UNIVERSITÉ TOULOUSE III PAUL SABATIER — Université de Toulouse

Continental

Rapport de stage

Développement d'une plateforme de tests automatisés : GreenT

Antoine de Roquemaurel

L3 Informatique – Parcours ISI
2013 – 2014

Maître de stage :
Stéphane Bride

Tuteur universitaire :
Joseph Boudou

Du 14 avril au 11 Juillet 2014
Version du 5 juin 2014

Je tiens à remercier toutes les personnes m'ayant permis de réaliser ce stage.

En premier lieu, un grand merci à Stéphane Bride pour m'avoir accepté dans son équipe et suivi tout au long du stage.
Je remercie les membres de l'équipe avec laquelle j'ai travaillé : Alain Fernandez et Olivier Ramel, sous-traitant. Ils ont su m'accompagner, me conseiller et me faire découvrir de nouvelles technologies, toujours dans la bonne humeur autour d'un café.
Je remercie Smaïl Bouaiche pour son aide dans ma recherche de stage.
Une pensée à Joelle Decol pour sa bonne humeur quotidienne, ainsi qu'à toute l'équipe du troisième étage, grâce à qui j'ai passé d'excellents moments au sein de l'entreprise.
Merci à mon tuteur universitaire Joseph Boudou pour son suivi et sa visite en entreprise.
Enfin, je remercie toutes les personnes m'ayant entouré durant ce stage et aidé à la rédaction ce rapport, à savoir Diane, Ophélie, Clément et Mathieu.

Introduction

Dans le cadre de ma formation en troisième année de Licence à l'université Toulouse III – Paul Sabatier, j'ai eu le choix entre effectuer un TER [1] ou un stage.

J'ai choisi la seconde option car je me sens plus attiré par le monde de l'entreprise que par la recherche. J'ai eu la chance d'avoir une opportunité de stage de trois mois dans l'entreprise Continental Automotive au sein de l'équipe Vérification et Validation pour un projet de développement d'une plateforme de tests de logiciels embarqués.

Souhaitant intégrer le Master « Développement logiciel », le sujet même du stage était particulièrement intéressant. En outre, après une expérience de quatre mois dans une PME au cours de mon cursus à l'IUT 'A' Paul Sabatier, il me semblait pertinent de travailler au sein d'une grande entreprise à caractère international pour mieux définir mon projet professionnel.

Après une première phase de contact, le projet m'a été présenté plus en détails par mail. Un fournisseur distribue un plugin à Continental : un bout de code sous forme d'objet, que l'entreprise doit intégrer dans le logiciel applicatif d'un calculateur de contrôle moteur. La mission de l'équipe Vérification & Validation est de tester la bonne intégration de ce plugin dans l'applicatif du calculateur, pour cela une plateforme qui permettra des tests automatisés est en cours de développement.

Dans ce rapport nous verrons en quoi le développement de cet outil est nécessaire à l'équipe en charge des tests de ce plugin. Dans une première partie nous présenterons l'entreprise Continental et plus particulièrement l'équipe Vérification & Validation(chapitres 1 et 2). Nous aborderons ensuite le problème que posent actuellement les tests de ce plugin(chapitre 3), avant de présenter la solution qui est en cours de développement et comment j'ai contribué à ce projet(chapitre 4).

1. Travail Etude Recherche

Table des matières

1 Continental

Comme nous l'avons vu dans l'introduction, j'effectue mon stage au sein de l'entreprise Continental, une entreprise ayant pris une très grande importance dans le monde de l'automobile.

Sommaire

Nous allons voir dans quel contexte opère cette société, et plus particulièrement l'équipe dans laquelle j'ai travaillé : l'équipe Vérification & Validation.

1.1 Organisation de l'entreprise

1.1.1 Continental AG

Continental AG est une entreprise allemande fondée en 1871 dont le siège se situe à Hanovre. Il s'agit d'une Société Anonyme (SA) dont le président du comité de direction est le Dr. Elmar Degenhart depuis le 12 août 2009. Elle est structurée autour de deux grands groupes : le groupe Rubber et le groupe Automotive.

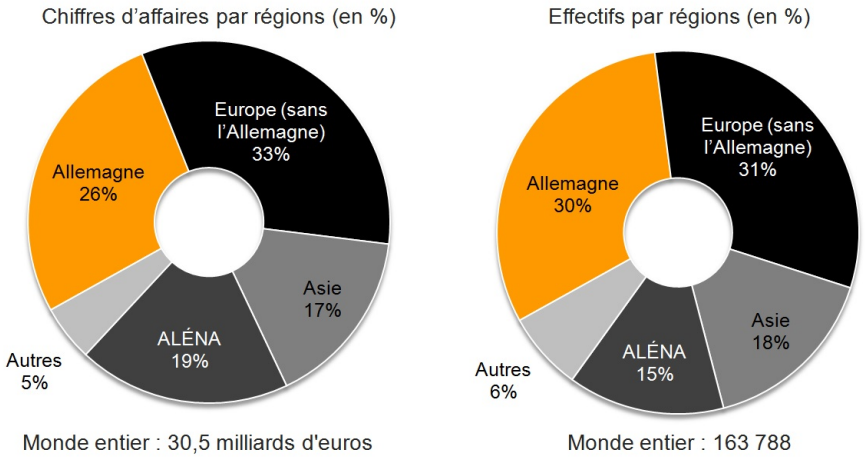

Figure 1.1 – Chiffre d'affaire et nombre d'employés (Année 2011)

En 2013, l'entreprise comptait plus de 177000 employés dans le monde[1] répartis dans 269 sites et 46 pays différents[2]. Avec un chiffre d'affaire de 30.5 milliards d'euros au total, Continental est le numéro un du marché de production de pneus en Allemagne et est également un important équipementier automobile.

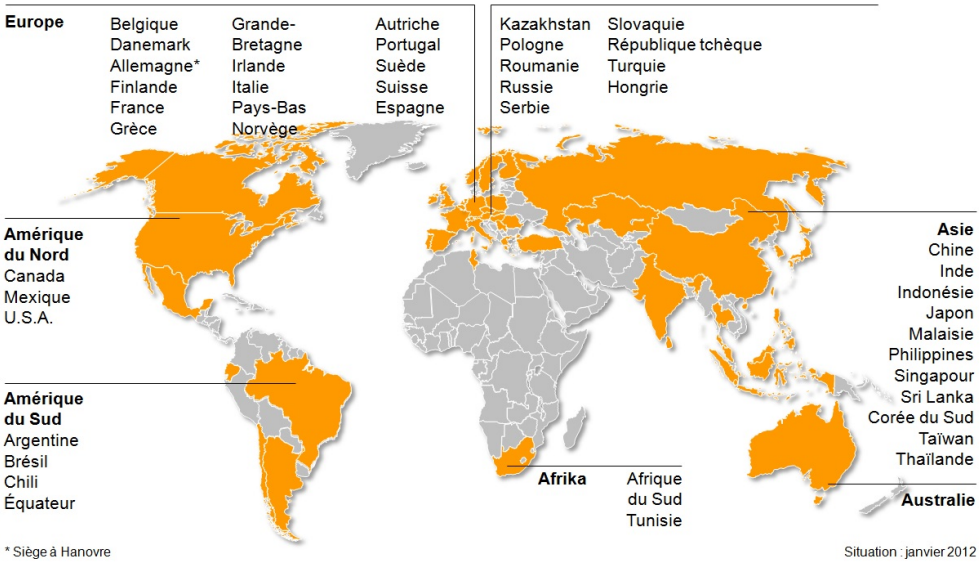

Figure 1.2 – Répartition du groupe continental dans le monde

1.1.2 Histoire de l'entreprise

Continental est fondée en 1871 comme société anonyme sous le nom de «Continental-Caoutchouc- und Gutta-Percha Compagnie» par neuf banquiers et industriels de Hanovre (Allemagne).

Continental dépose l'emblème du cheval représenté sur la figure 1.3, comme marque de fabrique à l'Office impérial des brevets de Hanovre en octobre 1882. Ce logo est aujourd'hui encore protégé en tant que marque distinctive.

Figure 1.3 – Logo de Continental

Le fabricant de pneus allemand débute son expansion à l'international en tant que sous-traitant automobile international en 1979, expansion qu'il n'a cessé de poursuivre depuis.

Entre 1979 et 1985, Continental procède à plusieurs rachats qui permettent son essor en Europe, celui des activités pneumatiques européennes de l'américain *Uniroyal Inc.* et celui de l'autrichien *Semperit*.

En 1995 est créée la division « Automotive Systems » pour intensifier les activités « systèmes » de son industrie automobile.

1. Cf figre 1.1
2. Cf figure 1.2

La fin des années 1990 marque l'implantation de Continental en Amérique latine et en Europe de l'Est.

En 2001, pour renforcer sa position sur les marchés américain et asiatique, l'entreprise fait l'acquisition du spécialiste international de l'électronique *Temic*, qui dispose de sites de production en Amérique et en Asie. La même année, la compagnie reprend la majorité des parts de deux entreprises japonaises productrices de composants d'actionnement des freins et de freins à disques.

En 2004, le plus grand spécialiste mondial de la technologie du caoutchouc et des plastiques naît de la fusion entre *Phoenix AG* et *Conti'Tech*.

Enfin en juillet 2007, Continental réalise sa plus grosse opération en rachetant le fournisseur automobile *Siemens VDO Automotive*. Ce rachat a permis à l'entreprise de multiplier son chiffre d'affaire par deux, passant ainsi de 13 milliards d'euros à plus de 30 milliards d'euros (chiffre de 2011).

