AltaRica et les Robots
Rapport

Matthieu Devert Julien Dumeste Julien Favarel
François Seimandi

Le 9 février 2009

Résumé

L'entreprise LEGO a développé un robot *MINDSTORMS NXT* qui possède les capacités d'intéragir avec son environnement au travers de capteurs. L'objectif de ce projet est de montrer l'applicabilité du cours de *Conception Formelle* à des systèmes programmables comme les robots. En effet, le langage AltaRica peut être employé afin de définir des tâches « sûres » à un ou des robots. Pour ce faire, nous allons nous attaquer à ce projet au travers de quatre axes majeurs.

Tout d'abord, il faut nous familiariser avec le robot *NXT* : quels sont les différents capteurs, comment réagissent-ils avec leur environnement, comment programmer le robot pour effectuer des tâches particulières ?

Le deuxième axe du projet est une phase de modélisation. Elle consiste à modéliser en AltaRica le robot, au travers de composants (capteurs, moteurs, contrôleur). Il faut également modéliser le protocole de communication *Bluetooth* afin de pouvoir représenter les possibilités de communication entre robots.

La troisième partie du projet consistera à définir une mission non triviale mettant en jeu plusieurs robots et de synthétiser automatiquement le contrôleur AltaRica associé. A partir de ce contrôleur, on pourra alors implémenter la mission désirée directement dans le langage de programmation du robot.

Le quatrième et dernier axe du projet est d'implémenter un traducteur entre AltaRica et le langage de programmation du LEGO *NXT* qui permettrait de passer automatiquement d'un noeud AltaRica « basique » (comme un contrôleur synthetisé automatiquement) vers un programme *NXT*.

Table des matières

Première partie

Documentation

Chapitre 1

Introduction au domaine d'application

1.1 Le LEGO Robot Mindstorms NXT

Fig. 1.1 – Le robot Mindstorms Education NXT

Ce LEGO est composé de :
- une brique « intelligente » NXT,
- 5 capteurs de différentes natures,
- 3 servo-moteurs et 3 lampes.

1.1.1 La brique « intelligente » NXT

Cette brique est la pièce maîtresse du robot. C'est le cerveau : elle possède deux processeurs : un ARM7 de 32 bits et un second microprocesseur de 8 bits. La brique possède aussi 2 mémoires RAM de 64 Kbits (pour le premier processeur) et de 512 bits (pour le second).

Elle a 4 ports d'entrée pour des capteurs, et 3 ports de sortie pour les moteurs ou la lumière. Elle possède de plus la capacité de communiquer via la technologie Bluetooth.

Fig. 1.2 – La brique intelligente. Crédit : [Gro06].

Cette brique permet de programmer différents comportements. On peut charger des programmes au travers d'un port USB ou du Bluetooth. On peut, de plus, programmer directement depuis l'interface de la brique NXT.

Cette brique a aussi la capacité d'emmetre des sons.

1.1.2 Les servo-moteurs

Fig. 1.3 – Un servo-moteur. Crédit : [Gro06].

Le LEGO comporte 3 servo-moteurs.

Ils permettent de faire bouger le robot de manière fluide grâce à un alignement automatique des vitesses des deux servo-moteurs branchés sur les ports B et C de la brique NXT.

Les moteurs ont des senseurs de rotations qui permettent de contrôler très précisément les mouvements. Ces senseurs mesurent la rotation en degré, et on peut donc par exemple demander une rotation de 180 degrés.

1.1.3 Les capteurs

On dispose de 4 types de capteurs :

Capteur de contact

Dans le LEGO, 2 capteurs de contact sont fournis.

Fig. 1.4 – Capteur de contact. Crédit : [Gro06].

Ce capteur possède un appendice qu'on peut enfoncer (comme un bouton poussoir). Ce capteur peut être dans 2 états différents : un état poussé, un état relaché.

Capteur de son

Fig. 1.5 – Capteur de son. Crédit : [Gro06].

Ce capteur détecte le niveau de décibel environnant. Il peut détecter aussi bien les dB que les dbA : aussi bien les sons audibles par les humains que ceux qui possèdent une fréquence trop faible ou trop élevée pour être audible. Il peut mesurer le son jusqu'à 90 décibels, et donne sa mesure en pourcentage du niveau sonore qu'il est capable de détecter. Par exemple : entre 10 et 30 % correspond au niveau détecté pour une conversation normale à coté du capteur.

Capteur de lumière

Fig. 1.6 – Capteur de lumière. Crédit : [Gro06].

Ce capteur permet de distinguer la nuance de clarté dans une pièce (en niveau de gris). Il permet de mesurer l'intensité de la lumière dans une pièce ou de mesurer l'intensité sur des surfaces colorées.

Capteur ultrasonique

FIG. 1.7 – Capteur ultrasonic. Crédit : [Gro06].

Ce capteur permet au robot de voir et de détecter des objets, d'éviter des obstacles, de mesurer des distances ou de détecter des mouvements.

Il utilise le principe du sonar : il mesure le temps que met une onde qu'il a envoyé pour lui revenir.

Sa portée va de 0 à 2,5 mètres, ceci avec une précision de plus ou moins 3 cm.

Il peut être important de noter que 2 capteurs à ultrasons peuvent interférer entre eux lorsqu'ils sont dans une même pièce, ce qui peut limiter les usages possibles.

1.2 Le langage de programmation

Le langage utilisé pour programmer le robot est le langage NXT-G.

Ce langage est purement graphique. On utilise des « blocs » que l'on place bout à bout. Ces blocs correspondent à diverses actions telles que les manoeuvres de base comme `avancer`, `reculer` ou plus poussées comme `jouer un son`, `afficher une image sur un écran`, etc. Ainsi, on peut commander chacune des parties du robot NXT.

Le NXT-G s'assimile donc à un langage impératif et dispose en conséquence de blocs concernants les boucles (loop blocks), les attentes (wait blocks) ou encore les aiguillages (switch blocks).

Le langage étant graphique, il ne permet pas de faire de programmes trop avancés de manière aisée, mais il simplifie la phase de programmation pour des utilisateurs lambdas.

Il semble qu'il n'existe pas d'interface entre le logiciel et un langage texte disponible directement. On ne peut alors pas transférer une suite d'instructions textuelles au robot. Il faut donc trouver une API qui permetterait d'effectuer une telle opération.

Il existe un tutorial intéressant sur le NXT-G [Yoc06] qui permet une prise en main rapide du logiciel et du matériel. Il met notamment à disposition des vidéos explicatives relativement bien réalisées.

FIG. 1.8 – L'interface de programmation du NXT-G. Crédit : [Mag08].

Chapitre 2

Analyse de l'existant

2.1 Les langages de programmation pour NXT Mindstorms

Voici une liste non exhaustive des différents langages qui permettent la programmation de robots NXT Mindstorms, ainsi qu'une brève présentation.

2.1.1 LeJOS

LeJOS est une API Java Open Source permettant de programmer pour le NXT Mindstorms. Étant basé sur Java, il nécessite l'installation de la machine virtuelle sur la brique en remplacement du firmware d'origine. Cet aspect rend son utilisation non intéressante dans notre projet [Esm08].

2.1.2 NXC, Not eXactly C

C'est un langage inspiré du langage C. On peut l'utiliser à travers une interface graphique (Bricx Command Center) [Han07]. Il ne nécessite aucun firmware et son utilisation est simplifiée de par sa ressemblance au C.

2.1.3 RobotC

C'est un autre langage inspiré du langage C. Il a cependant besoin d'un firmware et est payant [Mel07].

