Tuteur : Isabelle FERRANÉ
Encadrant : Sébastien RICHE

Développement, amélioration et intégration d'outils de génération de code

pour une plate-forme de prototypage de fonctions de contrôle moteur

Mathieu SOUM

Stage du 3 novembre 2014 au 31 juillet 2015
Chez Aboard Engineering

Master 2 – Développement Logiciel / 2014 - 2015

Tuteur : Isabelle FERRANÉ
Encadrant : Sébastien RICHE

Développement, amélioration et intégration d'outils de génération de code

pour une plate-forme de prototypage de fonctions de contrôle moteur

Mathieu SOUM

Stage du 3 novembre 2014 au 31 juillet 2015
Chez Aboard Engineering

Master 2 – Développement Logiciel / 2014 - 2015

Je souhaite remercier ici l'équipe pédagogique de la formation pour nous proposer une telle opportunité. Contrairement aux autres stages effectués lors de notre cursus, celui-ci s'étale sur toute une année et nous permet, non seulement de travailler sur un projet de longue haleine, mais également de montrer notre adaptation à une équipe sur le long terme.

Je tiens aussi à remercier Paul Martin pour m'avoir permis de réaliser ce stage chez Aboard Engineering ainsi que Vincent Huc et Sébastien Riche pour m'avoir épaulé tout au long de cette expérience. Ils ont su me faire une place et m'ont permis de me sentir utile et considéré et pour cela je leur en suis reconnaissant.
J'aimerais également remercier toute l'équipe d'Aboard Engineering pour l'accueil qu'ils m'ont accordé et leur bonne humeur quotidienne qui fait du bien et qui rend encore plus agréable l'environnement de travail.

Enfin, je souhaite remercier aussi ceux qui m'ont accompagné durant ce stage, certes plus moralement que techniquement, je veux parler de Clément, Ophélie, (petit) Clément, Antoine et d'autres que j'oublie sûrement. Ces gens de l'ombre sans qui les journées n'auraient pas le même parfum.

Sommaire

Chapitre 1

Introduction

Contexte du stage

Ce stage s'inscrit dans le cadre du Master 2 Développement Logiciel à l'Unversité Paul Sabatier de Toulouse. Cette formation a la particularité d'offrir un stage en alternance dès le début de l'année universitaire. C'est une très bonne expérience qui nous tient en haleine tout au long de notre dernière année d'étude et nous permet – pour la première fois dans notre cursus – de réaliser des projets d'envergure en entreprise et de pouvoir nous y impliquer pleinement sur une durée plus longue que lors des expériences de stage précédentes.

Ce stage s'est donc déroulé en alternance au rythme de 3 jours par semaine en entreprise (Lundi, Mardi, Mercredi) et 2 jours à l'université (Jeudi, Vendredi) sur une durée totale de 9 mois (3 Novembre 2014 – 31 Juillet 2015).

Structure du rapport

Ce rapport va rendre compte de mon expérience de stage. Il sera découpé selon 3 parties principales :

Le contexte dans lequel j'ai évolué durant les 9 mois de stage. Dans ce chapitre, je ferai une présentation de l'entreprise qui m'a accueillie ainsi que l'environnement de travail dans lequel j'ai évolué.

Le sujet sur lequel j'ai travaillé. Je ferai une description en détail de la problématique relative à mon stage et du travail à réaliser.

Le travail réalisé durant ce stage. Ce chapitre abordera les détails techniques de mon stage. J'y développerai les différentes tâches que j'ai réalisées et en quoi elles aboutissent à une solution viable à la problématique.

Conseils de lecture

Ce document est à destination de mon maître de stage, de mon encadrant universitaire ainsi que du jury de ma soutenance qui l'aura à disposition. Pour faciliter sa lecture, voici quelques précisions quant aux détails typographiques utilisés dans ce document.

Les citations relatives à du code source telles que des types de donnée ou des éléments du système de fichier seront écrit avec une `police à chasse fixe`.

Un glossaire à la fin du document répertorie certains termes techniques et acronymes qui nécessiteraient une définition plus précise. Les mots apparaissant dans ce glossaire seront écris *italique* lors de leur première apparition dans ce document.

