Génération de code à partir d'architecture logicielle

Année universitaire : 2017-2018
Formation : Licence 3 Informatique
Nom : Gayraud
Prénom : Matthieu

Sommaire

Remerciements

Je tiens à remercier toutes les personnes m'ayant aidé au cours de ce stage.

Tout d'abord, j'adresse mes remerciements à mon Maître de stage, Gilles ARDOUREL pour son accueil, ainsi que pour ses conseils et explications.

Je remercie également l'ensemble de l'équipe AeLoS pour son accueil lors de la première réunion à laquelle j'ai pû assister, et en particulier, Pascal ANDRE, pour ses conseils avisés sur la dernière partie de mon stage.

Introduction

Objet du stage, contexte et enjeux

Durant ce stage je me suis intéressé à la génération de code à partir d'architecture logicielle.

Pour pouvoir générer du code de manière automatique, il est nécessaire d'avoir un modèle qui sert de patron. Il est par conséquent indispensable d'avoir un ensemble de règles permettant de construire un modèle de la façon la plus rigoureuse possible, Cet ensemble de règles permet de garder une syntaxe similaire entre les différents modèles même si ces derniers sont créés par des utilisateurs différents. On peut donc déjà appréhender les premières étapes nécessaires à la génération de code, et la nécessité de devoir créer des outils permettant d'y arriver.

Afin de ne pas avoir à remonter encore plus haut et de devoir créer un langage permettant la création d'un autre langage permettant la modélisation, on utilisera un plug-in d'un environnement de développement. Il est nécessaire de poser une limite pour les langages définissant d'autres langages et on s'aperçoit ainsi rapidement qu'il est indispensable d'avoir un langage se définissant lui même.

Sachant cela, j'ai d'abord commencé à lire un article définissant les concepts liés à la modélisation. Une fois ces concepts compris et assimilés j'ai commencé la création d'une grammaire pour un langage définissant les modèles, que je présenterais dans la partie « Création de la grammaire»

Ensuite, je me suis intéressé à la modélisation, avec un exemple simple et fictif, ce qui m'a permis d'effectuer des tests et me rendre compte que les règles de grammaires seules ne suffisent pas pour assurer la cohérence du modèle. Nous verrons cela en détail dans la partie « Règles de vérification et modélisation »

Une fois la création du langage permettant la modélisation terminé, je me suis intéressé à Java LeJOS, un firmware java permettant de programmer des petits robots en lego : L'objectif étant, à partir d'un modèle, la génération d'un code permettant d'utiliser un robot mindstorms-ev3.

Présentation de l'université et de l'équipe :

Ce stage se déroule à l'université des Sciences et des Techniques de Nantes, dans le Laboratoire des Sciences du Numérique de Nantes (LS2N), une unité de recherche créée en janvier 2017, suite à la fusion du Laboratoire Informatique de Nantes Atlantique (LINA) et de l'Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN)[1]

Mon stage s'est effectué au sein de l'équipe AeLoS (Architectures et Logiciels Surs) à l'université des Sciences et des Techniques de Nantes. Cette équipe a comme principal objet d'étude le logiciel et cherche à garantir ses corrections. Les différentes difficultés de ces recherches sont abordées suivant plusieurs thématiques gravitant autour des modèles permettant de couvrir les différentes couches d'abstractions constituant un logiciel.

Présentation du contexte et du projet :

Ce projet permet, en plus du travail sur la génération de code, d'introduire des éléments de programmation par composants.

Concepts liés à la modélisation

Tout d'abord mon travail a consisté en la lecture d'un article de recherche où sont définis de manière rigoureuse, les concepts liés à la modélisation, aux méta-modèles et à l'ingénierie logicielle dirigée par les modèles. Les définitions utilisées sont celles issues de l'article de A.R.d.Silva « Model-driven engineering: A survey supported by the unified conceptual model »

Modèle, méta-modèle et langage de modélisation

D'après l'article, un modèle est un système aidant à définir et à répondre aux questions d'un système étudié, sans avoir à le considérer. Un méta-modèle est un modèle qui défini la structure d'un langage de modélisation. Il s'agit ici de la définition du méta-modèle pour le domaine du génie logiciel, en fonction des domaines il existe d'autres définitions qui ne seront pas abordés dans ce rapport.

L'utilisation de modèles dans la programmation n'est pas récente, ils étaient jusqu'alors considérés comme des artefacts annexes, principalement utilisés pour de la documentation. Cependant, depuis une dizaine d'année, une nouvelle approche concernant l'utilisation des modèles est apparue, ces derniers occupent, dans certains cas, une place importante lors du développement : On parle alors d'ingénierie logicielle dirigée par les modèles.[2]

Un langage de modélisation peut-être défini à l'aide d'un méta-modèle, il permet de représenter l'ensemble des modèles étant conforme à ce méta-modèle. (voir Fig 1)