1.1.3 Activités des différentes branches

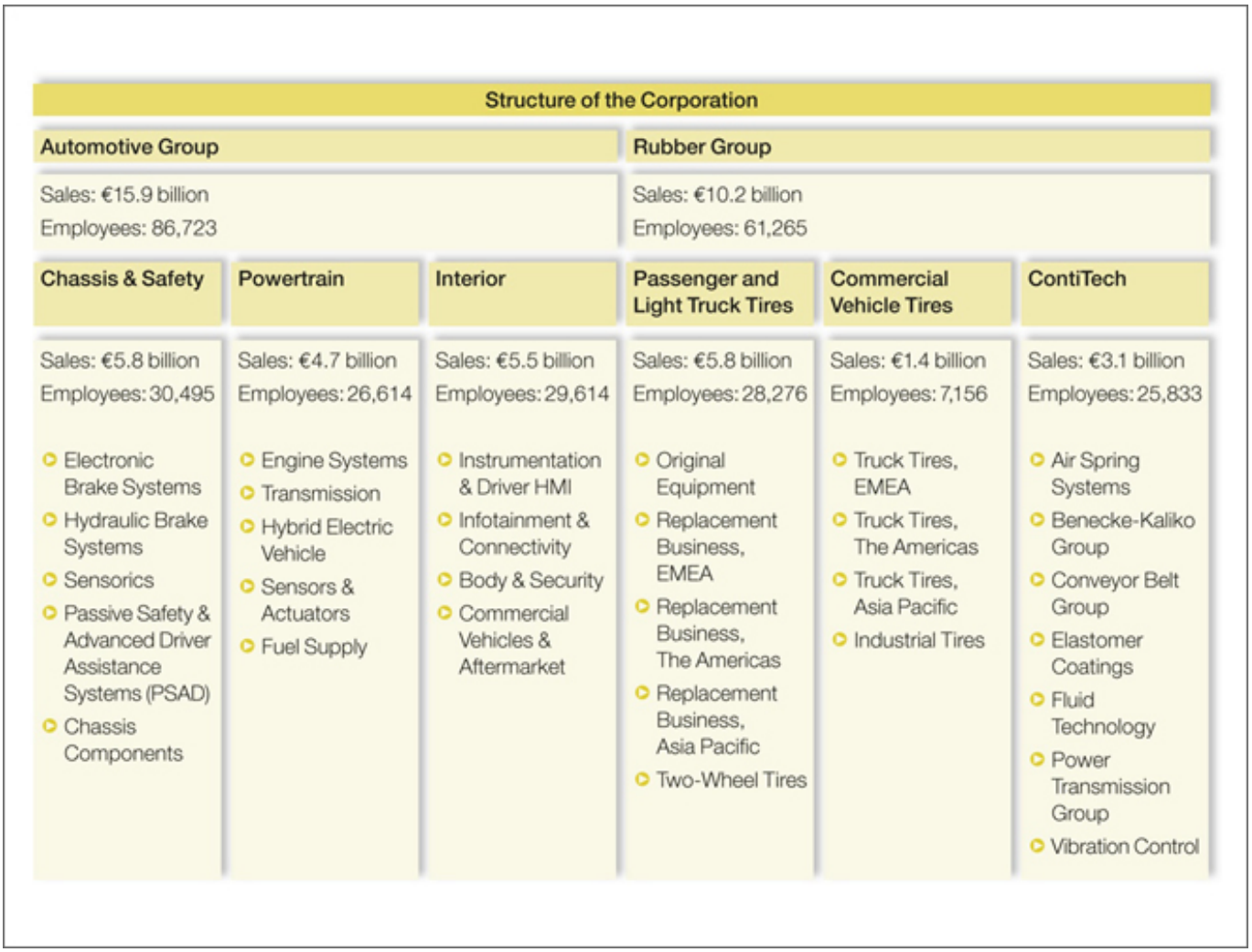

FIGURE 1.4 – Structure de continental

Comme on peut le voir sur la figure 1.4, Continental est composée de deux groupes et de six divisions. Ces dernières se chargent de développer et produire des équipements répondant aux besoins des clients. Pour cela elles sont composées de Business Units qui ont chacune une activité bien particulière dans leur domaine de compétence.

Durant mon stage, je travaillais au sein de la division *powertrain*. Elle s'occupe essentiellement du contrôle moteur, au niveau logiciel et materiel avec l'ECU [3] et de la mise au point des systèmes diesel et essence. Par exemple, la Business Unit *Engine Systems* est chargée de produire les équipements nécessaires au contrôle moteur tels que des calculateurs ou des injecteurs.

1.2 Le contexte de l'équipe Vérification & Validation

1.2.1 L'équipe

J'ai travaillé dans l'équipe en charge de la vérification et de la validation des logiciels. [4] dirigée par Stéphane Bride. Cette équipe est en charge du développement, de la configuration et de l'exécution de scripts de tests de non-régression automatique [5] sur bancs HIL [6] avant la livraison des projets.

1.2.2 Le besoin

Le calculateur du contrôle moteur d'une voiture est un dispositif très important et à haut risque, puisqu'une défaillance peut provoquer la mort de plusieurs personnes. Ainsi, le test est indispensable dans ce domaine, et doit être robuste.

L'automatisation des tests est rendue nécessaire pour deux raisons. Tout d'abord, un logiciel ne peut comporter aucun bug, cependant les erreurs et bugs critiques liés à l'inattention peuvent être grandement réduits grâce à ce processus. De plus, au vue du très grand nombre de cas à tester pour un contrôle moteur, une opération manuelle serait impensable.

C'est dans ce contexte que l'équipe Vérification et Validation intervient, elle doit fournir des outils aux développeurs afin de vérifier facilement et correctement leur travail, particulièrement pour des tests de régression, bien que l'outil que l'équipe est en train de développer soit à destination de tests d'intégration.

3. Electronic Control Unit
4. Plus précisément dans le service P-ES-E-SYS-ETV-V.
P : Powertrain
ES-E-SYS : Engine Systems
ETV : Engineering Tool and Verification
V : Vérification.
5. Aussi appelés FaST : Functions and Software Testing
6. Hardware in the Loop

2 Organisation du travail

Étant donné la complexité du projet et son importance, une bonne organisation est indispensable. Autant d'un point de vue humain, avec une gestion de projet et une gestion de l'équipe que technique en utilisant certaines technologies nous aidant dans la tache.

Sommaire

Nous allons voir l'organisation qui a été mise en place afin d'être le plus efficace possible.

2.1 L'équipe de développement

Au cours de mon stage, trois développeurs travaillaient sur le projet *GreenT* : Alain Fernandez, chef d'équipe et membre de l'équipe Vérification & Validation, Olivier Ramel, sous-traitant du groupe SII(Société pour l'informatique industrielle), et moi-même, stagiaire au sein de l'équipe Vérification & Validation.

En tant que chef d'équipe, Alain Fernandez organisait les réunions et supervisait notre travail tout en développant les tests managers [1]. Olivier Ramel se chargeait particulièrement de la partie serveur. Quant à moi je m'occupais de la partie parsing et génération [2].

À mon arrivée la conception du logiciel avait été commencée par mes deux collègues qui m'ont tous les deux formé afin que je puisse rapidement être opérationnel.

Ensemble nous avons convenu d'une réunion hebdomadaire tous les lundis matin afin de pouvoir faire un point sur nos avancements ou problèmes. Cela nous a permis d'avoir toujours une bonne vision du projet et de régler, ensemble, les problèmes au fur et à mesure. Pour autant, des réunions ponctuelles ont dû être organisées pour compléter ou revoir la conception en fonction des difficultés rencontrées durant la phase de développement.

2.2 Documentation

Une documentation, compte rendu des besoins du client et de tous nos choix de conception, a été mise en place au travers d'un document au format *Word*. Ainsi une trace de toutes nos réunions et de notre conception est accessible sur un disque réseau. Elle permet à l'équipe de consulter les détails de ce qui avait été décidé plusieurs semaines auparavant.

1. Cf 4.1.2
2. Cf 4.1.2

2.3 Outils de développement

Afin de travailler de façon efficace, nous avons utilisés des outils aidant au développement.

La partie client de notre plateforme est développée en Java dans sa version 6.0, Java nous permettant d'avoir un langage fortement typé, très puissant au niveau du paradigme Objet, connu de l'équipe, assez simple de déploiement et multiplateforme.

Les postes de Continental possédant pour la plupart Java 6, aucune fonctionnalité ultérieur à cette version n'a été utilisé.

Nous avons utilisé *Git* afin de faciliter le travail collaboratif d'une part, et de versionner le code du logiciel d'autres part. Git permet de fusionner les modifications de plusieurs développeurs, tant que nous ne modifions pas le même fichier en même temps. Ainsi, la fusion de nos modifications était faite automatiquement.

De plus, à chaque fois modification, un « commit », permet de crer un point de restauration : il est alors possible de récupérer n'importe quelle version de logiciel depuis son commencement. Nous y insérons un message clair expliquant ce qui a fait, cela permet aux autres développeurs de l'équipe de se tenir au courant de l'avancement.

Nous développions tous sous le même environnement de développement Eclipse Kepler, avec le plugin *Git* et le plugin *PyDev*. Le plugin Git permet d'avoir des outils aidant à la résolution d'éventuels conflits et le plugin PyDev permet de développer avec l'interpréteur et la coloration syntaxique Python. Je me suis rarement servi de ce dernier, mais il était indispensable pour développer la partie serveur de notre plateforme, qui fonctionne en Python.

Notre plateforme fonctionne avec une architecture client-serveur, le client écrit en Java et le serveur utilise Python. Afin de faire communiquer les deux parties de notre application, nous avons utilisé *Apache Thrift*. Une bibliothèque ayant pour but les communications réseau inter-langage, dans le même principe que le protocole RMI [3].

Nous avons travaillé avec la norme UML [4] 2 afin de concevoir la plateforme, en utilisant particulièrement des diagrammes de classes, mais aussi des diagrammes de cas d'utilisation ou d'activité.

Pour dessiner ces diagrammes, et les noter dans la documentation, nous les pensions d'abord sur tableau blanc, mais ensuite nous avions besoin d'un outil puissant afin de les dessiner sur informatique. Pour cela nous avons utilisés Enterprise Architect, un logiciel propriétaire permettant de créer tous les diagrammes de la norme UML 2.

Afin de rédiger ce rapport, et le diaporama de soutenance, j'ai utilisé LaTeX, un langage et un système de composition de documents fonctionnant à l'aide de macro-commandes. Son principal avantage est de privilégier le contenu à la mise en forme, celle-ci étant réalisée automatiquement par le système une fois un style définit.

LaTeX

3. Remote Method Invocation
4. Unified Modelling Language

3 Le problème

Depuis longtemps, l'entreprise avait un problème afin d'effectuer des tests d'intégrations, notamment pour les projets à destination de Ford. Les tests demandaient du temps et de l'argent à l'équipe en charge de ces tests. Ainsi, un an avant mon arrivée, une solution à été trouvée : le développement d'une nouvelle plateforme, *GreenT*.

Sommaire

3.1 Les tests

Comme expliqué dans la section 1.2.2, le calculateur moteur est un système critique, il est donc indispensable de tester correctement celui-ci.

3.1.1 Le plugin

Dans le cadre de projets pour Ford, Continental ne développe pas l'intégralité du logiciel, en effet une partie est fournie par le client sous forme de « plugin ». Le plugin est supposé correct, et ce n'est pas de notre ressort de le tester. Cependant, celui-ci va être interfacé avec les logiciels Continental : il est indispensable de vérifier que les deux parties fonctionnent ensemble lors de l'intégration.

Pour cela, le client fourni un fichier appelé `Walkthrough` [1] contenant la liste des variables du plugin avec toutes leur spécifications, ce fichier est au format *Excel* : et il contient environ 900 variables différentes. Il est impensable de tester le fonctionnement d'autant de paramètres manuellement, ainsi l'équipe en charge de tester cette intégration effectue des tests de différence d'une version à l'autre : seules les variables ayant pu être impactées par une *release* seront testées, il est supposé que les fonctionnement des variables restera inchangé.

Trois problèmes se posent à cette méthode :

La fiabilité des tests Le test des seules différences ne permet pas nécessairement de détecter tous les problèmes (notamment avec des effets de bords. . .). De plus, une tâche répétitive peut entrainer des erreurs humaines.

Le temps de tests Même en ne testant qu'une partie des variables, cela prend un temps considérable, il faut compter environ une semaine.

La disponibilité des bancs de tests Les tests s'effectuent sur des bancs de tests [2], ces équipements permettent de simuler un environnement voiture autour du contrôleur moteur comme

1. Ce fichier est expliqué plus en détail section 4.1.1
2. Une photo d'un banc est disponible figure 3.2

l'utilisation de la clé de démarrage, la tension de la batterie, la vitesse de rotation du moteur, ... Comme ces bancs de tests sont peu nombreux dans l'entreprise, en raison de leur coût et qu'il est nécessaire de réserver les bancs pendant une semaine afin d'effectuer les tests cette méthode peut bloquer d'autres personnes en ayant besoin.

Figure 3.1 – Exemple de banc de tests – HIL DSpace

3.1.2 La plateforme TA3

Actuellement, les équipes de tests disposent d'une plateforme appelée TA3. Celle-ci est une bibliothèque de classes écrites en Python. Jusqu'à présent, pour chaque objectif de test, il fallait écrire un script python utilisant la TA3. Ces scripts pilotent le banc Hil et le l'outil de debug afin d'envoyer des stimulis à l'Unité de contrôle moteur(Noté ECU pour *Electronic Control Unit*) et de vérifier que les réactions de celui-ci sont conforme aux spécifications de test.

