

Tutrice : Isabelle FERRANÉ
Encadrant : Sébastien RICHE



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

pour une plateforme de prototypage de fonctions de contrôle moteur

Mathieu SOUM

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



Tutrice : Isabelle FERRANÉ
Encadrant : Sébastien RICHE

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

pour une plateforme de prototypage de fonctions de contrôle moteur

Mathieu SOUM

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

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

Sommaire	v
1 Introduction	1
2 Contexte du stage	3
2.1 L'entreprise : Aboard Engineering	3
2.2 La plateforme Orianne	3
2.3 L'environnement de travail	5
3 Présentation du sujet	7
3.1 La génération de code embarqué	7
3.1.1 État des lieux	7
3.1.2 QGen	8
3.2 Les outils de la plateforme	9
3.2.1 La partie IHM	9
3.2.2 La partie génération	9
3.2.3 La problématique	9
4 Travail réalisé	13
4.1 La génération de code embarqué	13
4.1.1 Les premiers résultats	14
4.1.2 Les étapes d'amélioration	14
4.1.3 La première version majeure	14
4.2 La plateforme Orianne	15
4.2.1 L'existant	15
4.2.2 La génération CAN	16
4.2.3 La partie RTE	18
5 Bilan	19
Résumé	21
Summary	23
Table des figures	25

Introduction

Contexte du stage

Ce stage s'inscrit dans le cadre du Master 2 Développement Logiciel à l'Université 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 écrits *italique* lors de leur première apparition dans ce document.

Contexte du stage

2.1 L’entreprise : Aboard Engineering

Aboard Engineering est un bureau d’études en automatique, électronique et informatique industrielle. Composé d’experts en contrôle et instrumentation de systèmes embarqués temps réel, ses ingénieurs développent des solutions pour des applications civiles et militaires. De la R&D à la série, Aboard Engineering travaille dans le domaine des transports (automobile, aéronautique, marin), de l’off-road (agriculture, engins de chantier, ...) et de l’industrie. Les figures 2.1 et 2.2 sont tirées du site web d’Aboard Engineering et récapitulent les domaines et métiers de la société.

Domaines	Connaissances & expérience
Moteurs thermiques (essence , Diesel)	<ul style="list-style-type: none"> Combustion Stratégies de contrôle moteur, incl. dépollution Technologie des capteurs et actionneurs Problématiques de production série (diagnostic, adaptatif...) Mise au point moteur / véhicule
Machines électriques	<ul style="list-style-type: none"> Modélisation et simulation électronique de puissance et machine électrique Commande vectorielle Pilotage de machines électriques Mise au point et calibration
Hybrides	<ul style="list-style-type: none"> Contrôle & supervision d’énergie
Outils	<ul style="list-style-type: none"> Développement d’outils génériques et spécifiques client Instrumentation et essais pour tous les domaines

FIGURE 2.1 – Domaines – Source : Abord Engineering

2.2 La plateforme Orianne

Aboard Engineering fournie différents types de produits et de services. Je ne vais ici développer que le cas de la plateforme *Orianne* sur laquelle j’ai travaillé.

Orianne est une plateforme de prototypage rapide de fonctions de contrôle moteur. Elle permet de regrouper différents modules d’un calculateur moteurs (transmission, ABS, injection, couple moteur, etc.) afin de générer une unique application qui sera déployée dans un calculateur moteur. Les applications générées suivent les spécifications *AUTOSAR*, une architecture standardisée pour le développement d’applications embarquées (cf. figure 2.3). Pour cela, une interface graphique permet de choisir quelles fonctions sont à intégrer dans l’application et de les configurer mais aussi de définir les caractéristiques du moteur cible ou la configuration du *RTOS* (définition des récurrences des

Métiers	Compétences
Automatique	<ul style="list-style-type: none"> • Modélisation et simulation multi physiques • Algorithmes de contrôle - commande basés sur des modèles • Traitement du signal • Pilotage temps réel embarqué
Electronique analogique & numérique	<ul style="list-style-type: none"> • Mesure et instrumentation • Etude et dimensionnement de produits (calculateur...) • Réalisation de prototypes, incl. faisceaux de câblages
Informatique industrielle	<ul style="list-style-type: none"> • Logiciel embarqué temps réel critique : <ul style="list-style-type: none"> ◦ Basic Software ◦ Logiciel applicatif ◦ Piles de communication • Instrumentation & IHM (Labview) • Cibles: NI, DSPIC, Renesas, C167, Power PC... • Process : industriel ou prototype

FIGURE 2.2 – Métiers et Compétences – Source : Abord Engineering

tâches). Une fois l'application configurée, la plateforme va chercher les codes sources des modules sélectionnés, génère les fichiers sources non fonctionnels (non relatifs à des fonctions moteur) et effectue la compilation. La sortie produite est un fichier binaire représentant le code source compilé ainsi qu'un fichier « dictionnaire » associant chaque élément à l'adresse à laquelle il sera stocké dans la mémoire du calculateur. Ce fichier contient les adresses des constantes, des calibrations, des différentes fonctions appelées, etc. L'intérêt d'avoir des adresses fixes et connues pour ces éléments est qu'il est possible via un logiciel tiers de récupérer ces informations et de les modifier directement dans la mémoire sans avoir à déployer une nouvelle application. Pour rappel, nous sommes dans un cadre R&D et beaucoup de tests sont effectués sur banc de tests et le besoin d'adapter les calibrations pour obtenir le meilleur comportement est primordial.