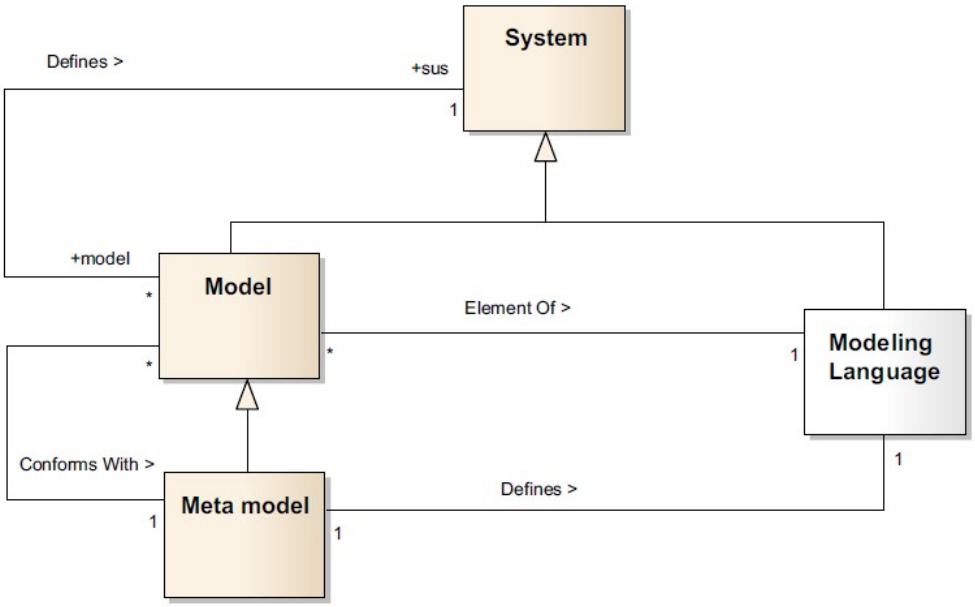


Fig 1 : Définition d'un méta-modèle, relations entre le modèle et le méta-modèle[1]

1 Schéma issu de l'article de A.R.d.Silva

Langage dédié

Un langage dédié, ou Domain Specific Language (DSL) est un langage de programmation dont les spécifications sont conçus pour répondre à un besoins précis[3]. Ce qui s'oppose à un langage de programmation généraliste. Le principal intérêt de ce genre de langage est de pouvoir utiliser et modifier un programme sans avoir de besoin de connaissance préalable en programmation avec des langages généralistes, comme nous le verrons dans la partie « Outils pour la modélisation ». De plus ces langages sont facilement réutilisable tant que l'on reste dans le domaine pour lequel ils sont prévus.

Conclusion sur l'article

L'article permet donc de mieux comprendre l'intérêt de la modélisation, ainsi que la notion de hiérarchie y étant liée, en définissant les différentes notions abordées. Cela permet de répondre à de nombreuses questions, notamment, « Qu'est-ce qu'un modèle ? », « Quelle est la relation entre un modèle et un méta-modèle ? » (Voir :*Modèle, méta-modèle et langage de modélisation*) « Comment utiliser les modèles dans le contexte d'un développement logiciel ?» , « Quelles sont les relations entre les différentes approches dirigées par les modèles ? ».

En fonction des besoins et du domaine dans lequel un utilisateur travaille, les besoins lors du développement d'un logiciel ne seront pas les mêmes

Présentation et installation des outils

Outils pour la modélisation

Afin de mener ce projet à bien, j'ai utilisé l'environnement de développement Eclipse[4] sur lequel le plug-in Xtext[5] est disponible. Il est aussi possible d'utiliser Xtext sur un navigateur, cependant on ne pourrait pas en exploiter toutes les fonctionnalités : Par exemple, le débogage sur navigateur n'est pas possible.

Il faut donc, une fois l'installation d'Eclipse terminée, installer Xtext à l'aide de la fonction « installer de nouveaux logiciels ». Il n'y a pas besoin d'effectuer de manipulations supplémentaires, l'environnement de développement se charge de l'installation des plug-in.

Xtext permet de créer des langages de programmation ainsi que des langages spécifiques à un domaine. Cela permet donc la création de langage très précis lié à un besoin particulier : L'intérêt étant d'éviter à l'utilisateur d'avoir à apprendre un nouveau langage complexe.

En effet, comme le langage est créé en fonction des besoins de l'utilisateur et du domaine dans lequel il travaille, il n'est pas nécessaire d'avoir un outil complexe, constitué de nombreuses bibliothèques qui ne seront jamais utiles dans un contexte précis.

On peut donc, ici, voir le lien avec les notions abordées dans le point précèdent, Xtext permet de définir un méta-modèle afin de créer un langage de modélisation et de ce fait est donc un méta-méta-modèle.

Outils pour le robot lego-mindstorms-ev3

C'est aussi en utilisant Eclipse que j'ai élaboré un projet en Java, afin de faire fonctionner le robot mindstorms-ev3. Pour des programmes simples, il est également possible d'utiliser le logiciel fourni avec les lego-mindstorms-ev3, appelé LEGO MINDSTORM Education EV3. Ce logiciel permet à des utilisateurs n'ayant que très peu de connaissances en informatique, de créer des programmes simples pour un robot. On peut donc voir ce logiciel comme une variante, bien plus complexe, du travail effectué sous Xtext. A l'aide de différents blocs, on peut créer un modèle donnant une suite d'instructions à un robot, plus simple à comprendre qu'une suite d'instructions textuelles comme ce qu'on pourrait faire grâce à la modélisation sur Xtext. Cependant, avec MINDSTORM un utilisateur est rapidement bloqué par ce que les constructeurs du logiciel ont implémentés ou non. Par exemple, il n'est pas possible d'effectuer des import, on ne peut pas non plus voir le fonctionnement interne des blocs d'instructions proposés. C'est pour cela qu'il est intéressant de créer un langage plus permissif, donnant davantage de choix à l'utilisateur, et lui permettant de

modéliser un problème précis.