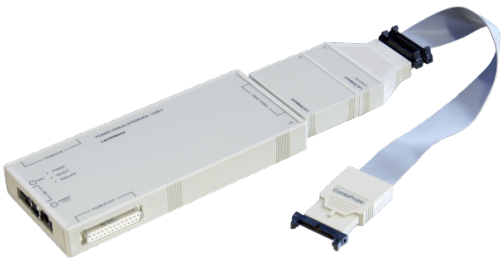

Figure 3.2 – Exemple de Debugger – Trace32

Cependant, cette plateforme pose un certain nombre de problèmes qui rend son utilisation difficile. D'une part, elle renvoie un trop grand pourcentage de faux-positifs, faisant perdre un temps considérable. D'autres part, elle ne prend pas en compte certains besoins apparu récemments Comme par exemple un système permettant de flasher automatiquement les ECU, ou la possibilité de vérifier la fréquence de mise-à-jour de la production de variables.

Afin d'améliorer cette situation, l'équipe Vérification et Validation a décidé de développer une nouvelle plateforme.

3.2 La solution : GreenT

Afin de résoudre les problèmes présentés dans la section 3.1, une solution a été pensée avant mon arrivée en étudiant les besoins de l'équipe en charge des tests du plugin : le développement d'une plateforme de tests, appelé *GreenT*.

3.2.1 Génération de tests automatiques

Les tests d'intégration du plugin Ford

À court terme, cette solution devrait permettre de tester le plugin pour les projets Ford facilement et de façon efficace. Pour cela, les testeurs Continental vont ajouter des colonnes dans le document `walkthrough`, afin de spécifier la manière de tester les variables. La plateforme, sera capable d'analyser le document `walkthrough`, et de générer les tests automatiques. Le testeur n'aura plus qu'à lancer l'exécutable le soir, il reviendra le lendemain, tous les tests auront été exécutés avec un rapport détaillé pour chaque test.

Ces tests s'effectueront sur des variables enregistrées lors de stimulation du contrôleur, afin de vérifier que celui-ci réagit de façon approprié.

Cette plateforme permettra donc de tester facilement la dizaine de projets Ford, et une fois le test d'une variable spécifiée, il n'est plus nécessaire de le réécrire. À chaque nouvelle *release* il suffira de relancer les tests : l'équipe n'aura à faire le travail qu'une fois, ensuite la réutilisation sera possible, les projets seront testés plus rapidement, plus efficacement, et plus souvent.

Les autres projets

À moyen terme, cette plateforme pourrait être utilisée pour les projets d'autres clients tel que Renault, afin d'effectuer là aussi des tests d'intégration, il est donc nécessaire de concevoir une plateforme qui puisse évoluer facilement, et avoir un fichier de spécifications en entrée qui soit légèrement différent d'un client à l'autre.

En effet, les autres clients peuvent aussi fournir une partie du logiciel, avec un document de spécification des variables, celui-ci ne serait pas totalement identique, mais l'approche des tests s'en approchera.

Il est également envisageable que la plateforme soit utilisée pour des tests d'intégration en interne, indépendamment des spécifications fournies par le client.

3.2.2 L'utilisation de GreenT comme une bibliothèque

Une autre approche de notre plateforme, serait de s'en servir pour écrire facilemet des tests en Java, de façon plus efficace et plus robuste qu'avec la TA3 : notre plateforme doit donc également fonctionner comme une bibliothèque sans utilisation de générateur ou de parser, pour que le testeur puisse effectuer un test rapide.

Celui-ci apprendra à se servir de la plateforme, écrira en règle générale des tests assez courts et moins complexes que ceux que nous générerons, ceux-ci doivent être faciles à écrire.

3.2.3 L'exécution des tests

À long terme, l'objectif serait de pouvoir effectuer de l'intégration continue.

Il faut savoir que l'exécution de ces tests pourrait durer une quinzaine d'heures, en espérant qu'elle tienne sur une nuit. Cependant, il va arriver que les tests débordent et que lorsque le testeur revient, les tests ne soient pas terminés : le banc va être occupé alors que d'autres personnes en ont besoin, et le testeur n'a toujours pas les verdicts.

Nous souhaitons concevoir un système permettant l'exécution des tests sur des bancs en parallèle : on pourrait ainsi diviser le temps d'exécution par 2 ou 3, les tests tiendront sur une nuit et l'objectif principal serait tenu.

Mais le plus intéressant, serait d'utiliser le décalage horaire à notre avantage : lancer l'exécution des tests sur des bancs non utilisés dans d'autres pays, ainsi à toute heure de la journée il serait possible de lancer les tests. En effet, Continental étant présent dans la plupart des continents, il fait toujours nuit sur un site sur la planète. Après chaque étape d'intégration, on pourrait relancer les tests afin de vérifier qu'aucun bug n'a été introduit : le problème de la disponibilité des bancs serait alors résolu, et nous atteindrons une excellente sécurité.

4 Développement de *GreenT*

Comme nous l'avons montré dans le chapitre 3, l'entreprise a besoin d'un nouvel outil aidant aux tests d'intégration : *GreenT*.

Nous allons donc voir le développement et la conception de cette plateforme de tests et plus partuclièrement le travail que j'ai effectué dans ce stage, c'est-à-dire le développement du parser, du générateur et le début de l'analyseur de trace.

Sommaire

4.1 Fonctionnement général

Lorsque je suis arrivé sur le projet, une partie de la conception était déjà réalisée, le fonctionnement général de la plateforme. Cette conception permet de répondre aux attentes du problème, tout en permettant le maximum d'évolutions facilement.

4.1.1 Le fichier Walkthrough

Le fichier Walkthrough est un fichier qui sera fourni par la personne en charge des tests, c'est un fichier au format Excel qui contient les informations de chacune des variables à tester. Il contient ainsi un très grand nombre de colonnes, bien que seule une partie des colonnes nous intéresse, certaines colonnes ont été fournies par le fournisseur du plugin, d'autres colonnes sont ajoutées dans le seul but de la génération de tests automatiques par *GreenT*. Voici les colonnes intéressantes :

Nom de la variable Le nom de la variable testée : il existe un nom court et un nom long.

Informations aidant à la conversion des données Certains *devices* tel que le debugger ne fonctionne qu'avec des valeurs Hexadécimales. À la charge de *GreenT* de convertir ces données vers des valeurs physiques exploitables par le testeur

Nécessité d'un test automatique un `GreenTTest` ne sera généré que si la colonne vaut *Yes*.

Statut du test Nous éditerons cette colonne afin de reporter le statut du test.

Precondition (cf section 4.1.2) Contient un scénario d'initialisation du *workbench* : tension de départ, lancement du debugger, ...

Scénario de stimulation (cf section 4.1.2) Contient un ou plusieurs scénarios de stimulations

`ExpectedBehavior`(comportement attendu, cf section 4.1.2) Contient une expression évaluant les variables ayant été enregistrées durant la stimulation : *GreenT* devra vérifier que cette expression est correct à toute instant de la stimulation.

Variable à enregistrer (cf section 4.1.2) Contient les variables devant être enregistrées durant un scénario, en plus des variables présentes dans l'expected behavior.

Alias locaux (cf section 4.1.2) Ce sont des alias déclarés uniquement pour le test courant.

Informations du test (cf section 4.1.2) Plusieurs colonnes tel que la sévérité, le responsable du test, les commentaires, ...

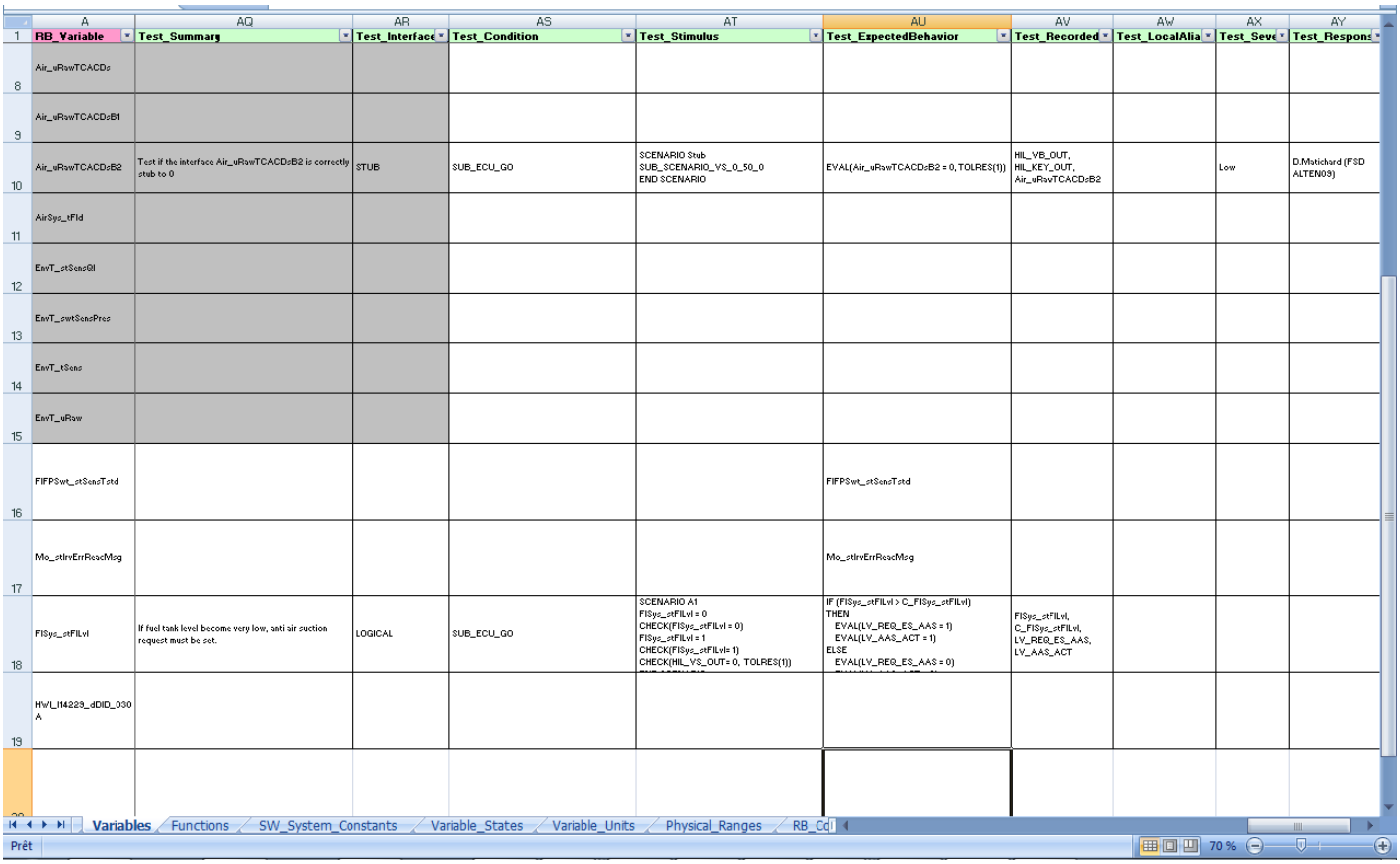

Figure 4.1 – Aperçu d'un fichier Walkthrough

4.1.2 Fonctionnalités principales

Le développement de *GreenT* va inclure un certain nombre de fonctionnalités attendues par le client et indispensable à son fonctionnement. D'autres fonctionnalités pourront apparaître plus tard en fonction des besoins.

Les principaux modules sont les suivants, avec leurs interactions schématisées figure **??** : Dans des objets ovales sont représentés des fichiers, les carrés représentent des modules de la plateforme et les flèches en pointillés un transfert réseau, les couleurs représentent les différents modules de la plateforme.

