Rapport de stage 4ième année

Nicolas GAILLARD-GROLÉAS

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

Résumé / Conclusions

Le laboratoire VSim (Visualisation and Simulation) de l'université de Carleton travaille sur un simulateur embarqué appelé ECadmium. Il permet de faire fonctionner tous types de systèmes, et utilise une représentation particulière appelé le formalisme DEVS.

Dans ce contexte, notre objectif est d'utiliser ce simulateur afin de vérifier son bon fonctionnement et déceler d'éventuels bugs. Pour cela, nous avons été chargés de réaliser le code permettant de piloter une petite voiture disposant de différents capteurs, en utilisant le formalisme DEVS.

Après avoir assimilé le fonctionnement d'un modèle DEVS, le travail s'est déroulé en quatre parties. Tout d'abord une partie hardware, puis une modélisation et une partie consacrée au code C++, enfin une phase finale de tests. La diversité des étapes menées a été un des attraits de ce stage.

Le travail effectué a permis de répondre au besoin exprimé. En effet la réalisation de ce robot a contribué à l'amélioration du simulateur ECadmium.

Cependant, bien que mon travail ait permis de corriger certains problèmes dans ECadmium, d'autres problèmes pourraient être révèles dans le développement d'un autre modèle utilisant d'autres capteurs ou ayant un fonctionnement différent. Mon travail s'inscrit donc dans un processus d'amélioration progressive.

REMERCIEMENTS

Je tiens à remercier Monsieur WAINER mon tuteur à l'Université de Carleton ainsi Ben EARLE, Kyle BJORNSON et Christina RUIZ MARTIN pour le temps qu'ils m'ont consacré pendant ce stage.

SOMMAIRE

1. INTRODUCTION 4
2. ÉTAT DE L'ART 4
2.1 Modèle atomique 4
2.2 Modèles couplés 6
3. TRAVAIL RÉALISÉ 8
3.1 Hardware 8
3.2 Modèle théorique 9
3.3 Codage 11
3.4 Tests 14
4. CONCLUSIONS ET PERSPECTIVES 15
Annexe 1 : EXPLICATION DU CODE DEVELOPPÉ 16

RÉFÉRENCES

[1] Modeling with parallel DEVS – Lecture 13. G. WAINER. Carleton University

[2] Review – Lecture 4. G. WAINER. Carleton University

TABLE DES FIGURES

Figure 1 : Modèle atomique

Figure 2 : Exemple de modèle atomique

Figure 3 : Modèle couplé

Figure 4 : Exemple de modèle couplé

Figure 5 : Carte Nucléo

Figure 6 : Module Shield Bot

Figure 7 : Montage électrique

Figure 8 : Modèle théorique

Figure 9 : Définition des entrées et sorties

Figure 10 : Fonction de transition interne

Figure 11 : Fonction de transition externe

Figure 12 : Extrait de la fonction de sortie

Figure 13 : Essais en fonctionnement

1. INTRODUCTION

Le laboratoire VSim (Visualisation and Simulation) de l'université de Carleton travaille sur un simulateur embarqué appelé ECadmium. Il permet de faire fonctionner tous types de systèmes, et utilise une représentation particulière appelé le formalisme DEVS.

Dans ce contexte, notre objectif est d'utiliser ce simulateur afin de vérifier son bon fonctionnement et déceler d'éventuels bugs.

Pour cela, nous avons été chargés de réaliser le code permettant de piloter une petite voiture disposant de différents capteurs, en utilisant le formalisme DEVS.

Hormis la prise en compte des spécifications et l'apprentissage du formalisme DEVS, une des difficultés réside dans la mise en œuvre du langage C++ qui est nouveau pour moi.

Après avoir défini le fonctionnement d'un modèle DEVS, le travail effectué pendant le stage sera présenté. Ce travail s'est déroulé en quatre parties, tout d'abord une partie hardware, puis une modélisation et une partie consacrée au code C++, enfin une phase finale de tests.

2. ÉTAT DE L'ART

La première phase consiste à étudier le formalisme DEVS imposé avant de pouvoir l'utiliser.