En fonction des règles de modélisation il est possible de créer des modèles assez simples permettant de générer un code assez complexe. Dans ce cas, le travail est plus long lorsque sont fixées les différentes règles de génération, mais la création de la grammaire est plus simple.

Les projets que l'on peut élaborer, grâce au logiciel proposé par Lego, se rapprochent davantage d'un programme, que la modélisation faite avec Xtext.

Bien que représenté sous forme de bloc, les instructions que l'on peut donner au robot se rapproche énormément d'un programme écrit en ligne de code : On y retrouve, par exemple les conditionnelles. De plus, un bloc est généralement l'équivalent d'une fonction, dont on donnera les paramètres.

Ci-dessous un exemple de programme avec le logiciel MINDSTORM : La suite d'instruction permet de faire avancer le robot en ligne droite.

Fig 2 : Exemple de programme avec le logiciel MINDSTORM

On peut ainsi distinguer 6 blocs. Cependant, dans cet exemple, il n'y a que 3 blocs différents. Le premier bloc permet d'exécuter le programme. Les blocs verts permettent de contrôler les moteurs B et C, chacun correspondant à une roue, à l'aide de plusieurs paramètres : Le premier pour la puissance à gauche, le second pour la puissance à droite. On peut ensuite donner le nombre de rotation, l'angle de rotation ou encore le temps de rotation. Chacune de ces options est représentée par l'un des blocs vert. Le bloc orange comportant le sablier permet d'effectuer une pause ; la durée de la pause est donnée en seconde.

Avec Xtext l'objectif est de générer des méthodes équivalentes à ces blocs d'instructions, pour que le robot effectue une action donnée.

Bien que l'objectif final du projet soit la génération de code, j'ai choisi de faire en sorte que la syntaxe utilisée dans le modèle, soit relativement proche de celle du java, sans que cela ne gène en rien une personne habituée au java, qui verrait les nombreuses différences syntaxiques lors de la modélisation, ou bien un utilisateur n'ayant aucune connaissances concernant ce langage.

Création et utilisation des outils pour la modélisation

Lorsque l'on crée le projet Xtext, 5 dossiers sont créés automatiquement, (voir fig.3) On s'intéressera seulement aux dossiers « com.project.foo » et « com.project.foo.tests ».

La majeure partie du contenu de ces dossiers est générée automatiquement par Xtext, dès qu'on demande l'exécution de la grammaire ; ce qui doit être fait après chaque modification.

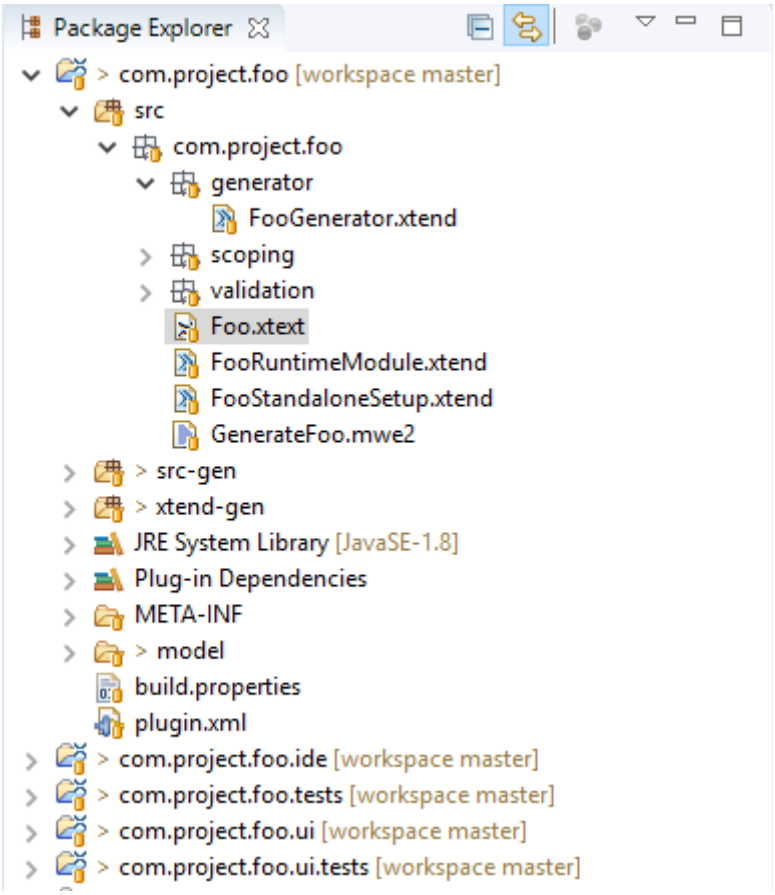


Fig 3 : Hiérarchie du projet

Le fichier *Foo.xtext* contient la grammaire qui sera utilisée pour la modélisation. (Voir partie « Création de la grammaire »)

Les fichiers présents dans les packages generator, scoping et validation sont à modifier après avoir créé la grammaire. Ils permettent respectivement de fixer, les règles de génération de code, les règles de scope, et les règles de validation.

