

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 27 mai 2014

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

En premier lieu, je remercie Stéphane BRIDE de m'avoir acceptée dans son équipe et suivi tout au long du stage.

Je remercie Alain FERNANDEZ, avec qui je travaillais pour ces excellents conseils, son suivi régulier et les excellents moments autour d'un café.

Également Olivier RAMEL, sous-traitant avec qui j'ai travaillé pour ses conseils, ses idées de technologies me facilitant la tâche, et les cafés toujours aussi agréables.

Je remercie Smail BOUAICHE pour son aide m'ayant permis de trouver un stage aussi intéressant et agréable.

Une pensée à Joelle DECOL pour sa bonne humeur quotidienne, ainsi qu'à toute l'équipe du 3ème étage m'ayant aidé à passer d'excellents moments au sein de l'entreprise.

Merci à mon tuteur universitaire Joseph BOUDOU pour son suivi et sa visite en entreprise.

Et enfin, je remercie toutes les personnes m'ayant entouré durant ce stage et aidé à travailler et rédiger ce rapport, à savoir Diane, Mathieu, Clément et Ophélie.

Introduction

Dans le cadre de ma formation en 3ème année de licence à l'université Toulouse III – Paul Sabatier, j'avais le choix entre effectuer un TER¹ ou un stage.

J'ai fait le choix d'un stage, car je suis bien plus attiré par le monde de l'entreprise que celui de la recherche, d'autant plus que j'étais à la recherche d'un stage de 3 mois me permettant de prolonger ce travail et d'avoir un sujet qui me paraissait plus intéressant.

J'ai eu la chance d'avoir une opportunité de stage dans l'entreprise Continental Automotive, afin d'effectuer du développement logiciel. J'ai été rapidement séduit par le sujet : le développement d'une plateforme de tests de logiciel embarqués. En effet, lors d'un précédent stage, j'ai travaillé dans le web, je voulais effectuer un stage dans le logiciel, cette branche du développement logiciel m'intéressant plus. Or les tests logiciels sont extrêmement importants, particulièrement dans le monde de l'automobile où une simple erreur peut être fatale.

Souhaitant travailler dans une grande entreprise afin de comparer le travail vis-à-vis d'une PME, Continental était donc le choix parfait pour mon stage.

Ainsi, le sujet m'a été présent plus en détails par mail : l'équipe Vérification et Validation doit automatiser les tests d'un plugin réalisant la plus grande partie des stratégies applicatives permettant de contrôler le moteur du véhicule cible : l'utilisateur devra donner en entrée un fichier décrivant les cas de tests, et son poste accédera à distance à des tables de tests afin d'exécuter ceux-ci.

L'automatisation de ces tests permettrait à l'équipe en charge de ceux-ci de gagner beaucoup de temps d'une part, et d'autres parts, limiterai au maximum les risques d'erreurs humaines.

Dans ce rapport, nous allons voir 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 l'équipe Vérification et Validation plus en détails, ensuite nous aborderons le problème que pose les tests de ce plugin actuellement et nous verrons ensuite la solution qui est en cours de développement, et comment j'ai contribué à ce projet.

1. Travail Etude Recherche

Table des matières

Remerciements	3
Introduction	5
1 Continental	9
1.1 Organisation de l'entreprise	9
1.2 Le contexte de l'équipe Vérification & Validation	12
2 Organisation du travail	13
2.1 L'équipe de développement	13
2.2 Documentation	13
2.3 Outils de développement	13
3 Le problème	15
3.1 Les tests	15
3.2 La solution : <i>GreenT</i>	17
4 Développement de <i>GreenT</i>	19
4.1 Fonctionnement général	19
4.2 Le parser	24
4.3 Le générateur	29
4.4 L'analyse des traces et la prononciation du verdict	31
5 Bilans	35
5.1 Bilan pour Continental	35

TABLE DES MATIÈRES

5.2 Bilan personnel	36
A Glossaire	39
B Templates	41
B.1 Template des stimulations	41
B.2 Template d'un <code>GreenTTest</code>	43
C Exemples de fichiers générés	45
C.1 Exemple de <code>StimScenario</code>	45
C.2 Exemple de <code>GreenTTest</code>	47
D Liste des codes sources	49
E Table des figures	51

1

Continental

1.1 Organisation de l'entreprise

1.1.1 Continental AG

Continental AG est une entreprise allemande dont le siège principal est à Hanovre. Il s'agit d'une Société Anonyme dont le président du comité de direction est depuis le 11 septembre 2001 Manfred Wenneber. Plus de 170.000 collaborateurs qui sont employés dans plus de 200 sites dans 45 pays appartenant à l'entreprise. En Allemagne Continental est une S.A. numéro un du marché de Production de pneu toutefois il s'agit aussi d'un équipementier automobile important.