Mon tuteur, le professeur G. Wainer dont la compétence est reconnue dans ce domaine m'a fourni quelques documents [1] [2], issus de ses cours, synthétisant la théorie des modèles DEVS. Ainsi il n'était pas nécessaire d'effectuer des recherches dans des articles de revues scientifiques sur le sujet.

Dans cette partie nous expliquerons les bases de ce formalisme.

L'acronyme DEVS vient de l'anglais Discrete Event System Specification. C'est un modèle se décomposant en quelques modules permettant de représenter une grande variété de systèmes à temps continu ou discret de manière structurée et efficace.

2.1 Modèle atomique

Un modèle DEVS est composé de modèles dits « Atomiques ». Un modèle atomique est décrit de la façon suivante (figure 1) :

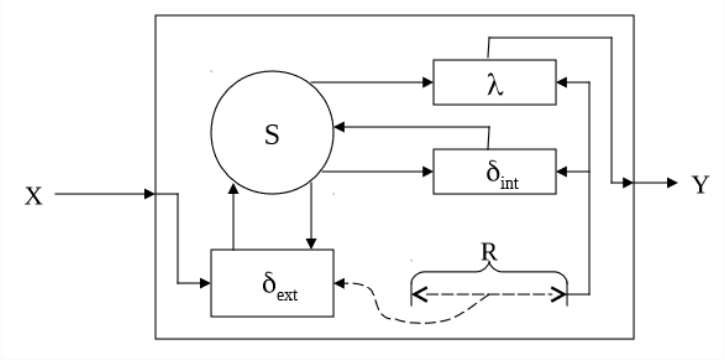

Figure 1: Modèle atomique

Atomic DEVS = < S, X, Y, δ_{int}, δ_{ext}, λ, Ta >

- S : Vecteur d'états

Il contient les différents états que peux prendre le modèle.

- X : Vecteur d'entrée

Il contient les signaux d'entrée du modèle.

- Y : Vecteur de sortie

Il contient les signaux de sortie du modèle.

- δ_{int} : Fonction de transition interne

Elle est activée lorsque la fonction Ta fait avancer le temps. Elle définit l'état suivant du système en fonction de son état courant.

- δ_{ext} : Fonction de transition externe

Elle est activée lorsqu'un signal arrive sur l'entrée du modèle X. Elle définit l'état suivant du système en fonction de l'état courant et du vecteur X.

- λ : Fonction de sortie

Elle définit la valeur du vecteur de sortie Y en fonction de l'état courant du système.

- Ta : Fonction d'avancée temporelle

Elle définit un temps d'attente entre deux transitions internes en fonction de l'état courant du système. C'est le temps pendant lequel le système conserve son état.

Pour bien comprendre, voici l'exemple donné par notre tuteur, le modèle DEVS atomique d'un joueur de ping-pong (figure 2) :

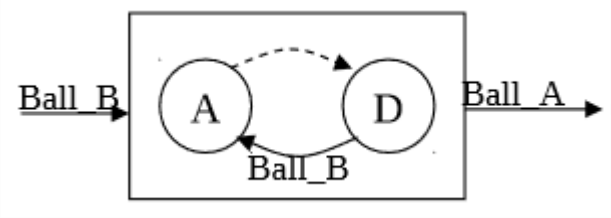

Figure 2: Exemple de modèle atomique

Joueur = < S, X, Y, δ_{int}, δ_{ext}, λ, Ta >

S = { A , D }

X = { Ball_B }

Y = { Ball_A }

δ_{int}(A) = D

δ_{ext}(Ball_B , D) = A

λ(A) = Ball_A

Ta(A) = temps de frappe

Ta(D) = ∞

Pour notre modélisation, le joueur est soit en attaque (état A), soit en défense (état D). Il reçoit une balle sur l'entrée Ball B, et renvoie une balle sur la sortie Ball A.

Supposons le joueur en défense initialement, il passe en attaque s'il reçoit une balle. C'est donc la fonction de transition extérieure qui intervient pour changer l'état, δ_{ext}(Ball_B , D) = A. Il n'y a pas de transition interne dans cet état de défense car le joueur attend théoriquement indéfiniment une balle, donc Ta(D) = ∞.