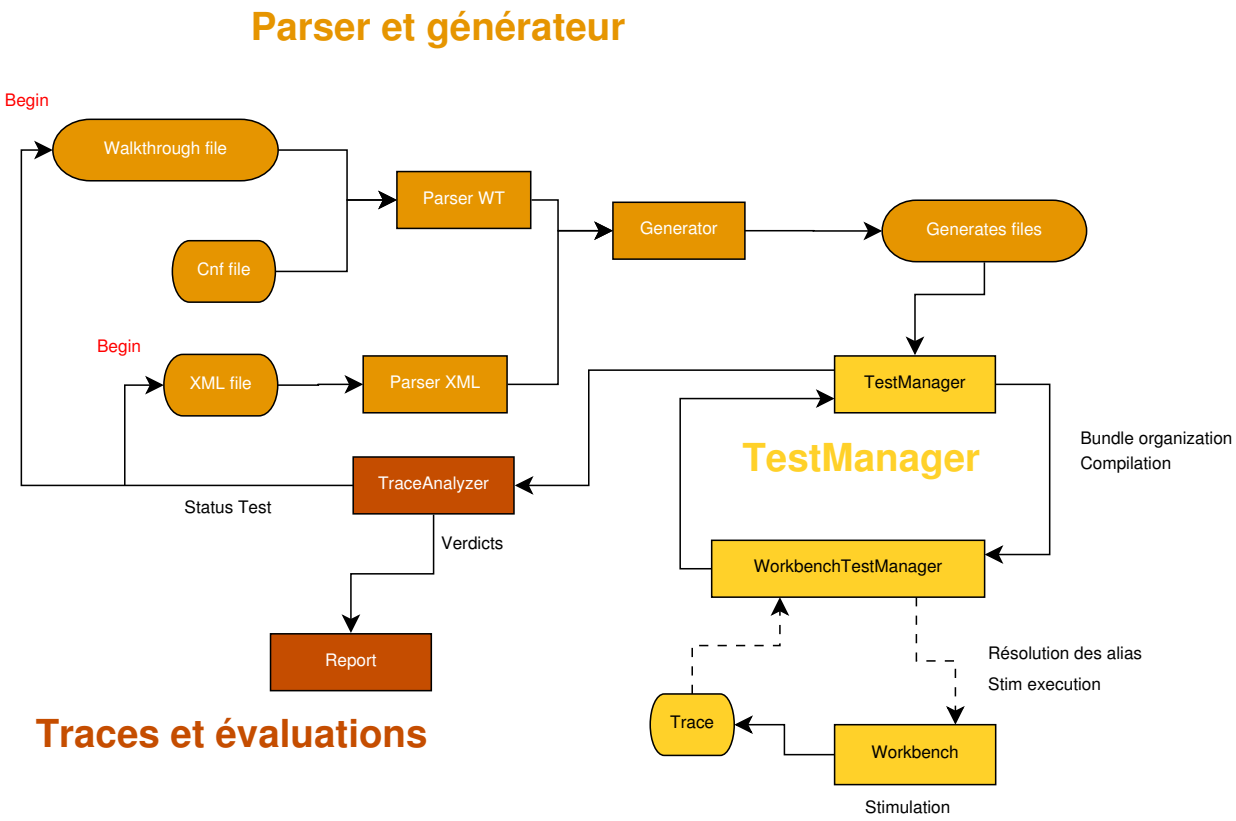

FIGURE 4.2 – Fonctionnement général de la plateforme *GreenT*

Parsing et Génération

Le but premier de la plateforme est d'effectuer des tests automatiques, il est ainsi indispensable d'avoir un système d'automatisation.

Pour cela, nous allons avoir un parser : il analysera un certain type de fichier [1] et en retirera pour chaque test, le scénario de pré condition, les différents scénarios de stimulations, leurs *Expected Behavior*, les données qui devront être enregistrées ainsi que différentes information sur le test [2].

Une fois toutes ces données acquises, il les transmettra à un générateur qui sera en charge d'écrire les fichiers Java de chaque test, tous seront organisés dans un dossier temporaire avec un dossier par test. Le `TestManager` pourra ensuite traiter ces données.

Résolution des alias

Les devices ont des fonctionnements différents pour leurs variables internes, le banc HIL (Hardware in the loop) par exemple utilise des adresses de base de données, alors que le debugger fonctionne avec des valeurs Hexadécimales permettant d'accéder directement à des cases RAM.

Il est donc indispensable de fournir au client une approche permettant de masquer ces différences, et le risque d'erreur en écrivant une adresse, pour cela nous avons pensé un système d'alias : un alias possède un nom qui permet de le lier une valeur sur le device à un nom de variable, le client n'aura ensuite plus qu'à travailler avec des noms clairs et explicites.

1. Nous ne commencerons qu'avec le Walkthrough pour débuter, mais dans le futur nous pourrions avoir des fichiers XML, des bases de données, ...
2. Responsable du test, sévérité, commentaires, nom de la variable, ...

Il y a deux types d'alias : ceux qui sont accessibles en écriture, et ceux accessibles en lecture, on pourra donc effectuer des actions différentes en fonction du type d'alias.

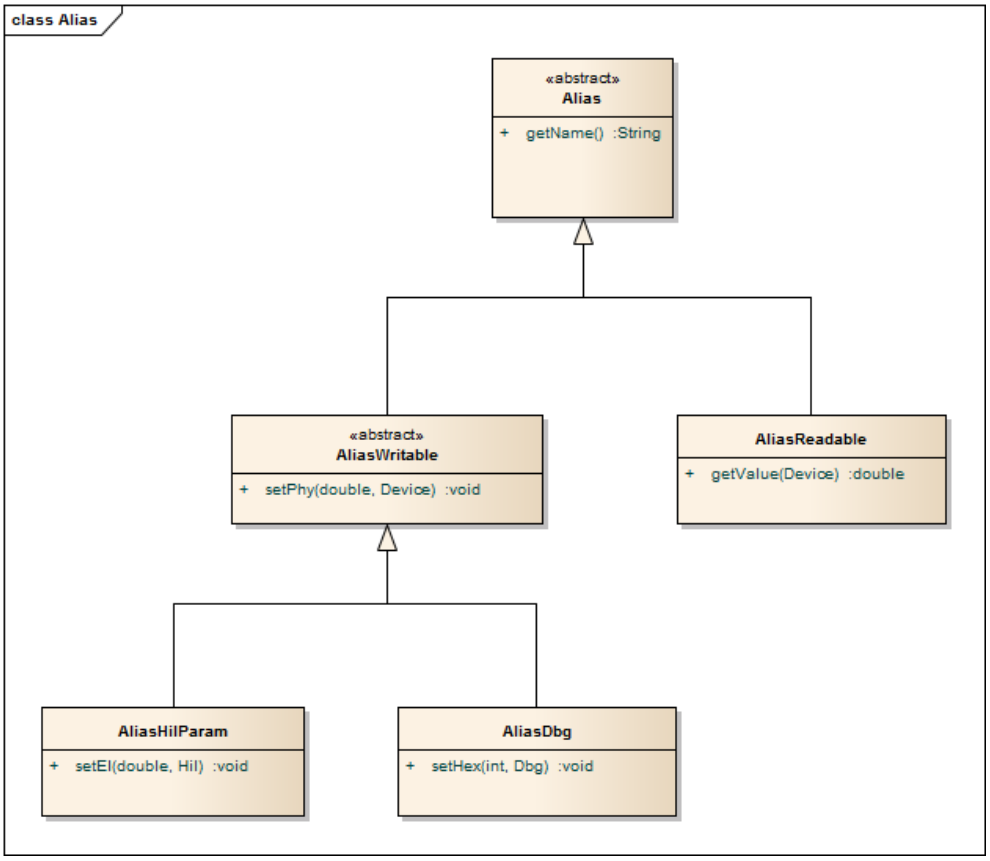

Figure 4.3 – Diagramme de classes des Alias

Les alias en écriture sont répartis en deux sous-types : les alias HIL et les alias Debugger, car ils n'ont pas les même méthodes : il est possible de faire un set hexadécimal sur un debugger, contrairement à un HIL qui lui permet d'effectuer un set électrique.

Stimulation

Afin de tester une variable du plugin, les développeurs vont utiliser des alias présents sur un device : actuellement, un HIL ou un debugger, prochainement nous pourrions en utiliser d'autres que ces deux derniers.

Le spécifieur va rédiger des scénarios de stimulation, ceci afin de mettre le contrôleur dans certaines conditions. Son but sera ensuite de vérifier que ces variables restent cohérentes vis-à-vis du scénario effectué.

Un scénario particulier doit être spécifié : une pré condition qui a pour but d'initialiser les *devices* et certains alias afin d'avoir un état de stimulation qui soit cohérent et identique à chaque lancement du scénario. Ce scénario sera effectué avant le lancement de chacun des scénarios de stimulation.

Les traces et leurs évaluations

Lorsqu'un scénario de stimulation s'exécute, un certain nombre de variables doivent être enregistrées : ces variables sont stockées sous la forme d'une trace au format CSV, qui pourra plus tard être représentée sous forme de courbe.

Une fois que la trace est complète, il est nécessaire de l'évaluer : le spécifieur a décrit le comportement attendu dans la colonne *Expected Behavior* détaillant dans quel cas le test est correct, ainsi cette expression va être transformée en arbre logique afin de l'évaluer à tout instant de la trace. Une fonctionnalité de couverture de tests sera fourni afin de s'assurer qu'un test de sévérité *High* est correctement spécifié.

Le module TestManager

Le `TestManager` est le chef d'orchestre de *GreenT*, il a donc un certain nombre de responsabilités.

Il va d'abord organiser les différents tests en un concept que nous avons appelé *Bundle* : afin de limiter le temps d'exécution qui atteindra plusieurs dizaines d'heures, il est intéressant de regrouper les tests possédant les mêmes scénarios de stimulations et les mêmes pré conditions. Seules leur *Expected Behavior* changent, mais celles-ci pourront être évaluées sur la même Trace.

Une fois les tests organisés en Bundle, il va les compiler et les donner à un `WorkbenchManager` : toujours pour une raison d'optimisation, il sera intéressant de pouvoir exécuter les enregistrements sur plusieurs bancs simultanément, pour cela le `TestManager` sera capable de savoir quels bancs peuvent être utilisés et pourra distribuer ses bundles en fonction.

Chaque `WorkbenchManager` sera en charge d'exécuter le code généré plus tôt et dialoguera en réseau avec son banc, une fois l'exécution terminée, il obtiendra une trace qui pourra être évaluée.

Afin d'être le plus souple possible, il existe plusieurs modes d'exécution du `TestManager` :

Check only Essaye de parser les différents fichiers, et vérifie que ceux-ci ne comportent aucune erreur de grammaire, d'alias introuvable, d'écriture sur un alias en lecture seule etc...

Parse and generate jar tests Parse les fichiers et génère des jars exécutables pour chacun des tests

Parse and genere bundles Parse les fichiers et génère des jars exécutables répartis en bundle

Parse and execute Parse les fichiers, génère les jars pour les bundles et les exécute : c'est le mode « classique ».

Restart test execution Redémarre une exécution qui se serait mal terminée.

Production de rapport détaillé

La plateforme aura en charge la production d'un rapport détaillé pour chaque test. Ce rapport contiendra un certain nombre d'informations, et permettra au testeur de comprendre pourquoi le test n'est pas passé. Voici les informations que contiendra ce rapport :

- Nom du test, de la variable à tester
- Nom du responsable du test
- Sévérité du test
- Pourcentage de branches de l'expectedBehavior renvoyant faux(Test « Rouge »), n'ayant pas pu être testé(Test « Gris ») et étant correct(Test Vert)
- Le testeur aura à sa disposition les expressions concernées par un résultat Rouge ou Gris.
- Les colonnes utiles du `Walkthrough`

Le format du rapport détaillé n'a pas encore été défini, mais cela pourrait être des fichiers textes, avec pour évolution une possibilité de le faire sous format Web avec affichage des courbes de trace par exemple.