Continental AG a été fondée en 1871 et est à nouveau membre depuis août 2003 de DAX. En 2007 elle a obtenu un chiffre d'affaires de 16 milliards d'Euros. L'entreprise est composée de 2 grands groupes auxquels sont rattachées 6 branches¹.

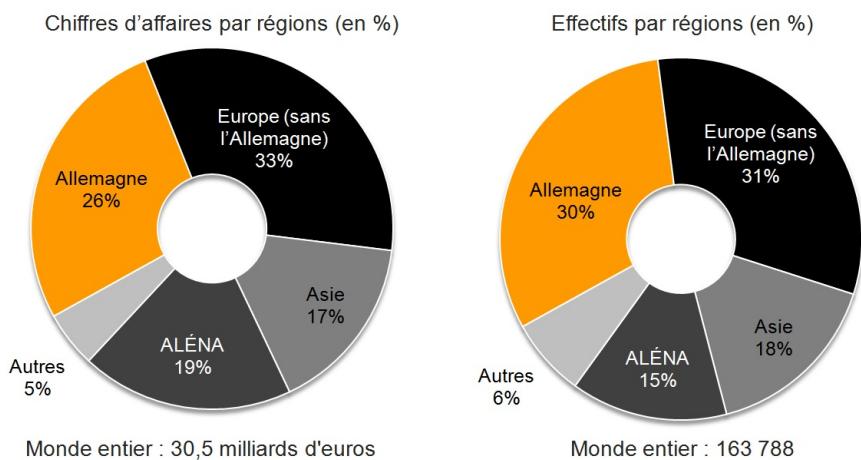


FIGURE 1.1 – Chiffre d'affaire et nombre d'employés

L'entreprise compte plus de 163000 employés dans le monde répartis dans 269 sites et 46 pays différents comme le montre la figure 1.2. (Chiffres 2011)

1. cf figure 1.4

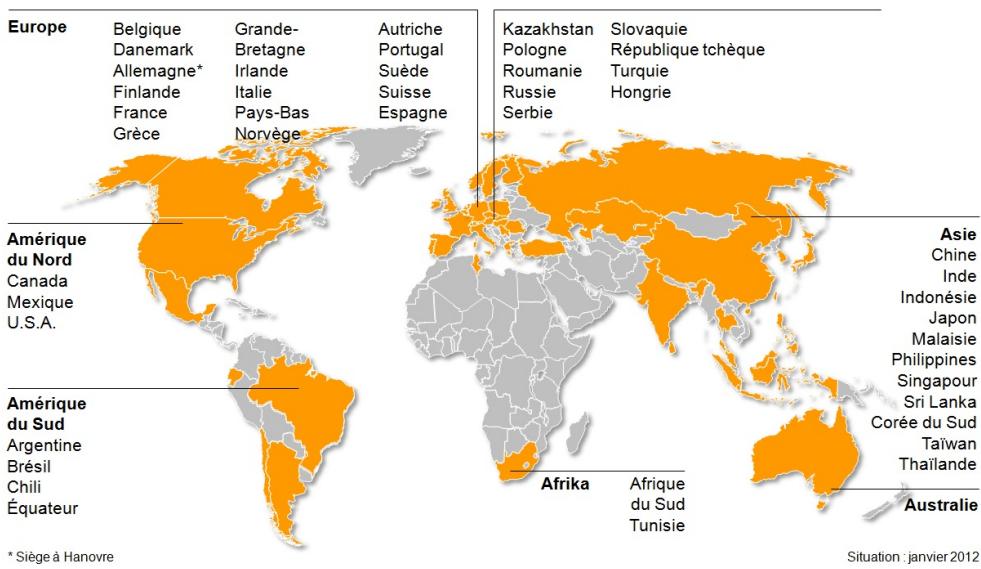


FIGURE 1.2 – Répartition du groupe continental dans le monde

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 comme marque de fabrique à l'Office impérial des brevets de Hanovre en octobre 1882. Il 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 de manière systématique.

Entre 1979 et 1985 Continental pose définitivement un pied en Europe avec le rachat l'acquisition des activités pneumatiques européennes de l'américain Uniroyal Inc. Et de la marque de pneus autrichienne Semperit.

En 1995 est créée la Division Automotive Systems pour intensifier les activités «systèmes» avec l'industrie automobile.