Si le joueur est en attaque, alors il repasse en défense juste après sa frappe. Il y a donc une transition interne (pas d'entrée sur le vecteur X). Cette transition s'effectue après le temps de frappe, donc Ta(A) = temps de frappe, et δ_{int}(A) = D.

Pour le vecteur de sortie, il ne contient rien lorsque le joueur est en défense, et renvoie une balle lorsqu'il passe en attaque. On a donc seulement λ(A) = Ball_A.

2.2 Modèles couplés

Plusieurs modèles atomiques peuvent être assemblés entre eux afin de créer un système plus complexe, appelé modèle couplé (figure 3). On crée ainsi une encapsulation de modèles permettant de bien structurer le système et de séparer différentes parties, fonctions etc…

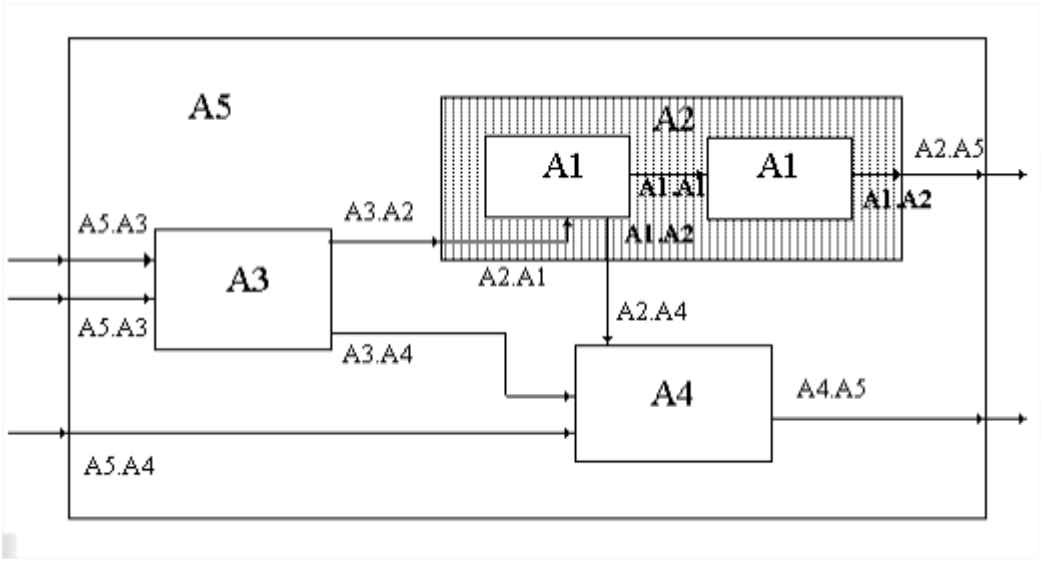

Figure 3: Modèle couplé

Le modèle A5 est un modèle couplé, composé des modèles A2, A3 et A4. Le modèle A2 est lui-même couplé, composé de deux instances du modèle A1.

Un modèle couplé est défini par le tuple CM = < X, Y, D, {M_i}, {I_i}, {Z_{ij}}, Select >

- X : Vecteur d'entrée

Il contient les signaux d'entrée du modèle.

- Y : Vecteur de sortie

Il contient les signaux de sortie du modèle

- D : Liste des noms des sous composants

Elle contient les noms de tous les sous composants utilisés.

- $\{M_i\}$: Liste des composants

$Mi = < S_i, X_i, Y_i, \delta_{int,i}, \delta_{ext,i}, \lambda_i, Ta_i > \forall\ i \in D$

- C_{XX} : Couplage des entrées externes

Relie les entrées du modèle couplé aux entrées des modèles atomiques.

- C_{XY}: Couplage interne

Relie les entrées et les sorties des modèles atomiques à l'intérieur du modèle couplé.

- C_{YY} : Couplage des sorties externes

Relie les sorties du modèle couplé aux sorties des modèles atomiques.

- Select : Fonction d'arbitrage.

Défini la priorité si une transition interne et externe arrivent simultanément.