La plateforme en elle-même est composée de plusieurs outils :

- Configuration du moteur cible
- Configuration des fonctions moteur
- Configuration du RTOS
- Génération des fichiers sources non fonctionnels
- Compilation de l'application

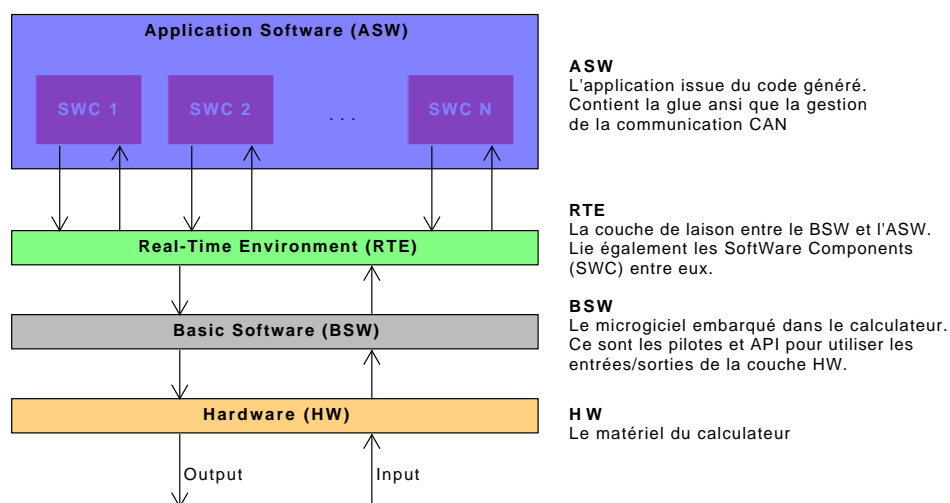


FIGURE 2.3 – Schéma de l'architecture d'une application suivant les spécifications AUTOSAR.

2.3 L'environnement de travail

J'avais mon poste de travail personnel sur lequel j'avais un contrôle suffisant pour installer les logiciels qu'il me fallait. L'entreprise possède également plusieurs ordinateurs portables sur lesquels sont installés certains logiciels comme Matlab®, *INCA* ou *CANalyzer*. Ces ordinateurs portables sont en libre service aux employés pour l'utilisation des ces logiciels en particuliers ou simplement pour avoir un environnement de test indépendant de l'environnement de développement.

J'ai utilisé les logiciels suivant durant mon stage :

Netbeans/Eclipse/IntelliJ J'ai utilisé plusieurs IDE. Les deux premiers pour reprendre le code existant et comprendre l'architecture des projets. Le dernier par préférence personnelle et par habitude.

VirtualBox J'ai installé deux machines virtuelles sur mon poste de travail en utilisant le logiciel VirtualBox. La première sous Fedora pour faire de l'analyse statique de code C. La deuxième sous Debian configurée en serveur pour installer Gitlab.

Gitlab J'ai mis en place un Gitlab pour gérer mes sources. Gitlab est une plateforme de gestion de projet permettant d'héberger un dépôt Git et de gérer des tâches (similaire au service Github mais sur un serveur local).

Présentation du sujet

Aboard Engineering est une entreprise spécialisée dans les domaines de l'informatique embarquée. Les problématiques logicielles qu'elle rencontre sont liées au matériel sur lequel sera déployé le code développé. Pour faciliter ce type de développement, les concepteurs de logiciels embarqués se tournent de plus en plus vers des outils de *MBD* qui permettent d'utiliser des langages de haut niveau – souvent graphiques – pour la conception et la spécification des systèmes à développer. On peut citer par exemple Matlab® Simulink® qui est un outil permettant de définir graphiquement sous forme de schéma logique le fonctionnement de systèmes complexes. C'est d'ailleurs l'outil utilisé par Aboard Engineering pour définir ses fonctions de contrôle moteur. Les avantages de ces modèles sont multiples. Premièrement, les outils de MBD permettent de simuler l'exécution du système afin de vérifier son fonctionnement. Ensuite, ces outils embarquent généralement un générateur de code afin de générer le code correspondant à ces modèles, allégeant grandement le travail des concepteurs et évitant au maximum les erreurs faites lors de développement « manuel ». En reprenant l'exemple de Matlab® Simulink®, la suite intègre un générateur de code embarqué temps réel appelé *RTW-EC*®. Il génère aujourd'hui le code des prototypes de fonctions de contrôle moteur développés par l'équipe. J'y reviendrai dans la section 3.1.1 plus en détail.

Comme dit précédemment, la génération de code facilite le travail de conception et de réalisation de prototype. Cependant, pour pouvoir compiler le code de plusieurs modules différents en une seule application, il reste des étapes à réaliser. C'est ici que la plateforme de prototypage rapide Orianne entre en jeu. Son rôle est de mettre en relation ces modules et de créer la « glue » entre tous les composants.

La plateforme Orianne est composée de plusieurs parties que nous appellerons « composants ». J'entrerai plus dans le détail dans la section 3.1.1.

3.1 La génération de code embarqué

3.1.1 État des lieux

Le développement d'une application pour un calculateur moteur est une tâche complexe qui peut prendre un certains temps. De nombreux modèles sont nécessaires à la spécification de tous les aspects du fonctionnement d'un moteur ou des différentes commandes utilisateur (pédales, boîte de vitesse, etc). À cela peut s'ajouter la gestion d'une configuration hybride – un moteur thermique couplé à un moteur électrique – nécessitant une gestion plus pointue de l'énergie.

Une fois complets, ces modèles ont vocation à être traduits en code source qui sera compilé puis intégré dans un calculateur. C'est donc ce code compilé qui est la base du contrôle moteur d'un véhicule. On comprend alors les contraintes de fiabilité et de performance sous-jacentes.

Pour générer ce code, nous utilisons Matlab® RTW-EC®. C'est un générateur de code embarqué temps réel puissant et reconnu pour fournir du code suivant les normes en vigueur dans l'automobile. Le code généré contient toute la logique métier et donc l'essentiel de la complexité de l'application. Sur ce point, Matlab® RTW-EC® fournit des résultats très bons en terme de temps d'exécution, de mémoire utilisée ou de taille de pile lors des appels aux fonctions des modules.