2.1.4 Micrososft Visual Programming Language

C'est l'environnement graphique de développement de Windows. Il permet, à la manière de NXT-G, de programmer sous forme de schémas et d'icônes [Cor08]. Ce langage n'est donc pas utile pour nous qui cherchons un langage textuel.

2.2 La communication par Bluetooth

Des communications avec les robots NXT peuvent être établies à l'aide de la technologie Bluetooth. On distingue les cas suivants :

- Des programmes peuvent être envoyés depuis un ordinateur sans utiliser de cable USB.
- Des programmes peuvent être envoyés depuis d'autres appareils qu'un ordinateur, tels qu'un robot NXT, un téléphone portable, etc.
- Des programmes peuvent être reçus par divers robots NXT, soit individuellement, soit collectivement. Dans ce dernier cas, jusqu'à trois robots NXT peuvent recevoir les mêmes informations.
- Enfin, des informations (par exemple une valeur numérique ou du texte) peuvent être envoyés d'un robot à 1 ou plusieurs récepteurs bluetooth.

Pour envoyer des informations via Bluetooth, une communication doit d'abord être établie entre les différents « interlocuteurs » . Ensuite seulement, des messages peuvent être envoyés. L'appareil qui démarre la communication est l'appareil « maître » , et pourra alors envoyer des données jusqu'à trois appareils « esclaves » , visibles par le maître sur les canaux 1, 2, et 3. Ces esclaves, eux, verront toujours le maître sur le canal 0. Il s'agit donc d'une communication Bluetooth unidirectionnelle allant d'un appareil maître vers un à trois appareils esclaves.

2.3 Différentes missions déjà éxécutées

Parmi les différentes missions possibles nous en avons recensé quelques-unes :

- Le robot, grâce au capteur de lumière, suit une ligne tracée au sol.
- Le robot, grâce au capteur à ultrasons ou à lumière, reste sur une table sans y tomber.
- Deux robots s'affrontent et tentent de se pousser hors d'une zone délimitée.
- Deux robots, un maître et un esclave, se fraient un chemin dans un labyrinthe, le maître guidant l'esclave.
- Un robot trie des billes par couleur.
- Un robot résoud un rubik's cube.

De nombreuses vidéos de ces missions sont évidemment visibles sur les sites de streaming vidéo.

Chapitre 3

Introduction au projet

3.1 But

Dans un premier temps, le but sera de modéliser le robot MINDSTORMS NXT en AltaRica.

Par la suite, nous devrons dans la mesure du possible, synthétiser automatiquement (grâce à `acheck`) le contrôleur qui permettra l'exécution d'une mission non triviale. De l'analyse de l'existant, il en ressort que les tâches envisageables sont diverses, notre choix se porte sur une mission qui verra le maître « lire » une figure ou un dessin et demander à l'esclave de reproduire son dessin (en reproduisant le parcours du maître). On pourra contrôler le mimétisme grâce au dessin que l'esclave tracera durant son déplacement.

Pour finir, il serait intéressant de pouvoir traduire automatiquement le noeud du contrôleur synthétisé du langage AltaRica vers le langage NXC. L'analyse de l'existant a révélé que c'était le langage textuel le plus accessible et celui qui ne nécessitait aucune installation de firmware sur la brique.

3.2 Justifications et priorités

Le projet vise donc à appliquer concrètement les acquis du cours de *Conception Formelle*, à savoir utiliser AltaRica et la vérification des modèles pour créer des missions sûres pour des robots. Nous effecturons les tâches dans l'ordre suivant :

- Modélisation des composants du robot en AltaRica.
- Modélisation du protocole de communication Bluetooth pour les communications inter-robots.
- Programmation de la mission souhaitée avec le NXT-G et vérification de sa faisabilité et de son bon fonctionnement.
- Si possible générer automatiquement un contrôleur AltaRica pour la mission.
- Enfin, traduire automatiquement le contrôleur engendré vers le langage NXC.

Deuxième partie

Cahier des charges

Chapitre 4

Besoins non fonctionnels

4.1 Modularité

On désire que la modélisation des composants nous permette d'assembler un robot donné pour une mission particulière, comme si on construisait un véritable Lego.

Ainsi, on doit pouvoir brancher n'importe quel capteur sans pour autant devoir changer la modélisation en profondeur. Ceci permettrait de conserver le modèle même si la mission souhaitée change du tout au tout.

4.2 Expressivité et vérification

Le principal souci au niveau non fonctionnel concerne l'abstraction. Il faut trouver un bon compromis entre l'expressivité du modèle et la possibilité de vérification des propriétés sur ce modèle.

En effet, plus un modèle est expressif, plus il est difficile de vérifier des propriétés dessus, ainsi que de générer un contrôleur. A contrario, moins le modèle est expressif, plus il est difficile de contrôler le robot pour une mission particulière et précise.

Chapitre 5

Besoins fonctionnels

5.1 Modélisation des composants en AltaRica

Les composants à modèliser sont les suivants :
- Périphériques d'entrée :
  - capteur de pression,
  - capteur à ultrasons,
  - capteur de lumière,
  - capteur de son.
- Périphériques de sortie :
  - moteur.
- Robot maître avec émission Bluetooth et contrôleur.
- Robot esclave avec réception Bluetooth.

5.2 Spécifications de la mission

Etant donnée une mission non triviale, le système modélisé doit respecter un certain nombre de spécifications :

1. La mission doit intégrer une relation maître-esclave entre 2 robots.
2. Le maître communique unilatéralement avec son esclave par Bluetooth.

5.3 OPTIONNEL : Génération automatique d'un contrôleur

On désire, si possible, générer automatiquement un contrôleur permettant au robot de mener à bien la mission souhaitée. On souhaite que ce contrôleur respecte les spécifications de la mission. Cette génération sera effectuée grâce à `acheck`.

5.4 OPTIONNEL : Traduction d'un contrôleur AltaRica en programme pour NXT

On désire traduire le contrôleur, généré automatiquement, en programme pour NXT de manière automatique. Un robot doté des mêmes composants que ceux du modèle, et dans lequel on charge ce programme, devra être capable de mener à bien la mission modèlisée par le contrôleur.

Troisième partie

Implémentation

Chapitre 6

Mission

6.1 Description

La mission proposée est de faire en sorte que le robot maître suive un parcours et qu'il communique au robot esclave des instructions pour qu'il effectue les mêmes déplacements (et puisse ainsi tracer le parcours vu par le robot maître).

Le robot maître est équipé d'un capteur de lumière dirigé vers le sol, et d'un capteur sonore. Il est également équipé de 2 moteurs formant un bloc de direction et servant à déplacer le robot.

Il est placé sur une "route" blanche bordée par une bande grise sur sa droite et une bande noire sur sa gauche. Lorsqu'il voit du blanc, le robot maître avance tout droit, lorqu'il voit du gris, il tourne à gauche, et lorqu'il voit du noir, il tourne vers la droite. A chaque changement de direction, le robot maître communique au robot esclave de nouveaux ordres.

La mission se termine quand le capteur sonore se déclenche.

Le robot esclave est équipé de 2 moteurs regroupés en un bloc de direction, et d'un support de feutre pour tracer le chemin parcouru.

6.2 Spécifications

La mission doit respecter les spécifications suivantes :

1. Le robot maître et le robot esclave doivent s'arrêter quand le capteur sonore se déclenche.
2. Le robot maître communique à son esclave avant d'effectuer une action.
3. Le robot esclave suit uniquement les ordres du robot maître, il ne peut prendre aucune décision par lui-même.