Si on reprend l'exemple du ping-pong, on peut représenter une partie entre deux joueurs grâce à un modèle couplé (figure 4) :

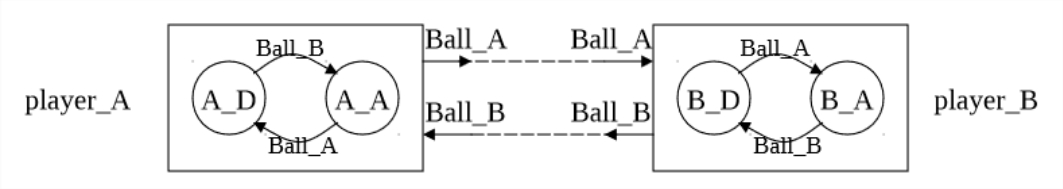

Figure 4: Exemple de modèle couplé

Ce modèle couplé est composé de deux modèles atomiques qui sont des instances du même modèle « Joueur » décrit plus haut. Il ne possède pas d'entrée ni de sortie extérieure, la sortie du premier joueur est liée à l'entrée du second et vice versa.

Ce modèle couplé est donc défini de la manière suivante :

$PingPong = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, Select >$

$X = \{\}$

$Y = \{\}$

$D = \{ Joueur \}$

$\{M_i\} = \{ player\_A, player\_B \}$

$C_{XX} = \{\}$

$C_{XY} = \{ (player\_A.Ball\_A, player\_B.Ball\_B) , (player\_B.Ball\_A, player\_A.Ball\_B) \}$

$C_{YY} = \{\}$

$Select = \emptyset$

3. TRAVAIL RÉALISÉ

Ce travail s'est déroulé en quatre parties, tout d'abord une partie hardware, puis une modélisation et une partie consacrée au code C++, enfin une phase finale de tests.

3.1 Hardware

Nous avons à notre disposition le matériel suivant :

Une carte Nucleo (figure 5), similaire à une carte Arduino, est un mini-ordinateur qui embarque un processeur STM32 ainsi que de nombreuses fonctionnalités et interfaces.

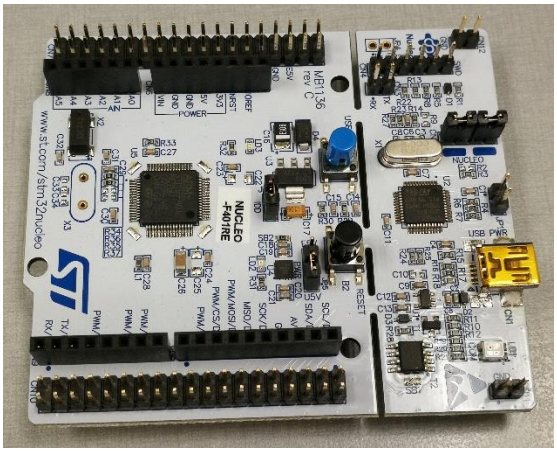

Figure 5: Carte Nucléo

Un module Shield Bot (figure 6). C'est une plateforme robotique conçue pour s'interfacer directement avec les cartes Nucleo et créer un robot suiveur de ligne avec ses capteurs infrarouges préinstallés. On peut y brancher une multitude de capteurs pour étendre ses capacités.

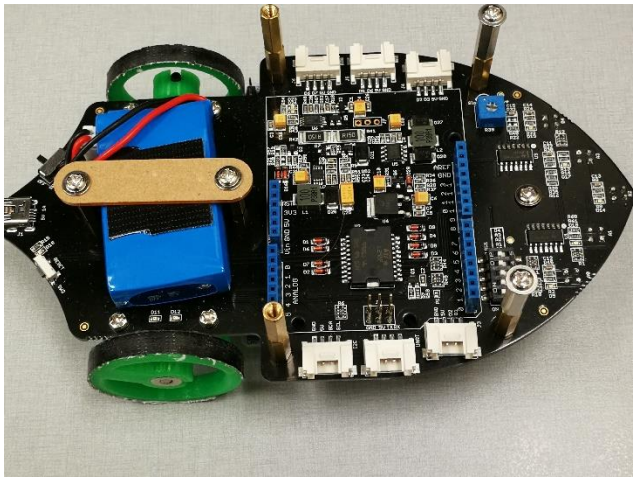

Figure 6: Module Shield Bot

Capteurs disponibles :

- 2x Capteur de température et humidité
- 2x Capteur de lumière
- 2x Capteur de mouvement

- 4x Capteur de température
- 1x Emetteur/Récepteur Ultrasonique
- Autres :
  - 4x LEDs RGB
  - 5x Boutons

Avec ce matériel, nous avons décidé de réaliser un robot suiveur de lumière, équipé d'un dispositif anticollision, et ne démarrant que lorsqu'il est posé sur le sol.