Chapitre 4

Travail réalisé

J'ai travaillé sur la plateforme et sur QGen en parallèle. En effet, lorsque AdaCore livrait une nouvelle version de QGen, nous effectuions les différents tests de performances que nous pouvions faire et nous leur retournions nos résultats. Cependant, entre deux livraisons, nous ne pouvions pas faire de tests supplémentaires. Nous profitions alors de ces intevalles pour se concentrer plus sur l'amélioration de la plateforme en attendant des nouveautés dans QGen.

4.1 La génération de code embarqué

La première étape était d'analyser le code généré par RTW-EC® pour avoir une base de comparaison sachant que ce code là est actuellement dans les applications déployés sur calculateur. Pour cela, j'ai installé un analyseur statique de code appelé OCLint. C'est un outil open source d'analyse statique de code C/C++ dont les développeurs ne fournissent de binaire que pour Linux 64 bits. C'est pourquoi j'ai installé une distribution Fedora sur une machine virtuelle. J'ai donc essentiellement travaillé sous Linux pour cette partie.

Une seconde étape a été de tester une application complète avec l'intégralité du code généré par RTW-EC® remplacé par le code généré par QGen. Pour cela, j'ai utilisé une application complète existante et ai remplacé le code généré. Cette étape a été bien plus fastidieuse que la première. En effet, la plateforme Orianne se base sur le code généré par RTW-EC® pour créer la « glue » entre les composants. Cette glue permet de lier les composants entre eux mais aussi de les faire communiquer avec les API internes d'un calculateur. Dans ce sens, il m'a fallu adapter le code généré par QGen de sorte que ses interfaces (points d'entrées, connexion avec les API de l'ECU, etc.) soient compatibles avec une application basée sur du code RTW-EC®. Pour cela j'ai modifié à la main un composant pour voir les modifications nécessaires et ensuite, j'ai construit une série de scripts python pour automatiser le processus à tous les composants et ne pas avoir à faire toutes ces modifications à la main à chaque nouvelle génération. Ces scripts nous font gagner du temps mais aussi nous donnent des détails sur les interfaces de QGen et leurs différences vis à vis de celles de RTW. Ces détails seront utiles pour la future intégration de QGen à la plateforme Orianne.

Une fois que l'application complète a été générée avec du code QGen, je l'a déployait dans un calculateur sur une table de tests. Je faisait ensuite des mesures comparatives entre l'applicaition QGen et l'application RTW-EC® vis à vis du temps d'execution de certaines tâches, de la taille de mémoire utilisé et de l'espace de stockage pris par le binaire de l'application.

Les mesures des quantités de mémoire et d'espace de stockage utilisés sont faites directement par le compilateur. Il produit un fichier A2L qui s'appuit sur le code assembleur compilé pour définir les tailles et addresses que les différents composants du code (fonctions, variables globales et locales, etc.) auront directement dans la mémoire du calculateur. En revanche, la mesure du temps d'execution est moins triviale. Le RTOS du calculateur nous permet de définir des tâches à executer à une certaine récurrence. J'ai choisit de façon arbitraire une tâche avec 10ms de récurrence. Ensuite, l'ECU possède des sorties digitales et analogiques que nous pouvons manipuler via les API et pilotes de cette ECU. J'ai alors utilisé une sortie digitale qu'aucun autre composant n'utilisait. Une sortie digitale ne peut prendre que deux valeurs (ou états) : 0 ou 1. En mettant cette sortie à 1 au début de l'execution d'une tâche précise et à 0 à la fin de cette même tâche, nous auront un 1 sur cette sortie durant toute l'execution de la tâche. Si maintenant nous visualisons le signal de cette sortie sur un oscilloscope,

on peut observer un signal carré représentant les oscillations de la sortie à chaque début et fin de la tâche. Enfin, en mesurant le temps que la sortie reste à 1, on obtient le temps d'execution de la tâche.

4.1.1 Les premiers résultats

La première analyse a montré que les deux codes générés étaient équivalents en terme d'analyse statique. Nous n'avons pas tenu rigueur des critères de bonne pratique d'écriture de code C. Étant du code généré, nous n'irons pas le modifier à la main – où en de très rare cas, ce n'est pas l'objectif de la génération de code. Le code RTW-EC® a une plus grande complexité cyclomatique [1] mais cela est du au générateur qui ne créer qu'une seule fonction de calcul alors que QGen créer plusieurs fichiers sources par modules et découpe ses traitements.