Ce processus a l'air de fonctionner. Cependant, les contraintes budgétaires sont également un point

3.1 La génération de code embarqué

important à ne pas négliger. La suite Matlab® coûte très cher. Même pour une entreprise – d’autant plus pour une entreprise de petite taille comme Aboard Engineering –, les coût des licences Matlab® couvrant toute la suite nécessaire à l’ensemble du processus de production sont significatif lors de la budgétisation d’un projet.

La problématique est alors de retrouver les même garanties de performance et de qualité du code délivrées par Matlab® sans le coût exorbitant qu’engendre l’achat de licences. C’est pourquoi Aboard Enginnering a décidé de se tourner vers une alternative libre : *QGen* et *Project P*.

3.1.2 QGen

A la base de QGen, on trouve un projet open source appelé Project P (cf. figure 3.1). Le but du projet P est de supporter le MBD pour des système temps réel embarqués en fournissant un framework de génération de code open source capable de :

1. vérifier l’intégrité d’un point de vue sémantique de systèmes conçus avec différents langages de modélisation (Matlab® Simuling®, Scicos, Xcos, SysML, MARTE, UML) ;
2. générer du code source optimisé pour des langages de programmation (Ada, C/C++) et des langages de synthèse (VHDL, SystemC) ;
3. supporter des processus de certification de différents domaines (avionique, spatial, automobile).

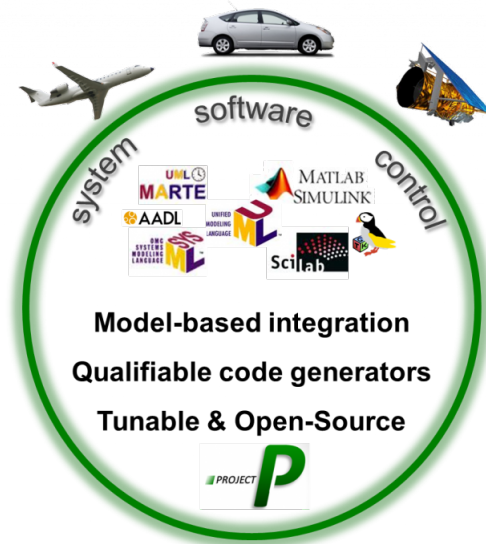


FIGURE 3.1 – Project P – Source : <http://www.open-do.org/projects/p/>

Parmi les collaborateurs de ce projet, on trouve de grands groupes industriels (Airbus, Astrium, Continental, Rockwell Collins, Safran, Thales), des PME (AdaCore, Altair, Scilab Enterprise, STI), des sociétés de services (ACG, Aboard Engineering, Atos Origins) mais aussi des centres de recherche (CNRS, ENPC, INRIA, ONERA).

AdaCore a ensuite développé un produit commercial à partir des technologies développées dans Project P : QGen.



FIGURE 3.2 – QGen – Source AdaCore

3.2 Les outils de la plateforme

La plateforme Oriante est développée en Java. Le projet est composé de plusieurs modules pour séparer les différents outils. Deux principaux modules séparent la partie IHM qui sert à la configuration des outils d'une part et les générateurs de code d'autre part.

3.2.1 La partie IHM

La partie IHM s'organise sous forme d'onglets correspondant à la configuration de chaque outil (cf. figure 3.4).

Configuration moteur Cette partie permet de configurer les éléments relatifs à un moteur. Le type de carburant, de combustion, d'injection, les fonctions à intégrer dans l'application finale.

Configuration fonctions Chaque fonction cochée dans le premier onglet, peut être configurée dans le deuxième onglet. Les fonctions sont donc configurables via des éléments graphiques comme des cases à cocher, des champs texte ou des liste déroulantes.

Configuration RTOS Cet onglet reprend les différences fonctions triées par leur récurrence sous forme de tableau. Chaque ligne représente un point d'entrée d'une fonction de contrôle moteur. L'ordre est significatif et peut être modifié via des boutons. L'ordre des fonctions dans cette table sera répercuté dans le fichier source de configuration du RTOS.

Configuration ECU & RTE Cette section fait le lien entre les API fournies par les pilotes du calculateur moteur et les variables d'entrée/sortie des fonctions de l'application. En configurant les capteurs et actionneurs du calculateur avec leurs données (variables), la plateforme permet de faire la correspondance entre ces variables et les données consommées par les différentes fonctions de l'application.

Configuration CAN L'ECU n'est pas le seul calculateur dans un véhicule. La communication entre ces différentes unités se fait généralement via un bus CAN. L'onglet CAN permet de configurer les différentes cellules de communication et accepte des fichiers XML représentant les différents messages à envoyer sur une cellule définie. L'interface permet également de faire correspondre les signaux reçus ou envoyés à des variables de l'application en entrée ou en sortie des fonctions.

3.2.2 La partie génération

L'interface de la plateforme a un menu regroupant les différentes actions de génération possibles via la plateforme (cf figure 3.3).

Génération CAN Cette action crée deux fichiers source `.c` et `.h` reprenant la configuration de la communication CAN.

Génération RTOS Cette action reprend les listes des points d'entrée des modules de l'application en fonction de leur place dans la configuration.

Génération RTE Cette action crée deux fichiers source `.c` et `.h` déclarant l'ensemble des entrées/sorties des fonctions qui seront échangées entre les différents modules. Elle génère aussi les fonctions d'accès en lecture et en écriture des données fournies par le matériel du calculateur moteur par l'intermédiaire de pilotes.

Génération Oriante Cette action met en place les fichiers aux emplacements requis pour la compilation d'une application complète. Elle copie les sources générées par la plateforme et celles importées depuis la génération de code des modèles Simulink®.

Compilation Cette action lance la compilation d'une application complète via le compilateur défini dans la configuration de la plateforme.

Génération A2L Cette action génère le fichier A2L servant de dictionnaire des variables dans la mémoire du calculateur.