Pour cela nous utiliserons deux capteurs de lumière, un émetteur/récepteur ultrasonique, et un capteur infrarouge.

Voici le montage électrique réalisé (figure 7):

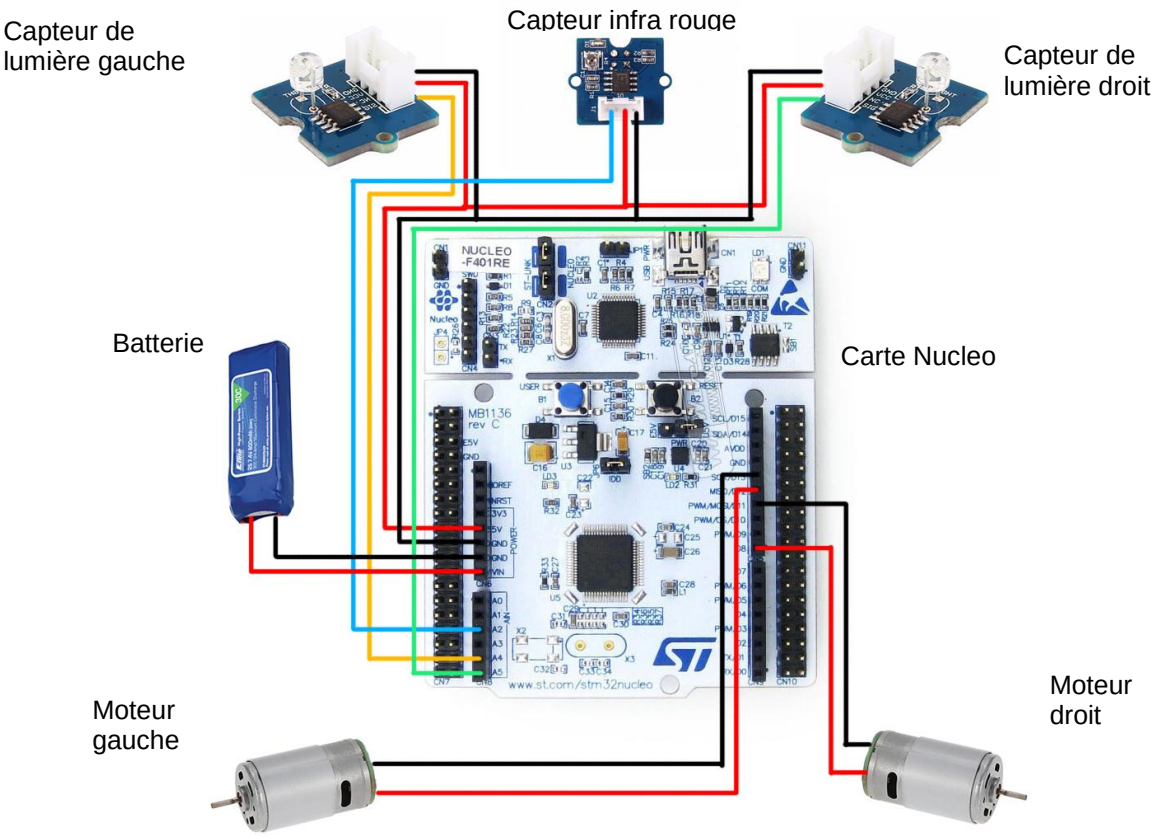

Figure 7: Montage électrique

Je me suis chargé de réaliser la partie correspondant au suivi de lumière et détection du sol, je ne parlerai donc pas de l'anticollision ici. En effet cette problématique a fait l'objet du sujet de stage de mon camarade Quentin De Montblanc.

