Rapport de TER :
Automatisation la production logicielle de contrôleurs d'automates

Oussama EL KOURRI - Ronan GUEGUEN - Antoine GODET

Mai 2019

GitLab : https://gitlab.univ-nantes.fr/E158479K/ter20198-2019_legos

Remerciements

Nous tenons à remercier Monsieur Pascal Andre, chercheur et maître de conférence, notre tuteur de stage pour le temps qu'il a accordé à la réponse à nos questions, pour tous les efforts qu'il a fournis pour mieux nous orienter et nous guider vers les meilleurs résultats, ses précieux conseils qui nous ont aidés à mener notre projet à bon port, ainsi que sa générosité intellectuelle, se manifestant dans le partage de sa connaissance, son savoir et de sa grande expérience.

Un grand merci également à Yannis Le Bars pour son aide précieuse, ses conseils sur le langage ATL et sa disponibilité.

Nous remercions chaleureusement tous les enseignants de M1 ALMA de l'université de Nantes, pour leur travail acharné afin que nos études se déroulent dans les meilleures conditions, nous en sommes reconnaissants.

Table des matières

1 Introduction

Dans le cadre de notre formation M1 ALMA, nous avons eu l'occasion de découvrir le domaine de la recherche, durant le module *Research Project*, qui vise à trouver des réponses à un ensemble de questions, de problèmes, à obtenir des informations, à apporter et formuler des preuves sur une hypothèse, à ouvrir ou bien rouvrir un débat sur une certitude, et à apporter des solutions logiques, pratiques et professionnelles.

Notre travail, fut d'étudier la possibilité d'inclure des spécificités du support dans le passage d'un modèle au code, pour objectif de mettre en place un outil qui permette la génération automatique du programme. Nous nous sommes, pour cela, penché sur la transformation d'un modèle standard, suivant la norme UML 5.0, à un modèle décrivant de la manière la plus précise possible les caractéristique relatives au support.

1.1 Problématique

A l'heure du développement exponentiel de l'informatique, des langages et techniques de programmation, et de l'augmentation toujours croissante du nombre de professionnels du milieu, la questions du devenir du code se pose. La programmation est une discipline pouvant être encore considérée comme récente, mais, le temps faisant son oeuvre, de plus en plus de professionnels du milieu partent en retraite et lèguent leurs codes aux nouvelles générations. Ces codes, bien que simples et clairs pour leurs auteurs, l'est souvent beaucoup moins pour les novices devant les reprendre et les développer.

Que cela provienne de manières de coder très spécifiques, de l'utilisation de langages depuis longtemps oubliés ou - à force de modifications successives - d'un code beaucoup trop lourd à comprendre ; le fait est que les héritiers de ce code sont bien souvent dépassés par la tâche de comprendre ce qu'ont appel du "legacy code".

Dans ce genre de situation, l'utilisation de modèles peut s'avérer salutaire : ceux-ci peuvent permettre aux novices d'avoir un aperçu plus global et visuel du code. Cela leur permet de comprendre plus aisément sa structure et son fonctionnement général.

De cet état de fait découle une envie compréhensible de modifier davantage le modèle, simple à comprendre, que le code, ardus à envisager : pourquoi donc modifier à la main le code s'il est possible de modifier directement le modèle dont il est issus, et d'avoir directement un aperçu de cette modification sur le code en résultant ?

Cette idée, optimiste, suppose toutefois l'existence de moyens efficaces de générations de code depuis un modèle. De tels moyens existent, mais sont intrinsèquement limités : nombre de concepts spécifiques à la situations et au support de développement ne sont pas descriptibles sur le modèle, quelle que soit la norme utilisée.

2 Contexte

2.1 État de l'art

Dans les années 80, le nombre d'entreprises travaillant dans l'informatique croit de manière exponentielle et, avec les capacités de calcul plus grandes, les projets réalisés deviennent toujours plus complexes, c'est ainsi qu'en 1989 est fondée l'Object Management Group (OMG), une association de professionnels de l'informatique voulant faire gagner du temps aux développeurs, rendre les projets plus maintenables et évolutifs et augmenter la productivité.

Ainsi, en 1994, Grady Booch débute les travaux sur la méthode unifiée (UM) chez Rationnal Software. Il sera rejoint par James RUMBAUCH, chercheur chez General Electric, ayant publié un ouvrage sur la technique de modélisation objet (OMT).
Ils seront rejoints plus tard par Ivar Jacobson dont les travaux sur les *uses cases* sont ajoutés à UM. UM changera de nom pour devenir Unified Modeling Laguage (UML) en 1996 et en 1997, l'OGM normalise UML, la version 1.0 sort.
UML continuera d'être amélioré jusqu'à arriver à la version 2.5 en 2013, il compte maintenant 14 types de diagrammes différents.

UML présente divers diagrammes, des diagrammes de structures, des diagrammes de comportements et des diagrammes d'interaction. On trouve notamment parmi les diagrammes de structure les diagrammes de classes représentant les différentes classes intervenant dans le système et, parmi les diagrammes de comportement, les diagrammes états-transitions représentant le comportement du système et de ses composants.

Les premiers outils supportant le développement par modèles ont été créés vers la fin des années 80, mais c'est vraiment après l'arrivée du logiciel Rationnal Rose et de l'UML que le MDE prend de l'importance.

Un autre standard ratifié par l'OMG est le Meta-Object Facility (MOF) utilisé pour définir un ensemble de règles de typage sur des entités dans l'architecture COBRA, un autre standard créé par l'OMG facilitant la communication entre différents systèmes déployés sur des plate-formes diverses, ainsi qu'un ensemble d'interfaces parmi lesquelles les types choisis peuvent être créés et manipulés.

Le MOF est au sommet d'une architecture à quatre couches :
— M3 : la couche méta-méta-modèle MOF,
— M2 : la couche méta-modèle comme UML,
— M1 : la couche modèle par exemple les modèle décrit par UML et
— M0 : la couche donnée ou monde réel qui décrit les objets du monde réel.

L'OMG créa aussi le XMI, un format supportant MOF qui définit des formats d'échanges de méta-données sous forme XML sur les couches M1, M2 et M3, ce standard est fréquemment utilisé par les logiciels de génération de code souvent dans des fichiers UML.

La génération automatique de code est utilisée dans plusieurs domaines de l'informatique, notamment en compilation où un code lisible par l'Homme est passé en entrée et est écrit selon un méta-modèle est ensuite converti en un code interprétable par une machine suivant un méta-modèle différent.

En Février 2019, Gregor Tolksdorf, Eric Esche, Günter Wozny et Jenz-Uwe Repke publient "Customised code generation based on user specifications for simulation and optimization" [Tol+19] dans Computer and Chemical Engineering, ils y présentent une méthode de génération de code utilisant un langage de spécification basé sur MathML et XML permettant de réduire les implémentations manuelles et aussi appliquer le développement dirigé par design au Génie Chimique.

En Décembre 2018, Samar Ali Abdallah, Ramadan Moawad et Esaam Eldeen Fawzy publient "An optimization approach for automated unit test generation tools using multi-objective evolutionary algorithms" [AMF18] dans Future Computing and Informatics Journal où ils proposent une méthode pour la génération de automatique de cas de tests.

En Décembre 2018, Hugo Bruneliere publie "Generic Model-based Approaches for Software Reverse Engineering and Comprehension"[Bru18]. Il y présente MODISCO, un projet Eclipse pour une approche de rétro-ingénierie dirigée par modèle permettant une réutilisation des modèles pour de futurs travaux et y décrit EMF Views, un framework implémentant une vue modèle générique décrite dans sa thèse.

Notre projet de recherche s'est déroulé à la faculté de Nantes, au laboratoire des science du numérique de Nantes (LS2N) de l'UFR des sciences et techniques. Le but de ce projet était d'étudier le passage d'un modèle uml à un code exécutable sur une plate-forme spécifique, en l'occurrence un robot LEJOS. Pour cela, nous avons étudier la suite des transformations nécessaire en modèle nécessaires pour passer d'un méta modèle [1] à un code. Il est toutefois à noter que ce code ne peut intrinsèquement pas être directement exécutable et beaucoup de choses doivent y être complétés à la main afin qu'il soit réellement exécutable.

2.2 Laboratoire des sciences du numérique de Nantes

Le laboratoire des sciences du numérique de Nantes (LS2N) est un pôle majeur de recherche en informatique sur le territoire Nantais, il regroupe vingt-cinq d'équipes de recherche dont l'équipe AeLoS dans laquelle nous avons réalisé notre projet de recherche.

2.3 Équipe AeLoS

L'équipe AeLoS compte sept membres permanents (enseignant-chercheurs), elle fait partie du pôle science du logiciel et des systèmes distribués, et a pour objectif l'étude de logiciels, la recherche de méthodes pour la correction de logiciels, la mise en place de modèles sémantiques, leur construction et leur vérification.

Notre projet de recherche était encadré par Pascal André. Le projet était à temps partiel, en parallèle des autres cours, et a duré 5 mois, entre le 18 Janvier et le 28 Mai, une journée par semaine, puis tout les jours à partir de la fin des examens. Nous voyions notre encadrant presque toutes les semaines afin de parler de l'avancement du projet.

1. Un méta-modèle est un modèle de modèle

2.4 Environnement et outils de développements

Avant d'introduire notre travail, il est primordial de présenter brièvement les différents outils de développement utilisés dans le cadre de notre projet.

2.4.1 IDE

Pour toute la partie programmation du projet, nous avons utilisé Visual Studio Code [2] et Eclipse [3]. Toutefois, la plupart des plugins utilisés (LeJos, Papryus, ATL...) n'étant disponibles que sur Eclipse et nous avons donc dû utiliser l'utiliser comme éditeur de texte et environnement de développement majoritaire pour programmer le contrôle des constructions Lego en Java.

2.4.2 Outils de modélisation

Pour la modélisation UML (Unified Modeling Language), nous avons utilisé : StarUML [4], un logiciel open source simple à l'utilisation ; et Papyrus [5], un outil open source du package Eclipse Modeling pour éditer les modèles de type Eclipse Modeling Framework (EMF). Ces derniers respectent l'approche Model Driven Architecture (MDA) permettant la génération automatique des "squelettes" de code dans un langage objet, et d'une représentation d'un diagramme UML via un fichier xmi ou uml.

De plus nous avons utilisé Yakindu [6], une boîte à outils contenant plusieurs fonctionnalités, parmi elles, est ce qui nous intéresse pour notre projet, est la génération du code. Yakindu génère des programme pour différents langages de programmation tels que java , C , C++, à partir d'un diagramme états transition. Ce dernier est disponible avec des licences gratuites pour certains utilisateurs (étudiants, utilisateurs non commerciaux . . .)