Pour renforcer sa position sur les marchés américain et asiatique, Continental fait l'acquisition en 2001 du spécialiste international de l'électronique Temic, qui dispose de sites de production en Amérique et en Asie. Deux autres reprises ont lieu en 2001. Continental reprend la majorité des parts de deux entreprises japonaises produisant des composants d'actionnement des freins et des 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 ContiTech.

En juillet 2007 Continental réalise son plus gros rachat sur le fournisseur automobile Siemens VDO Automotive, ce rachat à permis à l'entreprise de multiplier son chiffre d'affaire par deux : de 13 à plus de 30 milliards d'euros (chiffre 2011).

1.1.3 Activités des différentes branches

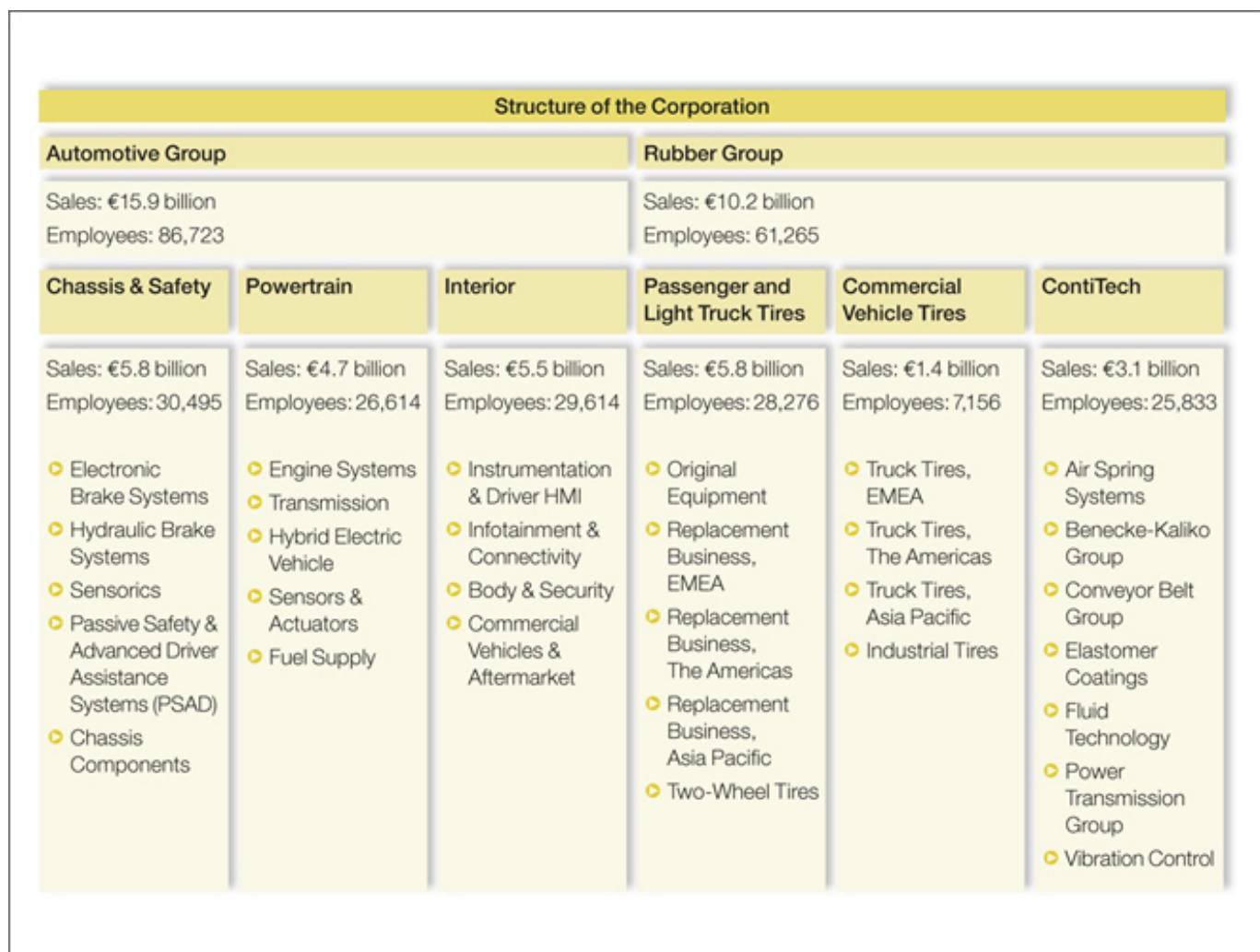


FIGURE 1.4 – Structure de continental

Comme on peut le voir sur la figure 1.4, le groupe est constitué de 6 divisions. Ces divisions sont ensuite divisées en Business Units qui ont une activité bien particulière dans leur domaine de compétence. Elles se chargent de développer et produire des équipements répondant aux besoins de nos clients.