La génération de code ne peut se faire qu'une fois un ensemble de règles fixées, c'est à dire que lorsque dans le modèle apparaît « A » le générateur doit savoir quel partie de code créer.

Le scope, permet de limiter ou non la portée d'une information dans le modèle. Par défaut, la portée d'un élément est celle de son conteneur : C'est à dire que si dans l'objet A se trouve un élément B alors il ne sera pas possible d'utiliser le même élément B dans le conteneur A'

Dans ce projet, j'ai modifié le scope afin que, lors de l'importation d'un composant d'un autre fichier on puisse utiliser seulement le nom du composant lors de la déclaration d'une instance, au lieu d'utiliser le nom complet. Le nom complet d'un composant est le nom du paquet dans lequel il se trouve, concaténé à son propre nom. La modification du scope est donc seulement faite pour rendre les modèles plus lisibles et moins lourds à écrire.

Avec Xtext, si on ne modifie que le fichier contenu dans le package scoping, la visibilité d'un élément s'arrête au fichier. Lorsqu'il y a plusieurs fichiers, il est nécessaire de faire des imports. On peut donc voir que certaines règles de la grammaire seront liées à la façon dont Xtext gère la visibilité des éléments.

Le package validation contient un fichier appelé *Validator. C*'est dans ce fichier que peuvent être rajoutées des règles en complément de celles de la grammaire. (Voir partie « Règle de vérification et modélisation »)

Création de la grammaire

La création de la grammaire est la première étape permettant de créer un langage de modélisation.

Lors de la création de la grammaire, il est nécessaire d'avoir à l'esprit l'objectif de la modélisation afin de créer un ensemble de règles, le plus petit possible, pour en simplifier la compréhension par l'utilisateur. Cependant, si la grammaire est trop simplifiée, c'est la génération de code qui deviendra bien plus complexe. Il est donc nécessaire de trouver un certain équilibre lors de la création du langage. De plus, il est souvent nécessaire de modifier la grammaire en cours de développement, elle doit donc être relativement expressive, afin de pouvoir être modifiable facilement.

Toutes les règles de la grammaire ont une structure similaire : Elles sont composées d'un nom et de plusieurs éléments. On distingue deux types de règles : Les règles terminales et les non-terminales que l'on notera respectivement RT et RNT.

Une règle est dite non-terminale si certains éléments qui la constitue sont d'autres règles.

Une règle est dite terminale si on utilise seulement des éléments terminaux, comme des chaînes de caractères.

Pour faciliter la distinction entre ces deux types de règles, Xtext affiche le nom des RNT en bleu et des RT en noir.(Voir fig. 4)

Une règle peut être composée de plusieurs éléments de même type, c'est à dire issus de la même règle. On peut utiliser « * » si l'objet peut être présent aucune fois, une fois ou plusieurs fois, ou « + » si l'élément doit être présent au moins une fois.

Foo.xtext

```
1  grammar com.project.foo.Foo with org.eclipse.xtext.common.Terminals
2
3  generate foo "http://www.project.com/foo/Foo"
4
5  //Concerne les modeles
6  DomainModel:
7      model=Model;
8
9  /*
10  * Structure d'un modele, correspond
11  * à un paquet pouvant contenir plusieurs
12  * composants et assemblages
13  */
14 Model:
15     'package' name=QualifiedName '{'
16     (imports+=Import)*
17     (components+=Component)*
18     (assembly+=Assembly)*
19     '}';
20
21
22 QualifiedName:
23     ID ('.' ID)*;
24
25 Import:
26     'import' importedNamespace=QualifiedNameWithWildcard;
27
28 QualifiedNameWithWildcard:
29     QualifiedName '.*'?;
30
```

Fig 4 : Morceau de grammaire

Dans la capture d'écran ci-dessus on peut voir une partie de la grammaire créée. J'ai fais le choix d'autoriser un seul modèle par fichier, ce qui est visible avec la règle DomainModel, qui est la racine d'un fichier. La règle appelé Model correspond donc à un unique fichier et par conséquent n'est pas obligatoirement un modèle complet.En effet, grâce aux imports il est possible de séparer plusieurs composants, ce qui améliore la lisibilité du modèle. Cependant, pour tout modèle il existe un équivalent que l'on peut écrire sur un unique fichier.

(Voir annexe A, pour la grammaire)

Un assemblage étant composé de plusieurs composants j'ai fais le choix de déclarer l'ensemble des composants avant les assemblages, par soucis de lisibilité.

Chaque ensemble de composant et d'assemblage présent dans un fichier est contenu dans un paquet, si deux fichiers appartiennent au même paquet il n'est pas utile de faire d'import. En effet, la séparation en plusieurs fichiers est une fois encore utile pour simplifier la lecture et hiérarchiser les informations, mais l'ensemble du contenu de fichiers présent dans un même paquet pourrait être contenu dans un unique fichier.

Ces premières règles permettent donc d'avoir la structure générale du modèle, c'est à dire que chaque fichier utilisé pour la modélisation suivra cette structure. C'est à dire : nom du paquet auquel le fichier appartient, puis des imports, suivi des composants et des assemblages s'il y en a.