2.4.3 Langage de transformation

Nous utiliserons comme modèle source un modèle UML sous format xml et décrit par un fichier .uml que nous passerons en paramètres d'un script ATL (ATLAS Transformation Language), un outil de transformation de modèle développé par l'équipe AtlanMod de l'université de Nantes, qui permet de produire un ensemble de modèles cibles à partir d'un ensemble de modèles sources pour la génération de code.

2.5 Méthode de travail

Dans le cadre de projet, nous avons choisi de favoriser le travail de groupe, nous avons donc réalisé la majeure partie de nos journées de travail ensemble, chacun cherchant des renseignements sur son pc et relayant au reste du groupe les informations adéquates trouvées. Chacun d'entre nous avais un domaine dans lequel il était plus à l'aise, mais nous avons réellement favoriser le travail groupé.

2. Visual Studio Code : https://code.visualstudio.com/
3. Eclipse : https://www.eclipse.org/
4. StarUML : http://staruml.io/
5. Papyrus : https://www.eclipse.org/papyrus/
6. Yakindu : https://www.itemis.com/en/yakindu/state-machine/

3 Comparaison Code/UML

3.1 Comparaison code Java Lejos et diagramme UML

Considérons le diagramme d'Etat-Transition représentant le fonctionnement du RileyRover réaliser en Java Lejos par le groupe de TER de l'année dernière :

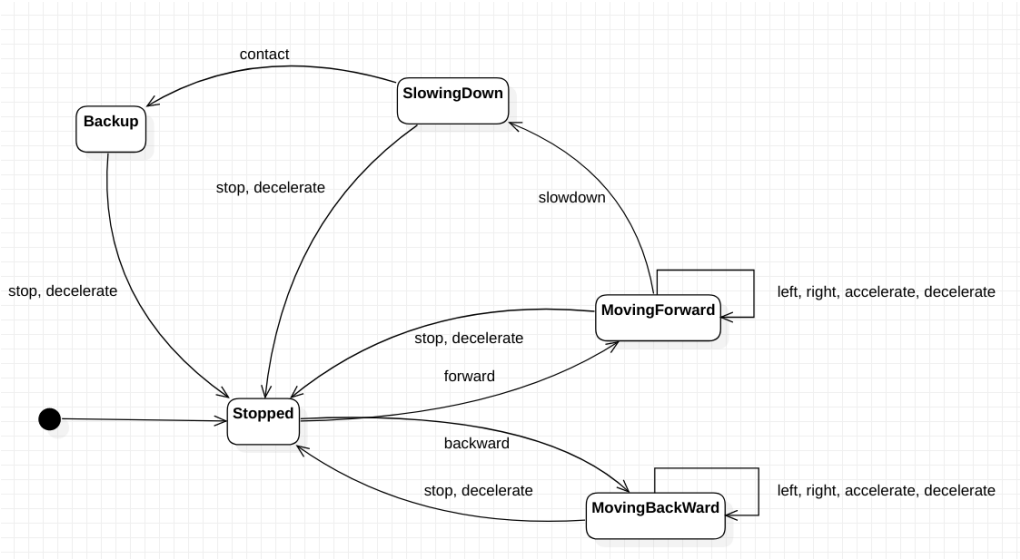

FIGURE 1 – Diagramme de transition du RileyRover

Transition Forward

La transition vers l'états movingForward, se fait si seulement si la machine était en état Stopped, et cela demande au niveau du code Java Lejos 7.3.1, un changement d'état des moteurs vers un nouveau état avec un rightMotor.movingForward() et lefttMotor.movingForward() en renseignant la vitesse. On en déduit la nécessité de passer en paramètre pour la génération de code, les moteurs à activer, la vitesse initiale à laquelle ils fonctionneront ainsi que le sens de rotation. Les getters et les setters de la classe Motor, ainsi que turnLeft(), turnRight(), accelerate() et deaccelerate(), sont des fonctions nécessaires pour la génération du code de la transition MovingForward. Ainsi que la création de l'interface StateCar et les deux classes StateCarMovingBackward et StateCarMoving-Backward qui l'implémente, ces derniers sont nécessaires pour définir quel est l'état actuel de l'EV3 et changer son état en StateCarMovingForward.

Transition Backward

Le code généré pour la transition movingBackward 7.3.2 est similaire au code généré pour la transition movingForward à ceci près que les moteurs sont actionnés en marche arrière. Ainsi, les paramètres à passer pour la génération de code sont les moteurs à activer, le sens de rotation des moteurs ainsi que leur vitesse initiale.

Transition slowDown

La transition vers un états slowingDown, se fait si seulement si la machine était en état MovingForward. Au niveau du code 7.3.3, la transition slowingDown se fait en changeant l'état du contrôleur ainsi que la vitesse du moteur leftMotor.setPreviousSpeed(30) et rightMotor.setPreviousSpeed(30), deux fonctions définies dans la classe Motor, les getters et les setters de cette dernière sont nécessaires. La génération automatique du code nécessite un passage en paramètre les moteurs ainsi que la vitesse cible.

Transitions vers l'état stopped

La génération du code pour passer à un état "Stopped" sachant que l'EV3 été en mouvement, se fait tout simplement en changeant l'état des deux moteurs avec un leftMotor.stop() et rightMotor.stop(), dans la fonction principale stoped(), en utilisant la fonction getMotor() de la classe Motor.7.3.4

3.2 Etude de la communication Bluetooth

Bluetooth est un protocole de communication suivant la norme IEEE 802.15.1, approuvé en 2002. Il fonctionne dans le domaine du PAN. Il émet dans la bande de fréquence des 2.4 Ghz, est utilisé pour de petites distances (jusqu'à 10m) et permet d'attendre 720Kb/s. Un réseau bluetooth peut regrouper jusqu'à 10 scatternets, chacun pouvant contenir jusqu'à 8 équipements, communicant chacun entre eux. Au vu de ces données, nous voyons donc que le protocole bluetooth s'adapte bien à une communication entre un appareil mobile, type tablette, et un robot de type EV3.
Comment passer d'un modèle UML utilisant des communications bluetooth à un code java gérant lesdites communication ?

L'idée majeure est de passer par des librairies spécifiques à Lejos coté ev3, et des librairies spécifiques pour Android coté tablette ; ce qui permet d'exploiter au mieux les possibilités du Bluetooth.

3.2.1 Coté récepteur

Prenons par exemple un modèle UML représentant ce genre de communications, et la façon dont cela a été implémenté dans le code.
Au niveau UML, nous avons simplement une classe d'attente d'ordre, celle-ci est la classe principale du programme et possède comme variable un contrôleur qui, lui, possède tout les composants (motors, sensor dans le cas de l'EV3).

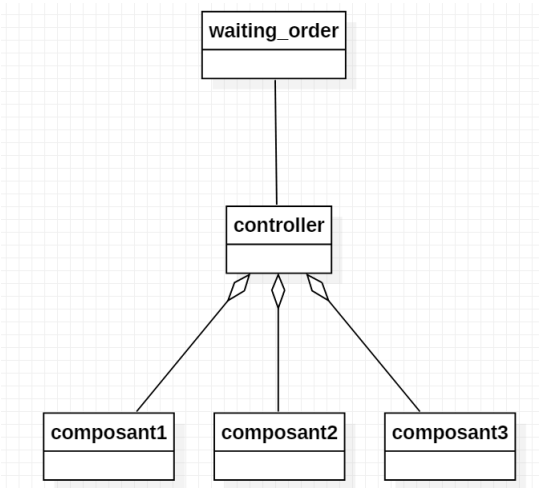

FIGURE 2 – UML Bluetooth coté récepteur générique

Au niveau de la logique d'action, la classe waiting_order gère la réception en Bluetooth des données (provenant de l'émetteur). En fonction du type de message reçu, elle appel la bonne méthode du contrôleur, qui lui même se chargera en local d'appeler les bonnes fonctions sur les bons composants, afin de refléter l'action demandée par l'émetteur.
Nous pouvons voir que cela correspond au diagramme coté client proposé par notre encadrant Pascal André. Voir figure 24 en annexe.
En l'occurence, la classe waiting_order, correspond au controlPanel, le controller à la classe vehiculeController, et les classes motors et sensors sont deux composants précis. En comparant le code et le diagramme UML, nous pouvons effectuer un petit exercice de rétro-ingénierie afin de déduire certaines opérations ayant permis de passer d'un à l'autre :

- La classe Controller est convertie d'une manière décrite dans la première partie.
- La classe waiting_order est renommée en MainClassRileyRover.
- Une méthode générique connect est rajoutée utilisant les bibliothèques lejos :
  - remote.BTConnection
  - remote.BTConnector
  - hardware.Bluetooth
- Une boucle tournant tant que l'app fonctionne est ajoutée dans le main, celle-ci s'occupe d'appeler la bonne méthode en fonction du message reçu par Bluetooth.

Une grande partie des étapes citées précédemment peut être automatisée. Toutefois, certains paramètres ont tout de même besoin d'être passés dans le convertisseur par le développeur/expert métier. Ces informations sont :

- L'action a effectué en fonction du code reçu. Peut être passé sous forme d'une hashmap avec la commande pour clé et l'action pour value.
- Le nom de la classe devant remplacer "waiting_order". Un simple string suffit pour cela.

3.2.2 Coté émetteur

Le code au niveau de l'émetteur est basé sur une architecture Android Studio (voir annexe). Coté UML, l'émetteur est représenté de la méthode suivante. Voir figure 25 en annexe.
Comme coté récepteur, le code utilise beaucoup de fonctions de bibliothèques spécifiques à la technologie utilisée, Lejos coté récepteur, Android Studio ici. La majorité de ces fonctionnalités sont appelés dans la classe MainActivity, ainsi que dans la classe ConnectionBluetoothActivity, utilisée dans MainActivity. L'idée générale de MainActivity est de créer une liste d'action, ici appelées "activité", pouvant être utilisée via l'interface graphique. Lorsque l'utilisateur appuie sur la tablette, ou tout autre support physique faisant tourner l'application, le système extrait l'activité devant être lancée dans la liste des activités possibles (relatif au bouton sur lequel l'utilisateur a appuyé).

```
1   private class OnItemClick implements AdapterView.OnItemClickListener {
2       public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
3        /* Find selected activity */
4           String activity_name = activities.get(position);
5           Class activity_class = name2class.get(activity_name);
6
7        /* Start new Activity */
8           Intent intent = new Intent(MainActivity.this, activity_class);
9           MainActivity.this.startActivity(intent);
10      }
11  }
```

FIGURE 3 – Fonction itemOnClick de la class MainActivity

Dans le code ci-dessus, startActivity est une méthode de ListActivity, la classe mère de MainActivity, et importée via les packages relatifs à android studio.