3.2.3 La problématique

La problématique autour de cette plateforme est simple : tous les outils ne sont pas encore créés.

Afin que la plateforme garde une certaine stabilité, les générateurs non implémentés sont remplacés par une simple copie de fichiers génériques embarqués en dur dans la plateforme. Ces fichiers doivent ensuite être modifiés à la main pour correspondre aux besoins de l'application conçue via la plateforme.

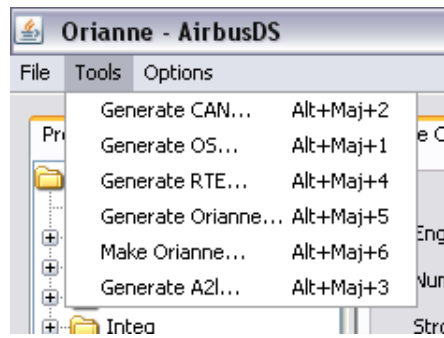


FIGURE 3.3 – Plateforme Orianne, menu Tools (capture du 17 juin 2015)

Ainsi, les fichiers sont en place pour faciliter la compilation d’une application. L’utilisateur n’a alors plus qu’à modifier les fichiers dont il a besoin et lancer la compilation.

Parmi les outils manquant à la plateforme, je vais me concentrer sur la génération CAN et la génération RTE, les outils sur lesquels j’ai travaillé. Il faudra donc développer les interfaces graphiques et les générateur correspondants. La prise en compte de ces nouveaux outils aura un impact sur les autres outils comme par exemple la génération RTOS qui devra prendre en compte les nouveaux points d’entrée de la communication CAN dont la structure pourra changer pas rapport à leur ancienne version.

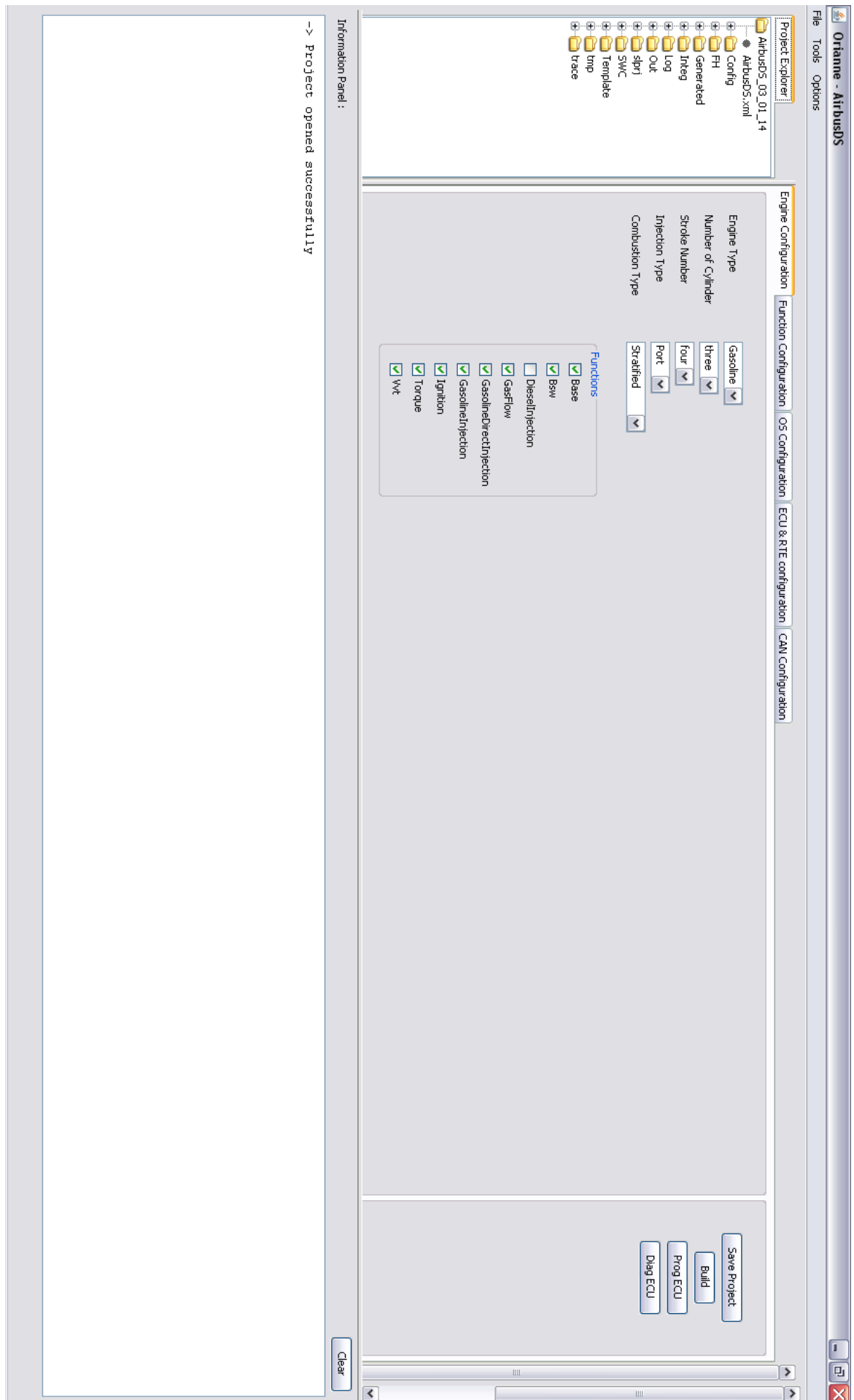


FIGURE 3.4 – Plateforme Oriane, configuration moteur (capture du 17 juin 2015)

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 intervalles 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 déployées 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 Oriante 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 Oriante.

