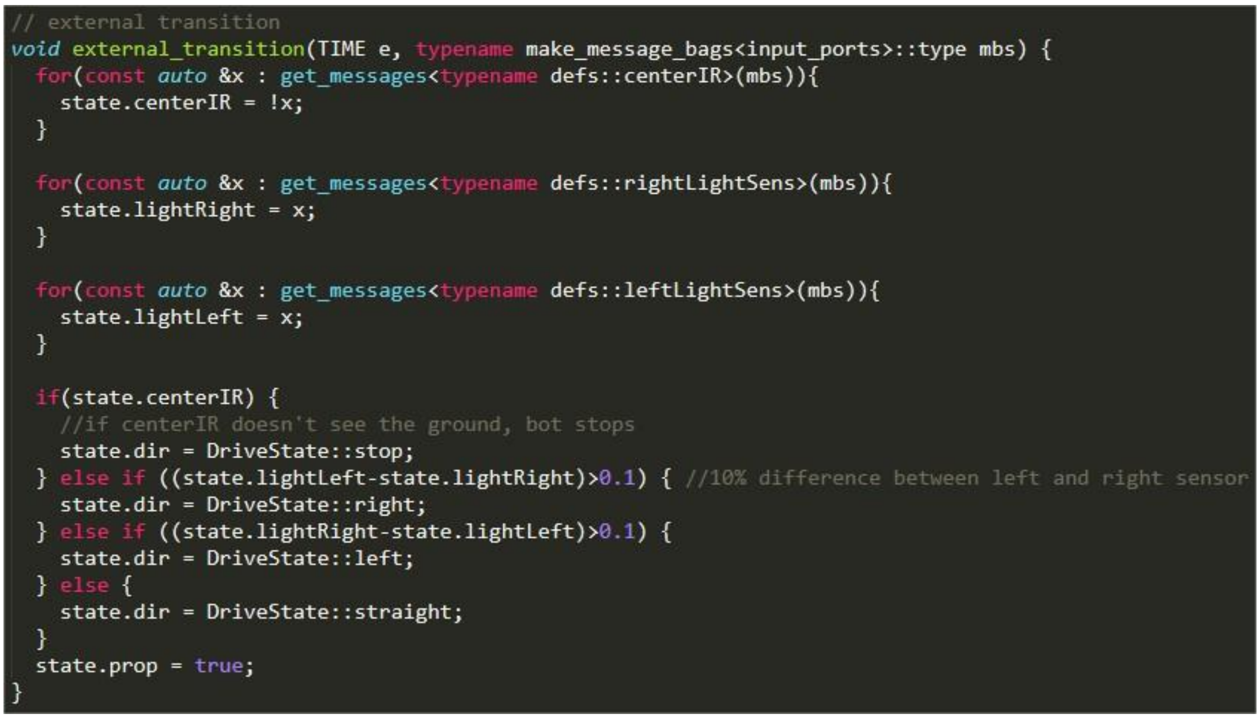
Then with a series of “if” statements, we can decide the next state of the robot.

It works by first checking the value of the infrared sensor. If it is 1, the state is set to stop.

If it is 0, then we check the difference between the values of the right and left light sensors.

If one is higher by more than 10%, we steer in that direction by setting the state to right or left.

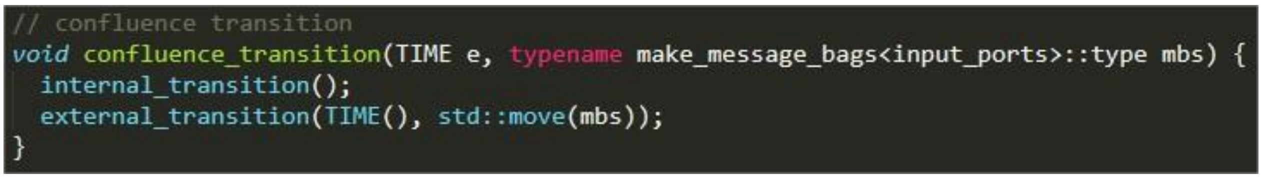
If none of the previous conditions are met, it means the robot is on the ground and the light is in front of it, so we go straight.



```cpp
// external transition
void external_transition(TIME e, typename make_message_bags<input_ports>::type mbs) {
  for(const auto &x : get_messages<typename defs::centerIR>(mbs)){
    state.centerIR = !x;
  }

  for(const auto &x : get_messages<typename defs::rightLightSens>(mbs)){
    state.lightRight = x;
  }

  for(const auto &x : get_messages<typename defs::leftLightSens>(mbs)){
    state.lightLeft = x;
  }

  if(state.centerIR) {
    //if centerIR doesn't see the ground, bot stops
    state.dir = DriveState::stop;
  } else if ((state.lightLeft-state.lightRight)>0.1) { //10% difference between left and right sensor
    state.dir = DriveState::right;
  } else if ((state.lightRight-state.lightLeft)>0.1) {
    state.dir = DriveState::left;
  } else {
    state.dir = DriveState::straight;
  }
  state.prop = true;
}
```