La totalité de cette partie là est toutefois générique et ne dépend pas réellement de ce qui a été fait sur l'UML. Pour trouver ce qui dépend de ce dernier, il convient de regarder du coté de la classe ControlPageActivity, qui possède comme variable de classe tout les boutons et autre moyen d'interactions (type barre de recherche), chacun étant de type Button, gauge (classes des bibliothèques d'Android Studio).

La classe ajoute ensuite un listener sur chacun de ces actionneurs via la méthode setOnTouchListener, ce qui override la méthode onTouch.

```
1   buttonR.setOnTouchListener(new View.OnTouchListener() {
2       @Override
3       public boolean onTouch(View v, MotionEvent event) {
4           switch (event.getAction()){
5               case MotionEvent.ACTION_DOWN:
6                   try {
7                       BTConnect.writeMessage((byte)2);
8                       if(currentSpeed == 0) {
9                           currentSpeed = 10;
10                          digit.updateSpeed(currentSpeed);
11                          gauge.setValue(currentSpeed);
12                          seekBar.setProgress(currentSpeed);
13                      }
14                      return true;
15                  } catch (InterruptedException e) {
16                      e.printStackTrace();
17                      return false;
18                  }
19          }
20          return false;
21      }
22  });
```

FIGURE 4 – Ajout d'un listener pour un boutton dans la class ControlPageActivity

Est-il possible d'automatiser cela ? La mise en place des variables de classe, oui : tout les actionneurs sont décrits dans le diagramme UML. Il suffit donc de créer une variable pour chacun d'eux, du type adapté.

Une partie du setup du listener peut également être automatisée, mais, à partir de la ligne 5, commence le comportement spécifique de chaque actionneur, ce qui ne peut pas être décris dans l'UML et doit donc être mis dans le code manuellement ou passé en paramètre du processus de traduction.

Ces paramètres peuvent prendre différentes formes, mais une hashmap est généralement une bonne initiative au vu de la structure du problème : nous pouvons avoir comme clé, pour le cas (ligne 5 de l'exemple), il est possible de les passer via une hashmap liant l'actionneur à un tableau de paramètres (son MotionEvent (ligne 5 de l'exemple), et le code dans son try par exemple).

3.2.3 Transmission entre l'émetteur et le récepteur

Dans le code, la transmission entre l'émetteur et le récepteur se fait via la classe ConnectBluetoothActivity. La classe est associée à une variable dans la classe ControlPageActivity. Si l'utilisateur n'a pas activé le Bluetooth sur sa machine, une nouvelle activité est créée avec la classe associée ConnectionBluetoothActivity et exécuter afin de mettre en place la connexion.

Par la suite, il est possible d'utiliser la fonction writeMessage associée, qui se chargera d'envoyer un message sur la tablette ; cette fonction est ainsi utilisée dans le code associé à l'appui sur tout type de buttons, sliders... afin de prévenir l'EV3 de la modification.

Qu'est-il possible d'automatiser dans cela ? La plupart des points sont généraux et ne dépendent pas de paramètres éventuels à passer par l'utilisateur, il est donc possible de presque entièrement l'automatiser.

Quelle message doit être envoyé à quel moment est la chose que nous pouvons pas déterminer sans intervention de l'utilisateur, et a déjà été discuté plus haut : Tout le code associé à chaque actionneur devait être placé directement par l'utilisateur.

4 Comparaison entre différents générateurs de code

Plusieurs programmes de modélisation UML proposent aujourd'hui de générer une partie du code représenté par les diagrammes de classes, le programme résultant est généralement un squelette du programme final, les classes du programmes ne possédant que leurs attributs et de méthodes vides devant être remplies par les développeurs.
Il est aussi possible d'associer, pour chaque classe du diagramme de classe, un diagramme d'Etat-Transition décrivant le comportement de la classe.

Ces programmes prennent le modèle créé par l'utilisateur par exemple un modèle UML, suivant un metamodèle d'entré comme l'UML, et suivant le modèle de transformation choisit, resort du code qui suit un métamodèle de sortie.
Par exemple, on passe en entré un modèle UML suivant le métamodèle UML, on utilise un metamodèle de transformation UML →Java, on precise le metamodèle de sortie comme étant le metamodèle Java, alors le fichier de sortie sera un fichier .java correspondant au modèle UML.

Ainsi pour le diagramme UML de classe ci-dessous suivant le metamodèle UML pour une tranformation UML→Java :

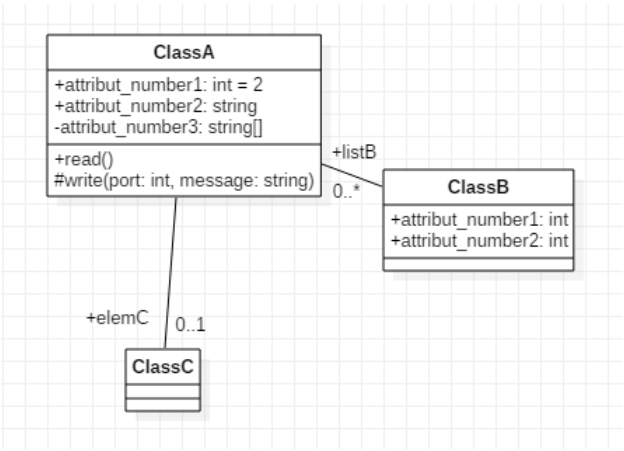

Figure 5 – Diagramme de classe simple

Les fichiers Java générés sont disponibles en annexe.

4.1 Génération sur StarUML

La classe ClassA possède cinq attributs, un entier attribut_number1 initialisé à 2 de visibilité publique, une chaîne de caractère attribut_number2 de visibilité publique, un ensemble de chaîne de caractère attribut_number3 de visibilité privée, un ensemble de ClassB listB de visibilité publique et un élément ClassC elemC de visibilité publique. la classe comporte aussi deux procédures read et write, la procédure write prennant un entier port et une chaîne de caractère message comme paramètre et etant de visibilité protected. Les deux procédures sont vides, n'ayant qu'un message TODO comme contenu.

La classe ClassB possède deux attributs, un entier attribut_number1 et un entier attribut_number2 tous les deux publiques. La classe ne possède aucune méthode.

La classe ClassC ne possède ni attributs, ni méthodes.

Les trois classes importent java.util.*

Cette méthode à l'avantage d'être rapide à utilisée mais nécessite d'utiliser des fichiers MDJ, StarUML utilisant ce format de fichier. StarUML est aussi incapable de générer des fichier .uml tout seul, il peut néanmoins générer des fichiers .xmi à l'aide de l'extension XMI.

4.2 Générations sur Papyrus

La classe ClassA possède cinq attributs, un entier attribut_number1 initialisé à 2 de visibilité publique, une chaîne de caractère attribut_number2 de visibilité publique, un ensemble de chaîne de caractère attribut_number3 de visibilité privée,un élément ClassC elemC de visibilité publique, et un tableau de ClassB listB de visibilité publique. la classe comporte aussi deux procédures read et write, la procédure write prennant un entier port et une chaîne de caractère message comme paramètre et étant de visibilité protected précéder par une documentation @param port et @param message. Les deux procédures sont vides.

La classe ClassB possède deux attributs, un entier attribut_number1 et un entier attribut_number2 tous les deux publiques. La classe ne possède aucune méthode.

La classe ClassC ne possède ni attributs, ni méthodes.

Étant une extension pour Eclipse, Papyrus a le défaut d'être gourmand en ressources, le programme est aussi plus difficile à utiliser que StarUML. Papyrus à néanmoins l'avantage de pouvoir générer des fichier .uml qui pourront être interprétés par ATL contrairement aux fichiers .xmi générés par l'extension de StarUML.

Nous avons donc choisit de travailler sur Papyrus.

Ni StarUML, ni Papyrus n'utilise les diagrammes d'Etats-Transtions associés aux classes des diagrammes de classes, il faut donc trouver un moyen de les exploiter et il est justement possible de le faire en utilisant ATL, une autre extension Eclispe du package Eclipse Modeling.

4.3 Générations sur Yakindu

Contrairement à StarUML et Papyrus, qui n'utilisent pas le diagramme états transition associé aux classe du diagramme de classe lors de la génération du code, Yakindu permet de transformer un diagramme états-transitions en code source d'un langage de programmation, par exemple Java, C ou C++.

Prenant par exemple le diagramme états-transitions suivant :

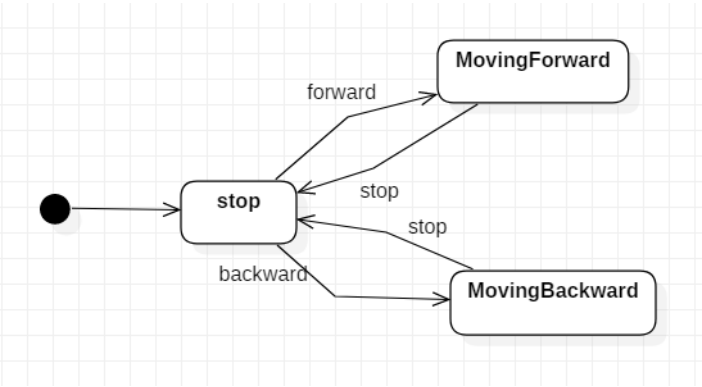

FIGURE 6 – Diagramme états-transitions simple

Le code généré par Yakindu est très riche. Yakindu ne s'arrête pas à la génération du squelette de code comme pour les autres générateurs ci-dessus, mais il génère aussi le corps des fonctions que nous pourrons utiliser pour la programmation. Lien vers le code

Le programme se compose d'une interface basic pour les states machines contenant la déclaration des fonctions ; init() qui initialise la machine d'état, enter() pour changer la machine à état dans un état défini, exit() pour sortir de la machine à état, isActive() afin de vérifier si la machine à état est active, isFinal() pour vérifier si tous états active sont finaux et runCycle() pour démarrer un cycle.
Une autre interface contient la déclaration des méthodes raiseBackward(), raiseStop() et raiseForward(), trois méthodes associés aux différentes transitions de notre diagramme UML.
De plus, deux classes qui implémentent les deux dernières interfaces, et qui contiennent la définition des méthodes déclarés dans ces derniers, d'autres méthodes, des variables ainsi qu'une énumération des états du diagramme de transition.

A partir du code généré par Yakindu, nous pouvons avoir ou extraire des idées par la suite, sur comment nous pouvons utiliser un diagramme états-transitions associée à chaque classe lors des transformations des modèles par ATL pour la génération du code.