Durant ce stage, j'étais dans la division *powertrain*. Celle-ci s'occupe essentiellement du contrôle moteur, au niveau logiciel, matériel avec l'ECU², mise au point et des systèmes diesel et essence.

2. Engine Control Unit

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.³ Cette équipe est en charge du développement, de la configuration et de l'exécution de scripts de Tests sur bancs HIL⁴, pour les tests de régression automatiques⁵, avant la livraison des projets.

1.2.2 Le besoin

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

Il est donc nécessaire d'éviter le maximum d'erreurs possibles dans les logiciels. Un logiciel ne peut pas comporter aucun bug, il est cependant possible d'éviter le maximum de bugs critiques et d'erreurs. Pour cela, il est nécessaire d'automatiser cette vérification, afin d'éviter les erreurs humaines due à l'innatention ou la répétition de.

De plus, un contrôle moteur possède un très grand nombre de cas de tests, il serait impensable de tout tester « à la main ».

C'est dans ce contexte que l'équipe Vérification & 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 est à destination de tests d'intégration.

3. 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.

4. Hardware in the Loop

5. Aussi appelés FaST : Functions and Software Testing

2

Organisation du travail

2.1 L'équipe de développement

Nous sommes 3 développeurs dans cet équipe, Alain FERNANDEZ le chef d'équipe, qui nous suivait et nous aider à planifier, organisais les réunions, tout en développant les tests managers¹. Olivier RAMEL sous-traitant travaillant chez SII, affecté à ce projet et développant particulièrement la partie Serveur. Et moi même, stagiaire, s'occupant de la partie parsing et génération². Alain et Olivier avait commencé la conception du logiciel avant mon arrivée, ils m'ont formés et expliqués tous les deux afin que je puiss rapidement les aider.

Peu de temps après mon arrivé, nous avons fixés une réunion hebdomadaire tous les lundis matins afin d'expliquer nos différents avancements ou problèmes, ce qui permettait de se projeter, et de régler les différents problèmes. Cela ne nous a pas empêché d'effectuer un certain nombre de réunions ponctuels pour continuer les points de conceptions n'ayant pas été terminés, ou les différents problèmes que nous avons rencontrés durant le développement.

2.2 Documentation

Lorsque nous fixions des choix de conceptions ou des choix stratégiques vis-à-vis du client, nous notions tout dans un document au format Word. Ainsi toutes les traces de nos réunions et de notre conception était accessible sur un disque réseau ce qui permet à l'équipe de lire ce qu'on avait dis plusieurs semaines avant, et de tenir au courant les autres.

2.3 Outils de développement

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

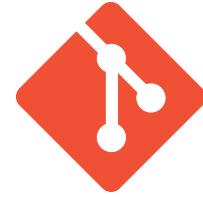
La partie client de notre plateforme est développée en Java à sa version 6, Java nous permettant d'avoir un langage fortement typé, très puissant au niveau du paradigme Objet, connu de l'équipe, assez simple de déploiement et multiplateforme.



Les postes de continental possédant pour la pluspart Java 6, aucune fonctionnalités ultérieur à cette version n'a été utilisé.

1. Cf 4.1.2
2. Cf 4.1.2

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 que nous effectuons une modification, nous faisons un « commit », dès lors un point de restauration se crée : il est possible de récupérer n'importe quelle version de logiciel depuis son commencement. Nous y insérions un message clair expliquant ce que l'on a fait, cela pouvait permettre aux autres développeurs de l'équipe de se tenir au courant de l'avancement.



Nous développions tous sous l'IDE³ Eclipse Kepler, avec le plugin Git et le plugin Python. Le plugin Git pour avoir des outils aidant à résoudre les conflits éventuels que nous avons rencontrés, et le plugin Python permet de développer avec l'interpréteur et la coloration syntaxique Python. Je ne m'en suis que rarement servi, mais il était indispensable pour développer la partie serveur de notre plateforme, qui fonctionne en Python.

Nous n'avons pas utilisé d'autres plugins, et restons avec une version simple nous permettant de développer facilement et sans problème.

Nous avons travaillé avec la norme UML⁴ 2 afin de concevoir la plateforme, en utilisant particulièrement des diagrammes de classes, mais aussi des diagrammes de cas d'utilisations ou d'activité.