Figure 10: External transition function

The confluence transition (Figure 12) comes into play if an internal and external transition happen at the same time. It gives the possibility to choose which one to process first. In this case, the internal transition will be executed first because else this would mean a state is skipped and the robot didn't react to an input.



```cpp
// confluence transition
void confluence_transition(TIME e, typename make_message_bags<input_ports>::type mbs) {
  internal_transition();
  external_transition(TIME(), std::move(mbs));
}
```

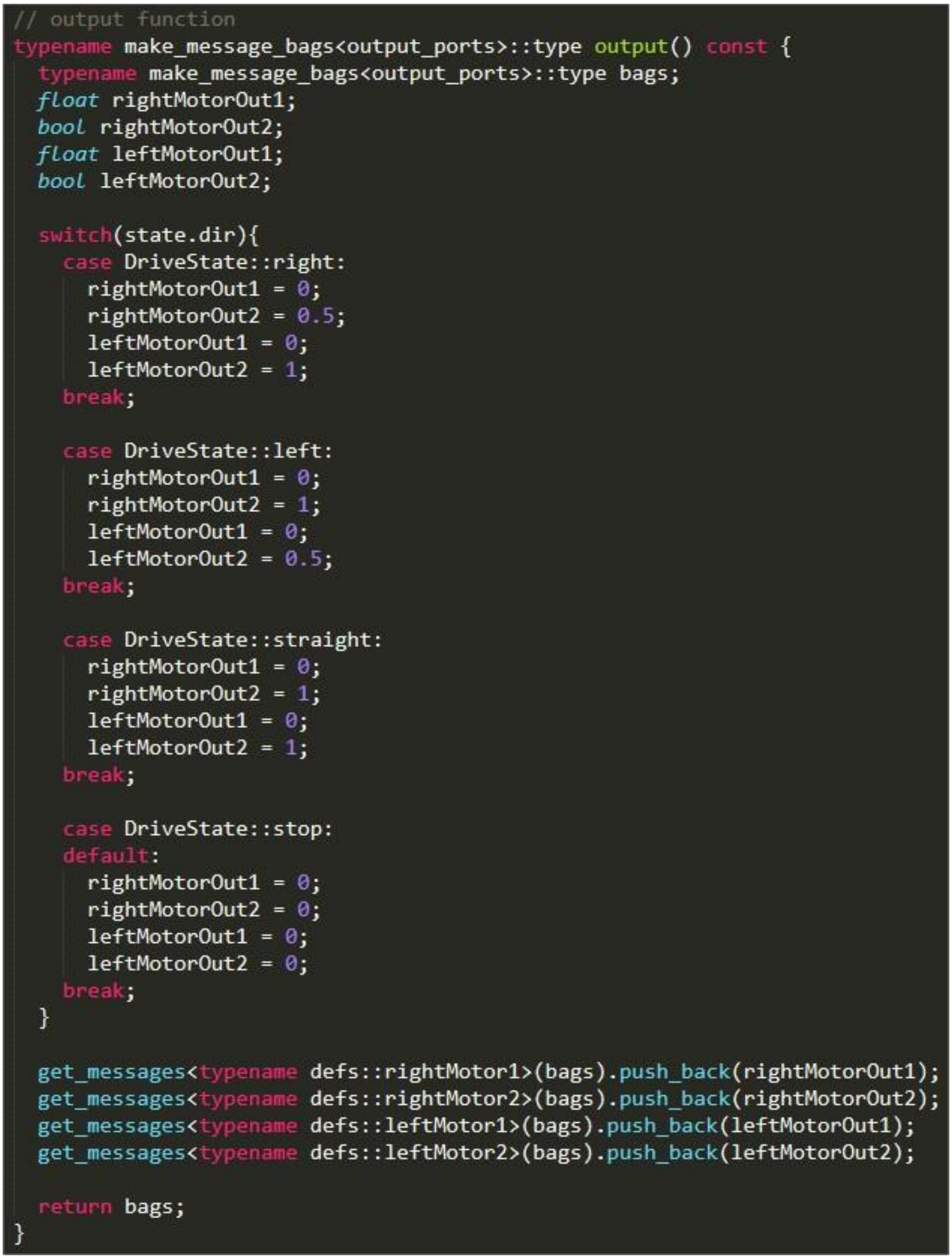
Figure 11: Confluence transition

For the output function (Figure 13), it is simply a switch on the different states, with different values for the motors to make it go in the associated direction.