5 Transformations des modèles

5.1 Chaîne de transformation

Afin de passer d'un méta-modèle le plus général possible à du code exécutable, il convient d'effectuer une série de modifications sur le modèle uml. Cette suite de transformation et passage d'un modèle uml à un autre jusqu'à en obtenir un transformable en code à remplir, est ce que l'on appel la chaîne de transformation.

L'idée principale de cette chaîne est d'avoir une suite de transformations autonomes et déterministes pour arriver à du code. L'idéal serait de pouvoir cliquer sur un simple bouton et d'exécuter en une fois toute la chaîne. Cela n'est malheureusement pas possible : si une majeure partie de cette chaîne peut être autonome, il n'est toutefois pas possible d'automatiser tout le processus de transformation car certains paramètres ont besoin d'être fournis par l'utilisateur, tel que décrit dans la partie *Comparaison Code/UML* (section 3) .

Notre étude a donc été la **minimisation de l'intervention de l'utilisateur et l'automatisation au maximum du processus de transformation**.
Nous commençons avec un méta-modèle général et effectuons les transformations suivantes :

1. **Renommage des variables :**
   Afin d'obtenir un code exécutable sur un support Lejos, certains concepts (comme des classes) ont besoin d'avoir un nom spécifique. Pour cela, l'utilisateur doit passer une hashmap avec les correspondances entre les noms dans le graphe uml de base et les noms dans le code cible afin que le script de transformation puisse effectuer son office.
2. **Création des énumérations :**
   Dans le graphe uml de base, nous avons des states pattern. Dans notre code toutefois, nous prévoyons d'avoir plutôt des énumérations regroupant tout ces états et une variable currentState dans la classe au dessus du type de cette énumération. Il convient donc de modifier le modèle UML pour **supprimer le state pattern** et **créer les énumérations pour chaque state pattern**.
3. **Création des état courant** :
   Une fois que les énumérations ont été créer, nous ajoutons une variable de classe dans chaque classe du type de l'énumération associée
4. **Pre-implementation des méthodes :**
   A partir du diagramme d'états-transition, certaines méthodes peuvent être déduites et pré-implémentée dans la classe adéquate. Chaque transition entre les états peut en effet être convertie en une méthode dans la classe associée et, en fonction des états liées à cette transition : le corps de la méthode peut même être pré-implémentée. Si une transition **A** va de l'état *state1* à l'état *state2*, nous pouvons en déduire un switch dans la méthode A : si l'état courant est *state1*, nous pouvons le passer à *state2*.

Une fois cela effectué, nous pouvons générer le code. Celui-ci n'est évidemment pas complet : certaines parties - notamment l'implémentation des méthodes - doivent être codés à la main.

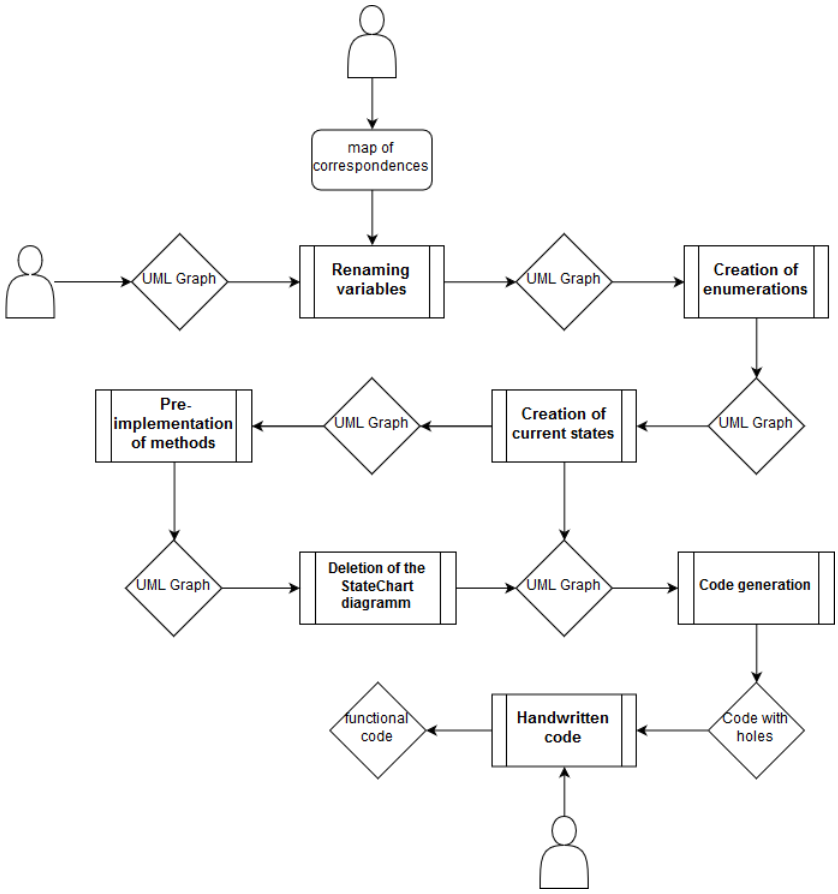

FIGURE 7 – séquence de transformations

5.2 Implémentation des transformations : atl

Le package Eclipse Modeling, une suite de plugins eclipse permettant de représenter et utiliser des modèles.

ATL est une extension eclipse faisant partie de ce package. Son but est de permettre de modifier des modèles à l'aide des règles écrites dans un langage spécifiques. Une transformation ATL est définie dans un fichier .atl possédant une suite de règles décrivant la transformation à effectuer. Pour éffectuer la transformation, il convient d'avoir quatre éléments :

— Le fichier **.atl** avec les règles de transformation
— Le fichier **.xmi** ou **.uml** à transformer
— Le modèle que dois suivre le fichier **.xmi** ou **.uml** à transformer
— Le modèle que dois suivre le fichier **.xmi** ou **.uml** en sortie

Ces quatre éléments combinés sont utilisés pour la génération d'un .xmi ou .uml en sortie vérifiant

le modèle passé en paramètre et étant le modèle d'entrée transformé à partir des règles décrites dans le .atl.

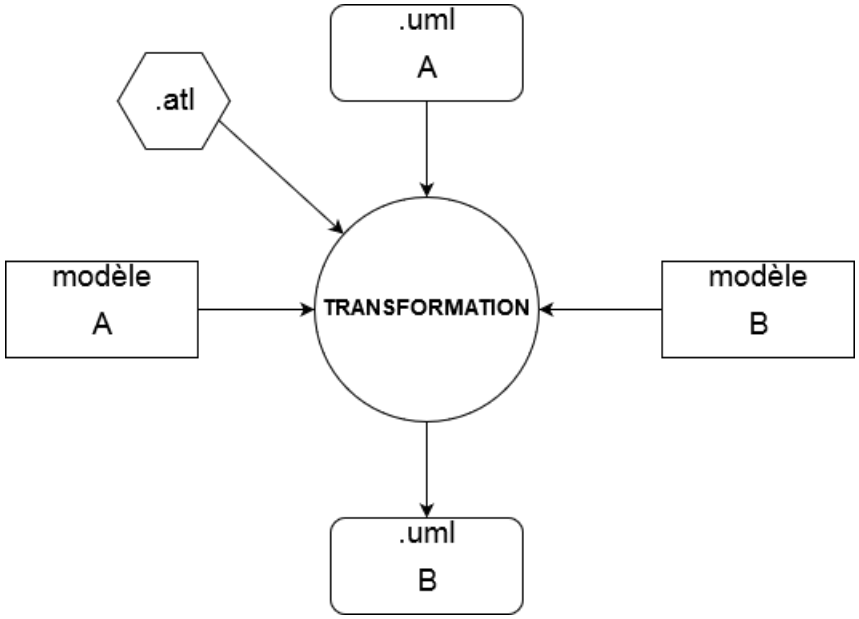

FIGURE 8 – Fonctionnement d'une transformation via ATL

Dans notre cas, les deux .uml suivent toujours la norme uml standard, il n'y a aucun autre conditions sur le format du fichier. Nous aurons donc toujours le modèle uml standard pour le modèle A, et le modèle B.

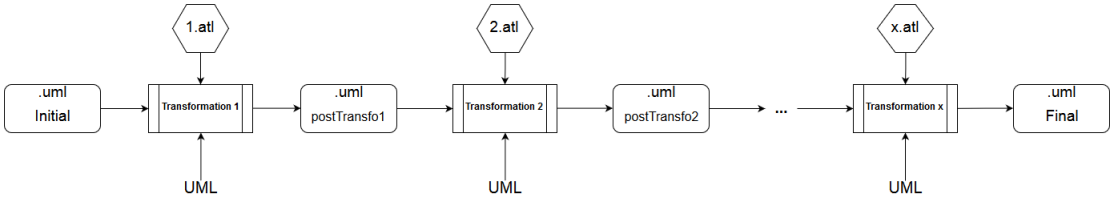

FIGURE 9 – Exemple de suite de transformation (le .uml final est celui duquel le code serra généré)

5.3 Explications des transitions

5.3.1 Contraintes relatives aux transformations

Une **classe** et sa balise de type **StateMachine**, ainsi que sa balise **region**, doivent **avoir le même nom**

5.3.2 Transformation 1 : renommage des variables

Afin d'avoir un modèle le plus générique possible, les noms dans les diagrammes initiales sont très succincts, nous avons ainsi par exemple des "module1" et "module2" pour les noms des variables de classe, et des "state1", "state2"... pour les noms des états dans le diagramme d'état transition. Pour renommer les variables, nous avons donc créé une règle sur atl qui se charge de renommer tout les variables, state, et autres éléments selon une hashmap passée en paramètres :

```
rule RennomageClasse {
    from
        m1 : MM!Class
   using {
        mapping : Map(String, String) = Map{('Main', 'MainClassRileyRover'), ('Ctrl',
            'Controlleur'), ('Module', 'Motor'), ('State1', 'StateCarMovingForward'),
            ('State2', 'StateCarMovingBackward'), ('State3', 'StateCarStopped'),
            ('State4', 'Forward'), ('state5', 'Backward'), ('State6', 'Stop'),
            ('StateMachineCrtl', 'Controlleur'), ('StateMachineModule', 'Motor')};
   }
    to
        m2 : MM!Class (
            name <- mapping.get(m1.name)
        )
}
```

FIGURE 10 – Renommage des classes

Plutôt que d'utiliser une hahsmap, nous aurions préféré utiliser un fichier de configuration json que nous aurions lu directement dans le code, mais cela n'est malheureusement pas possible. La seule solutions s'en approchant aurait été de passer en entrée un fichier de configuration xml avec ces informations. Nous avons donc préféré partir sur une hashmap directement dans le code.