La où les deux générateurs se sont démarqués, c'est au moment des tests d'une application complète. Premièrement, nous avons analysé l'empreinte mémoire d'une fonction en particulier. Dans le cas de l'application QGen, l'empreinte mémoire était deux fois supérieure à celle de l'application RTW-EC®. Ce résultat s'est confirmé lors du passage sur table de test. Le temps d'exécution était lui aussi deux fois supérieur pour l'application QGen. Nous passions de $400\mu s$ d'exécution pour RTW-EC® à $800\mu s$ pour QGen. Des temps de calculs qui semblent négligeables au premier abord, mais une telle augmentation suffisait à mettre en défaut le système temps réel qui n'arrivait plus à lancer toutes les tâches à leur correcte récurrence et qui n'exécutait même plus certaines fonctions. Un problème qu'il faudra régler pour que le code généré soit viable.

4.1.2 Les étapes d'amélioration

Après analyse du code généré, nous avons remarqué que les fonctions d'interpolation [2] et d'extrapolation [3] lors de recherche de correspondance dans les tables de calibration n'était pas les même entre RTW-EC® et QGen. Nous avons identifié un gain de performance potentiel à ce niveau là.

Après discussion avec AdaCore, ils ont revus ces fonctions ainsi que les options d'optimisation de leur générateur afin de, par exemple, réduire le nombre de variables intermédiaires entre la traduction des différents blocs Simulink® et utiliser la même variable en sortie d'un bloc et en entrée d'un autre si ces deux blocs sont reliés de manière simple et directe. Ces améliorations ont à la fois réduit le temps de calcul des méthodes (essentiellement avec la modification des fonctions de recherche et d'extrapolation dans les tables de calibration) mais aussi réduit l'impact du code en mémoire.

Après ces modifications, nous retombions sur des empreintes mémoires équivalentes entre les applications RTW-EC® et QGen. Cependant, il restait encore un écart d'environ $200\mu s$ au niveau du temps d'exécution.

4.1.3 La première version majeure

Sur la dernière partie des tests que nous avons effectués, nous nous concentrions sur une seule fonction, la plus significative en terme de performance dans notre application. Afin d'aider AdaCore dans leur recherche d'améliorations, nous leurs avons envoyé notre code généré par RTW-EC® ainsi que par QGen avec les modèles Simulink® nécessaires à cette génération. Ils pouvaient alors mieux identifier les morceaux de code moins performants dans le code généré par QGen par rapport au code généré par Matlab et ce, dans plusieurs configurations (compilateur, matériel). Au fur et à mesure de leurs recherches, ils nous faisaient parvenir une nouvelle version du code généré pour notre composant afin de l'intégrer rapidement dans notre application et de tester les dernières évolutions. Au terme de plusieurs cycles, nous sommes parvenus à une équivalence entre le temps d'exécution des deux applications. Une grande avancée qui aboutira à la livraison d'une première version majeure courant juin 2015. Une fois cette livraison effectuée, nous pourrons reprendre une application complète pour voir la viabilité de ce nouveau générateur dans un contexte concret, d'abord sur table de test et ensuite, si les résultats sont concluants, sur véhicule.

1. La complexité cyclomatique (ou nombre cyclomatique) représente le nombre de « chemins » possibles durant l'execution d'une fonction en tenant compte des conditions, boucles et sauts et de leurs niveaux d'imbrications.

2. L'interpolation consiste à construire la fonction d'une courbe à partir d'un nombre fini de ses valeurs. En se basant sur les données d'une table, on peut calculer ainsi les valeurs correspondantes à des paramètres compris entre le minimum et le maximum de ces points connus.

3. L'extrapolation consiste à calculer des valeurs en dehors de l'intervalle des paramètres initiaux. Plusieurs règles peuvent s'appliquer comme par exemple l'extrapolation linéaire ou l'extrapolation majorée/minorée.

Les derniers résultats obtenus sont très concluants. AdaCore a réussi à faire un générateur de code C temps réel à partir de modèles Simulink® aussi performant que celui embarqué dans les outils Matlab®.