For example in the straight state, we want both motors to run at the same speed, so we set a difference of 1 between the two values on both motors.

To make the robot turn, we slow down one motor by 50%.

At the end of the switch, the values are sent to the output ports, to the motors.



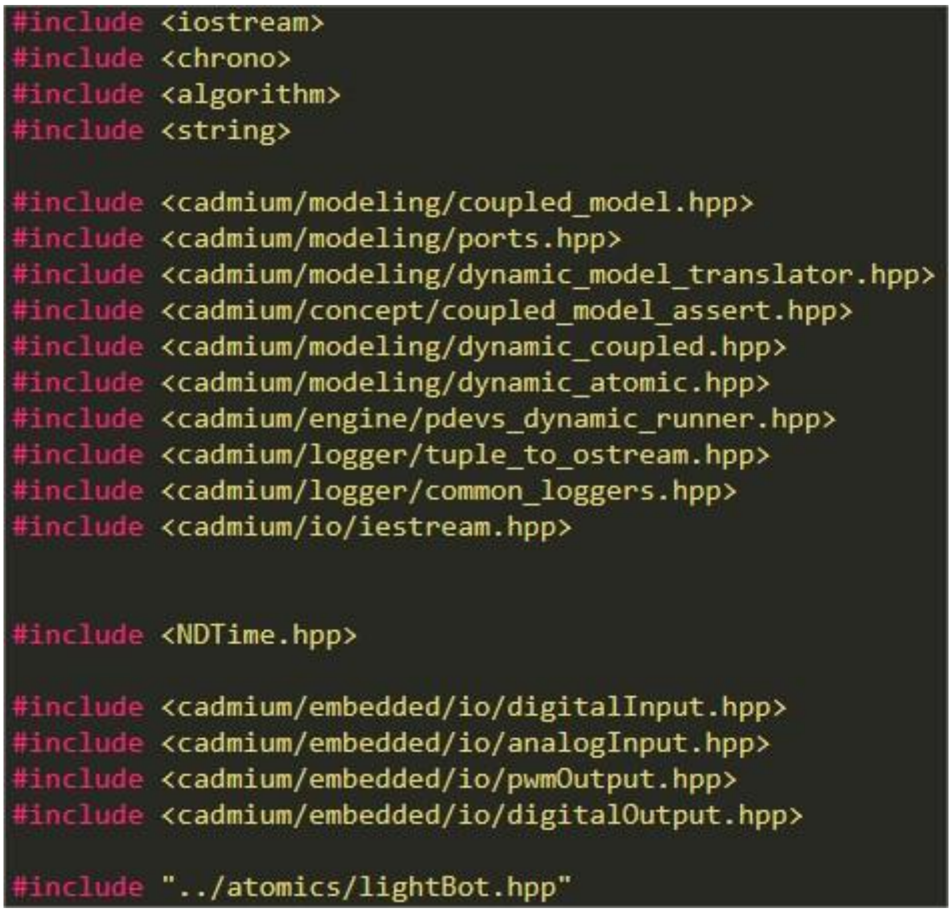
```
// output function
typename make_message_bags<output_ports>::type output() const {
  typename make_message_bags<output_ports>::type bags;
  float rightMotorOut1;
  bool rightMotorOut2;
  float leftMotorOut1;
  bool leftMotorOut2;

  switch(state.dir){
    case DriveState::right:
      rightMotorOut1 = 0;
      rightMotorOut2 = 0.5;
      leftMotorOut1 = 0;
      leftMotorOut2 = 1;
    break;

    case DriveState::left:
      rightMotorOut1 = 0;
      rightMotorOut2 = 1;
      leftMotorOut1 = 0;
      leftMotorOut2 = 0.5;
    break;

    case DriveState::straight:
      rightMotorOut1 = 0;
      rightMotorOut2 = 1;
      leftMotorOut1 = 0;
      leftMotorOut2 = 1;
    break;

    case DriveState::stop:
    default:
      rightMotorOut1 = 0;
      rightMotorOut2 = 0;
      leftMotorOut1 = 0;
      leftMotorOut2 = 0;
    break;
  }

  get_messages<typename defs::rightMotor1>(bags).push_back(rightMotorOut1);
  get_messages<typename defs::rightMotor2>(bags).push_back(rightMotorOut2);
  get_messages<typename defs::leftMotor1>(bags).push_back(leftMotorOut1);
  get_messages<typename defs::leftMotor2>(bags).push_back(leftMotorOut2);

  return bags;
}
```