5.3.3 Transformation 2 : création de l'énumération

Cette section a pour tâche de mettre en place l'énumération associée au diagramme de transition. Pour cela, la règle atl recopie chaque classe, mais en lui ajoutant une balise nestedClassifier possédant comme nom le nom de l'énumération associée.

```
rule CreationEnumeration {
   from
      m1 : MM!Class, enum : MM!StateMachine (
         m1.name = enum.name )
   to
      m2 : MM!Class (
         name <- m1.name,
         ownedOperation <- m1.ownedOperation,
         ownedAttribute <- m1.ownedAttribute,
         nestedClassifier <- m1.nestedClassifier.including(enumAdd)
      ),
      enumAdd : MM!Enumeration (
         name <- enum.name
      )
}
```

FIGURE 11 – Création de l'énumération

5.3.4 Transformation 3 : remplissage de l'énumération

Une fois le canvas de l'énumération créé, il nous faut le remplir avec tout les états possibles, états déduits du diagramme d'états-transition.
Pour cette transformation, nous ajoutons dans chaque énumération créée précédemment une balise ownedLiteral avec comme nom celui d'un State de la StateMachine associée à la bonne classe.

```
rule RemplissageEnumeration {
   from
      m1 : MM!Enumeration, enum : MM!State (
         m1.name = enum.owner.name )
   to
      m2 : MM!Enumeration (
         name <- m1.name,
         ownedLiteral <- m1.ownedLiteral.including(enumAdd)
      ),
      enumAdd : MM!EnumerationLiteral (
         name <- enum.name
      )
}
```

FIGURE 12 – Remplissage de l'énumération

5.3.5 Transformation 4 : création de l'état courant

Cette transformation ajoute une variable état courant à chaque classe du type de l'énumération associée et l'initialise à l'état initial de la stateMachine.

```
helper def : getInitState(s : String) : String =
     MM!Region.allInstances()->select(r | r.name = s)->collect(t | t.transition
         )->first()->select(y | y.source = MM!Pseudostate.allInstances()->select(e |
         e.owner.name = s).first())->first().target.name;

rule addEtatCourant {
   from
      class : MM!Class, enum : MM!Enumeration (
         class.nestedClassifier->collect(a | a.name)->first()->startsWith('State') and
             'State'.concat(class.name) = enum.name
      )
   to
      classeArrivee : MM!Class (
         ownedAttribute <- class.ownedAttribute.including(attr)
      ),
      attr : MM!Property(
         name <- '_etatCourant',
         type <- enum,
         defaultValue <- m3
      ),
      m3 : MM!OpaqueExpression(
         language <- 'JAVA',
         body <- enum.name.concat('.'.concat(thisModule.getInitState(class.name)))
      )
}
```

FIGURE 13 – Création de l'état courant

5.3.6 Transformation 5 : pré-implémentation des méthodes

Grâce aux diagrammes d'état-transition, il est possible de déduire toutes les méthodes devant être implémentées, et même de les pré-implémenter. En effet, si nous avons dans notre diagramme une transition **a** allant d'un état state1 à un état state2, nous aurons dans le code de la méthode **a** le switch suivant :

```
switch (this.currentState) {
    case state1:
        //TODO
        this.currentState = state2;
        break;
    default:
        break;
}
```

Bien que nous n'ayons pas encore programmé cette transformation via ATL, l'idée générale est de récupérer la liste des transitions du diagramme d'état-transition et d'ajouter une méthode du même nom dans la classe associée.
Pour pré-implémenter ces méthodes, il nous faut parcourir le diagramme d'états-transition, et afin de déduire comment remplir le switch.

5.4 Transformation 6 : suppression du diagramme d'état-transition

Une fois que toutes les transformations décrites ci-dessus ont été effectuées tous les concepts pouvant être déduits du diagramme d'état-transition ont été utilisés et modélisés dans la diagramme incident. Nous pouvons donc supprimer le diagramme d'état afin d'alléger le modèle. Cette transformation est réalisée en atl via la commande drop.

```
rule SuppressionStateMachine {
   from
      class : MM!StateMachine
   to
      drop
}
```

FIGURE 14 – Suppression du diagramme d'état transition

6 Conclusion

La question de la programmation par modèles est actuelle, et de plus en plus prépondérante, à mesure que les programmes, et l'informatique de manière générale, se développent. Celle-ci permet en effet de s'abstenir d'une partie du développement, parfois considérée comme la plus longue et fastidieuse par certains professionnels du milieu.

Cette idée présuppose l'existence d'un moyen de générer du code à partir du modèle réalisé mais, comme nous avons pu le voir et le développer au cours de ce projet, cette génération nécessite une suite de transformation afin d'obtenir un modèle duquel nous pouvons générer du code.

Durant ce projet, nous nous sommes donc attelé à étudier les transformations successives à réaliser pour atteindre un modèle duquel nous pouvons générer du code. Nous en avons conclu que ces transformations pouvaient être réalisées à l'aide de divers outils : d'un simple parseur jusqu'à un langage de transformation spécialisé, comme ATL.

Afin de découvrir de nouvelles technologies nous sommes, pour notre part, partie sur l'utilisation d'ATL. Il nous a fallu apprendre le langage et avons pu, au terme de plusieurs essais, réaliser une suite de transformations convenables pour passer d'un modèle le plus abstrait possible à un modèle utilisable.

Pour notre projet, nous sommes partis d'un modèle UML avec un diagramme de classe et un diagramme d'état transition associé à chaque classe et sommes arrivés à un diagramme de classe plus complet pour la génération du code. Pour réaliser ceci, nous effectuons la liste des transformations suivantes : renommage des éléments selon ce que désire l'utilisateur, création des énumérations à partir des diagrammes d'états transition, remplissage des énumérations avec les bons états, et création une variable etatCourant dans la classe du type associé à l'énumération...

Nous avons voulu cette chaîne la plus automatique possible. Toutefois, certains paramètres doivent toujours être indiqués par l'utilisateur, comme la hashmap des noms. Il est donc nécessaire

pour l'utilisateur d'intervenir au cours de l'opération, mais également de prendre en compte certaines restrictions liées à l'implémentation. De nombreuses critiques peuvent donc être soulevées par la génération automatique de modèle à modèle : est-il réellement plus pratique de créer un modèle et d'adapter localement certaines transformations plutôt que de coder le tout directement ? N'est-il pas restrictif d'utiliser des scripts de transformations exécutables uniquement avec un IDE spécifique (Eclipse en l'occurrence) ?

Ce projet fut pour nous l'occasion d'apprendre à nous servir de nouvelles technologies, telles que ATL ou Papyrus, mais également à développer nos compétences avec d'autres technologies comme la syntaxe UML générale.

7 Annexes

7.1 Mise en place de la liaison USB entre un PC et le robot LEJOS

Note importante : afin de configurer la liaison usb il convient de configurer le EV3 reconnu comme périphérique inconnu à l'aide d'un driver RNDIS. Toutefois, avec plusieurs essais, l'ev3 n'est pas reconnu (même pas en temps que périphérique inconnu) sur les dernières versions de windows (8 et 10 à ce jour) et sur un ubuntu 18. Dans la suite, nous utiliserons une machine tournant sous windows 7.

Ré-installation de leJos sur le win7 → ok
Installation du plugin eclipse pour l'EV3 → pas ok

Après vérification, la dernière version d'Eclipse pouvant supporter les plugins LeJos est la version 2018-2019 4.9. Il convient donc de la télécharger, elle ou une version antérieure, si l'on souhaite pouvoir installer les plugins.

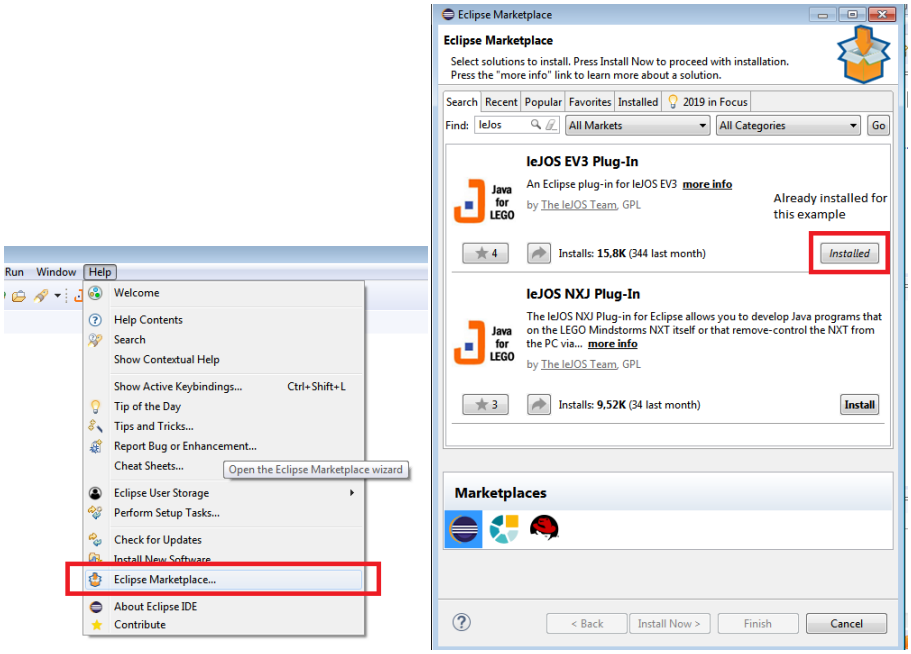

FIGURE 15 – Où trouver le marketplace ?

Après l'installation des plugins, il convient de configurer la liaison usb, pour cela, le mieux est de suivre ce tutoriel [7]

Quelques petits détails sont toutefois à noter :

— Sur les ordinateurs sous win8, win10 ou ubuntu testés, l'ev3 n'est même pas détecté, il convient donc d'utiliser une autre distribution.

7. www.java-online.ch

— Le lien fournit pour l'étape 1 sur le site indiqué ci-dessus n'est plus disponible en date du 08/03/2019. Celui de la version allemande du site, l'est, lui, toujours.

La communication usb entre le robot et la machine se fait donc via un driver RNDIS, une protocole de communication permettant de créer un lien ethernet virtuel, et donc de transmettre des données, entre une machine windows et une autre machine (potentiellement non windows).

Après avoir suivi ce tutoriel, l'EV3 était bien reconnu en temps que réseau RNDIS, mais le ping ne marchait toujours pas. Pour régler ce problème, il nous a donc fallut configurer la connexion.