Une fois ces règles écrites, il est important de savoir ce qui constitue un import, un composant ou encore un assemblage.

Comme dit précédemment, un import permet d'avoir accès à un composant présent dans un autre fichier. On peut donc en déduire la règle Import.

Un composant est constitué d'une liste de services qu'il fournit et d'une liste de services requis, cette dernière pouvant être vide. Un service correspond donc à une méthode dont on doit donner la signature. Il est donc nécessaire de faire le distinguo entre les services requis et des services fournis, ce qui signifie qu'il y a besoin de créer deux nouvelles règles qui seront des éléments de la règle Component.

Un assemblage est constitué d'un ensemble de composant, pour qu'un assemblage soit correct il faut que chaque service requis par un des composants soit lié à un service fourni par un autre composant. La règle Assembly est constitué de plusieurs éléments. On list d'abord les composants qui forment l'assemblage, puis on donne l'ensemble des liens entre services requis et services fournis.

Cependant, la grammaire toute seule ne suffit pas à créer un langage cohérent et non ambigü Il est donc nécessaire de rajouter un ensemble de règles de validation permettant de compléter cette grammaire. Une règle de validation permet, par exemple, de faire en sorte que l'utilisateur respecte certaines conventions, comme le fait de commencer le nom d'un composant ou d'un assemblage par une majuscule, cela permet de tout de suite d'identifier le type de l'objet lors de la lecture du modèle.

Règles de vérification et modélisation

La création de règles de validation a été la partie la plus complexe . C'est seulement en modélisant et en affinant la modélisation que l'on se rend compte que d'autres vérifications sont nécessaires. Ces dernières sont parfois associées aux premières règles de validations créées.

Les méthodes de vérifications, ainsi que les test de ces méthodes, sont écrits en Java. Cependant, un utilisateur, doit simplement utiliser les règles qui lui sont proposées par le créateur du langage pour modéliser l'ensemble des assemblages sur lesquels il travaille. Il n'y a donc pas de connaissances préalables requises en Java, ni même besoin d'avoir l'habitude de Xtext, ce qui est l'un des principaux intérêts de ce plug-in.

Les règles de vérification sont exécutées lors de la modélisation, ce qui permet de constater les erreurs directement.

Vérification de la validité d'un assemblage

Lorsqu'on souhaite créer un assemblage il est nécessaire que chaque service requis par un composant soit satisfait. (Voir «Création de la grammaire»)

Il est donc impératif de vérifier que l'ensemble des services requis par les composants constituant l'assemblage soit satisfait. Pour simplifier la lecture, j'ai créé plusieurs règles permettant cette vérification. Dans un premier temps, il faut vérifier que chaque liaison est valide : Pour cela on compare un à un les paramètres du service fourni et du service requis, présents dans la liaison. Une erreur sera générée si le type de retour des méthodes n'est pas le même, si le nombre de paramètre n'est pas identique, ou bien s'il existe au moins un paramètre ayant un type différent. Cette méthode s'appelera *checkBindingIsValid.*

La vérification des liaisons une à une ne suffit pas pour assurer qu'un assemblage est correct. En effet, on pourrait avoir un cas où toutes les liaisons présentes sont correctes, mais un service requis d'un composant n'est pasfourni. Il faut donc aussi vérifier que pour chaque instance des composants présents dans l'assemblage, tous les services requis sont liés. Une autre méthode doit être rajoutée afin d'effectuer cette vérification que l'on appellera *checkAssemblyIsCorrect*. Cette méthode de vérification prend en entrée un assemblage et vérifie que chaque service requis de chaque composant constituant l'assemblage, est correctement lié.

Vérification de l'unicité des noms

Il est important d'éviter toute ambiguïté pour que la génération de code fonctionne correctement. Pour cela, il est nécessaire de ne pas avoir deux éléments possédant le même nom. De plus, la lisibilité du modèle s'en trouve simplifiée. J'ai donc créé un ensemble de méthodes permettant de vérifier l'unicité des noms des composants et de leurs attributs, des assemblages ainsi que des services. Si le type des éléments est différent, ils peuvent avoir le même nom car le type sera connu lors de la génération.

Pour vérifier l'unicité, la méthode la plus simple est l'utilisation d'une fonction proposée par Xtext permettant d'obtenir la liste de tous les éléments d'un fichier pour un type donné. Il suffit ensuite de simplement vérifier qu'un élément n'est pas présent dans la liste pour savoir si le nom est unique ou non. Cette méthode est néanmoins coûteuse : De manière brute cela implique que pour chaque ajout d'un élément, il y a obligation de le comparer à tous les éléments du même types déjà présents dans le modèle. Stocker les noms lors de leur ajout dans une table de hachage aurait été une meilleure solution, cependant je n'ai pas trouvé de méthodes fonctionnant de cette façon.

Validation de la vérification

Il est, en plus, important de fournir des jeux de test, afin de prouver que les règles de vérification fonctionnent. Cependant il n'est pas possible de faire un ensemble de tests exhaustifs. On cherche donc à tester des cas limites du modèle, afin de s'assurer du bon fonctionnement de ces règles lorsqu'il y a des erreurs dans la modélisation. J'ai donc créé une méthode de test pour chaque règle de validation créée. Chaque méthode doit être nécessairement testée indépendamment, afin d'en assurer le bon fonctionnement : Il s'agit là des briques de base du programmes. Il faut donc chercher à tester les choses les plus élémentaires, pour que l'ensemble de la modélisation fonctionne.