4.2 La plateforme Orianne

Pour la partie du travail sur la plateforme Orianne, j'ai dans un premier temps repris du code existant. Il a fallu me familiariser avec la plateforme et son architecture. La première étape a donc été d'analyser ce code pour identifier les interfaces sur lesquelles se connecter pour développer de nouveaux outils.

4.2.1 L'existant //TODO WIP Correction

La plateforme était construite sur une architecture de composants *OSGi* où chaque module de génération était indépendant, connecté à un composant de référence qui définissait l'IHM. Je ne trouvais pas pertinent ce choix d'architecture dans la mesure où l'application ne pouvait se lancer que si tous les composants étaient présent. L'aspect dynamique qu'apporte la plateforme OSGi n'était pas du tout exploité.

Après discussion avec mes collègues, nous avons constaté que l'utilisation la plateforme OSGi n'avait pas d'intérêt suffisant et ajoutait de la complexité dans le développement. De plus, le découpage des composants n'était pas très optimisé, introduisant de la redondance dans le code ce qui freine le développement et la maintenance. Nous avons donc décidé de migrer ces composants vers une unique application Java gérée avec *Maven* afin de mieux identifier les améliorations possibles et faciliter la factorisation du code redondé.

On peut citer par exemple la configuration relative à un projet édité par la plateforme qui était passée de composant en composant en fonction des actions demandées par l'utilisateur. Ce qui était réellement passé en paramètre n'était pas des objets sérialisés représentant les données de configuration mais un chemin vers le fichier principal de configuration du projet. Ce fichier XML était relu par chaque composant qui traduisait alors les données en objets Java. Or l'ajout d'un nouvel outil entraîne – avec une forte probabilité – le changement de ce fichier de configuration. Un changement même mineur devait être répercuté sur chaque outil afin de garder une cohérence dans la gestion de la configuration des projets dans tous les outils. Ceci représente une surcharge de travail qui en plus induit une perte au niveau des performances par des accès disque inutile et des traitements réalisés plusieurs fois sans raison valable.

Ensuite, toujours dans une optique de factorisation du code afin d'améliorer la maintenabilité de l'application et éliminer la duplication de code, j'ai commencé à regrouper les classes d'interface graphiques basées sur la bibliothèque Swing. J'ai trouvé plusieurs portions de code commune qui pourraient être abstraites pour faciliter le développement de nouvelles vues pour de futurs outils. De telles classes graphiques abstraites aident à garder une cohérence dans les différentes vues sans avoir à s'en soucier à chaque nouveau développement.

Heureusement, les IDE modernes sont des outils très puissants qui peuvent nous aider dans les tâches de factorisation et d'optimisation de code. C'est pourquoi j'ai pris la liberté d'installer sur ma machine un outil comme IntelliJ avec lequel je suis familier et qui me permet de factoriser du code de façon sûre et efficace.

Nous allons maintenant nous concentrer sur le premier outil que j'ai eu à intégrer dans cette plateforme. Il s'agit de l'outil de génération du code relatif à la communication de l'application sur un bus CAN.

4.2.2 La génération CAN

L'ECU n'est pas le seul organe communiquant au sein d'un véhicule. Elle peut communiquer avec d'autres composants électroniques comme les commandes (pédales), les injecteurs, la boîte de vitesse, le contrôleur hybride, etc. Cette communication se fait généralement via un bus CAN. Ce module de communication n'est pas directement une fonction moteur et est commun à tous les modules de l'application dans la mesure où ils peuvent tous communiquer via ce bus. Il n'a donc pas de modèle Simulink® associé ni de code généré. C'est pourquoi la plateforme a besoin d'un générateur CAN pour gérer cette communication et lier certaines entrées/sorties de modules à des signaux émis ou transmis sur le bus.

Là encore, j'ai repris un générateur existant qui produisait du code C pas suffisamment performant d'un point de vue utilisation mémoire pour le calculateur cible. Il a donc fallu reprendre les modèles de code existants – aussi appelés « templates » – et de les modifier pour générer du code moins volumineux, identifier les données et variables superflues, adapter les structures de données pour être plus compactes. La modification des modèles de code a forcément entrainé une modification du générateur de code. Je l'ai repris quasiment dans son intégralité, là aussi pour des raisons de factorisation, de duplication de code et de complexité non optimisée.

Les objectifs étaient donc de réduire la consommation mémoire du code C généré et de gagner en modularité et en clarté dans le code du générateur. Une fois ces deux points éclaircis, il faudra intégrer le générateur à la plateforme.