Figure 12: Output function

Main/Coupled model :

This file contains the code enabling the lightbot atomic model to communicate with the input and outputs and links it to the Cadmium environment.

First of all, we include all the required libraries (Figure 14). We use some standard libraries but also cadmium libraries. And of course we import our lightbot model at the end.



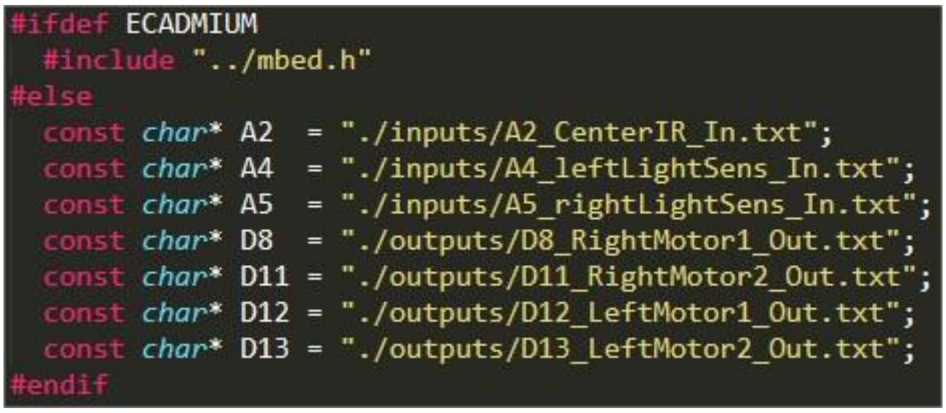
```
#include <iostream>
#include <chrono>
#include <algorithm>
#include <string>

#include <cadmium/modeling/coupled_model.hpp>
#include <cadmium/modeling/ports.hpp>
#include <cadmium/modeling/dynamic_model_translator.hpp>
#include <cadmium/concept/coupled_model_assert.hpp>
#include <cadmium/modeling/dynamic_coupled.hpp>
#include <cadmium/modeling/dynamic_atomic.hpp>
#include <cadmium/engine/pdevs_dynamic_runner.hpp>
#include <cadmium/logger/tuple_to_ostream.hpp>
#include <cadmium/logger/common_loggers.hpp>
#include <cadmium/io/iestream.hpp>


#include <NDTime.hpp>

#include <cadmium/embedded/io/digitalInput.hpp>
#include <cadmium/embedded/io/analogInput.hpp>
#include <cadmium/embedded/io/pwmOutput.hpp>
#include <cadmium/embedded/io/digitalOutput.hpp>

#include "../atomics/lightBot.hpp"
```

Figure 13: Coupled model libraries

Then we can either run the program in simulation mode or embedded mode.

In simulation mode, the inputs are red from pre made text files containing values simulating sensors associated with a time stamp, and the outputs are written to text files in the same fashion. We can then look at the output files and analyse if we get the correct output at the correct time based on the input files.