Fig. 6.1 – Programme NXT-G du robot maître

FIG. 6.2 – Programme NXT-G du robot esclave

Chapitre 7

Architecture

7.1 Prémices

Avant de choisir l'architecture finale, nous avons réfléchi à plusieurs solutions.

7.1.1 Version 1

Principe

Dans cette première version, chaque capteur est décrit spécifiquement. Différents niveaux d'abstraction sont ainsi utilisés pour chacun des capteurs. Chaque capteur dispose d'un ensemble d'états et d'un ensemble identique de seuils.

On dipose d'un noeud Brick qui fonctionne de la manière suivante : on regarde un capteur, on attend le signal, on modifie les variables du premier moteur, du second, puis du troisième, on modifie le seuil du capteur, on éxécute les commandes moteur et enfin, on passe au capteur suivant.

Problèmes

Cette solution permet d'associer un ordre très précis des moteurs à chaque information d'un capteur. Malheureusement, cette expressivité a un coût : l'explosion combinatoire.

Le fait de « regarder » chaque capteur un par un augmente inutilement le nombre d'état. En effet, c'est une description qu'on peut abstraire puisqu'en vrai le temps que met le système à regarder un capteur puis un autre est vraiment trop négligeable pour qu'il soit pris en compte dans un modèle.

7.1.2 Version 2

Principe

Dans cette version on utilise des variables de flux pour observer les capteurs. De plus, on réduit les capteurs à de simples automates à deux états : déclenché ou non déclenché. Ceci nous fait perdre de l'expressivité mais permet de gagner un nombre important d'états.

Problèmes

Grâce aux variables de flux le nombre d'états se réduit, mais les combinaisons de toutes les modifications possibles des variables des moteurs posent encore problème. De plus, la réduction des capteurs à de simples booléens ne nous permet pas de décrire la mission souhaitée de manière satisfaisante.

7.2 Architecture finale

Le problème majeur de nos solutions précédentes se situait sur la « quantité » d'abstraction (le rapport expressivité/nombre d'états du système).

Pour réduire les états du systèmes, nous avons décidé de rajouter un noeud *Direction*. Cela nous impose un comportement précis pour le robot (à savoir un véhicule), mais réduit considérablement le nombre d'états et de transitions.

Pour pallier au manque d'expressivité du modèle, nous sommes revenu à une solution où les capteurs ont plusieurs états. Cependant, nous n'avons pas mis de notion de seuil pour le déclenchement du signal. Nous avons laissé le travail de juger du déclenchement ou non au contrôleur, nous rapprochant ainsi du comportement réel des capteurs.

Voici l'architecture :

FIG. 7.1 – Architecture du système pour la mission considérée.

Les communications Bluetooth ont été modélisées au travers de synchronisations de transitions entre le robot maître et le robot esclave. Ainsi, nous

avons choisi de ne pas créer des noeuds *BluetoothMaster* et *BluetoothSlave* car dans notre cas, nous n'avons qu'un seul esclave. Il nous semble qu'écrire un noeud modélisant le Bluetooth est intéressant dans le cas où il serait nécessaire d'adresser des ordres à un robot esclave particulier, c'est-à-dire dans le cas où nous aurions plusieurs esclaves pouvant effectuer des tâches différentes des autres esclaves en même temps, ce qui n'est pas notre cas ici.

Nous avons choisis de ne pas mettre de noeuds *Environnement* car dans notre cas, il est toujours possible pour chacun des capteurs qu'un changement soit détecté. C'est le rôle du contrôleur de décider si le déclenchement de tel ou tel signal entraîne tel ou tel comportement. De plus, pour notre mission, nous n'avons pas de parcours « figé », c'est à dire que nous ne pouvons pas connaître l'enchaînement des couleurs détectées par le capteur de lumière. Cet enchaînement dépendra juste du comportement du robot et sera soumis en grande partie au hasard.

Chapitre 8

Les composants

8.1 Le noeud « UltraSonicSensor »

8.1.1 Description

Il correspond au capteur à ultrasons. Il dispose d'une variable à 3 états `signal` qui correspond au seuil du capteur. Dans les événements, on représente le fait qu'on ne peut passer qu'à une valeur directement supérieure ou directement inférieure.

8.1.2 Le source AltaRica

```
node UltraSonicSensor
  state
    signal : [0,2] : public;
  event
    trigger;
  trans
    signal < 2 |- trigger -> signal := signal + 1;
    signal > 0 |- trigger -> signal := signal - 1;
  init
    signal := 0;
edon
```

8.1.3 La sémantique

FIG. 8.1 – Le noeud « UltraSonicSensor »

8.1.4 Les propriétés

Les spécifications

```
with UltraSonicSensor do
 quot()                >  '$NODENAME.dot';
 dead                  := any_s - src(any_t - self_epsilon);
 notCFC                := any_t - loop(any_t,any_t);
 signal0               := [signal = 0];
 signal1               := [signal = 1];
 signal2               := [signal = 2];
 trigger               := label trigger;
 show(all)             >  '$NODENAME.prop';
 test(dead,0)          >  '$NODENAME.res';
 test(notCFC,0)        >> '$NODENAME.res';
done
```

Les résultats

```
/*
 * # state properties : 6
 *
 * signal0 = 1
 * signal1 = 1
 * signal2 = 1
 * initial = 1
 * any_s = 3
 * dead = 0
 *
 * # transition properties : 7
 *
 * trigger = 4
 * self_epsilon = 3
 * notCFC = 0
 * epsilon = 3
 * self = 3
 * not_deterministic = 2
 * any_t = 7
 */
```

Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

8.2 Le noeud « TouchSensor »

8.2.1 Description

Ce noeud représente le capteur de pression. C'est un noeud classique à deux états.

8.2.2 Le source AltaRica

```
node TouchSensor
  state
    signal : bool : public;
  event
    trigger;
  trans
    true |- trigger -> signal := ~signal;
  init
    signal := false;
edon
```

8.2.3 La sémantique

FIG. 8.2 – Le noeud « TouchSensor »

8.2.4 Les propriétés

Les spécifications

```
with TouchSensor do
 quot()                >  '$NODENAME.dot';
 dead                  := any_s - src(any_t - self_epsilon);
 notCFC                := any_t - loop(any_t,any_t);
 signalOn              := [signal = true];
 signalOff             := [signal = false];
 trigger               := label trigger;
 show(all)             >  '$NODENAME.prop';
 test(dead,0)          >  '$NODENAME.res';
 test(notCFC,0)        >> '$NODENAME.res';
done
```

Les résultats

```
/*
 * # state properties : 5
 *
 * initial = 1
 * signalOn = 1
 * signalOff = 1
 * dead = 0
 * any_s = 2
```

```
 *echiquier nicois
 * # transition properties : 7
 *
 * trigger = 2
 * self_epsilon = 2
 * notCFC = 0
 * epsilon = 2
 * self = 2
 * not_deterministic = 0
 * any_t = 4
 */
```

Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

8.3 Le noeud « SoundLightSensor »

8.3.1 Description

Ce noeud représente deux capteurs, celui du son et celui de lumière. Ces deux capteurs ont en commun de pouvoir recevoir directement une certaine valeur de seuil (contrairement au capteur à ultrasons).