Mise à jour du Walkthrough

Une fois un test exécuté, un résultat sera mis dans le fichier Excel, en fonction de la sévérité du test. En effet, un test *High* ne devra comporter aucune branche non testée contrairement à un test *Low* par exemple.

4.2 Le parser

Avant mon arrivée, le client et l'équipe avaient conçu une grammaire de langage permettant de spécifier le fonctionnement du test, afin d'effectuer un scénario de stimulation ou une *Expected Behavior* il faut respecter une syntaxe précise, dans le cas contraire une exception est levée.

4.2.1 La grammaire

Les scénarios de stimulations

Une précondition de stimulations, notée *precondstim* peut contenir des affectations, des *check* et des actions debugger :

Affectation Une affectation s'effectue de la façon suivante : `Alias = valeur`, l'alias `Alias` possédera ensuite la valeur `valeur`.

Check Cette action permet de vérifier qu'un alias possède bien la valeur espérée, à une tolérance prés. Dans le cas contraire le bundle ne pourra pas être exécuté et une exception sera levée. Cela s'effectue ainsi : `CHECK(Alias = Valeur, TOLRES(tolerance))`, la tolerance étant facultative.

Actions débugger Il est actuellement possible d'effectuer deux actions : démarrer le debugger, ou l'arrêter : `T32_GO` et `T32_CPU_STOP`.

Les scénarios de stimulations, notés *precondstim* quant-à eux ont le même fonctionnement, la seule différence est qu'ils doivent être encadrés des balises `BEGIN SCENARIO nomScenario` et `END SCENARIO`.

Le fichier cnf

Comme nous l'avons expliqué, les tests sont spécifiés dans le fichier *Walkthrough* qui est au format Excel, étant donné la difficulté d'éditer de longs texte dans la même cellule, il parait important de mettre les scénarios de stimulations dans un autre fichier : c'est l'utilité du fichier cnf, celui-ci contiendra des groupes d'actions à effectuer, spécifiés par un nom [3] : le fichier Excel pourra ainsi ne contenir que le nom du groupe d'action, et de la place sera gagnée dans la cellule.

La syntaxe du fichier est assez simple :

```
// Section for group of parameters
SECTION SUB
  SUB_ECU_GO : {
    HIL_VB = 13;
    HIL_KEY = 1;
    CHECK(HIL_KEY_OUT = 1);
```

3. Cela pourrait s'approcher du fonctionnement de sous-programmes

```
    T32_GO;
  }

  SUB_ECU_OFF : {
    T32_CPU_STOP;
    HIL_KEY = 0;
    HIL_VB = 0;
  }

  SUB_SCENARIO_VS_0_50_0 : {
    HIL_VS = 0;
    CHECK(HIL_VS_OUT = 0,  TOLRES(1)) ;
    HIL_VS = 0;
    CHECK(HIL_VS_OUT = 50,  TOLPER(1)) ;
    HIL_VS = 0;
    CHECK(HIL_VS_OUT = 0, TOLRES(1));
  }

END SUB
```

Listing 4.1 – Exemple de fichier cnf

> **R** Il y a une différence sémantique entre `TOLRES` et `TOLPER` : `TOLRES` est une tolérance en terme de valeur, alors que `TOLPER` est une tolérance en pourcentage.

Un simple appel à `SUB_ECU_GO` dans le fichier Excel fera les actions nécessaires. Cette approche permet de gagner de la place comme indiqué, mais elle permet aussi de factoriser les tests !

Les Expected Behavior

Comme expliqué précédemment, les *Expected Behavior* sont des expressions permettant d'évaluer une trace, celles-ci ont une syntaxe proche d'un langage algorithmique :

```
if tco > c1 then
  EVAL(lv_cfa = 1)
else
  if(tco < c2) then
    EVAL(lv_cfa = 0)
  else
    EVAL(lv_cfa = PRE(lv_cfa))
  endif
endif
```

Listing 4.2 – Exemple d'expected Behavior

La grammaire est récursive, il est possible d'effectuer plusieurs `Eval` à la suite : ceux-ci doivent être séparés par un retour chariot.

Le mot clé `PRE`[4] permet de récupérer la valeur précédente de la variable en paramètre, l'instruction `lv_cfa = PRE(lv_cfa)` permet de vérifier que la variable n'a pas changée.

4. Pour previous

Les alias locaux

Une autre grammaire nécessaire est celle des alias locaux : les alias déclarés uniquement pour le test courant. Un certain nombre d'informations est nécessaire : le nom de l'alias, l'adresse, et les informations permettant de convertir celui-ci en valeur physique.

```
1 Alias_name = @HEXA;tailleOctet[8000...7FFFH]
```

Listing 4.3 – Exemple de définition d'alias local

L'alias aura une valeur hexadécimale, une taille en nombre d'octet, ainsi qu'une valeur de début et une valeur de fin afin de savoir les limites de définition de cette variable.

Variables à enregistrer

Il est possible de fournir des variables à enregistrer en plus de celle présente dans l'*Expected Behavior* : cela permet d'avoir le contexte d'exécution, et de mieux comprendre la raison de l'échec d'un test. La définition de ces alias est très simple, ceux-ci doivent simplement être séparés par des virgules.

```
1 FlSys_stFlLvl, C_FlSys_stFlLvl, LV_REQ_ES_AAS, LV_AAS_ACT
```

Listing 4.4 – Exemple de définition d'alias à enregistrer

4.2.2 Implémentation du parser

La grammaire : utilisation de Antlr

J'étais en charge d'écrire le parser spécifié dans la section 4.2.1. Comme nous l'avons vu, celui-ci intègre beaucoup de grammaires différentes, notamment celle des *Expected Behavior* qui est assez complexe.
Une solution a été choisie afin d'optimiser au maximum le temps de développement et de limiter les erreurs : utilisation de Antlr [5].

Antlr est un framework libre de construction d'interpreteurs, il prend en entrée une grammaire, définissant notre langage à reconnaître et produit automatiquement le code Java permettant de reconnaître ce langage, ceci en parcourant l'arbre syntaxique. Il ne nous est donc plus nécessaire d'effectuer cette partie du travail, il faut simplement que nous effectuions les bonnes actions en fonction de notre emplacement dans l'arbre.

Antlr à une syntaxe très simple pour définir une grammaire, il est nécessaire de déclarer tous nos mots clés tels que `if`, `check`, mais aussi les caractères utilisés dans le langage tel que `(`, `)`, `;` où simplement le retour chariot [6]. Une fois que nos mots clés sont déclarés, il est possible de construire nos grammaires : pour cela, Antlr possède un outil de conception de grammaire, celui-ci a la particularité de construire le diagramme du langage en temps réel. Voici deux exemples de grammaires que j'ai rédigé :

Grammaire du Check

5. ANother Tool for Laguage Recognition
6. Celui-ci correspondant à `\n` ou `\n\r` sous Windows

```
checkFunc : Check OBracket Alias Equals (Real|Integer) (Coma (tolres|tolper))? ↩
    CBracket;
```

Listing 4.5 – Grammaire Check

Dans la partie gauche de cette expression, nous avons le nom du *token*, sur la partie droite sa définition. Les mots commençant par une majuscule sont des constantes se rapportant à un caractère, le pipe `|` signifie un «ou», et le point d'interrogation `?` signifie que l'expression le précédant est facultative, quant-au mots commençant par une minuscule, ce sont des mots se rapportant à la définition d'un autre *token*, en l'occurrence ici à la définition de `tolres` et `tolper`. Ainsi le diagramme de cette expression est le suivant :

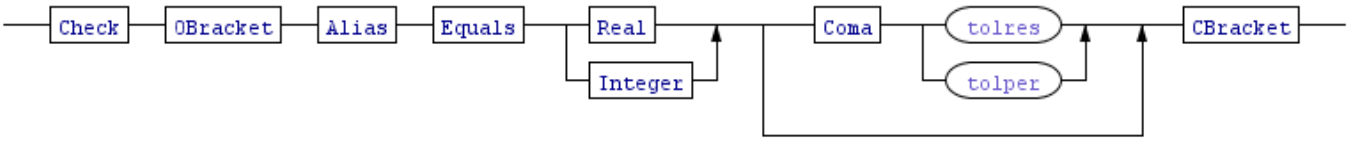

FIGURE 4.4 – Diagramme syntaxique du Check

Grammaire de l'Expected Behavior Une autre grammaire plus complexe, serait la grammaire des expected Behavior, disponible ci-dessous :

```
/* Expected Behavior
 * Represents what we can meet in an expected Behavior Cell.
 */
ExpectedBehavior: NewLine?((
                  (If OBracket comparison CBracket Then NewLine? ↩
                     ExpectedBehavior NewLine?)
                  (Else NewLine? ExpectedBehavior NewLine? Endif))
                  | eval+)
                  NewLine?;
```

Listing 4.6 – Grammaire Check

Le caractère plus `+` qui n'était pas présent précédemment représente la répétition de l'expression le précédent. Il est ainsi possible d'effectuer n'importe quelle grammaire avec ces caractères de base. Comme nous pouvons le voir, la grammaire est récursive. Son diagramme syntaxique est présent figure 4.5.

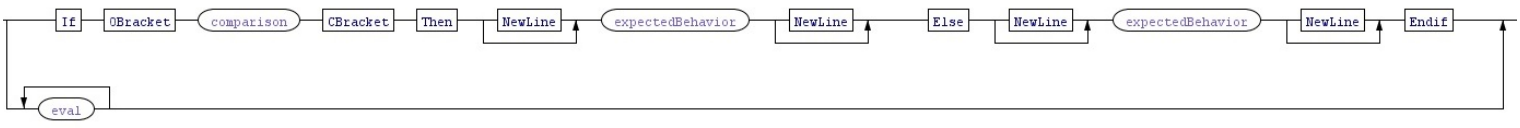

FIGURE 4.5 – Diagramme syntaxique du Check

> R Le diagramme devrait contenir des `Newline` facultatifs au début et à la fin, ceux-ci ont été omis par soucis de lisibilités du diagramme en raison de sa largeur.

Une fois toutes les grammaires construites avec l'intégralités des nœuds de l'arbre d'expression, il suffit de générer les fichiers java, et de parcourir l'arbre d'expression.

La visite de l'arbre d'expression

Antlr propose plusieurs méthodes afin de reconnaître notre langage, nous avons choisis d'utiliser les *visitors*. Le concept est assez simple : Nous devons hériter d'une classe générée par Antlr, qui est nommée WalkthroughBaseVisitor [7], c'est une classe générique nécessitant un type, notre grammaire retournera des `String`, mais dans le cas d'un parser d'expression, il peut être intéressant de retourner des `Numbers`.

Cette classe contient une méthode par token présente dans la grammaire, il est possibles de les surcharger afin d'effectuer nos actions : chacune de ces méthodes a pour convention de commencer par `visit` suivi par le nom du token. Ces méthodes sont appelées à chaque fois que nous passons dans ce nœud de l'arbre. Celles-ci contiennent en paramètre le nœud où l'on se trouve, il est donc possible de chercher des nœuds fils si-besoin, et ensuite d'appeler la suite de l'évaluation de l'arbre.