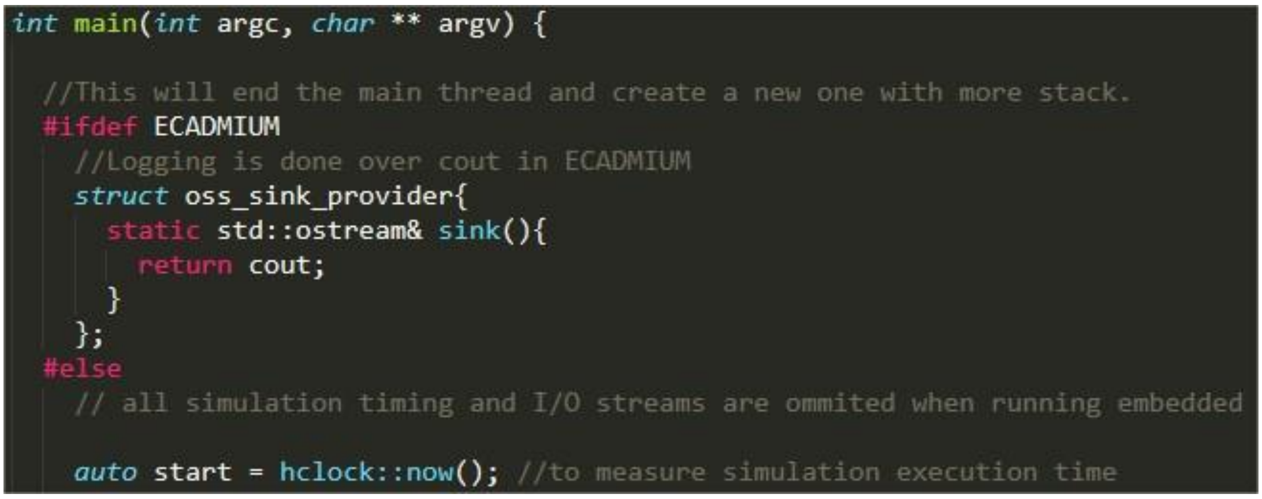
In embedded mode, the inputs and outputs are directly red from the sensors using the mbed library.



```
#ifdef ECADMIUM
  #include "../mbed.h"
#else
  const char* A2  = "./inputs/A2_CenterIR_In.txt";
  const char* A4  = "./inputs/A4_leftLightSens_In.txt";
  const char* A5  = "./inputs/A5_rightLightSens_In.txt";
  const char* D8  = "./outputs/D8_RightMotor1_Out.txt";
  const char* D11 = "./outputs/D11_RightMotor2_Out.txt";
  const char* D12 = "./outputs/D12_LeftMotor1_Out.txt";
  const char* D13 = "./outputs/D13_LeftMotor2_Out.txt";
#endif
```

Figure 1414: Simulation mode inputs and outputs

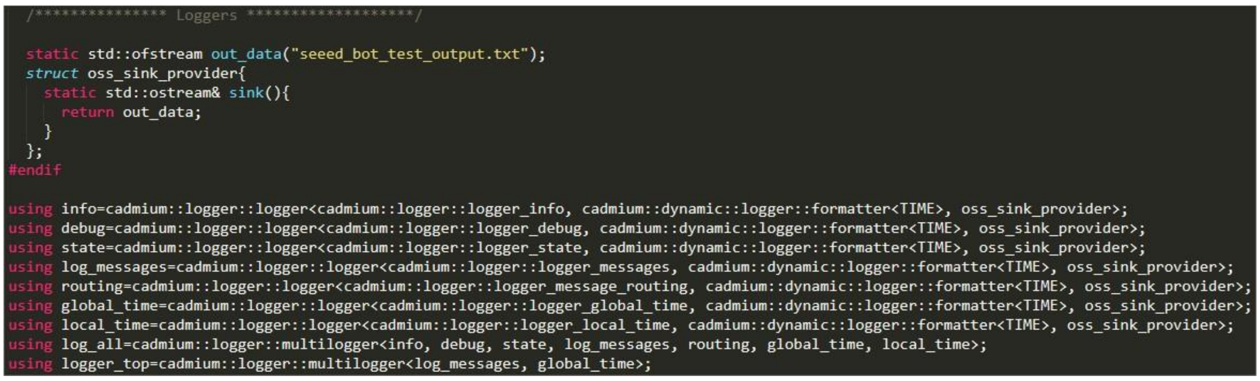
In the main function (Figure 16), we first define how the logging of information is handled. If we use ECadmium, the logging is done over cout in ECadmium, else all simulation timing and I/O streams are omitted when running embedded.



```
int main(int argc, char ** argv) {

  //This will end the main thread and create a new one with more stack.
  #ifdef ECADMIUM
    //Logging is done over cout in ECADMIUM
    struct oss_sink_provider{
      static std::ostream& sink(){
        return cout;
      }
    };
  #else
    // all simulation timing and I/O streams are ommited when running embedded

    auto start = hclock::now(); //to measure simulation execution time
```