8.3.2 Le source AltaRica

```
node SoundLightSensor
  state
    signal : [0,2] : public;
  event
    trigger;
  trans
    true |- trigger -> signal := 0;
    true |- trigger -> signal := 1;
    true |- trigger -> signal := 2;
  init
    signal := 0;
edon
```

8.3.3 La sémantique

FIG. 8.3 – Le noeud « SoundLightSensor »

8.3.4 Les propriétés

Les spécifications

```
with SoundLightSensor do
 quot()                >  '$NODENAME.dot';
 dead                  := any_s - src(any_t - self_epsilon);
 notCFC                := any_t - loop(any_t,any_t);
 signal0               := [signal = 0];
 signal1               := [signal = 1];
 signal2               := [signal = 2];
 trigger               := label trigger;
 show(all)             >  '$NODENAME.prop';
 test(dead,0)          >  '$NODENAME.res';
 test(notCFC,0)        >> '$NODENAME.res';
done
```

Les résultats

```
/*
 * # state properties : 6
 *
 * initial = 1
 * signal0 = 1
 * signal1 = 1
 * signal2 = 1
 * any_s = 3
 * dead = 0
 *
 * # transition properties : 7
 *
 * trigger = 9
 * self_epsilon = 3
 * notCFC = 0
 * not_deterministic = 9
 * self = 6
 * epsilon = 3
 * any_t = 12
 */
```

Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

8.4 Le noeud « DummySensor »

8.4.1 Description

Ce noeud permet de simuler une absence de capteur. Sa valeur de signal est toujours à false.

8.4.2 Le source AltaRica

```
node DummySensor
  state
    signal : bool : public;
  event
    trigger;
  trans
    false |- trigger -> signal := ~signal;
  init
    signal := false;
edon
```

8.4.3 La sémantique

FIG. 8.4 – Le noeud « DummySensor »

8.4.4 Les propriétés

Les spécifications

```
with DummySensor do
 quot()                >  '$NODENAME.dot';
 dead                  := any_s - src(any_t - self_epsilon);
 notCFC                := any_t - loop(any_t,any_t);
 signalOn              := [signal = true];
 signalOff             := [signal = false];
 trigger               := label trigger;
 show(all)             >  '$NODENAME.prop';
 test(dead,0)          >  '$NODENAME.res';
 test(notCFC,0)        >> '$NODENAME.res';
```

```
done
```

Les résultats

```
/*
 * # state properties : 5
 *
 * signalOn = 0
 * signalOff = 1
 * dead = 1
 * initial = 1
 * any_s = 1
 *
 * # transition properties : 7
 *
 * trigger = 0
 * self_epsilon = 1
 * notCFC = 0
 * self = 1
 * epsilon = 1
 * not_deterministic = 0
 * any_t = 1
 */
```

Interprétations

```
TEST(dead=0) [FAILED] actual size = 1
TEST(notCFC=0) [PASSED]
```

8.5 Le noeud « ServoMotor »

8.5.1 Description

Ce noeud représente les moteurs. Il dispose d'une variable à 3 états `speed` qui symbolise le comportement des roues du moteur, ainsi que les 3 évènements associés pour pouvoir modifier la variable.

8.5.2 Le source AltaRica

```
/* Speed -1 -> marche arriere. */
/* Speed 0  -> arret.          */
/* Speed 1  -> marche avant.   */

node ServoMotor
  state
    speed : [-1,1] : public;
  event
    setVarR, setVar0, setVar1;
  trans
    true |- setVarR -> speed := -1;
```

```
    true |- setVar0 -> speed := 0;
    true |- setVar1 -> speed := 1;
edon
```

8.5.3 La sémantique

FIG. 8.5 – Le noeud « ServoMotor »

8.5.4 Les propriétés

Les spécifications

```
with ServoMotor do
 quot()                 >  '$NODENAME.dot';
 dead                   := any_s - src(any_t - self_epsilon);
 notCFC                 := any_t - loop(any_t,any_t);
 speedR                 := [speed = -1];
 speed0                 := [speed = 0];
 speed1                 := [speed = 1];
 setVarR                := label setVarR;
 setVar0                := label setVar0;
 setVar1                := label setVar1;
 show(all)              >  '$NODENAME.prop';
 test(dead,0)           >  '$NODENAME.res';
 test(notCFC,0)         >> '$NODENAME.res';
done
```

Les résultats

```
/*
 * # state properties : 6
 *
 * speedR = 1
 * speed0 = 1
 * speed1 = 1
 * dead = 0
 * initial = 1
 * any_s = 3
 *
 * # transition properties : 9
 *
 * setVarR = 3
 * setVar0 = 3
```

```
 * setVar1 = 3
 * self_epsilon = 3
 * notCFC = 0
 * epsilon = 3
 * not_deterministic = 0
 * self = 6
 * any_t = 12
 */
```

Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

8.6 Le noeud « Direction »

8.6.1 Description

Ce noeud simule la direction de deux roues. Il s'occupe donc principalement de synchroniser les différentes manoeuvres avec les moteurs concernés.

8.6.2 Le source AltaRica

```
/* Direction -1 -> recule. */
/* Direction 0  -> arret.  */
/* Direction 1  -> gauche. */
/* Direction 2  -> droite. */
/* Direction 3  -> avance. */

node Direction
  sub
    A : ServoMotor;
    B : ServoMotor;
  state
    direction : [-1,3] : public;
  event
    backward, forward, stop, left, right;
  trans
    true |- backward -> direction := -1;
    true |- stop -> direction := 0;
    true |- left -> direction := 1;
    true |- right -> direction := 2;
    true |- forward -> direction := 3;
  sync
    <backward, A.setVarR, B.setVarR>;
    <stop, A.setVar0, B.setVar0>;
    <left, A.setVar0, B.setVar1>;
    <right, A.setVar1, B.setVar0>;
    <forward, A.setVar1, B.setVar1>;
  init
```

```
    A.speed := 0, B.speed := 0;
edon
```

8.6.3 La sémantique

La sémantique de ce noeud étant trop riche pour être proposée ici, on se référera au fichier `Direction.eps` joint au rapport.

8.6.4 Les propriétés

Les spécifications

```
with Direction do
 quot()                 >  '$NODENAME.dot';
 dead                   := any_s - src(any_t - self_epsilon);
 notCFC                 := any_t - loop(any_t,any_t);
 directionR             := [direction = -1];
 direction0             := [direction = 0];
 direction1             := [direction = 1];
 direction2             := [direction = 2];
 direction3             := [direction = 3];
 backward               := label backward;
 forward                := label forward;
 stop                   := label stop;
 left                   := label left;
 right                  := label right;
 show(all)              >  '$NODENAME.prop';
 test(dead,0)           >  '$NODENAME.res';
 test(notCFC,0)         >> '$NODENAME.res';
done
```

Les résultats

```
/*
 * # state properties : 8
 *
 * initial = 1
 * directionR = 1
 * direction0 = 1
 * direction1 = 1
 * direction2 = 1
 * direction3 = 1
 * dead = 0
 * any_s = 5
 *
 * # transition properties : 11
 *
 * backward = 5
 * self_epsilon = 5
 * forward = 5
 * stop = 5
```

```
 * left = 5
 * right = 5
 * epsilon = 5
 * notCFC = 0
 * not_deterministic = 0
 * self = 10
 * any_t = 30
 */
```

Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

8.7 Le noeud « MissionController »

8.7.1 Description

Il s'agit du contrôleur spécifique à la mission pour le robot maître. Comme tout contrôleur, il dispose des transitions de l'entité qu'il contrôle, à savoir le robot maître. Il dispose aussi des informations des capteurs du robot maître, mais aussi la vitesse et la direction des moteurs.