3.2 Modèle théorique

L'étape suivante de cette réalisation est la traduction en modèle DEVS du fonctionnement souhaité du véhicule.

Il faut pour cela définir les entrées du modèle, les sorties, les états, les fonctions de transition interne et externe ainsi que la fonction de sortie.

Les entrées du modèle sont les différents capteurs installés :

- Capteur de lumière droit : rightLightSens
- Capteur de lumière gauche : leftLightSens
- Capteur infrarouge : centerIR

X = { rightLightSens , leftLightSens , centerIR }

Les sorties sont les deux moteurs couplés aux roues droites et gauches. Les moteurs sont contrôlés par deux valeurs comprises entre 0 et 1. La vitesse de rotation est maximale lorsque la différence est de 1 entre ces deux valeurs, tandis que la vitesse est nulle lorsqu'elles sont égales :

- Moteur droit : rightMotor1, rightMotor2
- Moteur gauche : leftMotor1, leftMotor2

Y = { rightMotor1 , rightMotor2 , leftMotor1 , leftMotor2 }

Ensuite, les différents états nécessaires au déplacement sont :

- Avancer tout droit : straight
- Tourner à droite : right
- Tourner à gauche : left
- S'arrêter : stop

Il y a également un état « prop » qui est associé aux autres états. Il peut être vrai ou faux.

S = { straight , right , left , stop , prop }

La fonction de transition externe permet de changer n'importe quel état en fonction des entrées. Elle prend en paramètre les entrées des capteurs.

δ_{ext}(rightLightSens , leftLightSens , centerIR)

Son fonctionnement sera détaillé par la suite.

La fonction de sortie associe chaque état à des valeurs sur le vecteur de sortie :

λ(straight , right , left , stop)

La fonction de transition interne change simplement la valeur de l'état « prop » à false et permet l'appel de la fonction de sortie.

La fonction d'avancement temporel retourne une valeur de 0 si l'état prop = T. Cela permet d'avoir une transition interne immédiate et donc une sortie immédiate lorsqu'une transition externe intervient. Quand prop = F, on attend indéfiniment une transition externe, donc la fonction d'avancement temporel retourne une valeur infinie.

On obtient alors le modèle suivant (figure 8) :

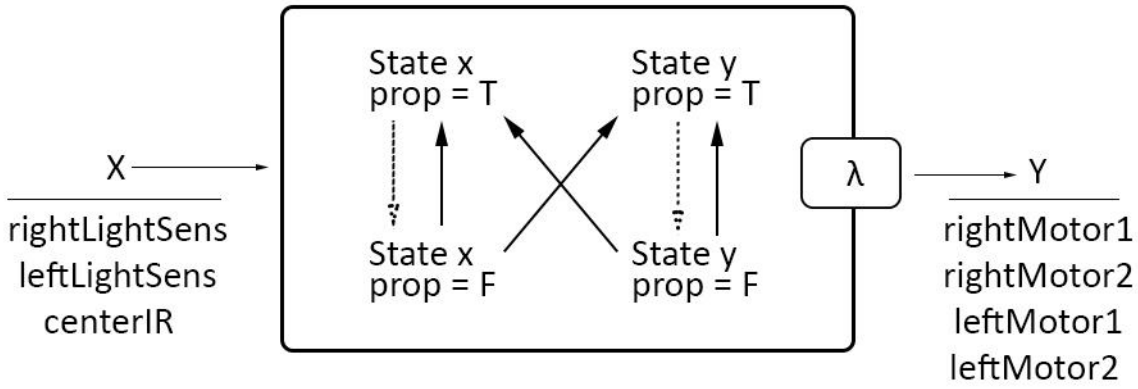

Figure 8: Modèle théorique

LightBot = < S, X, Y, δ_{int}, δ_{ext}, λ, Ta >

{x,y} ∈ S = { straight , right , left , stop }

X = { rightLightSens , leftLightSens , centerIR }

Y = { rightMotor1 , rightMotor2 , leftMotor1 , leftMotor2 }

δ_{int} : prop = F

δ_{ext}(rightLightSens , leftLightSens , centerIR)

λ(straight , right , left , stop)

Ta = 0 si prop = T, infini si prop = F

3.3 Codage

Une fois le modèle théorique établi, il faut le traduire en code compilable et téléchargeable sur la carte Nucléo. Pour cela le code est écrit en C++ dans un éditeur de texte classique, sublime text, et il est compilé et exécuté grâce à l'invite de commande de l'environnement Linux.