J'ai surchargé les méthodes qui étaient intéressantes ici tel que `visitCheckFunc(CheckFuncContext ctx)` présenté ci-dessous, le code est commenté afin d'aider à sa compréhension.

```
@Override
public String visitCheckFunc(CheckFuncContext ctx) {
  String aliasName = ctx.getChild(2).toString(); // The alias name to check
  if(ctx.getChildCount() > 6) { // If tolerance is present
    // It's private method who call the generator, it will be explained after
    addCheckParam(aliasName, ctx.getChild(4).toString(),
        (ctx.getChild(6).getChild(0).getText().equals("TOLRES") ? ↩
           TolType.TOLRES : TolType.TOLPER),
        ctx.getChild(6).getChild(2).getText());
  } else {
    addCheckParam(aliasName, ctx.getChild(4).toString());
  }
  // Generator too
  aliasReadableRequired.add(new AliasReadable(aliasName));

  return super.visitCheckFunc(ctx); // call next node of tree
}
```

Listing 4.7 – Surcharge de `visitCheckFunc`

La gestion des exceptions

Étant donné le temps d'exécution des tests [8], il était nécessaire d'effectuer le maximum de vérifications avant la phase d'exécution afin de ne pas avoir une erreur qui remonte au bout de plusieurs heures d'exécution.

Pour cela, nous avons utilisé plusieurs stratagèmes, tout d'abord l'utilisation d'un langage fortement typé tel que le Java, à l'opposé du Python utilisé sur la TA3, nous permet de générer du

7. Walkthrough étant le nom du fichier contenant la grammaire antlr
8. Environ une quinzaine d'heures

Java qui en cas d'erreur de type ne compilerait pas[9], mais l'autre approche était l'utilisation des exceptions lors du parsing.

Une fois le parsing de l'intégralité des variables du *Walkthrough* effectué, si au moins une ligne n'a pas pu être parsé, une exception est retournée contenant le message de toutes les erreurs trouvées durant ce parsing, toutes les lignes n'ayant pas obtenu d'erreur sont tout de même générés.

J'ai également mis en place un système de log avec `log4j` qui permet d'afficher proprement des messages de logs et de gérer la sévérité. Si une exception de parsing est renvoyée, celle-ci sera affichée dans la sortie standard, et dans le fichier de log.

Nous pouvons les dissocier en deux types d'exceptions : les erreurs de syntaxe et les erreurs d'Alias.

Erreurs de syntaxe Ces erreurs sont en partie gérées par Antlr, c'est purement syntaxique tel que l'oubli d'un caractère, cependant les erreurs Antlr n'étant pas claires, j'ai surchargé leurs méthodes d'affichage d'une part, afin d'avoir un affichage comme montré ci-dessous, mais j'ai également surchargé leurs gestionnaires afin de renvoyer une exception, ce qui n'était pas fait précédemment. L'envoi d'une exception permet de traiter le problème plus haut : c'est le `TestManager` qui la traitera.

```
line 34:10 mismatched input '0' expecting '='
HIL_VB  0;
        ^
```

Listing 4.8 – Affichage d'une erreur de syntaxe

Erreurs d'alias Ces erreurs sont des erreurs sémantiques au niveau des Alias, elles sont regroupées en 3 types différents :

Alias non connu Cela signifie que l'alias n'est connu nul part, il ne pourra donc pas être résolu lors de l'execution des tests

Alias en lecture seule L'utilisateur a essayé d'écrire sur un alias en lecture.

Alias en écriture seule Il n'est pas possible de lire la valeur d'un alias en écriture.

```
Error: Alias unknown [HL_VSd, Alias, Test]
Error: Alias not readable [HIL_VS, HIL_VB]
Error: Alias not writable [HIL_VS_OUT, HIL_VB_OUT]
```

Listing 4.9 – Affichage d'une erreur d'alias

4.3 Le générateur

Comme montré dans le schéma 4.2, une fois le walkthrough parsé, il faut générer les fichiers nécessaire au test. Pour cela, j'ai créé un *package* de classes ayant des services permettant d'ajouter les fonctionnalités à générer : Ajouter un check, ajouter une affectation, ajouter une action debugger, ajouter les informations des tests, ... et lancer la génération du fichier.

9. Comme l'écriture sur un alias en lecture

4.3.1 Génération des tests

Je dois générer 3 types de fichiers : un précondstim, des stimscenarios et un `GreenTTest` pour chaque test. Chacune des classes que je génère hérite d'une classe présente dans *GreenT*, respectivement `PrecondStim`, `StimScenario` et `GreenTTest`.

La génération des fichiers possède des points communs en fonction du type de fichier, principalement entre un `PrecondStim` et un `StimScenario`, ainsi j'ai conçu un arbre d'héritage assez simple me permettant de factoriser le code :

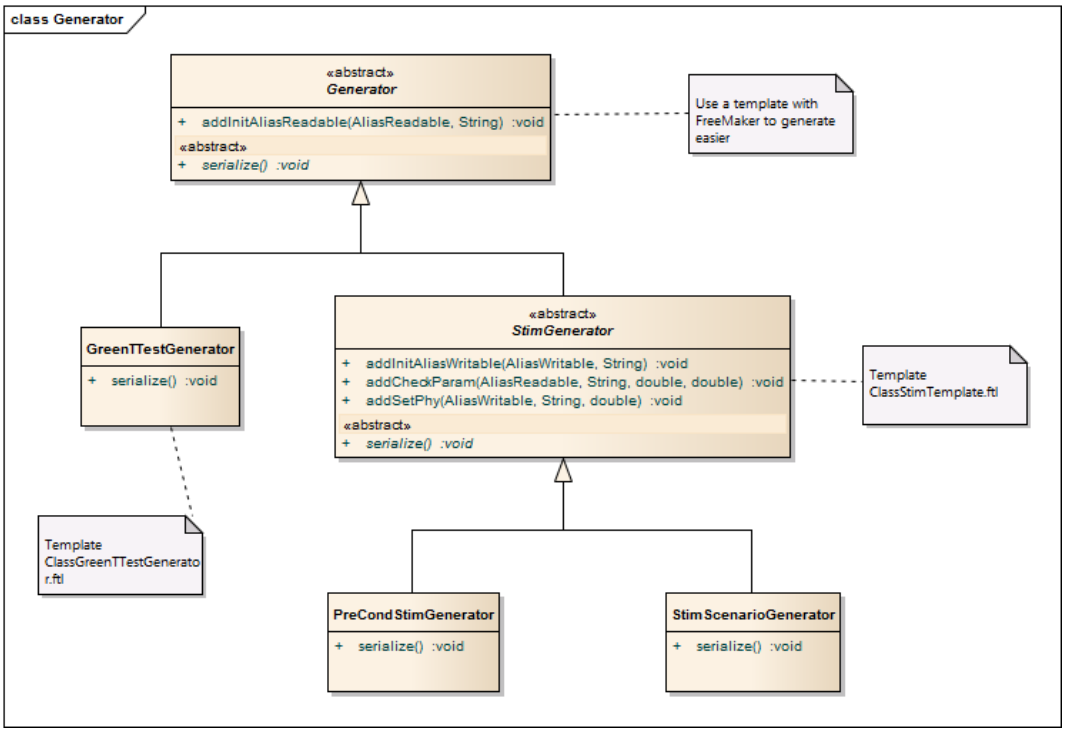

Figure 4.6 – Diagramme de classes du générateur

La méthode *serialize* écrit le code Java, avant d'appeler cette méthode il est donc nécessaire de « remplir » l'objet avec les informations nécessaires via les autres méthodes.

4.3.2 Le moteur de template : freemarker

<FreeMarker>

Les classes que je génère ont un gabarit [10] commun, seule l'implémentation des méthodes change, ainsi plutôt que réécrire systématiquement tout le corps de la classe, il parraissait intéressant d'avoir un moteur de *template*, j'ai choisi FreeMarker.

J'ai développé un fichier gabarit utilisant le format *FreeMarker*, à l'interieur de celui-ci, j'utilise des objets. Lors de la sérialisation, je fournis les valeurs des objets concernés, l'adresse du fichier *template*, le fichier `.java` est ensuite généré par FreeMarker.

Deux fichiers de templates ont été nécessaire : un pour les stim, et un pour le *GreenTTest*. En

10. Également appelé *template*

effet un precondstim et un stimscenario n'ont que peu de différence, il était donc possible de les regrouper avec le même template.

Le template des stim contient une liste d'actions à exécuter dans la méthode d'exécution, et une liste d'alias nécessaire qui doivent être ajoutée dans une méthode adéquate.
Le template des *GreenTTest* lui contient une méthode permettant de remplir le rapport et une méthode qui créé l'*Expected Behavior*. Les deux templates ont également une liste d'import qui est ajoutés automatiquement en début du fichier, ceux-ci sont disponibles dans l'annexe B suivis de deux exemples de fichiers générés annexe C.

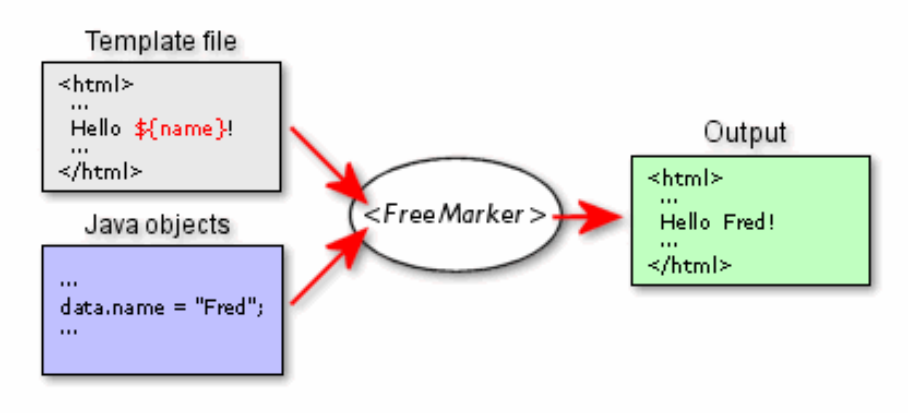

FIGURE 4.7 – Schéma de fonctionnement de FreeMarker

R Cet exemple qui vient du site officiel de FreeMarker utilise des templates HTML, pour ma part j'ai créé des templates Java, mais le principe est le même.

4.4 L'analyse des traces et la prononciation du verdict

L'analyse des traces et la prononciation des verdicts, est le cœur de *GreenT*, mais aussi une des parties les plus complexes. Comme expliqué précédemment, lors des stimulations un certain nombre de variables sont enregistrées, ensuite il faut évaluer l'*Expected Behavior* à chaque instant T de la trace, qui est représenté au format CSV.

4.4.1 Arbre de l'expectedBehavior

Nous avons choisi de représenter l'*Expected Behavior* sous la forme d'un arbre d'évaluation, cela nous permettra d'avoir plus de souplesse, en pouvant donner un verdict à une partie de l'arbre.

Chaque condition de l'*Expected behavior* peut être transformée en une *implication logique*, plusieurs evals à la suite correspondent à une conjonction logique de chacune des conditions des evals, et enfin, un else implique d'avoir eu la négation de la condition précédente. Figure 4.9, nous pouvons voir un exemple d'*Expected Behavior* et sa transformation en arbre.

L'*Expected Behavior* contiendra des variables, celles-ci pourront être modifiées à tout moment ce

qui relancera son évaluation, elle sera notifiée de cette modification à l'aide du patron de conception *Observateur*. Une variable peut être spécifique en fonction du type de la variable, ceci afin d'aider les conversions.