8.7.2 Le source AltaRica

```
node MissionController

  state
    mode : [0,1];
    order :[0,4];

  init
    mode := 0,
    order := 0;

  flow
    s1state, s2state : [0,2];
    s3state, s4state : bool;
    direction : [-1, 3];
    cSpeed : [-1, 1];

  event
    noop,
    BackwardSetVarCR, BackwardSetVarC0, BackwardSetVarC1,
    StopSetVarCR, StopSetVarC0, StopSetVarC1,
    ForwardSetVarCR, ForwardSetVarC0, ForwardSetVarC1,
    LeftSetVarCR, LeftSetVarC0, LeftSetVarC1,
    RightSetVarCR, RightSetVarC0, RightSetVarC1,
    sendNoop,
    sendBackwardSetVarCR, sendBackwardSetVarC0, sendBackwardSetVarC1,
```

```
    sendStopSetVarCR, sendStopSetVarC0, sendStopSetVarC1,
    sendForwardSetVarCR, sendForwardSetVarC0, sendForwardSetVarC1,
    sendLeftSetVarCR, sendLeftSetVarC0, sendLeftSetVarC1,
    sendRightSetVarCR, sendRightSetVarC0, sendRightSetVarC1;

  trans
    mode = 0 & s2state = 2
                  |- sendStopSetVarC0 -> mode := 1, order := 0;
    mode = 0 & s2state < 2 & s1state = 0 & direction < 3
                  |- sendForwardSetVarC0 -> mode := 1, order := 1;
    mode = 0 & s2state < 2 & s1state = 1
                  |- sendLeftSetVarC0 -> mode := 1, order := 2;
    mode = 0 & s2state < 2 & s1state = 2
                  |- sendRightSetVarC0 -> mode := 1, order := 3;
    mode = 0 & s2state < 2 & direction = 3 & s1state = 0
                  |- sendNoop -> mode := 1, order := 4;

    false |- BackwardSetVarCR, BackwardSetVarC0, BackwardSetVarC1,
             StopSetVarCR, StopSetVarC1,ForwardSetVarCR,
             ForwardSetVarC1, LeftSetVarCR, LeftSetVarC1,
             RightSetVarCR, RightSetVarC1 -> ;

    mode = 1 & order = 0 |- StopSetVarC0    -> mode := 0;
    mode = 1 & order = 1 |- ForwardSetVarC0 -> mode := 0;
    mode = 1 & order = 2 |- LeftSetVarC0    -> mode := 0;
    mode = 1 & order = 3 |- RightSetVarC0   -> mode := 0;
    mode = 1 & order = 4 |- noop            -> mode := 0;

    false |- sendBackwardSetVarCR, sendBackwardSetVarC0,
             sendBackwardSetVarC1, sendStopSetVarCR,
             sendStopSetVarC1, sendForwardSetVarCR,
             sendForwardSetVarC1, sendLeftSetVarCR,
             sendLeftSetVarC1, sendRightSetVarCR, sendRightSetVarC1 -> ;
edon
```

8.7.3 La sémantique

La sémantique de ce noeud étant trop riche pour être proposée ici, on se référera au fichier `MissionController.eps` joint au rapport.

8.7.4 Les propriétés

Le nombre des valeurs possibles pour l'ensemble des variables du noeud « MissionController »implique un très grand nombre d'états et de transitions. Il nous a donc été impossible d'obtenir la terminaison de `acheck` pour le calcul des propriétés de ce noeud.

8.8 Le noeud « MasterRobot »

8.8.1 Description

Ce noeud représente le robot maître. Il est constitué de sous-noeuds correspondant aux différents capteurs, ainsi qu'aux moteurs. Il peut modifier les valeurs des moteurs, communiquer des ordres au robot esclave grâce aux synchronisations.

8.8.2 Le source AltaRica

```
node MasterRobot
  sub
    S1 : SoundLightSensor;
    S2 : SoundLightSensor;
    S3 : DummySensor;
    S4 : DummySensor;

    Dir : Direction;
    C : ServoMotor;

    Co : MissionController;

  event
    noop,
    BackwardSetVarCR, BackwardSetVarC0, BackwardSetVarC1,
    StopSetVarCR, StopSetVarC0, StopSetVarC1,
    ForwardSetVarCR, ForwardSetVarC0, ForwardSetVarC1,
    LeftSetVarCR, LeftSetVarC0, LeftSetVarC1,
    RightSetVarCR, RightSetVarC0, RightSetVarC1,

    sendNoop,
    sendBackwardSetVarCR, sendBackwardSetVarC0, sendBackwardSetVarC1,
    sendStopSetVarCR, sendStopSetVarC0, sendStopSetVarC1,
    sendForwardSetVarCR, sendForwardSetVarC0, sendForwardSetVarC1,
    sendLeftSetVarCR, sendLeftSetVarC0, sendLeftSetVarC1,
    sendRightSetVarCR, sendRightSetVarC0, sendRightSetVarC1;

  trans
    // On continue l'action en cours
    true |- noop -> ;

    // On modifie l'action en cours (direction + troisieme moteur)
    true |- BackwardSetVarCR, BackwardSetVarC0, BackwardSetVarC1,
            StopSetVarCR, StopSetVarC0, StopSetVarC1,
            ForwardSetVarCR, ForwardSetVarC0, ForwardSetVarC1,
            LeftSetVarCR, LeftSetVarC0, LeftSetVarC1,
            RightSetVarCR, RightSetVarC0, RightSetVarC1 -> ;

    // On envoie un ordre
```

```
  true |- sendNoop,
          sendBackwardSetVarCR, sendBackwardSetVarC0, sendBackwardSetVarC1,
          sendStopSetVarCR, sendStopSetVarC0, sendStopSetVarC1,
          sendForwardSetVarCR, sendForwardSetVarC0, sendForwardSetVarC1,
          sendLeftSetVarCR, sendLeftSetVarC0, sendLeftSetVarC1,
          sendRightSetVarCR, sendRightSetVarC0, sendRightSetVarC1 -> ;

sync
  <noop, Co.noop>;

  <BackwardSetVarCR, Co.BackwardSetVarCR, Dir.backward, C.setVarR>;
  <BackwardSetVarC0, Co.BackwardSetVarC0, Dir.backward, C.setVar0>;
  <BackwardSetVarC1, Co.BackwardSetVarC1, Dir.backward, C.setVar1>;
  <StopSetVarCR, Co.StopSetVarCR, Dir.stop, C.setVarR>;
  <StopSetVarC0, Co.StopSetVarC0, Dir.stop, C.setVar0>;
  <StopSetVarC1, Co.StopSetVarC1, Dir.stop, C.setVar1>;
  <ForwardSetVarCR, Co.ForwardSetVarCR, Dir.forward, C.setVarR>;
  <ForwardSetVarC0, Co.ForwardSetVarC0, Dir.forward, C.setVar0>;
  <ForwardSetVarC1, Co.ForwardSetVarC1, Dir.forward, C.setVar1>;
  <LeftSetVarCR, Co.LeftSetVarCR, Dir.left, C.setVarR>;
  <LeftSetVarC0, Co.LeftSetVarC0, Dir.left, C.setVar0>;
  <LeftSetVarC1, Co.LeftSetVarC1, Dir.left, C.setVar1>;
  <RightSetVarCR, Co.RightSetVarCR, Dir.right, C.setVarR>;
  <RightSetVarC0, Co.RightSetVarC0, Dir.right, C.setVar0>;
  <RightSetVarC1, Co.RightSetVarC1, Dir.right, C.setVar1>;

  <sendNoop, Co.sendNoop>;
  <sendBackwardSetVarCR, Co.sendBackwardSetVarCR>;
  <sendBackwardSetVarC0, Co.sendBackwardSetVarC0>;
  <sendBackwardSetVarC1, Co.sendBackwardSetVarC1>;
  <sendStopSetVarCR, Co.sendStopSetVarCR>;
  <sendStopSetVarC0, Co.sendStopSetVarC0>;
  <sendStopSetVarC1, Co.sendStopSetVarC1>;
  <sendForwardSetVarCR, Co.sendForwardSetVarCR>;
  <sendForwardSetVarC0, Co.sendForwardSetVarC0>;
  <sendForwardSetVarC1, Co.sendForwardSetVarC1>;
  <sendLeftSetVarCR, Co.sendLeftSetVarCR>;
  <sendLeftSetVarC0, Co.sendLeftSetVarC0>;
  <sendLeftSetVarC1, Co.sendLeftSetVarC1>;
  <sendRightSetVarCR, Co.sendRightSetVarCR>;
  <sendRightSetVarC0, Co.sendRightSetVarC0>;
  <sendRightSetVarC1, Co.sendRightSetVarC1>;

assert
  // Les informations des capteurs.
  Co.s1state = S1.signal;
  Co.s2state = S2.signal;
  Co.s3state = S3.signal;
  Co.s4state = S4.signal;
```

```
    // Les informations des moteurs.
    Co.direction = Dir.direction;
    Co.cSpeed = C.speed;

  init
    Dir.direction := 0, C.speed := 0;
edon
```