Pour m'aider dans la rédaction de ce langage inconnu, mon tuteur m'a fourni un code faisant uniquement clignoter une LED sur la carte. A partir de cela j'ai pu comprendre comment implémenter le modèle.

Je vais maintenant présenter ici les grandes lignes du code. Pour une explication détaillée (en anglais), se référer à l'annexe 1. La structure de ce dernier suit assez bien la manière dont est construit le modèle théorique, ce qui m'a permis de m'y retrouver sans trop de difficultés même si le langage C++ m'était inconnu.

Deux fichiers « source » décrivent le fonctionnement du robot. Le fichier constituant le modèle atomique décrit précédemment, appelé lightBot.cpp, et le fichier constituant un modèle couplé avec les modules d'entrée sortie embarqués et permettant l'exécution du modèle atomique dans l'environnement ECadmium, appelé top_model.cpp.

Dans le modèle atomique lightBot.cpp, nous y retrouvons entre autres la définition des ports d'entrée et sortie correspondant aux capteurs et moteurs (figure 9).

```
//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 9: Définition des entrées et sorties

La fonction de transition interne (figure 10) qui met simplement la variable « prop » à false pour déclencher l'envoi des commandes aux moteurs.

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

Figure 10: Fonction de transition interne

La fonction de transition externe (figure 11) qui détermine le comportement du robot lorsqu'une donnée provenant d'un capteur est détectée.

On y lit d'abord les valeurs des capteurs de lumière ainsi que du capteur infrarouge.

Si le capteur infrarouge ne détecte le sol l'état du robot est mis sur « stop ».

Sinon on compare les valeurs des deux capteurs de lumière. S'il y a une différence de plus de 10% entre les valeurs des deux capteurs de lumière, l'état du robot est changé sur la direction « left » ou « right » correspondant à la valeur la plus élevée.

Si aucune des conditions précédentes n'est remplie, c'est que le robot est sur le sol et la différence de lumière est de moins de 10%, l'état du robot est mis sur « straight » pour aller tout droit.

```
// 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 11: Fonction de transition externe

Ces états sont traduits en sortie moteurs par la fonction de sortie (figure 12) qui grâce à un switch assigne les bonnes valeurs sur les sorties en fonction de l'état courant du robot.

```
// 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;
```

Figure 12: Extrait de la fonction de sortie

Lors du développement de ce code, de nombreux problèmes ont été rencontrés aux cours des compilations intermédiaires. Il s'agissait principalement de problématiques liées au simulateur Ecadmium développé par le laboratoire, ce qui a permis grâce à leur identification de les résoudre au fur et à mesure et ainsi de contribuer à l'amélioration du simulateur.

3.4 Tests

Ce code est ensuite compilé puis chargé sur le robot présenté précédemment.

Conformément aux attentes, le robot actionne les roues uniquement lorsqu'il est posé sur le sol, et tourne à gauche ou à droite afin de suivre la source de lumière présentée devant lui (figure 25).