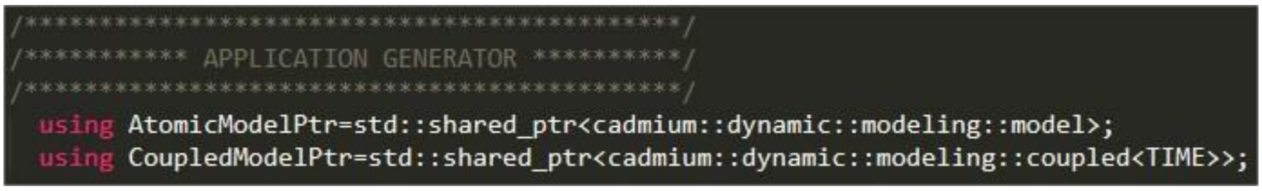
Figure 15: Main function

We can display a multitude of information in the console while the model is running, which helps a lot with debuging. Here we can choose what to display (Figure 17).



```
  /*************** Loggers *******************/

  static std::ofstream out_data("seeed_bot_test_output.txt");
  struct oss_sink_provider{
    static std::ostream& sink(){
      return out_data;
    }
  };
#endif

using info=cadmium::logger::logger<cadmium::logger::logger_info, cadmium::dynamic::logger::formatter<TIME>, oss_sink_provider>;
using debug=cadmium::logger::logger<cadmium::logger::logger_debug, cadmium::dynamic::logger::formatter<TIME>, oss_sink_provider>;
using state=cadmium::logger::logger<cadmium::logger::logger_state, cadmium::dynamic::logger::formatter<TIME>, oss_sink_provider>;
using log_messages=cadmium::logger::logger<cadmium::logger::logger_messages, cadmium::dynamic::logger::formatter<TIME>, oss_sink_provider>;
using routing=cadmium::logger::logger<cadmium::logger::logger_message_routing, cadmium::dynamic::logger::formatter<TIME>, oss_sink_provider>;
using global_time=cadmium::logger::logger<cadmium::logger::logger_global_time, cadmium::dynamic::logger::formatter<TIME>, oss_sink_provider>;
using local_time=cadmium::logger::logger<cadmium::logger::logger_local_time, cadmium::dynamic::logger::formatter<TIME>, oss_sink_provider>;
using log_all=cadmium::logger::multilogger<info, debug, state, log_messages, routing, global_time, local_time>;
using logger_top=cadmium::logger::multilogger<log_messages, global_time>;
```

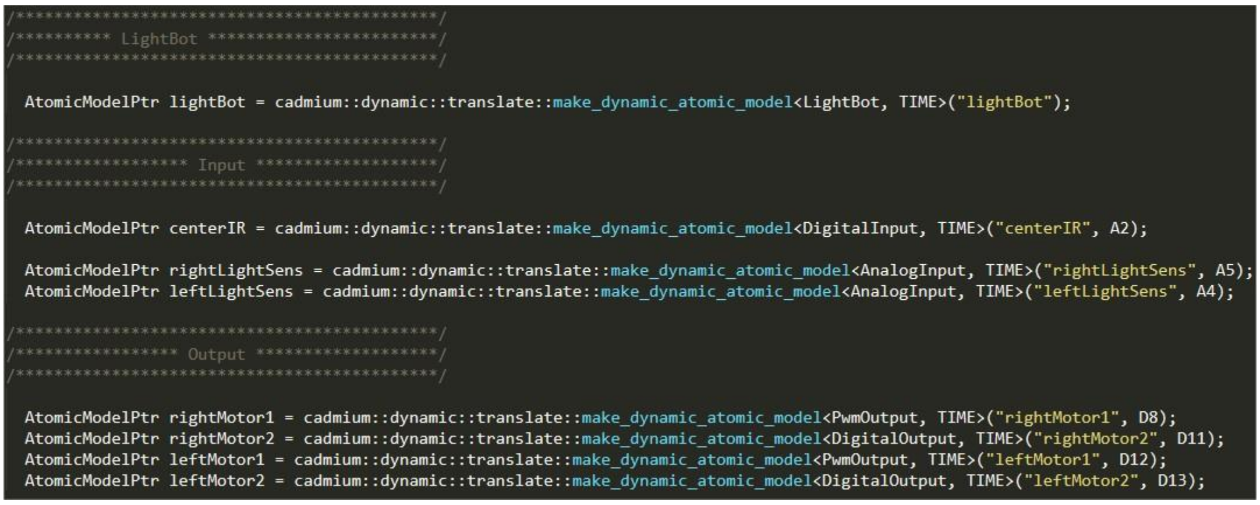
Figure 16: Logging options



```
/*********************************************/
/*********** APPLICATION GENERATOR **********/
/*********************************************/
  using AtomicModelPtr=std::shared_ptr<cadmium::dynamic::modeling::model>;
  using CoupledModelPtr=std::shared_ptr<cadmium::dynamic::modeling::coupled<TIME>>;
```

Figure 17: Application Generator

Then we define all the atomic models composing the coupled model (Figure 19). We have of course the lightbot model, but also one atomic model for each input and output.