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

Les mesures des quantités de mémoire et d'espace de stockage utilisées sont faites directement par le compilateur. Il produit un fichier A2L qui s'appuie sur le code assembleur compilé pour définir les tailles et adresses 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'exécution est moins triviale. Le RTOS du calculateur nous permet de définir des tâches à exécuter à une certaine récurrence. J'ai choisi de façon arbitraire une tâche à évaluer 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'exécution d'une tâche précise et à 0 à la fin de cette même tâche, nous aurons un 1 sur cette sortie durant toute l'exécution 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'exécution 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 rares cas –, ce n'est pas l'objectif de la génération de code. Le code RTW-EC® a une plus grande complexité cyclomatique¹ mais cela est du au générateur qui ne crée qu'une seule fonction de calcul alors que QGen crée plusieurs fichiers sources par modules et découpe ses traitements. En revanche, le code généré par QGen est plus long en terme de lignes de code.

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 passons de 400 μ s d'exécution pour RTW-EC® à 800 μ 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'interpolation² et d'extrapolation³ lors de recherche de correspondance dans les tables de calibration n'étaient pas les mêmes 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'optimisations 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 μ 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 concentrons sur une seule fonction, la plus significative en terme de performance dans notre application. Afin d'aider AdaCore dans leur recherche d'améliorations, nous leur 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 durant l'été 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'exécution 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.

Il manque actuellement deux composants majeurs à la plateforme : la génération CAN et la génération RTE. La génération CAN se charge de gérer la communication de l’application sur un bus CAN à l’intérieur du véhicule. La génération RTE comprend la liaison entre les modules – leurs entrées/sorties – et les capteurs et actionneurs accessibles via l’ECU. C’est également dans la partie RTE que nous configurons le « basic software » – c’est-à-dire le micrologiciel embarqué dans l’ECU définissant les pilotes et API fournis par cette ECU – et ainsi couplerons complètement notre application avec le matériel sur lequel elle sera déployée.

4.2.1 L’existant

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 présents. L’aspect dynamique qu’apporte la plateforme *OSGi* n’était pas du tout exploitée.

Après discussion avec mes collègues, nous avons constaté que l’utilisation de la plateforme *OSGi* n’avait pas un 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*. Cela nous a permis 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 inutiles 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 sera « utilisée » par 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.

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 entraîné une modification du générateur. 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 l'utilisation mémoire du code C généré pour la communication CAN 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. Cependant, le complètement de ces objectifs est primordiale pour avoir un outil viable et fonctionnel avant l'intégration 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 système 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 étaient 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 de deux 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 lues 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 contrepartie, la configuration des trames – qui sont des données statiques, 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 une 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 liés à cette configuration. J'ai ensuite ajouté ces composants dans un nouvel onglet de la plateforme (cf. figure 4.1 et 4.2).

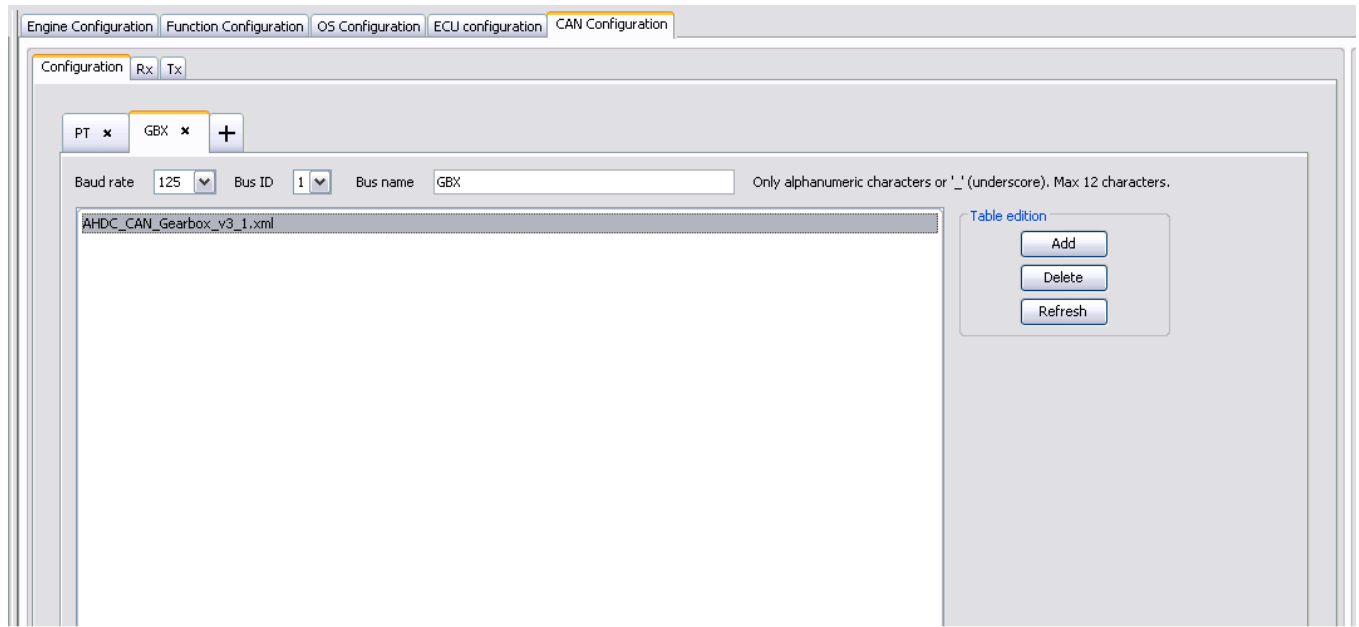


FIGURE 4.1 – Interface de configuration des bus CAN utilisés par l'application (capture du 17 juin 2015)