Conclusion sur la création des outils

Une fois les différents outils permettant de modéliser des assemblages et leurs composants implémentés, on peut commencer à s'intéresser aux programmes que l'on cherche à générer de manière automatique. La modélisation ne peut être faite qu'après cette étude.

Java LeJOS

Dans cette partie, la programmation des robots lego-mindstorms-ev3 est le principal interêt. Comme indiqué précédemment, les programmes que les robots utilisent peuvent être codés en java, ou bien grâce au logiciel MINDSTORM. Ces robots constitués de différentes pièces, tels que des capteurs ou des moteurs, on peut voir le lien avec les composants. Il s'agit dans le cas du robot de composant matériel : Lors de la modélisation, on s'intéressera à des composants logiciels.

Des exemples au modèles

Dans un premier temps j'ai commencé par analyser de nombreux exemples de code permettant de faire fonctionner des composants du robot, tels que les moteurs ou un capteur de proximité.

Les exemples commencent toujours de la même manière : Les différents composants nécessaires pour que le robot effectue le programme, sont déclarés. Par exemple, dans le cas où l'on veut faire déplacer le robot, il est impératif d'initialiser les moteurs et de leur donner une vitesse.

Le principal défaut des différents exemples est que la totalité du code se trouve dans la fonction main d'une classe Java. Il est donc nécessaire d'extraire ce qui est lié à l'architecture logicielle de ce code, par exemple la déclaration d'un moteur. (voir fig 5)

```
// create two motor objects to control the motors.
UnregulatedMotor motorA = new UnregulatedMotor(MotorPort.A);
UnregulatedMotor motorB = new UnregulatedMotor(MotorPort.B);

// set motors to different power levels. Adjust to get a circle.
motorA.setPower(70);
motorB.setPower(30);
```

Fig 5 : Déclaration des moteurs et de leur puissance [6]

Dans l'extrait de code ci-dessus, il apparaît que deux moteurs sont déclarés ainsi que l'utilisation de leur port. Il est donc impératif de laisser à l'utilisateur le choix du port qu'il utilisera lors de la modélisation.

La méthode setPower(.) permet d'ajuster la puissance d'un moteur, c'est donc un service fourni par le composant moteur.

Avec la grammaire précédemment créée, le contenu des services était inexistant, et un service était seulement défini par une signature. Pour pouvoir modéliser les différents services, il est donc impératif de rajouter la possibilité d'avoir des boucles qui permettent plusieurs appels à un autre service.

Retour sur la grammaire

Suite à l'étude de différents exemples pour Java LeJOS, la grammaire que j'ai créée au début du stage s'est avérée incomplète. En effet, il est possible qu'un composant soit constitué d'un assemblage ou de plusieurs assemblages. De plus, dans la modélisation, on ne peut pas toujours faire un lien direct entre les composants matériels et les composants logiciels.

J'ai donc rajouté dans la grammaire précédente, des éléments dans la règle Component. Après la déclaration des différents services requis et proposés, on peut déclarer un ou plusieurs assemblages appartenant au composant.

Une règle permettant de donner la liste des composants et des liaisons pouvant être raffinés a aussi été ajoutée.

Les services fournis ne doivent plus être une simple signature, l'utilisateur du modèle doit pouvoir donner un ensemble d'instructions simples et pouvoir utiliser des conditionnelles et des boucles while. Les paramètres donnés à ces services seront des types simples ou primitifs, tels que des entiers naturels (int) ou des flottants. J'ai donc ajouté un ensemble de règles permettant de répondre à ces besoins, la principale difficulté se trouve dans l'écriture de la règle If, dans le cas où l'on veut avoir des conditionnelles imbriquées, l'utilisation de else peut devenir ambiguë et le langage n'est pas capable de savoir à quel if un else est lié.

Par exemple si l'on a «if A then if B then C else D », cela peut être interprété comme «if A then (if B then C else D) », c'est à dire que le else appartient à la conditionnelle « if B », mais il peut aussi être interprété comme «if A then (if B then C) else D », dans ce cas le else est lié à la conditionnelle « if A ». [2] Afin de lever cette ambiguïté Xtext fourni un mot clé « => », qui permet de forcer l'analyseur syntaxique à lier le else au if le plus proche. Ce qui nous donne la règle suivante :

```
If :
    'if' '(' (conditions+=Condition('||'|'&&'))*conditions+=Condition ')' '{'
        expressions+=Expression
    '}'(=>'else'(else=If | '{'
        (expression=Expression)
    '}'))?;
```

Fig 6 : Syntaxe du if

L'ensemble des règles utilisé pour le projet prenant assez de place, j'ai utilisé l'outil de génération de graphe proposé par Xtext afin d'obtenir une version plus lisible et intuitive de la grammaire. (Voir annexes)

2 Exemple extrait de https://dslmeinte.wordpress.com/2011/12/05/using-syntactic-predicates-in-xtext-part-1/

Conclusion

Durant le déroulement de mon stage j'ai eu l'opportunité de travailler sur les différentes étapes de la génération de code. Le travail réalisé a été enrichissant pour mon expérience professionnelle d'un point de vue technique, notamment grâce à la découverte des bases de la programmation par composant ainsi qu'à la création et à l'utilisation de modèles.