La première étape fut d'autoriser le partage de connexion au niveau du pc, pour cela :

— Panneau de configuration → Réseaux et Internet → Afficher l'état et la gestion du réseau → Modifier les paramètres de la carte
— Clique droit sur le réseau RNDIS (le robot), et le renommer en EV3 (par commodité)
— Cliquer sur la connexion internet actuellement active
— Propriétés → Partage
— Cocher la case "Autoriser d'autres utilisateurs du réseau à se connecter via la connexion Internet de cet ordinateur" et mettre dans le champ de texte "EV3"

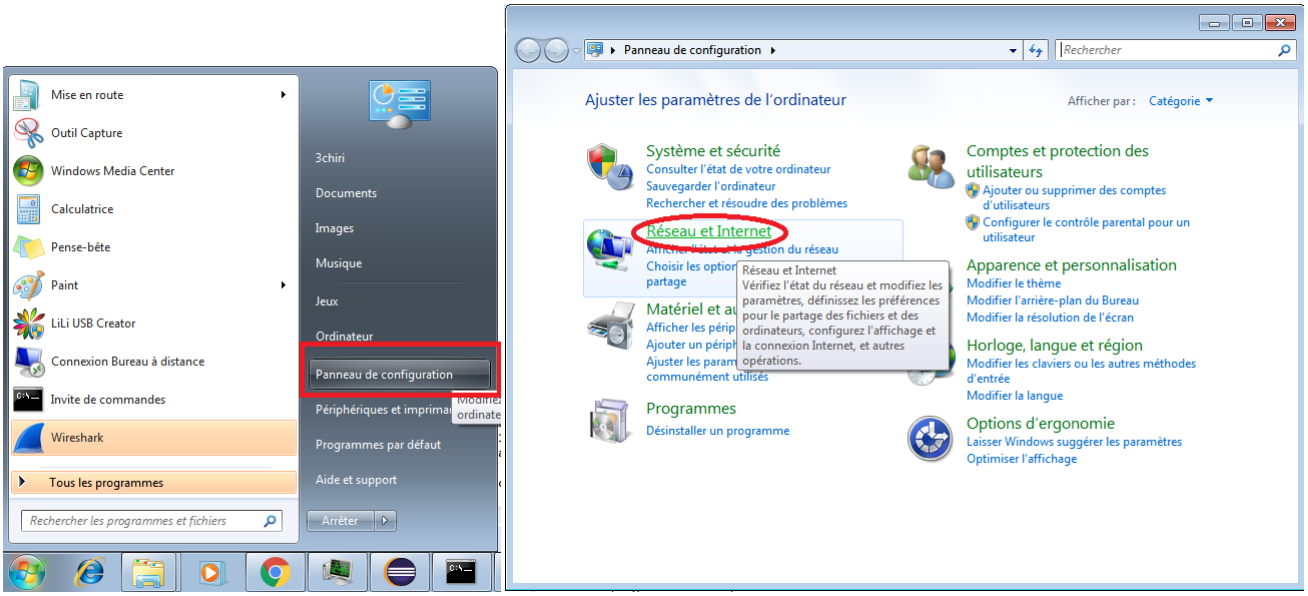

FIGURE 16 – Panneau de configuration → Réseau et Internet

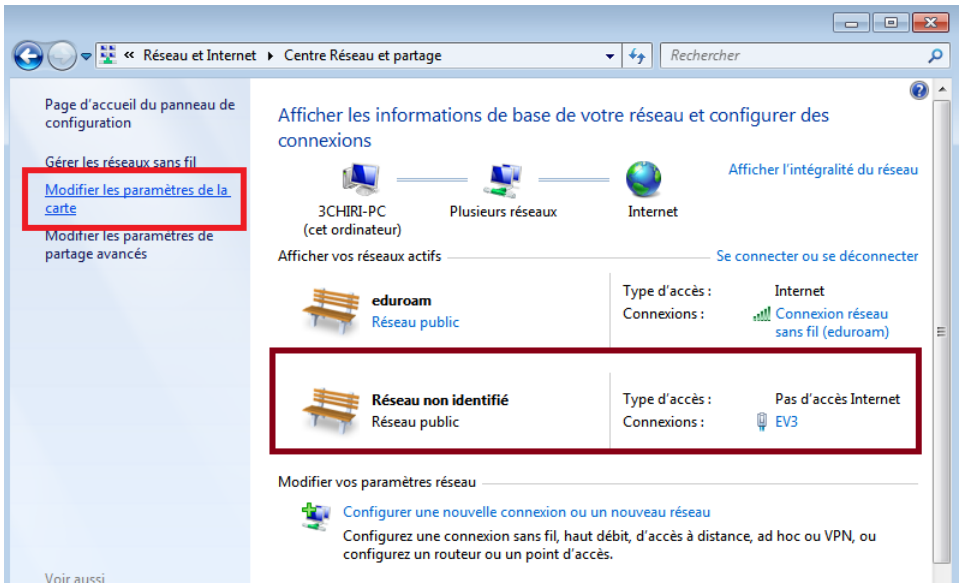

FIGURE 17 – Modifier les paramètres de la carte

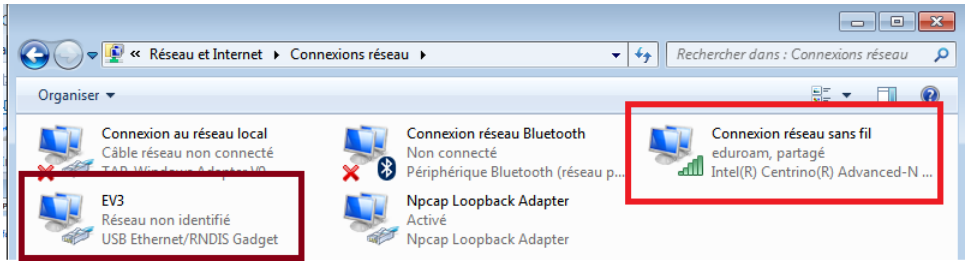

FIGURE 18 – Connexion réseau sans fil

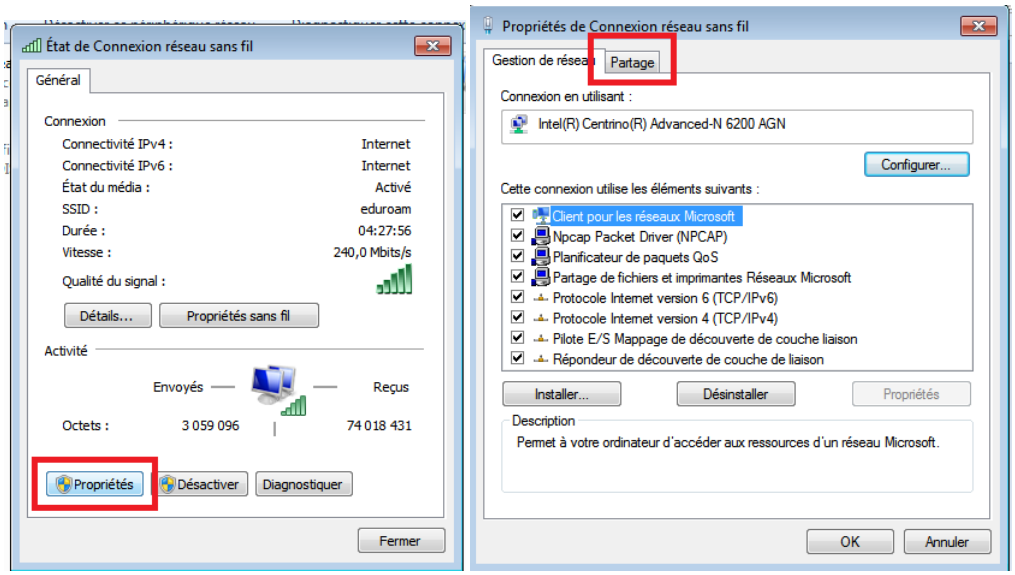

FIGURE 19 – Propriétés → Partage

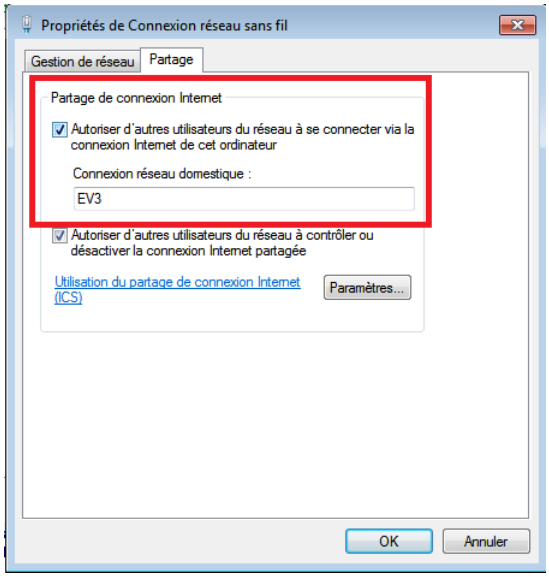

FIGURE 20 – Autoriser d'autres utilisateurs du réseau à se connecter via la connexion internet de cet ordinateur

Aller dans les configurations de connexion de l'EV3 et choisir une adresse réseaux automatique pour USB. Après retour sur le menu de l'EV3, une adresse du réseau local est attribué à l'EV3 ; l'adresse attribuée s'affiche maintenant au-dessus du menu principal. Si l'affichage reste bloqué sur

"Start system", enlever les batteries et redemmarer l'EV3 (il se peut que l'attribution automatique de l'adresse ne fonctionne plus, si c'est le cas, aller dans les configurations de partage du reseau, décocher le partage puis recocher-le).

Tester la connexion par USB avec un ping vers l'adresse de l'EV3. Si l'EV3 répond au ping l'adresse peut être utilisée pour les programmes leJos.

Sur Eclipse, aller dans Windows/Preferences/LeJOS EV3, cocher *Connect to named brick* et taper l'adresse de l'EV3 dans le champs.

Une fois ces étapes effectuées, il est normalement possible de ping l'adresse indiquée sur le robot, et de lancer le programme via eclipse sur runAS/Lejos EV3 Programm

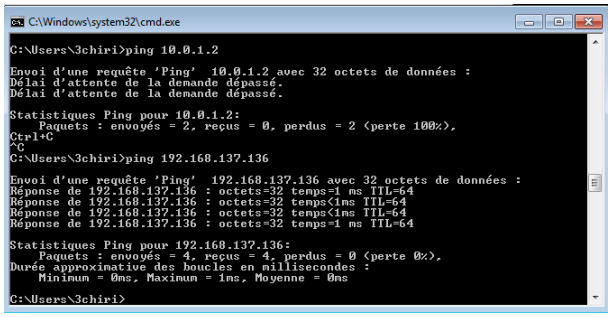

FIGURE 21 – Le ping marche bien...