8.8.3 La sémantique

La sémantique de ce noeud étant trop riche pour être proposée ici, on se référera au fichier `MasterRobot.eps` joint au rapport.

8.8.4 Les propriétés

Les résultats

```
/*
 * # state properties : 3
 *
 * dead = 0
 * any_s = 189
 * initial = 1
 *
 * # transition properties : 6
 *
 * self = 567
 * notCFC = 0
 * epsilon = 189
 * not_deterministic = 1134
 * self_epsilon = 189
 * any_t = 1512
 */
```

Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

8.9 Le noeud « SlaveRobot »

8.9.1 Description

Il s'agit du robot esclave, il dispose d'un sous-noeud `Direction`, d'un moteur et d'un contrôleur. Il ne fait que recevoir les ordres de l'esclave et les synchronise avec les moteurs.

8.9.2 Le source AltaRica

```
/* Version simple : pas de capteurs. */
```

```
node SlaveRobot
  sub
    Dir : Direction;
    C : ServoMotor;
    Co : IdleSlaveController;

  event
    recvNoop,
    recvBackwardSetVarCR, recvBackwardSetVarC0, recvBackwardSetVarC1,
    recvStopSetVarCR, recvStopSetVarC0, recvStopSetVarC1,
    recvForwardSetVarCR, recvForwardSetVarC0, recvForwardSetVarC1,
    recvLeftSetVarCR, recvLeftSetVarC0, recvLeftSetVarC1,
    recvRightSetVarCR, recvRightSetVarC0, recvRightSetVarC1;

  trans
    // On recoit un ordre.
    true |- recvNoop,
            recvBackwardSetVarCR, recvBackwardSetVarC0, recvBackwardSetVarC1,
            recvStopSetVarCR, recvStopSetVarC0, recvStopSetVarC1,
            recvForwardSetVarCR, recvForwardSetVarC0, recvForwardSetVarC1,
            recvLeftSetVarCR, recvLeftSetVarC0, recvLeftSetVarC1,
            recvRightSetVarCR, recvRightSetVarC0, recvRightSetVarC1 -> ;

  sync
    <recvNoop, Co.noop>;
    <recvBackwardSetVarCR, Co.BackwardSetVarCR, Dir.backward, C.setVarR>;
    <recvBackwardSetVarC0, Co.BackwardSetVarC0, Dir.backward, C.setVar0>;
    <recvBackwardSetVarC1, Co.BackwardSetVarC1, Dir.backward, C.setVar1>;
    <recvStopSetVarCR, Co.StopSetVarCR, Dir.stop, C.setVarR>;
    <recvStopSetVarC0, Co.StopSetVarC0, Dir.stop, C.setVar0>;
    <recvStopSetVarC1, Co.StopSetVarC1, Dir.stop, C.setVar1>;
    <recvForwardSetVarCR, Co.ForwardSetVarCR, Dir.forward, C.setVarR>;
    <recvForwardSetVarC0, Co.ForwardSetVarC0, Dir.forward, C.setVar0>;
    <recvForwardSetVarC1, Co.ForwardSetVarC1, Dir.forward, C.setVar1>;
    <recvLeftSetVarCR, Co.LeftSetVarCR, Dir.left, C.setVarR>;
    <recvLeftSetVarC0, Co.LeftSetVarC0, Dir.left, C.setVar0>;
    <recvLeftSetVarC1, Co.LeftSetVarC1, Dir.left, C.setVar1>;
    <recvRightSetVarCR, Co.RightSetVarCR, Dir.right, C.setVarR>;
    <recvRightSetVarC0, Co.RightSetVarC0, Dir.right, C.setVar0>;
    <recvRightSetVarC1, Co.RightSetVarC1, Dir.right, C.setVar1>;

  assert
    // Les observations du controleur.
    Co.direction = Dir.direction;
    Co.cSpeed = C.speed;

  init
    Dir.direction := 0, C.speed := 0;
edon
```

8.9.3 La sémantique

La sémantique de ce noeud étant trop riche pour être proposée ici, on se référera au fichier `SlaveRobot.eps` joint au rapport.

8.9.4 Les propriétés

Les résultats

```
/*
 * # state properties : 3
 *
 * dead = 0
 * initial = 1
 * any_s = 15
 *
 * # transition properties : 6
 *
 * not_deterministic = 0
 * notCFC = 0
 * self = 45
 * epsilon = 15
 * self_epsilon = 15
 * any_t = 255
 */
```

Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

Chapitre 9

Le système

9.1 Le noeud « System »

9.1.1 Description

Le noeud « System » est l'endroit où sont regroupés les deux robots : l'esclave et le maître. C'est ici que les communications entre chacun d'entre eux sont spécifiées. Cela s'effectue au travers de synchronisations entre les évenements d'envoi du maître et des évenements de réception de l'esclave.