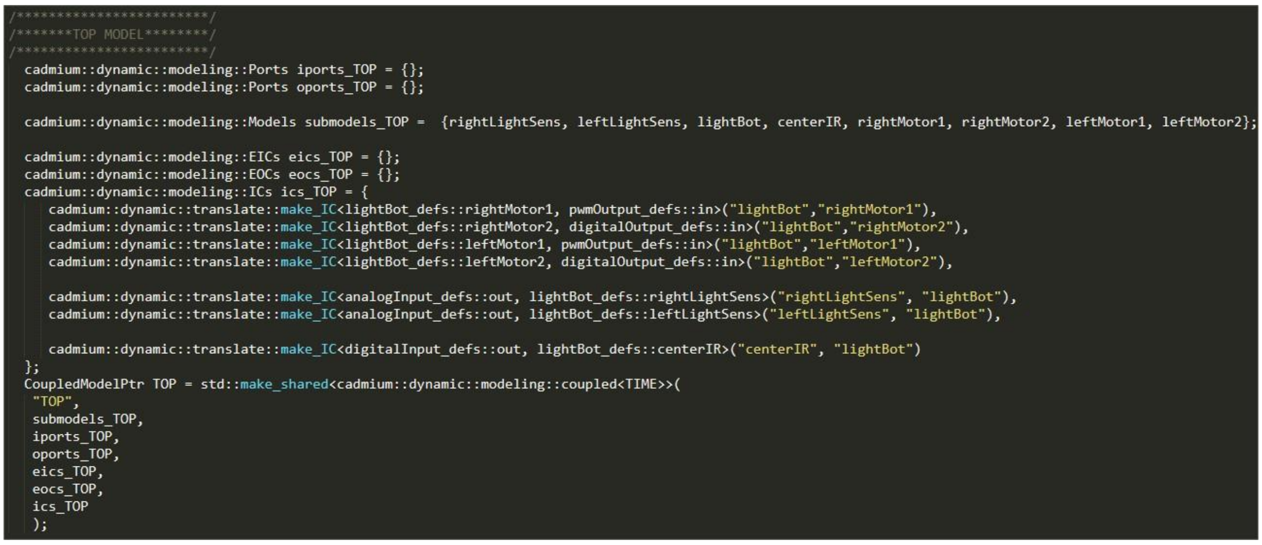
```
/*********************************************/
/********** LightBot ***********************/
/*********************************************/

  AtomicModelPtr lightBot = cadmium::dynamic::translate::make_dynamic_atomic_model<LightBot, TIME>("lightBot");

/*********************************************/
/****************** Input ******************/
/*********************************************/

  AtomicModelPtr centerIR = cadmium::dynamic::translate::make_dynamic_atomic_model<DigitalInput, TIME>("centerIR", A2);

  AtomicModelPtr rightLightSens = cadmium::dynamic::translate::make_dynamic_atomic_model<AnalogInput, TIME>("rightLightSens", A5);
  AtomicModelPtr leftLightSens = cadmium::dynamic::translate::make_dynamic_atomic_model<AnalogInput, TIME>("leftLightSens", A4);

/*********************************************/
/***************** Output ******************/
/*********************************************/

  AtomicModelPtr rightMotor1 = cadmium::dynamic::translate::make_dynamic_atomic_model<PwmOutput, TIME>("rightMotor1", D8);
  AtomicModelPtr rightMotor2 = cadmium::dynamic::translate::make_dynamic_atomic_model<DigitalOutput, TIME>("rightMotor2", D11);
  AtomicModelPtr leftMotor1 = cadmium::dynamic::translate::make_dynamic_atomic_model<PwmOutput, TIME>("leftMotor1", D12);
  AtomicModelPtr leftMotor2 = cadmium::dynamic::translate::make_dynamic_atomic_model<DigitalOutput, TIME>("leftMotor2", D13);
```

Figure 18: Atomics models definition

This is the part where we build the actual coupled model (Figure 20).