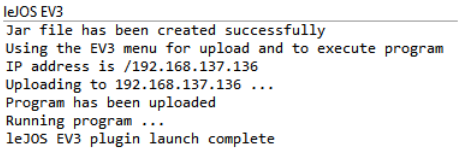

FIGURE 22 – ... de même que l'upload du code

Note : il peut arriver que windows désactive inopportunément le partage de connexion. Dans ce cas, il convient de refaire la procédure décrite figure 12

7.2 Diagrammes Annexes

7.2.1 Figure 1

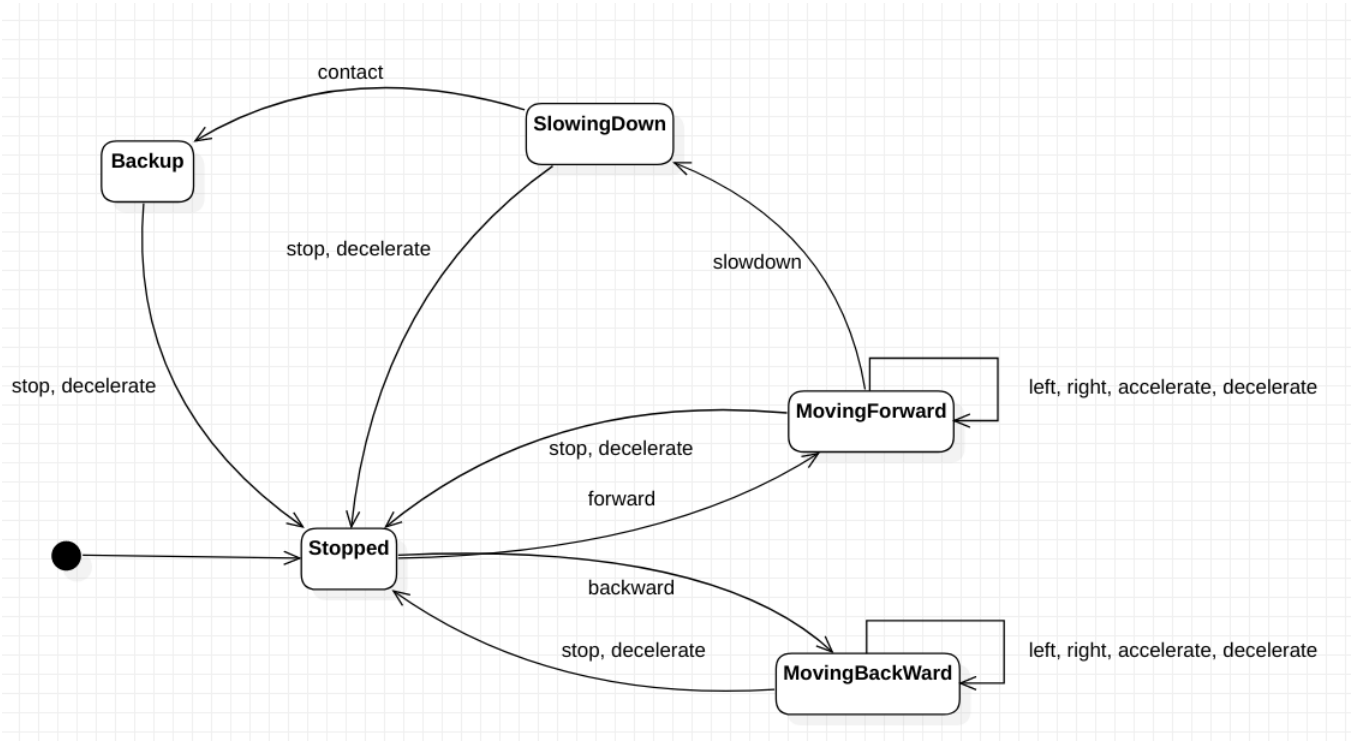

FIGURE 23 – diagramme Etats-Transitions du RileyRover

7.2.2 UML coté récepteur

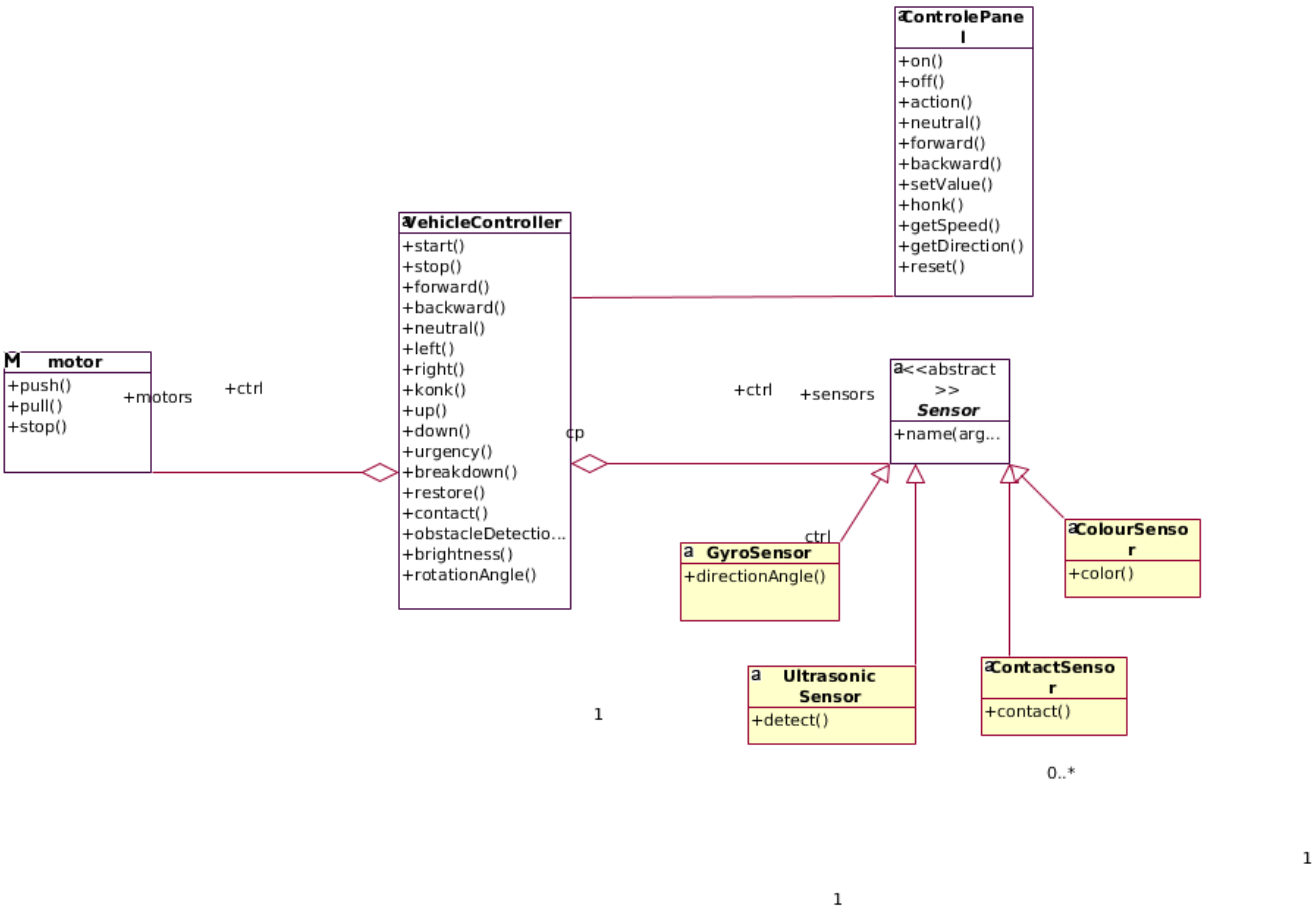

FIGURE 24 – UML Bluetooth coté récepteur exemple

7.2.3 UML coté émetteur

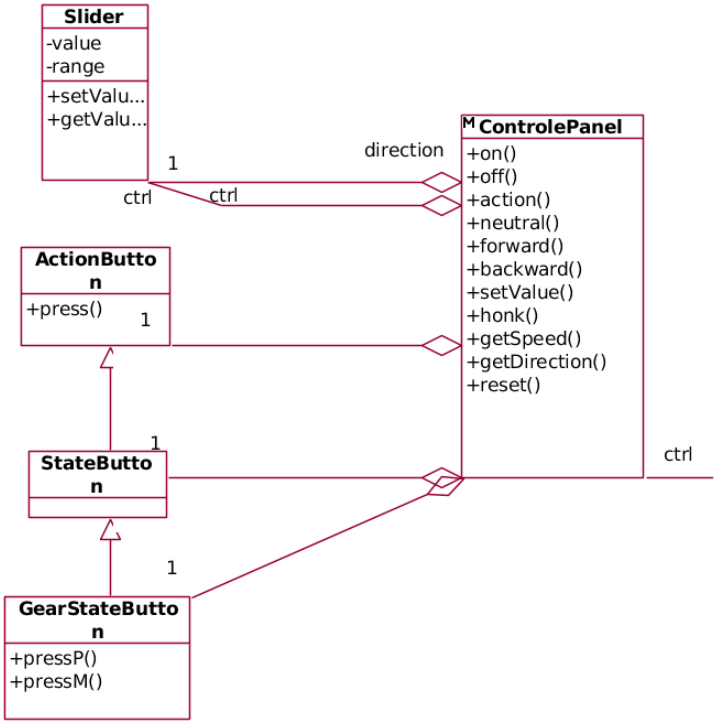

Figure 25 – UML Bluetooth coté émetteur exemple

7.3 Comparaison Code/UML

7.3.1 Code de transition Forward

```
1  private StateCar actualState ;
2  private StateCar stateMovingForward ;
```

```
1  private Motor leftMotor;
2   private Motor rightMotor;
```

```
1   public void movingForward () {
2      if(actualState instanceof StateCarStopped) {
3         System.out.println("want to go forward after stop");
4
5         leftMotor.getMotor().startSynchronization();
6         leftMotor.movingForward();
7           rightMotor.movingForward();
8           actualState = stateMovingForward;
9           leftMotor.getMotor().endSynchronization();
10     } else if(actualState instanceof StateCarMovingBackward) {
11        System.out.println("Can't moving forward if you're moving backward, better stop
              first");
12     } else if(actualState instanceof StateCarMovingForward) {
13        System.out.println("Already movingForward");
14     }
15  }
```

```
  public void turnLeft(int ratio) {
     leftMotor.getMotor().startSynchronization();
     leftMotor.setSpeed(leftMotor.getSpeed());
       rightMotor.setSpeed(rightMotor.getSpeed() *ratio);
       leftMotor.getMotor().endSynchronization();
  }

  public void turnRight(int ratio) {
     leftMotor.getMotor().startSynchronization();
     leftMotor.setSpeed(leftMotor.getSpeed() * ratio);
       rightMotor.setSpeed(rightMotor.getSpeed());
       leftMotor.getMotor().endSynchronization();
  }

  public void accelerate(int value) {
     leftMotor.getMotor().startSynchronization();
     leftMotor.setSpeed(leftMotor.getSpeed() + value);
       rightMotor.setSpeed(rightMotor.getSpeed() + value);
       leftMotor.getMotor().endSynchronization();
  }

  public void decelerated(int value) {
     if(leftMotor.getSpeed() - value <= 0 || rightMotor.getSpeed() - value <= 0) {
        leftMotor.getMotor().startSynchronization();
        leftMotor.stop();
        rightMotor.stop();
          actualState = stateStopped;
          leftMotor.getMotor().endSynchronization();
     } else {
        leftMotor.getMotor().startSynchronization();
        leftMotor.setSpeed(leftMotor.getSpeed() - value);
          rightMotor.setSpeed(rightMotor.getSpeed() - value);
          leftMotor.getMotor().endSynchronization();
     }
  }
```