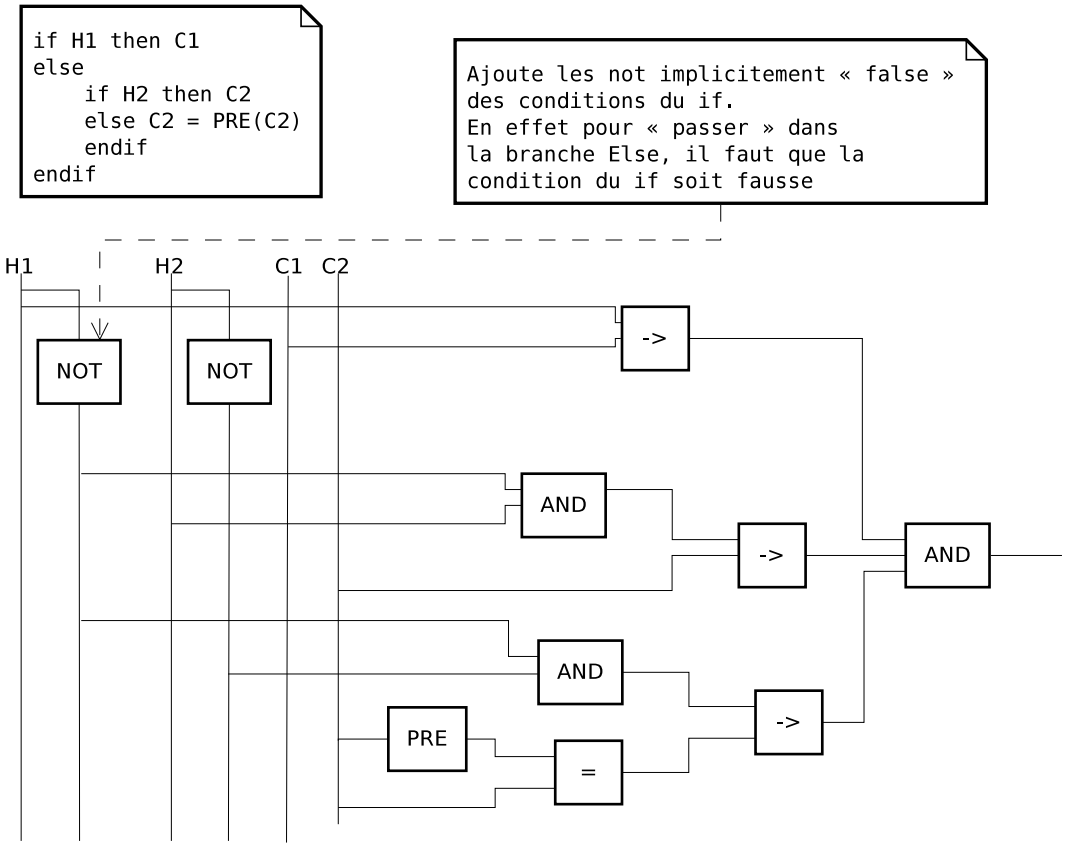

Figure 4.8 – Transformation d'une *Expected Behavior* en arbre logique

Une *Expected Behavior* peut recevoir 3 états différents à l'issue de l'évaluation de la trace :

Rouge Au moins un eval a renvoyé faux

Gris Aucun eval n'a renvoyé faux, mais au moins une branche n'a pas été testée

Vert Toutes les branches ont été testées, et tous les eval ont renvoyés vrais.

4.4.2 Analyse de la trace

Le fichier de trace au format CSV va être transformé en un dictionnaire ayant pour clé le timestamp et pour valeur la liste des variables ayant subi une modification, ainsi nous allons avoir une boucle itérant sur de dictionnaire qui à chaque itération modifiera les variables nécessaires, lors de cette modification une notification va être envoyée à l'*Expected Behavior* qui relancera l'évaluation de la trace, le `TraceAnalyzer` stockera ce résultat dans le rapport.

De cette manière, il sera possible d'analyser l'*Expected Behavior* à tout instant T de la trace.

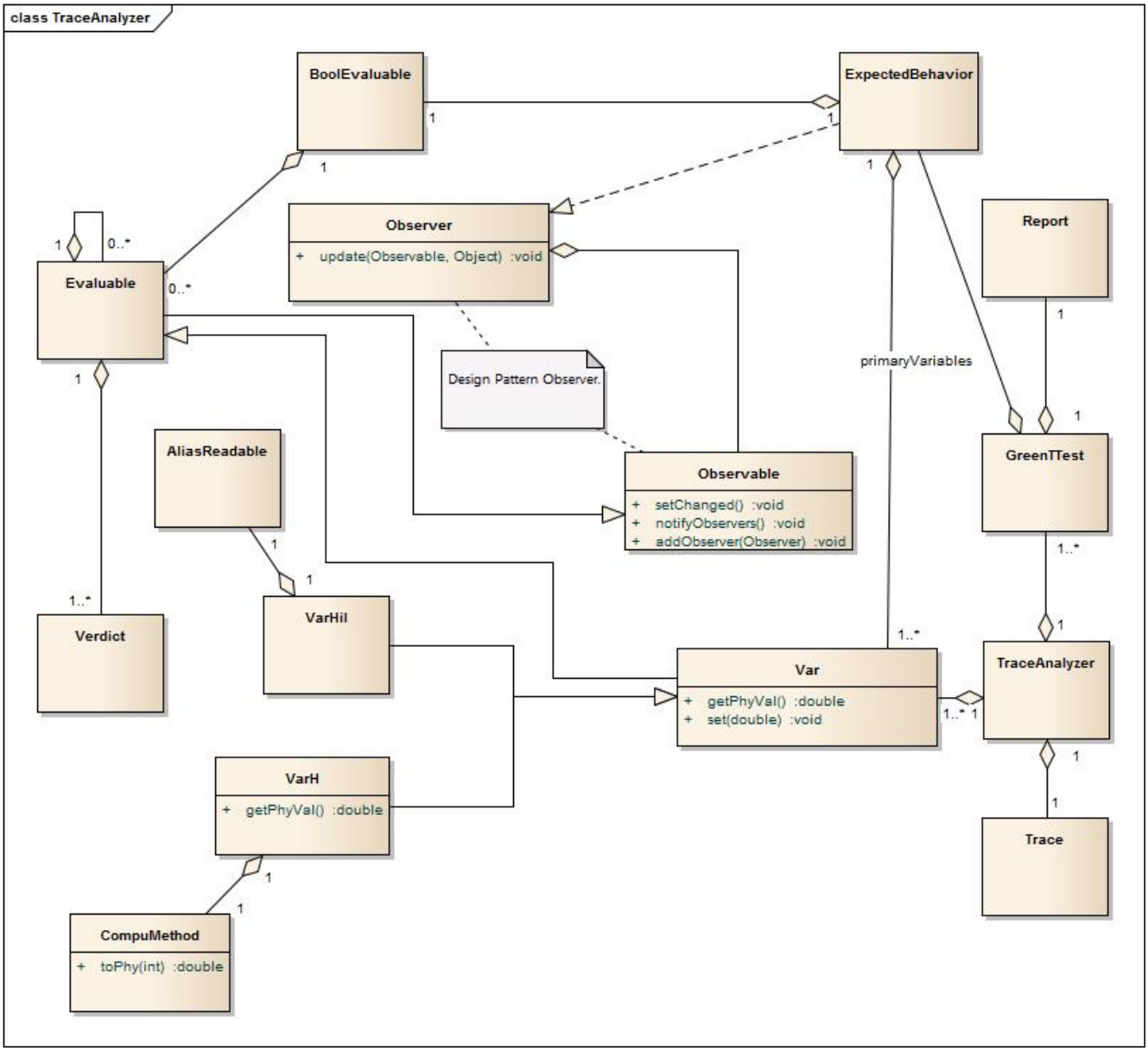

FIGURE 4.9 – Diagramme de classe de l'analyse de trace

Le nœud racine de l'arbre d'expression sera un `BoolEvaluable` possédant un verdict(Rouge, Gris, Vert), et chaque nœud de l'arbre et donc les `Variables` seront des `Evaluables`.

4.4.3 Prononciation du verdict

Chaque nœud de l'arbre d'expression est susceptible d'avoir son propre verdict : ceci afin d'avoir plus de souplesse. Dans notre cas, seul les implications correspondant à une branche de `if...else` possèdera un verdict.

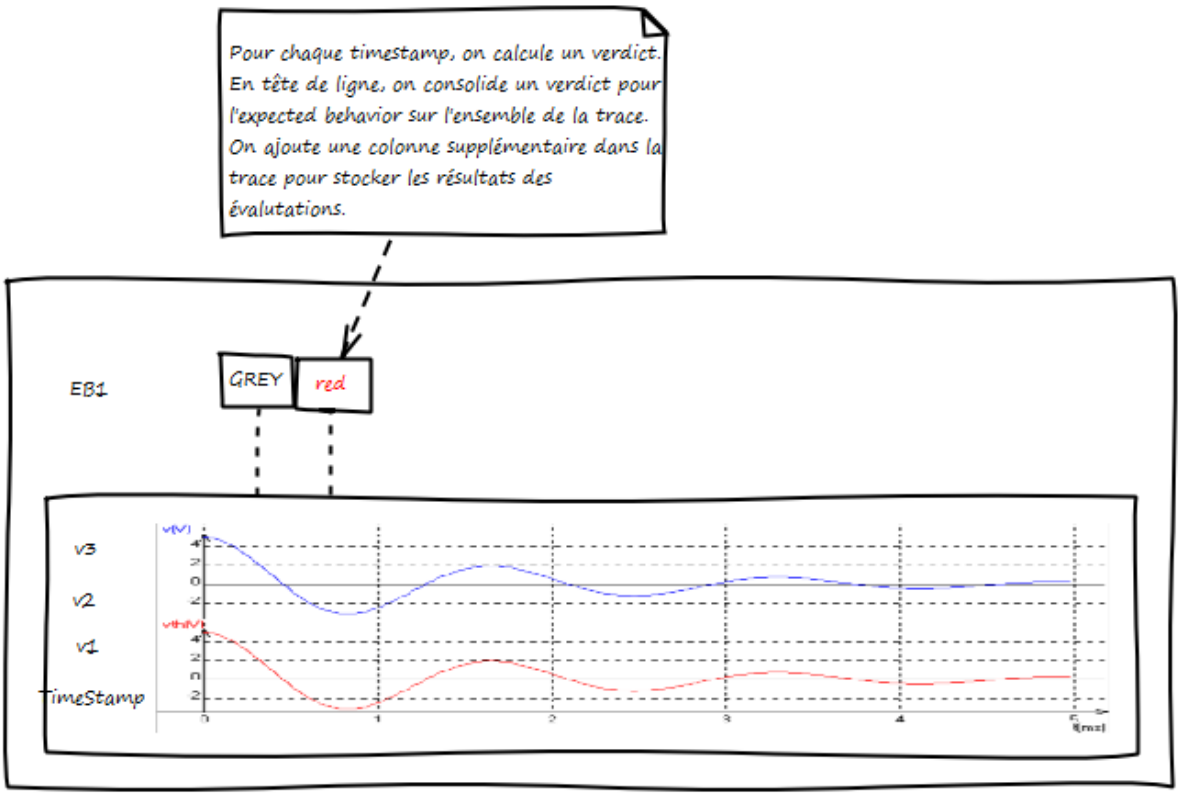

Figure 4.10 – Exemple de Trace sous forme de courbes

R À l'heure de l'écriture de ce rapport, cette partie n'a pas encore été implémentée, ayant 3 mois de stages, cela se fera durant le mois de stage supplémentaire.

5 Bilans

Bien que mon travail au sein de l'entreprise ne soit pas enore terminé, il est possible de dresser un bilan provisoire de ces deux mois de stage, tant au point de vue de l'entreprise, ce que mon travail leur a apporté, mais également en quoi ce stage m'a été bénéfique.

Sommaire

5.1 Bilan pour Continental

Mon stage va se prolonger jusqu'au 11 Juillet, ainsi le bilan du projet au moment de la rédaction de ce rapport n'est que partiel.