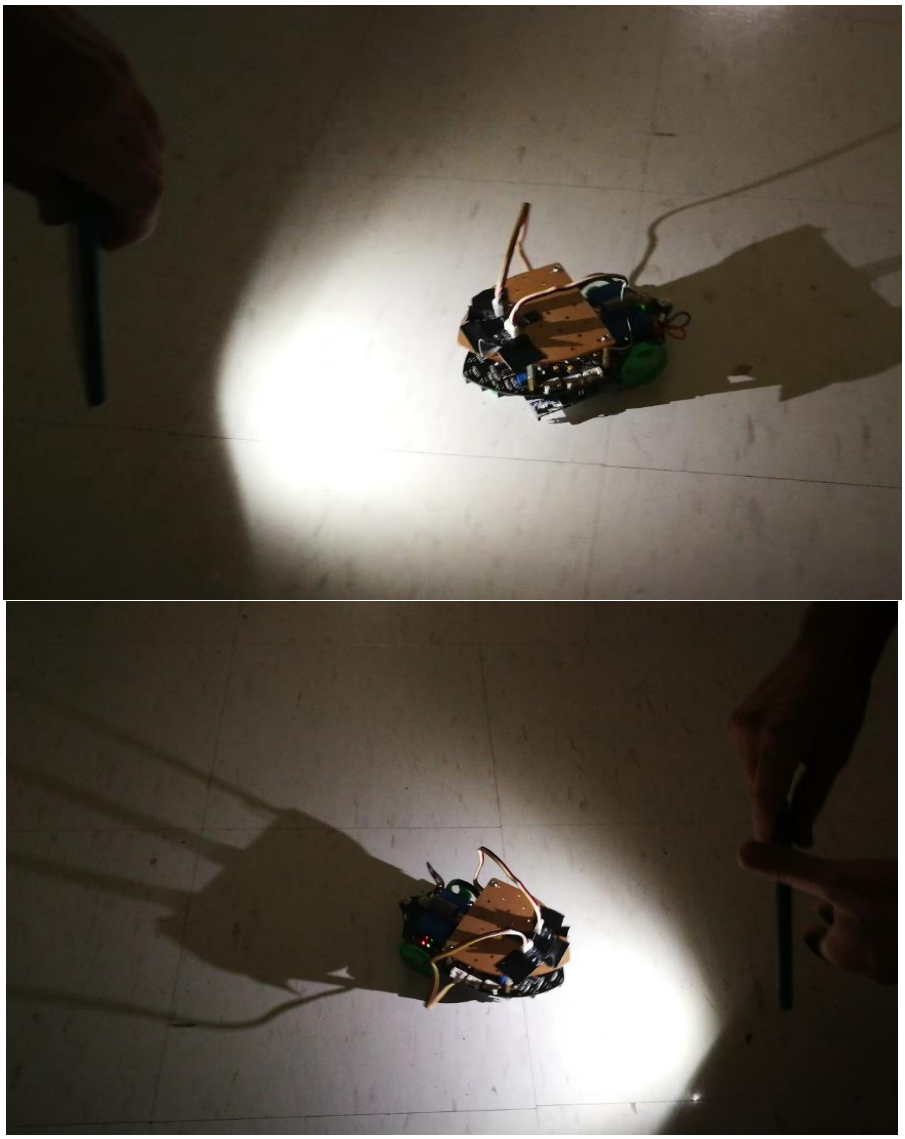

Figure 13 : essais en fonctionnement

4. CONCLUSIONS ET PERSPECTIVES

Le travail effectué a permis de répondre au besoin exprimé. En effet la réalisation de ce robot a contribué à l'amélioration du simulateur développé par le laboratoire VSim suite à l'identification et la résolution de plusieurs bugs liés au simulateur ECadmium.

Un aspect intéressant de ce projet a été la diversité des étapes menées. J'ai pu travailler sur des aspects matériels, modélisation, logiciels, et aussi expérimentaux au travers des tests et débogage effectués avec le robot qui a m'a permis d'avoir une visualisation directe des effets du code.

De plus ce stage m'a permis d'aborder un nouveau langage, le C++. Cet apprentissage a néanmoins été facilité par les compétences acquises à Polytech car ce langage s'apparente à un mélange de C et Java, qui ont été vus en 3A et 4A.

Cependant, bien que mon travail ait permis de corriger certains problèmes dans ECadmium, d'autres problèmes pourraient être révèles dans le développement d'un autre modèle utilisant d'autres capteurs ou ayant un fonctionnement différent. Mon travail s'inscrit donc dans un processus d'amélioration progressive. De plus, en raison de sa mise au point tardive, l'intégration de la partie de détection d'obstacle n'a pas pu être effectuée.

Ce stage m'a permis de découvrir l'environnement et la vie d'un laboratoire de recherche. J'ai ainsi pu échanger en anglais avec de nombreux interlocuteurs experts dans leur domaine afin de progresser dans mon travail et j'ai particulièrement apprécié l'aspect collaboratif des projets menés, ou chacun a l'occasion d'apporter son expertise.

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.

```
// 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.

```
// 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