Durant la première partie du stage, l'utilisation de Xtext m'a permis de mieux comprendre comment la création d'un langage spécifique à un domaine peut apporter des outils faciles d'utilisation pour la modélisation de cas concrets.

Pendant la seconde partie du stage, j'ai pu prendre connaissance de java LeJOS et des outils pour manipuler un robot ev3, construit avec des legos. Cependant, par manque de temps je n'ai pas pu créer un modèle générant un programme permettant de faire bouger le robot.

Glossaire

AeLoS : Architectures et Logiciels Sûrs

IRCCyN : Institut de Recherche en Communications et Cybernétique de Nantes

LS2N : Laboratoire des Sciences du Numérique de Nantes

RNT : règle non terminale

RT : règle terminale

Bibliographie et sitographie

[1]site du LS2N [En ligne]. [consulté le 15 mai 2018]. Disponible sur : https://www.ls2n.fr/

[2]A.R.d. Silva, Model-driven engineering: A survey supported by the unified conceptual modelComput Lang Syst Struct, 43 (Supplement C) (2015), pp. 139-155

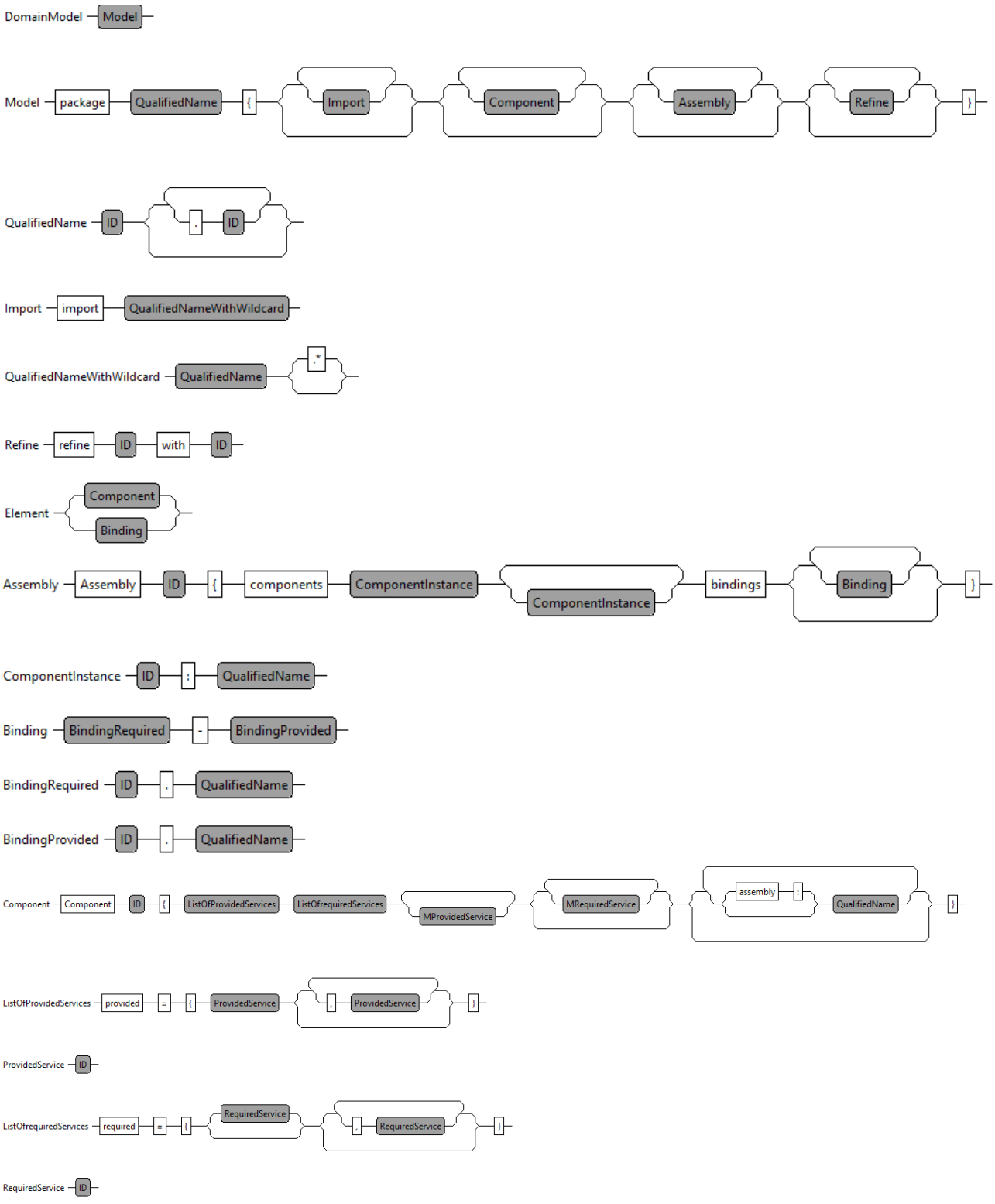
[3]Wikipédia, page sur les langages dédiés [consulté le 31 mai 2018]. Disponible sur : https://fr.wikipedia.org/wiki/Langage_dédié