Mon travail dans l'équipe a été bénéfique, durant ce stage, j'ai aidé à la conception de la plateforme, notamment au niveau des Alias, des Devices, de l'analyse des traces et du parser, j'ai aussi apporté un regard neuf sur le travail déjà effectué.

Mon arrivée à apporté une personne de plus à la conception, or il est plus facile de concevoir un système facilement maintenable à 3 personnes. Au vu de nos approches différentes, concevoir un système nous satisfaisant tous les trois devrait être compréhensible pour le plus grand nombre.

J'ai développé une bonne partie de la génération, j'ai ainsi pu permettre à l'équipe de gagner du temps et de tenir au mieux les délais. À l'heure actuelle, la partie sur les *Expected Behavior* n'est pas terminée, cependant la conception étant faite cela devrait aller assez rapidement.

Je n'ai malheureusement pu tester mon travail qu'en local. Afin de pouvoir tester sur table, il faut terminer un prototype de TestManager. Celui-ci étant bien avancé et la création des traces étant presque finie, une fois l'analyse des traces fonctionnelles implémentée, nous pourrons effectuer nos premiers tests.

Le bilan pour l'entreprise semble donc être très positif, ce qui me motive pour la suite, notamment le mois de stage restant !

5.2 Bilan personnel

Cette expérience en entreprise m'a beaucoup apporté, tout d'abord d'un point de vue technique, j'ai acquis de l'expérience en conception logicielle, grâce à toutes nos réunions où nous réfléchissions à la meilleure approche possible. De plus lors de problèmes, les propositions des autres m'ont permis

d'avoir une autre vision du problème et une autre manière de le résoudre !

Mais j'ai aussi acquis des connaissances humaines avec notamment le travail en équipe, communiquer sur nos avancements, et être capable de synthétiser ses propositions ou de réussir à poser un problème rapidement tout en se faisant comprendre.

Ce stage m'a permis de découvrir le monde d'une grande multinationale. Jusqu'à maintenant je ne connaissais que les PME et je n'étais pas sûr de pouvoir travailler dans une grande entreprise.

Un bilan très positif, qui m'a réconforté dans mon projet professionnel : le développement logiciel et la conception sont vraiment les domaines de l'informatique qui m'intéressent le plus, je souhaite poursuivre vers un master Développement Logiciel.

A Acronymes et Glossaire

Antlr Another Tool for Language Recognition, outil permettant de faciliter l'interprétation d'une chaîne de caractère, celui-ci prend en entrée une grammaire, et génère un arbre syntaxique dans plusieurs langages.

Device Les différents équipements dont pourrait avoir besoin l'utilisateur : Hil, Debugger, ...

ECU Electronic Control Unit, calculateur du contrôle moteur

Grammaire Formalisme permettant de définir une syntaxe clair et non ambigüe.

HIL Hardware in the loop, permet de simuler un environnement véhicule autour du calculateur du contrôleur moteur : celui-ci réagira comme s'il était embarqué dans une voiture.

JAR Java ARchive est un fichier ZIP utilisé pour distribuer un ensemble de classes Java.

Java Langage de programmation orienté Objet soutenu par Oracle. Les exécutables Java fonctionnent sur une machine virtuelle Java et permettent d'avoir un code qui soit portable peut importe l'hôte.

JSON JavaScript Object Notation est un format de données textuelles, générique, dérivé de la notation du langage JavaScript, il permet de représenter de l'information structurée.

JVM Java Virtual Machine

Logiciel de versionnement Logiciel, tel que *Git*, permettant de maintenir facilement toutes les versions d'un logiciel, mais aussi facilitant le travail collaboratif.

Parsing Processus d'analyser de chaîne de caractère, en supposant que la chaîne respecte un certain formalisme.

Apache Thrift Langage de définition d'interface conçu pour la création et la définition de services pour de nombreux langages. Il est ainsi possible de faire communiquer deux problèmes dans deux langages différents : Python et Java dans notre cas.

Trace32 Debugger, permet de debugger un programme embarqué, ceci en permettant de lire la mémoire, mettant des points d'arrêts, ...

UML Unified Modeling Language est un langage de modélisation graphique. Il est utilisé en développement logiciel et en conception orienté Objet afin de représenter facilement un problème et sa solution.

XML Extensible Markup Language est un langage de balisage générique permettant de stocker des données textuelles sous forme d'information structurée.

B Templates

B.1 Template des stimulations

```
<#if package = "">
package com.continental.gt.generation.test;
<#else>
package com.continental.gt.generation.test.${package};
</#if>

import java.util.List;

<#if package = "">
import com.continental.gt.test.${nameClass};
<#else>
import com.continental.gt.test.${package}.${nameClass};
</#if>

import com.continental.gt.devices.Device;
import com.continental.gt.exception.CheckFailedGreenTTexception;

<#foreach import in imports>
import ${import};
</#foreach>

import org.apache.thrift.TException;

/**
 * Test of stubbed class generated
 * Generated by GreenT
 */
public class ${nameClass}_${nameTest} extends ${nameClass} {

  public ${nameClass}_${nameTest}(){
    super("Anonymous ${nameClass}_${nameTest}");
  }

  public ${nameClass}_${nameTest}(String name) {
    super(name);
  }

  @Override
  public void addAllRequiredAlias() {
    <#foreach alias in initAlias>
    ${alias};
    </#foreach>
  }
  /*
   * @see com.continental.gt.test.stim.${nameClass}#exec()
```

```
     * Generated by GreenT.
47   */
   @Override
49 public void exec(List<Device> devices) throws CheckFailedGreenTTexception {
     showMsg(".exec() : executing stimulation code of ${nameClass}_${nameTest} ↩
         class...");
51   try {
       double n;
53     Thread.sleep(500); // TODO remove me
       <#foreach device in deviceDeclarationList>
55     ${device};
       </#foreach>
57
       <#foreach exec in instructionsExec>
59     ${exec};
       </#foreach>
61   } catch (InterruptedException e) {
       e.printStackTrace();
63   } catch (TException e) {
       e.printStackTrace();
65   }

67   showMsg("... complete ok!");
   }
69
}
```

Listing B.1 – Template des stimulations

B.2 Template d'un GreenTTest

```
package com.continental.gt.generation.test;

import com.continental.gt.test.GreenTTest;
import com.continental.gt.test.report.Severity;
<#foreach import in imports>
import ${import};
</#foreach>

import org.apache.thrift.TException;

/**
 * Test of stubbed class generated
 * Generated by GreenT
 */
public class GreenTTest_${nameTest} extends GreenTTest {

  public GreenTTest_${nameTest}() {
    super("Anonymous GreenTTest_${nameTest}");
  }

  public GreenTTest_${nameTest}(String name) {
    super(name);
  }

  @Override
  public void addAllRequiredContextualData() {
    <#foreach alias in recordedVars>
    ${alias};
    </#foreach>
  }

  @Override
  public void createReport() {
    report.setVariableLongName("${variableLongName}");
    report.setVariableName("${variableName}");
    report.setResponsible("${responsible}");
    report.setSeverity(${severity});
    report.setTestSummary("${testSummary}");

    <#foreach comment in comments>
    report.addComment("${comment}");
    </#foreach>
  }

  @Override
  public void verdict() {
    // TODO Auto-generated method stub

  }
}
```

Listing B.2 – Template d'un GreenTTest

C Exemples de fichiers générés

C.1 Exemple de StimScenario

```
package com.continental.gt.generation.test.stim;

import java.util.List;

import com.continental.gt.test.stim.StimScenario;

import com.continental.gt.devices.Device;
import com.continental.gt.exception.CheckFailedGreenTTexception;

import com.continental.gt.devices.Hil;
import com.continental.gt.test.alias.AliasHilParam;
import com.continental.gt.test.alias.AliasReadable;

import org.apache.thrift.TException;

/**
 * Test of stubbed class generated
 * Generated by GreenT
 */
public class StimScenario_Stub_1 extends StimScenario {

  public StimScenario_Stub_1(){
    super("Anonymous StimScenario_Stub_1");
  }

  public StimScenario_Stub_1(String name) {
    super(name);
  }

  @Override
  public void addAllRequiredAlias() {
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
    addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
    addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
    addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_KEY"));
    addAliasWritable(Hil.class, new AliasHilParam("HIL_VB"));
  }
  /*
   * @see com.continental.gt.test.stim.StimScenario#exec()
   * Generated by GreenT.
   */
  @Override
```

```
46    public void exec(List<Device> devices) throws CheckFailedGreenTTexception {
        showMsg(".exec() : executing stimulation code of StimScenario_Stub_1 ↩
            class...");
48      try {
          double n;
50        Thread.sleep(500); // TODO remove me
          Hil hil = (Hil)getDeviceByClass(devices, Hil.class);
52        Dbg dbg = (Dbg)getDeviceByClass(devices, Dbg.class);

54        ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);

56        n = getAliasReadable(Hil.class,"HIL_VS_OUT").getValue(hil);
          if(!(n >= -0.2 && n <= 0.2)) {
58          throw new ↩
                CheckFailedGreenTTexception("CHECK(HIL_VS_OUT,0.0,TOLRES(1.0))");
          };
60        ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);

62        n = getAliasReadable(Hil.class,"HIL_VS_OUT").getValue(hil);
          if(!(n >= 49.5 && n <= 50.5)) {
64          throw new ↩
                CheckFailedGreenTTexception("CHECK(HIL_VS_OUT,50.0,TOLPER(1.0))");
          };
66        ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);

68        n = getAliasReadable(Hil.class,"HIL_VS_OUT").getValue(hil);
          if(!(n >= -0.2 && n <= 0.2)) {
70          throw new ↩
                CheckFailedGreenTTexception("CHECK(HIL_VS_OUT,0.0,TOLRES(1.0))");
          };
72        dbg.stop();
          ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_KEY"))).setPhy(0, hil);
74        ((AliasHilParam)(getAliasWritable(Hil.class, "HIL_VB"))).setPhy(0, hil);
        } catch (InterruptedException e) {
76        e.printStackTrace();
        } catch (TException e) {
78        e.printStackTrace();
        }
80
        showMsg("... complete ok!");
82    }

84  }
```

Listing C.1 – Exemple de StimScenario

C.2 Exemple de GreenTTest

```java
package com.continental.gt.generation.test;

import com.continental.gt.test.GreenTTest;
import com.continental.gt.test.report.Severity;
import com.continental.gt.test.alias.AliasReadable;

import org.apache.thrift.TException;

/**
 * Test of stubbed class generated
 * Generated by GreenT
 */
public class GreenTTest_AIRT_Air_uRawTCACDsB2 extends GreenTTest {

  public GreenTTest_AIRT_Air_uRawTCACDsB2() {
    super("Anonymous GreenTTest_AIRT_Air_uRawTCACDsB2");
  }

  public GreenTTest_AIRT_Air_uRawTCACDsB2(String name) {
    super(name);
  }

  @Override
  public void addAllRequiredContextualData() {
    addRecordedVariable(new AliasReadable("Air_uRawTCACDsB2"));
    addRecordedVariable(new AliasReadable("HIL_VB_OUT"));
    addRecordedVariable(new AliasReadable("HIL_KEY_OUT"));
  }

  @Override
  public void createReport() {
    report.setVariableLongName("Sensed value of down stream charged air ↩
       temperature (bank-2)");
    report.setVariableName("Air_uRawTCACDsB2");
    report.setResponsible("D.Matichard (FSD ALTEN09)");
    report.setSeverity(Severity.LOW);
    report.setTestSummary("Test if the interface Air_uRawTCACDsB2 is correctly ↩
       stub to 0");

    report.addComment("");
    report.addComment("");
    report.addComment("");
  }

  @Override
  public void verdict() {
    // TODO Auto-generated method stub

  }
}
```

Listing C.2 – Exemple de GreenTTest

D Liste des codes sources

E Table des figures