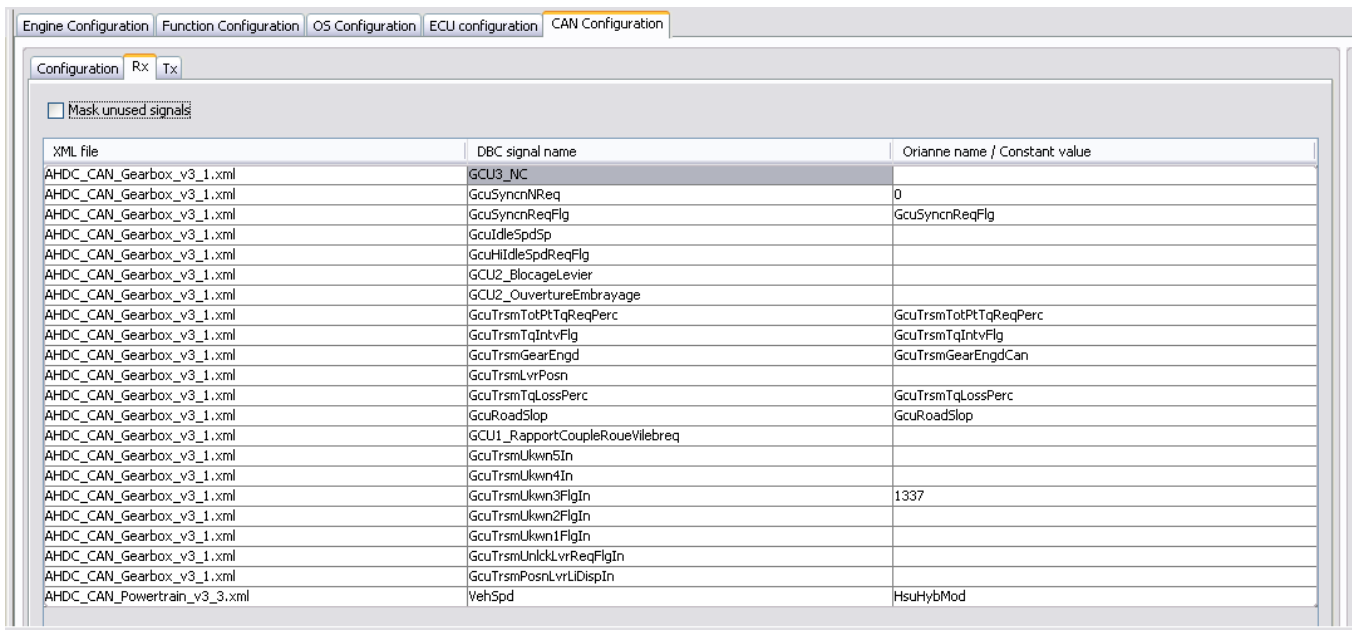
4.2.3 La partie RTE

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 générera 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 graphique. Comme pour la partie CAN, elle a vocation à faciliter la configuration de l'application via des éléments graphiques simples (formulaires, tables, etc.). Elle doit s'articuler autour de la configuration possible de l'ECU afin de fournir le maximum de détails à l'utilisateur tout en restant cohérente avec le processus de configuration. Par exemple, il faudra que l'utilisateur définisse les capteurs et actionneurs qu'il souhaite utiliser avant de définir la liaison des données de ces composants avec les paramètres des modules de l'application. L'IHM doit être construite pour que cet enchaînement de tâche se fasse de la manière la plus naturelle et la plus intuitive possible. Il faudra ensuite développer le générateur basé sur cette IHM qui créera le code relatif à cette configuration et adaptera la « glue » pour prendre en compte ce nouveau code.

4.2 La plateforme Orianne



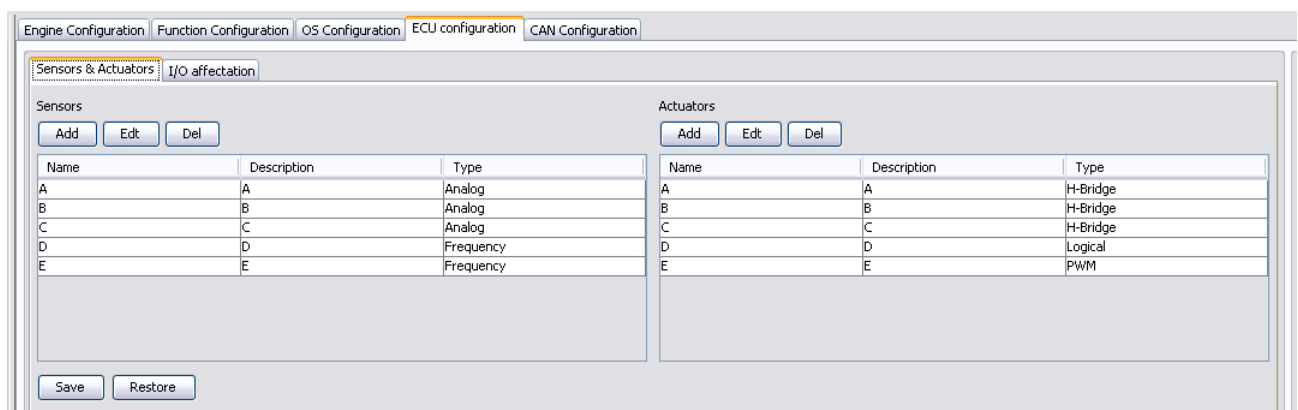
The screenshot shows the 'CAN Configuration' tab in a software interface. It features a table with three columns: 'XML file', 'DBC signal name', and 'Orianne name / Constant value'. The table lists various signals from the 'AHD_CAN_Gearbox_v3_1.xml' and 'AHD_CAN_Powertrain_v3_3.xml' files, mapping them to specific Orianne names or constant values.