[4]Site officiel d'eclipse [En ligne]. [consulté le 16 mai 2018]. Disponible sur : https://www.eclipse.org/

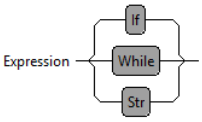
[5]Site officiel de Xtext [En ligne]. [consulté le 18 mai 2018]. Disponible sur : https://www.eclipse.org/Xtext/

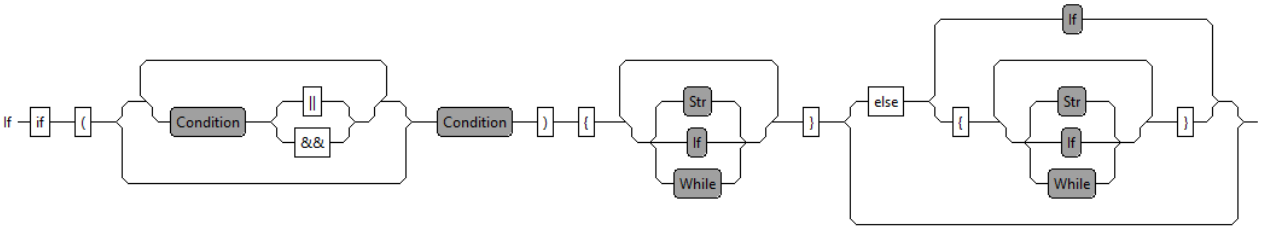
[6]STEMRobotics [En ligne]. [consulté le 29 mai 2018]. Disponible sur : http://stemrobotics.cs.pdx.edu/node/4766

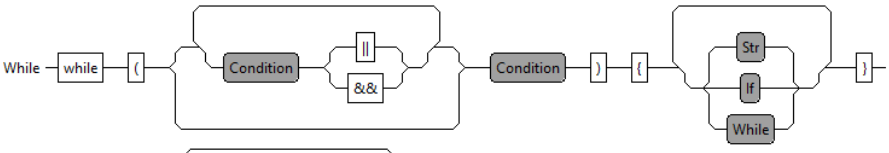
ANNEXE A

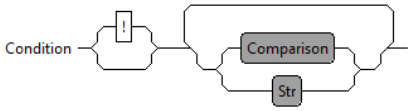

DomainModel
Model
Model
package
QualifiedName
{
Import
Component
Assembly
Refine
}
QualifiedName
ID
.
ID
Import
import
QualifiedNameWithWildcard
QualifiedNameWithWildcard
QualifiedName
.*
Refine
refine
ID
with
ID
Element
Component
Binding
Assembly
Assembly
ID
{
components
ComponentInstance
ComponentInstance
bindings
Binding
}
ComponentInstance
ID
:
QualifiedName
Binding
BindingRequired
-
BindingProvided
BindingRequired
ID
.
QualifiedName
BindingProvided
ID
.
QualifiedName
Component
Component
ID
{
ListOfProvidedServices
ListOfrequiredServices
MProvidedService
MRequiredService
assembly
:
QualifiedName
}
ListOfProvidedServices
provided
=
{
ProvidedService
,
ProvidedService
}
ProvidedService
ID
ListOfrequiredServices
required
=
{
RequiredService
,
RequiredService
}
RequiredService
ID

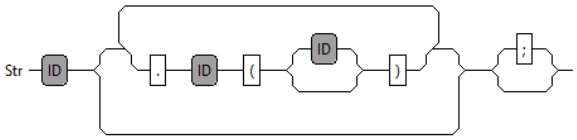

MProvidedService
service
provided
PSignature
{
Expression
}

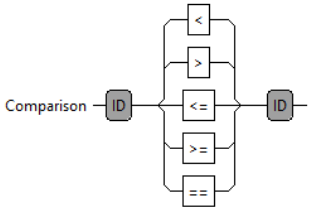

Expression
If
While
Str

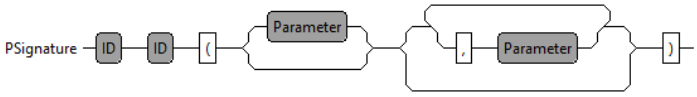

If
if
(
Condition
||
&&
Condition
)
{
Str
If
While
}
else
If
{
Str
If
While
}


While
while
(
Condition
||
&&
Condition
)
{
Str
If
While
}

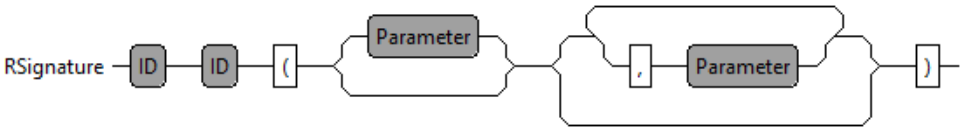

Condition
!
Comparison
Str

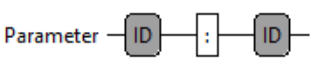

Str
ID
.
ID
(
ID
)
;


Comparison
ID
<
>
<=
>=
==
ID


PSignature
ID
ID
(
Parameter
,
Parameter
)


MRequiredService
service
required
RSignature
;


RSignature
ID
ID
(
Parameter
,
Parameter
)


Parameter
ID
:
ID