9.1.2 Le source AltaRica

```
node System

  sub
    RM : MasterRobot;
    RE : SlaveRobot;

  event
    comNoop,
    comBackwardSetVarCR, comBackwardSetVarC0, comBackwardSetVarC1,
    comStopSetVarCR, comStopSetVarC0, comStopSetVarC1,
    comForwardSetVarCR, comForwardSetVarC0, comForwardSetVarC1,
    comLeftSetVarCR, comLeftSetVarC0, comLeftSetVarC1,
    comRightSetVarCR, comRightSetVarC0, comRightSetVarC1;

  sync
    <comNoop, RM.sendNoop, RE.recvNoop>;
    <comBackwardSetVarCR, RM.sendBackwardSetVarCR, RE.recvBackwardSetVarCR>;
    <comBackwardSetVarC0, RM.sendBackwardSetVarC0, RE.recvBackwardSetVarC0>;
    <comBackwardSetVarC1, RM.sendBackwardSetVarC1, RE.recvBackwardSetVarC1>;
    <comStopSetVarCR, RM.sendStopSetVarCR, RE.recvStopSetVarCR>;
    <comStopSetVarC0, RM.sendStopSetVarC0, RE.recvStopSetVarC0>;
    <comStopSetVarC1, RM.sendStopSetVarC1, RE.recvStopSetVarC1>;
    <comForwardSetVarCR, RM.sendForwardSetVarCR, RE.recvForwardSetVarCR>;
    <comForwardSetVarC0, RM.sendForwardSetVarC0, RE.recvForwardSetVarC0>;
```

```
    <comForwardSetVarC1, RM.sendForwardSetVarC1, RE.recvForwardSetVarC1>;
    <comLeftSetVarCR, RM.sendLeftSetVarCR, RE.recvLeftSetVarCR>;
    <comLeftSetVarC0, RM.sendLeftSetVarC0, RE.recvLeftSetVarC0>;
    <comLeftSetVarC1, RM.sendLeftSetVarC1, RE.recvLeftSetVarC1>;
    <comRightSetVarCR, RM.sendRightSetVarCR, RE.recvRightSetVarCR>;
    <comRightSetVarC0, RM.sendRightSetVarC0, RE.recvRightSetVarC0>;
    <comRightSetVarC1, RM.sendRightSetVarC1, RE.recvRightSetVarC1>;

  trans
    true |- comNoop,
            comBackwardSetVarCR, comBackwardSetVarC0, comBackwardSetVarC1,
            comStopSetVarCR, comStopSetVarC0, comStopSetVarC1,
            comForwardSetVarCR, comForwardSetVarC0, comForwardSetVarC1,
            comLeftSetVarCR, comLeftSetVarC0, comLeftSetVarC1,
            comRightSetVarCR, comRightSetVarC0, comRightSetVarC1 ->;
edon
```

9.1.3 La sémantique

La sémantique de ce noeud étant bien trop riche pour être proposée ici, on se référera au fichier `System.eps` joint au rapport.

9.1.4 Les propriétés

Les résultats

```
/*
 * # state properties : 3
 *
 * dead = 0
 * any_s = 189
 * initial = 1
 *
 * # transition properties : 6
 *
 * not_deterministic = 1134
 * notCFC = 0
 * self_epsilon = 189
 * any_t = 1512
 * self = 567
 * epsilon = 189
 */
```

Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

Chapitre 10

Synthèse de contrôleur

La principale difficulté dans cette partie réside dans la capacité à définir l'événement redouté pour notre mission. Dans notre cas, les évenements redoutés ne peuvent être que du type : « un signal est déclenché et les opérations qui devraient le suivre ne sont pas effectuées ».

La synthèse automatique dépend aussi de l'expressivité de notre modèle. On a réussis à construire un contrôleur pour la mission « à la main », cela laisse entendre que notre modèle est suffisamment expressif pour la mission souhaitée.

Nous n'avons pas pu concrétiser notre tentative de synthétisation. Malgré de nombreux essais, tous nos contrôleurs générés étaient de type « sûr »avec toutes les transitions à `false`. Nous pensons que nous n'avons pas réussi :

- Soit à bien traduire l'évenement redouté sus-nommé en propriétés `acheck`. En effet, cet évenement redouté fait intervenir une succession de transitions à traverser dans un ordre bien précis, il ne s'agit pas de simples états à ne pas atteindre.
- Soit à bien choisir l'évenement redouté, le notre n'étant alors pas le bon. Puisque c'est la difficulté principale dans la synthèse de contrôleur, il est probable que nous ayons fait une erreur sur ce point.

10.1 Tentative

Voici les sources `acheck` de notre tentative de synthétisation.

```
with MasterRobot do

 dead                    := any_s - src(any_t - self_epsilon);
 notCFC                  := any_t - loop(any_t,any_t);

 show(any_s, any_t);

 test(initial, 1);

 test(dead,0) >  '$NODENAME.prop';
 test(notCFC,0) >> '$NODENAME.prop';
```

```
/*
   S1 light sensor
   S2 sound sensor
   S3 & S4: no sensor
*/

 S1_Black := [S1.signal = 0];
 S1_Grey := [S1.signal = 1];
 S1_White := [S1.signal = 2];

 S2_NoSnd := [S2.signal = 0];
 S2_Snd := [S2.signal = 1] | [S2.signal = 2];

 S3 := [S3.signal];
 S4 := [S4.signal];


 Move_Rev := [Dir.direction = -1];
 Move_Stop := [Dir.direction = 0];
 Move_Left := [Dir.direction = 1];
 Move_Right := [Dir.direction = 2];
 Move_Fwd := [Dir.direction = 3];

 Servo_Rev := [C.speed = -1];
 Servo_Stop := [C.speed = 0];
 Servo_Fwd := [C.speed = 1];


 Act := [Co.mode = 0];
 Send := [Co.mode = 1];


 S1_trigger_B := rsrc(S1_Grey | S1_White) & rtgt(S1_Black);
 S1_trigger_G := rsrc(S1_Black | S1_White) & rtgt(S1_Grey);
 S1_trigger_W := rsrc(S1_Black | S1_Grey) & rtgt(S1_White);
 S1_trigger := S1_trigger_B | S1_trigger_G | S1_trigger_W;

 S2_trigger_ON := (rsrc(S2_NoSnd) & rtgt(S2_Snd));
 S2_trigger_OFF := (rsrc(S2_Snd) & rtgt(S2_NoSnd));
 S2_trigger := S2_trigger_ON | S2_trigger_OFF;

 S3_trigger := (rsrc(S3) & rtgt(not S3))
| (rsrc(not S3) & rtgt(S3));

 S4_trigger := (rsrc(S4) & rtgt(not S4))
| (rsrc(not S4) & rtgt(S4));


/* mouvements */
```

```
noop := label noop;

backward_R := label BackwardSetVarCR;
backward_0 := label BackwardSetVarC0;
backward_1 := label BackwardSetVarC1;

stop_R := label StopSetVarCR;
stop_0 := label StopSetVarC0;
stop_1 := label StopSetVarC1;

forward_R := label ForwardSetVarCR;
forward_0 := label ForwardSetVarC0;
forward_1 := label ForwardSetVarC1;

left_R := label LeftSetVarCR;
left_0 := label LeftSetVarC0;
left_1 := label LeftSetVarC1;

right_R := label RightSetVarCR;
right_0 := label RightSetVarC0;
right_1 := label RightSetVarC1;



/* communication */
s_noop := label sendNoop;

s_backward_R := label sendBackwardSetVarCR;
s_backward_0 := label sendBackwardSetVarC0;
s_backward_1 := label sendBackwardSetVarC1;

s_stop_R := label sendStopSetVarCR;
s_stop_0 := label sendStopSetVarC0;
s_stop_1 := label sendStopSetVarC1;

s_forward_R := label sendForwardSetVarCR;
s_forward_0 := label sendForwardSetVarC0;
s_forward_1 := label sendForwardSetVarC1;

s_left_R := label sendLeftSetVarCR;
s_left_0 := label sendLeftSetVarC0;
s_left_1 := label sendLeftSetVarC1;

s_right_R := label sendRightSetVarCR;
s_right_0 := label sendRightSetVarC0;
s_right_1 := label sendRightSetVarC1;



nonControle := S1_trigger | S2_trigger | S3_trigger
```

```
| S4_trigger;
controle := any_t - nonControle;

show(nonControle, controle) >> '$NODENAME.prop';

/* Conditions pour tourner a gauche */
s_left_0_OK := S2_NoSnd & Send & tgt(S1_trigger_G)
& src(s_left_0);
left_0_OK := S2_NoSnd & Act & tgt(rsrc(s_left_0_OK))
& src(left_0) & S1_Grey;

show(s_left_0_OK, left_0_OK) >> '$NODENAME.prop';

/* Conditions pour tourner a droite */
s_right_0_OK := S2_NoSnd & Send & tgt(S1_trigger_B)
& src(s_right_0);
right_0_OK := S2_NoSnd & Act & tgt(rsrc(s_right_0_OK))
& src(right_0) & S1_Black;

show(s_right_0_OK, right_0_OK) >> '$NODENAME.prop';

/* Conditions pour aller tout droit */
s_forward_0_OK := S2_NoSnd & Send & tgt(S1_trigger_W)
& src(s_forward_0);
forward_0_OK := S2_NoSnd & Act & tgt(rsrc(s_forward_0_OK))
& src(forward_0) & S1_White;

show(s_forward_0_OK, forward_0_OK) >> '$NODENAME.prop';

/* Conditions pour stopper le moteur */
s_stop_0_OK := Send & tgt(S2_trigger_ON) & src(s_stop_0);
stop_0_OK := S2_Snd & Act & tgt(rsrc(s_stop_0_OK))
& src(stop_0);

show(s_stop_0_OK, stop_0_OK) >> '$NODENAME.prop';


go_fwd := reach(reach(s_forward_0_OK, s_forward_0), forward_0);
go_left := reach(reach(s_left_0_OK, s_left_0), left_0);
go_right := reach(reach(s_right_0_OK, s_right_0), right_0);
go_stop := reach(reach(s_stop_0_OK, s_stop_0), stop_0);


ER := any_s
- (s_left_0_OK | left_0_OK
   | s_right_0_OK | right_0_OK
   | s_forward_0_OK | forward_0_OK);

show(ER) >> '$NODENAME.prop';
```

```
/* Gagne pour le controleur */
Gagne := any_s - ER;

/* Perdu pour l'environnement */
Perdu := any_s - ER;

show(Gagne, Perdu) >> '$NODENAME.prop';


Ctrl -= controle & rtgt(Perdu &
                   (src(nonControle & rtgt(Gagne & src(Ctrl))) -
                    src(nonControle - rtgt(Gagne & src(Ctrl)))));


Controlable := initial & src(Ctrl);

// Generation des controleurs
project(any_s, Ctrl, '$NODENAMEControler', true, Co) > '$NODENAMEControler.alt';

show(Ctrl, Controlable) >> '$NODENAME.prop';

done
```