The coupled model or top model has no input or output, hence the empty eics_TOP and eocs_TOP lists. However, there are internal connections between all the atomic models, listed in ics_TOP. The first four lines of this list are linking the outputs of the lightbot model to the input of each motor model. The next three lines are linking the outputs of the sensors models to the inputs of the lightbot model.

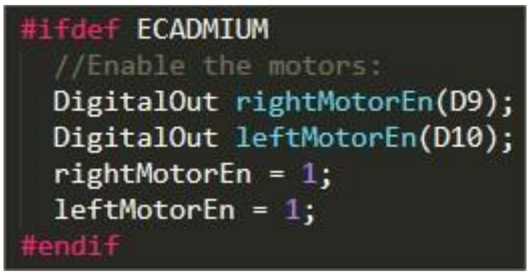
Then the coupled model is created using all these parameters and is called TOP.



```
/************************/
/*******TOP MODEL********/
/************************/
  cadmium::dynamic::modeling::Ports iports_TOP = {};
  cadmium::dynamic::modeling::Ports oports_TOP = {};

  cadmium::dynamic::modeling::Models submodels_TOP =  {rightLightSens, leftLightSens, lightBot, centerIR, rightMotor1, rightMotor2, leftMotor1, leftMotor2};

  cadmium::dynamic::modeling::EICs eics_TOP = {};
  cadmium::dynamic::modeling::EOCs eocs_TOP = {};
  cadmium::dynamic::modeling::ICs ics_TOP = {
     cadmium::dynamic::translate::make_IC<lightBot_defs::rightMotor1, pwmOutput_defs::in>("lightBot","rightMotor1"),
     cadmium::dynamic::translate::make_IC<lightBot_defs::rightMotor2, digitalOutput_defs::in>("lightBot","rightMotor2"),
     cadmium::dynamic::translate::make_IC<lightBot_defs::leftMotor1, pwmOutput_defs::in>("lightBot","leftMotor1"),
     cadmium::dynamic::translate::make_IC<lightBot_defs::leftMotor2, digitalOutput_defs::in>("lightBot","leftMotor2"),

     cadmium::dynamic::translate::make_IC<analogInput_defs::out, lightBot_defs::rightLightSens>("rightLightSens", "lightBot"),
     cadmium::dynamic::translate::make_IC<analogInput_defs::out, lightBot_defs::leftLightSens>("leftLightSens", "lightBot"),

     cadmium::dynamic::translate::make_IC<digitalInput_defs::out, lightBot_defs::centerIR>("centerIR", "lightBot")
  };
  CoupledModelPtr TOP = std::make_shared<cadmium::dynamic::modeling::coupled<TIME>>(
   "TOP",
   submodels_TOP,
   iports_TOP,
   oports_TOP,
   eics_TOP,
   eocs_TOP,
   ics_TOP
   );
```

Figure 19: Top model

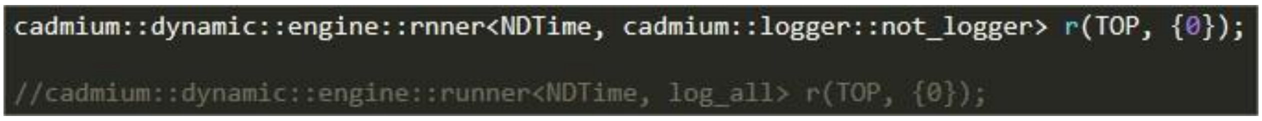
If we are in embedded mode, the motors are enabled (Figure 21).



```
#ifdef ECADMIUM
  //Enable the motors:
  DigitalOut rightMotorEn(D9);
  DigitalOut leftMotorEn(D10);
  rightMotorEn = 1;
  leftMotorEn = 1;
#endif
```

Figure 20: Enabling the motor output

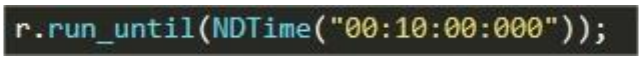
These two lines (Figurer 22) allow us to choose whether to log the information we defined previously or not. Logging, although quite useful, comes with a drawback which is a significant latency in the execution. This latency grows overtime. So it should be turned off when not debugging.



```
cadmium::dynamic::engine::rnner<NDTime, cadmium::logger::not_logger> r(TOP, {0});

//cadmium::dynamic::engine::runner<NDTime, log_all> r(TOP, {0});
```

Figure 21: Logging on or off

Finally, this function (Figure 23) sets a time drift limit that will stop the execution if the latency becomes too high.



```
r.run_until(NDTime("00:10:00:000"));
```

Figure 22: Time drift limit