La configuration CAN est basée sur des fichiers XML de description des différents messages et signaux envoyés et reçus sur le bus avec les variables de l'application associées à ces signaux. Afin d'optimiser les accès à ces données, j'ai mis en place un système de cache qui garde en mémoire les données extraites des fichiers de configuration et facilite l'accès via des sous-ensemble pour récupérer facilement et rapidement uniquement les messages envoyés sans les messages reçus par exemple. Ainsi, les algorithmes de génération ont gagné en clarté et les futurs ajouts au générateur nécessiteront moins de travail.

Du côté des templates, je souhaitais modifier uniquement la gestion des données sans toucher aux algorithmes de traitement des trames qui sont des algorithmes complexes qui ont été testés et validés. De plus les modèles se basent déjà sur un systèmes de balises de la forme `\$nom_balise$\ ... \$nom_balise$\`, une séquence qu'on ne retrouve pas dans le langage et donc qui se détache bien du code C effectif dans le template. À l'intérieur de ces balises, on trouve des séquences du type `$variable$`. Les premières représentent des blocs de code regroupés par sémantique ou parce qu'ils seront générés plusieurs fois (déclaration et initialisation de variables par exemple). Les dernières seront remplacées par des valeurs calculées ou issues des fichiers XML. J'ai tenu à garder ce système en ajoutant mes propres balises et variables. Je trouvais le système intéressant et bien conçu. De plus, je n'ai ainsi pas eu à toucher aux algorithmes présents dans les templates et donc éviter des erreurs durant une migration éventuelle.

J'ai ensuite cherché à identifier chaque structure de données à l'intérieur de ces templates pour voir leur pertinence et comment elles intervenaient dans la communication. J'ai alors pu constater par exemple que certaines données était générées car présentes dans le XML de configuration mais inutilisées dans la communication CAN. J'ai aussi identifié un peu de redondance dans les données sur un point précis. Sur un bus CAN, la partie données de la trame peut être formatée selon deux formats : le format Intel ou le format Motorola. Dans le premier cas elles sont traitées comme deux mots de 32 bits et donc stockées dans un tableau à deux cases d'entiers 32 bits. Dans le second cas, elles sont traitées comme un tableau de 8 octets. Cependant, les données restent les mêmes. Pour éviter cette duplication, j'ai donc définie une `union`, c'est une structure de données en C qui permet de représenter des données prenant la même place en mémoire mais qui seront lus de plusieurs manières différentes (dans notre cas un `uint8[8]` et un `uint32[2]` prennent tous les deux 64 bits en mémoire mais fournissent les données sous deux formats différents).

Du côté du générateur en lui-même, j'ai repris le code existant pour l'organiser différemment. Le code était très linéaire et séquentiel ce qui engendrait beaucoup de modification dans le générateur en cas de modification des templates. J'ai donc extrait des fonctions afin d'identifier des similarités de comportement mais aussi d'améliorer la lecture du code. L'algorithme que j'ai mis en place remplace les blocs au fur et à mesure de la lecture du modèle par du code généré. Ainsi l'ajout, la suppression ou la modification d'une balise dans le template n'influe pas sur le traitement des autres balises.

Côté performance, l'objectif était de prendre le moins de place possible dans la mémoire du calculateur. J'ai réussi à diviser par 7 l'espace mémoire utilisé. En contre partie, la configuration des trames – qui sont des données statiques et constantes – est stockée directement dans l'espace mémoire du calculateur avec le code compilé.

La partie génération terminée, la prochaine tâche a été de créer un interface graphique à intégrer dans la plateforme pour permettre une configuration plus agréable de la partie CAN en utilisant des formulaires et des tables pour faire correspondre les variables transmises sur le CAN avec les variables de l'application en entrée et en sortie des composants. J'ai utilisé la bibliothèque Swing – déjà utilisée pour le reste de la plateforme – pour créer les différents composants graphiques lié à cette configuration. J'ai ensuite ajouté ces composant dans un nouvel onglet de la plateforme (cf. figure 4.1).