XML file	DBC signal name	Orianne name / Constant value
AHD_CAN_Gearbox_v3_1.xml	GCU3_NC	
AHD_CAN_Gearbox_v3_1.xml	GcuSynchNReq	0
AHD_CAN_Gearbox_v3_1.xml	GcuSynchReqFlg	GcuSynchReqFlg
AHD_CAN_Gearbox_v3_1.xml	GcuIdleSpdSp	
AHD_CAN_Gearbox_v3_1.xml	GcuHIdleSpdReqFlg	
AHD_CAN_Gearbox_v3_1.xml	GCU2_BlocageLevier	
AHD_CAN_Gearbox_v3_1.xml	GCU2_OuvertureEmbrayage	
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmTotPtTqReqPerc	GcuTrsmTotPtTqReqPerc
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmTqIntvFlg	GcuTrsmTqIntvFlg
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmGearEngd	GcuTrsmGearEngdCan
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmLvrPosn	
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmTqLossPerc	GcuTrsmTqLossPerc
AHD_CAN_Gearbox_v3_1.xml	GcuRoadSlop	GcuRoadSlop
AHD_CAN_Gearbox_v3_1.xml	GCU1_RapportCoupleRoueVilebreq	
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmUkwn5In	
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmUkwn4In	
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmUkwn3FlgIn	1337
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmUkwn2FlgIn	
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmUkwn1FlgIn	
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmUnlckLvrReqFlgIn	
AHD_CAN_Gearbox_v3_1.xml	GcuTrsmPosnLvrLIDispIn	
AHD_CAN_Powertrain_v3_3.xml	VehSpd	HsuHybMod

FIGURE 4.2 – Interface de configuration des signaux CAN (vue des signaux d'entrée, capture du 17 juin 2015)

La partie CAN est à présent terminée, l'interface est complète et le générateur fonctionnel. Il est désormais possible de configurer et généré le code relatif à la communication CAN entièrement depuis l'interface graphique de la plateforme.

La partie RTE est en pleine conception et le développement devrait démarrer sous peu. Les spécifications ne sont pas figées encore mais nous avons les premiers prototypes de l'IHM (cf. figure 4.3). Ils vont nous permettre de simuler une utilisation de l'outil et de voir les améliorations éventuelles ou modifications nécessaires dans la spécification pour que l'outil soit agréable à utiliser mais surtout qu'il permette une configuration poussée et complète de la partie RTE.



The screenshot shows the 'Sensors & Actuators' configuration window. It has two tabs: 'Sensors & Actuators' (selected) and 'I/O affectation'. The 'Sensors' section contains a table with columns 'Name', 'Description', and 'Type'. The 'Actuators' section contains a similar table. Both sections have 'Add', 'Edit', and 'Delete' buttons. At the bottom, there are 'Save' and 'Restore' buttons.

Name	Description	Type
A	A	Analog
B	B	Analog
C	C	Analog
D	D	Frequency
E	E	Frequency

Name	Description	Type
A	A	H-Bridge
B	B	H-Bridge
C	C	H-Bridge
D	D	Logical
E	E	PWM

FIGURE 4.3 – Prototype de la configuration des capteurs et actionneurs utilisés.(capture du 17 juin 2015)

Chapitre 5

Bilan

Expérience professionnelle

Au cours de mes expériences professionnelles précédentes, j'ai eu l'occasion de voir plusieurs domaines, plusieurs technologies, plusieurs structures d'entreprise et plusieurs méthodes de travail. J'ai donc eu l'occasion de faire la part des choses et identifier les aspects que j'aime et ceux que je n'aime pas.

Les stages sont un très bon moyen de découvrir des environnements professionnels différents. C'est pourquoi j'ai été tenté par un stage dans l'automobile, un domaine que je n'avais pas encore exploré. De plus, je me sens bien dans les petites structures où on a généralement plus le droit à la parole et où notre avis compte en terme de technique et de solution à adopter. Je retiens des discussions que j'ai eu avec mes camarades qu'il n'est pas toujours évident de défendre ses idées en terme de conception ou de technologie quand on est obligé de suivre des politiques communes à plusieurs équipes comme c'est le cas dans certaines grandes structures.

Ce stage constitue une expérience pour moi dans le domaine de l'informatique industrielle embarquée et plus particulièrement dans l'automobile. Il m'a aussi permis de découvrir le MBD dont on nous a parlé en cours et que je souhaitais voir dans un contexte professionnel pour voir les contraintes que cela pose et leur impact sur un projet concret.

Sans oublier le format du stage qui est un point essentiel de cette expérience. Le fait d'effectuer ce stage en alternance m'a permis de suivre un projet d'envergure avec un avenir concret dans le temps et d'y participer pleinement.

D'un point de vue plus objectif, je dirais que j'ai rempli mes objectifs de stage. J'ai réussi les missions qui m'ont été confiées et je suis satisfait du travail que j'ai effectué durant ces 9 mois. J'ai travaillé avec une combinaison de plusieurs technologies, utiliser des compétences différentes et même appris de nouvelles. Et ce, pour atteindre la même finalité. Je trouve cela très enrichissant et je pense que ça nous permet de mieux appréhender les problèmes en abordant des solutions sous plusieurs angles.

Apports personnels

D'un point de vue plus personnel, je me suis conforté dans l'idée que je suis intéressé par l'informatique embarquée. Ce stage a aussi éveillé ma curiosité dans le développement plus orienté système. Je ne pense pas que je souhaiterais faire toute ma carrière dans ce domaine technique mais c'est ce qui m'intéresse actuellement.

J'ai pu aussi conforter mes compétences en anglais. Toute la communication avec AdaCore autour de QGen s'est faite en anglais via leur gestionnaire de bug ou par e-mail. Je me sens à l'aise dans la communication en anglais et je songe à chercher du travail à l'étranger pour enrichir encore mon vocabulaire et découvrir d'autres rythmes de vie.

J'ai récemment commencé ma recherche d'emploi pour l'après stage. J'ai plusieurs opportunités dans des domaines différents. Je suis actuellement en discussion avec Aboard Engineering à propos d'un poste de développeur logiciel et système. Je recherche des offres qui me permettraient d'approfondir mes compétences techniques et pourquoi pas me familiariser avec de nouvelles en les mettant en application dans des projets concrets.