FIGURE 26 – Code généré dans la classe Controller

```
   public void setSpeed(int speed) {
      this.speed = speed;
      motor.setSpeed(speed);
   }
```

```
   public void movingForward() {
      setSpeed(50);
      motor.forward();
   }

   public RegulatedMotor getMotor() {
      return motor;
   }
```

FIGURE 27 – Code généré dans la class Motor

```
package state;

public class StateCarMovingBackward implements StateCar {

}
```

```java
package state;

public class StateCarMovingForward implements StateCar {

}
```

```java
package state;

public interface StateCar
{
// public void accelerate(int value);
// public void decelerated(int value);
// public void turnLeft(int ratio);
// public void turnRight(int ratio);
// public void stop();
}
```

FIGURE 28 – Code généré dans le dossier state

7.3.2 Code de transition Backward

```java
  private StateCar stateMovingBackward ;
```

```java
  public void movingBackward() {
     if(actualState instanceof StateCarStopped) {
        System.out.println("want to go backward after stop");
        System.out.println(leftMotor.getMotor().getSpeed());
        leftMotor.getMotor().startSynchronization();
        leftMotor.movingBackward();
        rightMotor.movingBackward();
          actualState = stateMovingBackward;
          leftMotor.getMotor().endSynchronization();
     } else if(actualState instanceof StateCarMovingForward) {
        System.out.println("Can't moving backward if you're moving forward, better stop
            first");
     } else if(actualState instanceof StateCarMovingBackward) {
        System.out.println("Already moving backward");
     }
  }
```

FIGURE 29 – Code généré dans la classe Controller

```
1    public void setSpeed(int speed) {
2       this.speed = speed;
3       motor.setSpeed(speed);
4    }
5
6    public void movingBackward() {
7       setSpeed(50);
8       motor.backward();
9    }
```

FIGURE 30 – Code généré dans la classe Motor

```
1  package state;
2
3  public class StateCarMovingBackward implements StateCar {
4
5  }
```

FIGURE 31 – Code généré dans le dossier state du projet

7.3.3 Code de transition SlowDown

```
1    public void slowDown() {
2       if(actualState instanceof StateCarMovingForward) {
3          leftMotor.setPreviousSpeed(leftMotor.getSpeed());
4          rightMotor.setPreviousSpeed(rightMotor.getSpeed());
5          actualState = stateSlowingDown ;
6          leftMotor.getMotor().startSynchronization();
7          leftMotor.setSpeed(30);
8            rightMotor.setSpeed(30);
9            leftMotor.getMotor().endSynchronization();
10      }
11   }
```

FIGURE 32 – Code généré dans la classe Controller

```
1    public int getPreviousSpeed() {
2       return previousSpeed;
3    }
4
5    public void setPreviousSpeed(int previousSpeed) {
6       this.previousSpeed = previousSpeed;
7    }
8
9    public int getSpeed() {
10      return speed;
11   }
12
13   public void setSpeed(int speed) {
14      this.speed = speed;
15      motor.setSpeed(speed);
16   }
```

FIGURE 33 – Code généré dans la classe Controller

```
package state;

public class StateCarSlowingDown implements StateCar{

}
```

FIGURE 34 – Code généré dans le dossier state du projet

7.3.4 Code des transitions vers l'état Stopped

```
  private StateCar stateStopped ;
```

```
  private Motor leftMotor;
   private Motor rightMotor;
```

```
  public void stop() {
    if(actualState instanceof StateCarStopped) {
      System.out.println("AlreadyStopped");
    } else if(actualState instanceof StateCarMovingForward) {
      leftMotor.getMotor().startSynchronization();
      leftMotor.stop();
      rightMotor.stop();
        actualState = stateStopped;
        leftMotor.getMotor().endSynchronization();
    } else if(actualState instanceof StateCarMovingBackward) {
      leftMotor.getMotor().startSynchronization();
      leftMotor.stop();
      rightMotor.stop();
        actualState = stateStopped;
        leftMotor.getMotor().endSynchronization();
    } else if(actualState instanceof StateCarSlowingDown) {
      leftMotor.getMotor().startSynchronization();
      leftMotor.stop();                                    .
      rightMotor.stop();
        actualState = stateStopped;
        leftMotor.getMotor().endSynchronization();
    } else if(actualState instanceof StateCarBackUp) {
      leftMotor.getMotor().startSynchronization();
      leftMotor.stop();
      rightMotor.stop();
        actualState = stateStopped;
        leftMotor.getMotor().endSynchronization();
    } else if(actualState instanceof StateCarSlowingDown) {
      leftMotor.getMotor().startSynchronization();
      leftMotor.stop();
      rightMotor.stop();
        actualState = stateStopped;
        leftMotor.getMotor().endSynchronization();
    }
  }
```

FIGURE 35 – Code généré dans la classe Controller

```
  public RegulatedMotor getMotor() {
     return motor;
  }
```

FIGURE 36 – Code généré dans la classe Motor

```
package state;


public class StateCarStopped implements StateCar {


}
```

FIGURE 37 – Code généré dans le dossier state du projet

7.4 Code Java généré par StarUML pour la figure 5

Classe A :

```java
package Package;

import java.util.*;

/**
 *
 */
public class ClassA {

    /**
     * Default constructor
     */
    public ClassA() {
    }

    /**
     *
     */
    public int attribut_number1 = 2;

    /**
     *
     */
    public String attribut_number2;

    /**
     *
     */
    private Set<String> attribut_number3;

    /**
     *
     */
    public Set<ClassB> listB;

    /**
     *
     */
    public ClassC elemC;

    /**
     *
     */
    public void read() {
        // TODO implement here
    }

    /**
     * @param port
     * @param message
     */
    protected void write(int port, string message) {
        // TODO implement here
    }

}
```

Classe B :

```java
package Package;

import java.util.*;

/**
 *
 */
public class ClassB {

    /**
     * Default constructor
     */
    public ClassB() {
    }

    /**
     *
     */
    public int attribut_number1;

    /**
     *
     */
    public int attribut_number2;


}
```

Class C :

```java
package Package;

import java.util.*;

/**
 *
 */
public class ClassC {

    /**
     * Default constructor
     */
    public ClassC() {
    }


}
```

7.5 Code Java généré par Papyrus pour la figure 5

Classe A :

```java
// -------------------------------------------------------
// Code generated by Papyrus Java
// -------------------------------------------------------

package TestPapyrus;

/*************************************************************/
/**
 *
 */
public class ClassAPapyrus {
   /**
    *
    */
   public int attribut_number1;
   /**
    *
    */
   public String attribut_number2;
   /**
    *
    */
   public ClassBPapyrus[] listB;
   /**
    *
    */
   public ClassCPapyrus elemC;
   /**
    *
    */
   private String[] attribut_number3;

   /**
    *
    */
   public void read() {
   }

   /**
    *
    * @param port
    * @param message
    */
   protected void write(int port, String message) {
   }
};
```

Classe B :

```
// ----------------------------------------------------------
// Code generated by Papyrus Java
// ----------------------------------------------------------

package TestPapyrus;

/*********************************************************/
/**
 *
 */
public class ClassBPapyrus {
   /**
    *
    */
   public int attribut_number1;
   /**
    *
    */
   public int attribut_number2;
};
```

Class C :

```
// ----------------------------------------------------------
// Code generated by Papyrus Java
// ----------------------------------------------------------

package TestPapyrus;

/*********************************************************/
/**
 *
 */
public class ClassCPapyrus {
};
```

8 Lexique

— Digital Twin : Un digital Twin est le "jumeau digital" du robot. Alors que les variables, ainsi la tête de lecture, du programme en local sur la carte microSD du robot sont dans un état précis au cours de l'exécution du dit programme ; la tablette contient elle également un "double" de cet état afin de garantir une parfaite synchronisation entre les deux (le code étant également et trivialement identique).
— ATL : ATLAS Transformation Language
— UML : Unified Modeling Language
— OMG : Object Management Group -> Créée en 1989, OMG est une association américaine dont le but est la standardisation et la promotion du modèle objet, cette association est notamment à la base de l'UML.
— MDE : Model-Driven Engineering est une méthode de développement basée sur la création et l'exploitation de modèles représentant le fonctionnement, comportement d'un programme.
— MOF : Meta-Object Facility est standard de l'OMG pour la représentation des méta-modèles.
— XMI : XML Metadata Intercharge est un standard de l'OMG pour l'échange de méta-données par XML.
— XML : Extensible Markup Language est un langage de balisage qui definit un ensemble de rêgles d'encodage pour des documents dans un format interpretables par l'Homme et la machine.

Références

[AMF18] Samar Ali Abdallah, Ramadan Moawad et Esaam Eldeen Fawzy. "An optimization approach for automated unit test generation tools using multi-objective evolutionary algorithms". In : *Future Computing and Informatics Journal* 3.2 (2018), p. 178–190. ISSN : 2314-7288. DOI : https://doi.org/10.1016/j.fcij.2018.02.004. URL : http://www.sciencedirect.com/science/article/pii/S2314728818300072.

[Bru18] Hugo Bruneliere. "Generic Model-based Approaches for Software Reverse Engineering and Comprehension". Thèse de doct. University of Nantes, 2018.

[Tol+19] Gregor Tolksdorf et al. "Customized code generation based on user specifications for simulation and optimization". In : *Computers & Chemical Engineering* 121 (2019), p. 670–684. ISSN : 0098-1354. DOI : https://doi.org/10.1016/j.compchemeng.2018.12.006. URL : http://www.sciencedirect.com/science/article/pii/S0098135418306987.