FIGURE 4.1 – Partie CAN de l'interface graphique de la plateforme

4.2.3 La partie RTE

La génération CAN n'est pas le seul outil manquant à la plateforme. Ma prochaine tâche se tourne vers la partie RTE. Cette partie consiste à définir les liens entre le code interne à l'ECU (pilotes) et notre application. Pour cela, on se base sur les différents types d'entrée/sortie fournis par l'ECU. Les entrées/sorties sont définies dans un fichier de configuration XML propre à chaque ECU. C'est sur ce fichier là que nous allons nous baser pour la configuration RTE de l'application.

Du point de vue utilisateur, la configuration RTE se découpe en 3 parties. La première est la saisie des capteurs et actionneurs, la deuxième est l'affectation des entrées/sorties de l'ECU aux entrées/sorties des composants via les capteurs et actionneurs définis précédemment et la troisième est le lien entre ces entrées/sorties et les variables des modules de l'application. Cela se traduira par 3 vues différentes dans l'application. Ces 3 étapes permettront de configurer le générateur qui génèrera les appels aux API pour interfacer les accès aux variables des capteurs de l'ECU.

Je suis actuellement en cours de conception de l'interface et le développement du générateur se fera ensuite.

Glossaire et acronymes

A2L //TODO

CANalyzer Logiciel permettant de visualiser les trames émises et reçues sur un bus CAN développé par Vector.

ECU **E**ngine **C**ontrol **U**nit. Le boitier de contrôle moteur contenant des pilotes et API pour manipuler les entrées sorties du boitier. L'application de contrôle moteur sera déployée dans ce boitier et utilisera les pilotes de ce matériel pour interagir avec le moteur.

INCA Logiciel de mesure et de calibration d'*ECU* via plusieurs protocoles de communication développé par ETAS. Il permet de modifier les calibrations d'une *ECU* et de vérifier que les changements au niveau des mesures sont corrects.

Maven Maven est un outil pour la gestion de l'automatisation de production des projets logiciels Java. L'objectif recherché est de produire un logiciel à partir de se sources en optimisant les tâches réalisées à cette fin et en garantissant le bon ordre de fabrication.

MBD **M**odel **B**ased **D**esign. Utilisé notamment dans des applications automobiles, aérospatiales ou pour de l'informatique industrielle, MBD est un méthode mathématique et visuelle pour comprendre et résoudre des problèmes complexes de système de contrôle, de traitement du signal ou de système communicant. Elle est beaucoup utilisée pour des logiciels embarqués.

Orianne **O**util numé**RI**que pourme m**A**quettage de fo**N**ctions de co**N**trole mot**E**eur. Plate-forme de prototypage rapide de fonctions de contrôle moteur développée par Aboard Engineering. Voir 2.2 pour plus de détails.

OS temps réel (**R**eal **T**ime **O**perating **S**ystem en anglais). Système d'exploitation multi-tâches destinée aux applications temps réel. Il facilite la création d'un système temps réel via des ordonnances de tâches spécialisés afin de fournir aux développeurs des systèmes temps réel les outils et les primitives nécessaires pour produire un comportement temps réel souhaité dans le système final.

OSGi **O**pen **S**ervices **G**ateway **i**nitiative est une plate-forme de services fondée sur le langage Java qui permet de créer des composants dynamiques fournissant des services. Ces composants peuvent se connecter et se déconnecter « à la volée » à la plate-forme.

Project P Projet open source visant à réduire les coût des approches de développement *MBD* dans le cadre d'applications temps réel embarquées. Le projet comprend de grands groupe industriel comme de plus petites structure qui collaborent dans un but commun. Pour en savoir plus `http://www.open-do.org/projects/p/`

QGen Générateur open source de code embarqué temps réel en C et Ada à partir de modèles Matlab® Simulink®. Basé sur *Project P*, son développement est piloté par la société AdaCore.

RTE **R**eal **T**ime **E**nvironment. Dans la plate-forme *Orianne*, RTE correspond à l'application développée à partir du code généré. En comparaison à l'application gérant le matériel de l'*ECU* (pilotes, API, etc.).

RTW-EC® **R**eal-**T**ime **W**orkshop-**E**mbedded **C**oder®. Aujourd'hui renommé Simulink Coder™. Outil de génération de code C et C++ à partir de diagrammes Simulink®. Le code généré peut-être utilisé pour des applications temps réel et hors temps réel, notamment pour le prototypage rapide.

Table des figures