Résumé

Mots-clés Matlab, Langage C, Java Swing, MBD, Génération de code, Informatique embarqué, Communication CAN, Automobile, Optimisation, Factorisation.

J'ai effectué ce stage dans le cadre du Master 2 Développement Logiciel. Il s'est étalé sur 9 mois du 3 novembre 2014 au 31 juillet 2015 en alternance au rythme de 3 jours par semaine en entreprise et 2 jours à l'université. J'ai rejoint Aboard Engineering, un bureau d'étude expert dans les domaines de l'électronique et de l'informatique industrielle. J'ai occupé un poste d'ingénieur en développement logiciel et j'ai travaillé sur deux sujets en parallèle : la plateforme Orianne et le générateur de code QGen.

La plateforme permet de faire du prototypage rapide de fonctions de contrôle moteur à partir de code source généré par Matlab® RTW-EC®. Elle est en cours de développement. J'ai donc repris du code existant que j'ai factorisé dans le but d'optimiser le code et de faciliter l'évolution de la plateforme. J'ai ensuite ajouté des outils manquants à cette plateforme comme un outil de génération de code pour de la communication CAN. Une ébauche de l'outil existait mais le code généré était trop gourmand en terme de mémoire pour être intégré dans une application de contrôle moteur.

Le générateur QGen est lui aussi en cours de développement mais pas par Aboard Engineering. QGen est un générateur open-source de code embarqué temps réel en C et Ada. Le projet est piloté par AdaCore et est issu d'un autre projet open-source baptisé Project P. Le rôle que joue Aboard Engineering dans l'évolution de ce générateur relève du test et de la revue. Nous avons participé à l'évolution de ce générateur de code de manière collaborative avec AdaCore via l'intermédiaire d'une plateforme de suivi de bug. Du fait que nous possédons les qualifications techniques pour analyser le code généré par QGen, notre objectif était de comparer en terme de temps d'exécution et d'impact mémoire les performances du code généré par QGen par rapport au code généré par Matlab® RTW-EC®, outil que nous utilisons et que nous avons pris pour référence de comparaison sur ces points précis.

Au terme de ce stage, les objectifs sont remplis. Le générateur CAN est fonctionnel et les résultats de l'amélioration du code généré sont concluants. En effet, j'ai pu réduire de manière significative l'impact en mémoire (7 fois moins d'espace utilisé) tout en gardant un outil opérationnel. J'ai aussi développé une partie de configuration graphique pour s'intégrer au reste de la plateforme. Je travaille actuellement à la conception et au développement d'un outil de génération de code faisant le lien entre l'application en question et le matériel sur lequel elle sera déployée.

Les résultats des tests obtenus via la collaboration avec AdaCore autour de QGen sont eux aussi très concluants. Nous sommes arrivés à obtenir des performances équivalentes entre le code généré par RTW-EC® et QGen. Une première version majeure doit sortir courant juin 2015. Avec cette version finale, nous devrions être en mesure d'effectuer des tests plus poussés sur banc de test et, si ces derniers sont validés, des tests directement sur véhicule.

Summary

Key words Matlab, C language, Java Swing, MBD, Source code generation, Embedded computing, CAN communication, Automotive, Optimisation, Factoring.

This internship took place during the last year of my Master's Degree in Software Development. It was 9 months long, from November, 3rd 2014 to July, 31th 2015. It was a work-study internship with 3 days a week at work and 2 days a week at university. I joined Aboard Engineering, a service company expert in electronics and industrial computing. I was a software designer engineer and I worked on 2 subjects at the same time : the Orianne platform and the QGen source code generator.

The platform allows to do rapid prototyping for engine management functions development based on source code generated using Matlab® RTW-EC®. It is currently in development. So I used already written code that I factored for clarity and complexity reasons. Then I added some missing tools to the platform like a code generator for CAN communication. A first draft of the tool existed but it was taking too much memory space to be integrated in an engine control application.

The QGen generator was in development too but not by Aboard Engineering. It is an open source real time source code generator. The project is driven by AdaCore and comes from an other open source project called Project P. The role played by Aboard Engineering in the evolution of that tool is about tests and reviews. We have the technical qualifications to analyze the source code generated by QGen. Our objective was to compare this code to the one generated by Matlab® RTW-EC® regarding execution time and memory usage. Matlab® RTW-EC® is a tool that we used at Aboard Engineering so we took it as a benchmark for our tests and comparisons on those particular points.

At the end of the internship, the objectives are completed. The CAN generator is functional and the results from the improvements of the generated code are conclusive. In fact, I achieve to reduce in a significant way the memory usage (7 times less memory space used) without losing features. I also developed a graphical interface for the tool configuration to be integrated in the rest of the platform. I currently work on the design and development of another tool that will generate the code to bind the application on the hardware in which it will be deployed.

The results from the tests about QGen obtained in collaboration with AdaCore are also conclusive. We achieved to obtain similar performance between the code generated by RTW-EC® and the one generated by QGen. A first major version would be released during June 2015. With this final version, we should be able to do further tests on test bench and, if those are validated, more tests directly on a vehicle.

Table des figures

2.1	Domaines – Source : Abord Engineering	3
2.2	Métiers et Compétences – Source : Abord Engineering	4
2.3	Schéma de l’architecture d’une application suivant les spécifications AUTOSAR.	4
3.1	Project P – Source : http://www.open-do.org/projects/p/	8
3.2	QGen – Source AdaCore	8
3.3	Plateforme Orianne, menu Tools (capture du 17 juin 2015)	10
3.4	Plateforme Orianne, configuration moteur (capture du 17 juin 2015)	11
4.1	Interface de configuration des bus CAN utilisés par l’application.	17
4.2	Interface de configuration des signaux CAN (vue des signaux de sortie).	17
4.3	Prototype de la configuration des capteurs actionneurs utilisés.	18