Chapitre 11

Traduction en langage NXC

11.1 Langage NXC

Au vu de l'étude de l'existant, le langage NXC est le plus adapté pour écrire un programme NXT textuellement.

On dispose en effet de toutes les fonctions disponibles avec le NXT-G, mais sous une forme ressemblant beaucoup au langage C.
Il est donc possible de lister toutes les « commandes » disponibles dans notre `MissionController` au travers de fonctions NXC.

On peut regarder les valeurs des capteurs au travers des macros suivantes :
- `s1state -> SENSOR_1`
- `s2state -> SENSOR_2`
- `s3state -> SENSOR_3`
- `s4state -> SENSOR_4`

On peut utiliser la variable `order` pour coder automatiquement les ordres envoyés :
- `sendStopSetVarC0 -> order = 0 -> SendRemoteNumber(1, OUTBOX, 0)`
- `sendForwardSetVarC0 -> order = 1 -> SendRemoteNumber(1, OUTBOX, 1)`
- `sendLeftSetVarC0 -> order = 2 -> SendRemoteNumber(1, OUTBOX, 2)`
- `sendRightSetVarC0 -> order = 3 -> SendRemoteNumber(1, OUTBOX, 3)`
- `sendNoop -> order = 4 -> SendRemoteNumber(1, OUTBOX, 4)`

Les mouvements peuvent être codés comme ceci :
- `StopSetVarC0 -> Off(OUT_BC) ;`
- `ForwardSetVarC0 -> OnFwd(OUT_BC, 30) ;`
- `LeftSetVarC0 -> OnFwd(OUT_C, 30) ;`
- `RightSetVarC0 -> OnFwd(OUT_B, 30) ;`
- `noop -> ;`

On peut donc coder sans problème toutes les commandes du contrôleur de

la mission.

11.2 Méthode de traduction

Nous n'avons pas eu le temps d'implémenter cette partie, cependant, nous avons réfléchi à la théorie.

Nous pensons qu'un contrôleur AltaRica doit être codé en NXC de cette façon :

1. On définit toutes les variables d'états comme des variables globales.
2. On met une boucle infinie qui englobe toutes les instructions.
3. Une succession de `if(GARDE DE TRANSITION) then (ORDRE CORRESPONDANT A LA TRANSITION TIREE + MISE A JOUR DES VARIABLES GLOBALES)` pour chaque transition du noeud à traduire.
4. Si une `GARDE DE TRANSITION` fait intervenir une variable de flux des capteurs, on utilise dans le `if` les macros d'acces aux états des capteurs.
5. Si une transition du modèle met fin au programme, on l'implémente par un `break`.

Quatrième partie

Conclusion

Améliorations

Le modèle choisi est satisfaisant et les contrôleurs « à la main » sont corrects. Cependant ils restent quelques points à aborder :

- Génération automatique d'un contrôleur.
- Meilleure modélisation du protocole Bluetooth dans le cas d'une communication maître/multi-esclave.
- En effectuant les recherches pour la traduction automatique contrôleur/NXC, nous nous sommes rendus compte qu'il aurait mieux fallu ne pas joindre les ordres sur la direction avec les ordres sur le troisième moteur. Cela n'avait pas d'incidence grave sur notre mission donc nous ne nous en sommes pas préoccupé.

Extensions

En plus du travail effectué, on peut trouver d'autres idées pour continuer l'étude sur la modélisation des robots NXT. En voici quelques unes :

- Ajouter des capteurs au robot esclave.
- Expérimenter une communication bilatérale si du matériel le permet.
- Implémenter un traducteur AltaRica/NXT-G.

Pour conclure

Tout l'intérêt de ce projet se situe au niveau de l'abstraction. En effet, jusqu'à présent, les cas que nous étudions concernaient des modèles déjà abstraits ou nécessitant une abstraction relativement facile. Ici, nous avions un cas concret avec des composants complexes où il nous fallait restreindre les domaines de tolérance des capteurs et les paramètres des moteurs.

Bibliographie

[Cor08] Microsoft Corporation. Vpl introduction, 2008. http ://msdn2.microsoft.com/fr-fr/library/bb483088(en-us).aspx.

[Esm08] Esmeta. Robotc.net, 2008. http ://lejos.sourceforge.net.

[Gro06] The LEGO Group. Nxt user guide, 2006. http ://assets.lego.com/downloads/education/9797_LME_UserGuide_US_low.pdf.

[Han07] John Hansen. Not exactly c (nxc) programmer's guide, 2007. http ://bricxcc.sourceforge.net/nbc/nxcdoc/NXC_Guide.pdf.

[Mag08] Robot Magazine. Programming solutions for the lego mindstorms nxt, 2008. http ://www.botmag.com/articles/10-31-07_NXT.shtml.

[Mel07] Carnegie Mellon. Robotc.net, 2007. http ://www.robotc.net.

[Yoc06] Dale Yocum. Nxt tutorial, 2006. http ://www.ortop.org/NXT_Tutorial/index.html.