Pour dessiner ces diagrammes, et les noter dans la documentation, nous les pensions d'abord sur tableau blanc, mais ensuite il nous fallait un outil puissant afin de les dessiner sur informatique. Pour cela nous avons utilisés Enterprise Architect, un logiciel propriétaire payant permettant de créer tous les diagrammes de la norme UML2.



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, celles-ci étant réalisé automatiquement par le système une fois un style définis.

3. Integrated Development Tools
4. Unified Modelling Language

3

Le problème

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 calculateur, en effet une partie est fournie par le client sous forme de « plugin ». Le plugin est supposé correct, et ce n'est pas du ressort de continental 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 fournit un fichier appelé *Walkthrough*¹ contenant la liste des variables du plugin avec toutes leur spécifications, ce fichier est au format Excel : environ 900 variables différentes. Il est impensable de tester le fonctionnement d'autant de paramètres « à la main », ainsi l'équipe en charge de tester cette intégration effectue des tests de différence d'une version à l'autre : seul les variables ayant pu être impacté par une *release* seront testés, il est supposé que les autres resteront inchangées.

Trois problèmes se posent à cette méthode :

La fiabilité des tests Rien ne nous garantie qu'une variable ne va pas être impacté par une modification, un problème peut donc passer à côtés du test.

Le temps de tests Même en ne testant qu'une partie des variables, cela prend un temps considérable, il faut compter environ une semaine pour tester ce qu'il faut, sans compter le fait qu'une tâche répétitive peut entraîner des erreurs d'inattention.

La disponibilité des bancs de tests Les tests s'effectuent sur des bancs de tests², ces équipements permettent de simuler un environnement voiture autour du contrôleur moteur³. Or ces bancs de tests sont peut nombreux dans l'entreprise, en raison de leurs couts. Il est donc nécessaire de réserver les bancs pendant une semaine afin d'effectuer les tests, cela peut bloquer d'autres personnes en ayant besoin.

1. Ce fichier est expliqué plus en détail section 4.1.1

2. Une photo d'un banc est disponible figure 3.2

3. tel que l'utilisation de la clé de démarrage, la tension de la batterie, la vitesse de rotation du moteur, ...



FIGURE 3.1 – Exemple de banc de tests à Continental – HIL DSpace

3.1.2 La TA3

Actuellement, afin de tester, les équipes de tests disposent d'une plateforme appelé TA3. Cette plateforme leur permet de dialoguer avec des contrôleurs embarqué dans des simulateurs d'environnement⁴ et le debugger, et ainsi d'effectuer des scripts de tests en Python.



FIGURE 3.2 – Exemple de Debugger à continental – Trace32

4. Hardware In the Loop (HIL) par exemple

Cependant, cette plateforme pose un certain nombre de problème qui rends sont utilisation difficile.

Problème de temps réel Un certain nombre de calculs à des fins de tests sont effectués par le calculateur, ainsi les tests peuvent être faussés car le calculateur n'est pas en conditions réels mais utilise une partie de sa charge afin de tester.

Mauvaise gestion des exceptions Un autre problème récurrent, est la gestion des exceptions et erreurs de scripts. Cette plateforme utilise python et est donc interprété. Certaines erreurs de syntaxe ne sont détectées qu'à l'exécution : éventuellement plusieurs minutes après avoir écrit le test. D'autre part, certaines erreurs sont inexplicables

Bugs Enfin, celle-ci est buggé, il peut arriver qu'elle renvoi des faux positifs, faisant perdre un temps considérable

Il est donc extrêmement difficile de tester dans ces conditions, et celles-ci ayant été mal conçue, il est compliqué de la modifier ou de la faire évoluer. En effet, le résultat serait une plateforme toujours instable, ou alors passer énormément de temps pour un résultat qui ne serait pas parfait : une nouvelle solution partant de zéro afin d'effectuer des tests fiables, simplement et efficacement semble nécessaire.

3.2 La solution : *GreenT*

Afin de résoudre les problèmes vus section 3.1, une solution à été pensé 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

A 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. Et la plateforme, sera capable d'analyser le 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 enregistrés 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 spécifié, 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

A moyen terme, cette plateforme pourrait être utilisé 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 spécifications 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 fournis par le client.

3.2.2 L'utilisation de GreenT comme une bibliothèque

Une autre approche de notre plateforme, serait de s'en servir afin de facilement écrire ses propres 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, à la charge du testeur d'effectuer un test rapide.

Celui-ci apprendra à se servir de la plateforme, écrira en règle général 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

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

Il faut savoir que le lancement de ces tests pourrait durer une quinzaine d'heures, en espérant que l'exécution des tests 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, ainsi 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 à un site sur la planète. Ainsi, 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.

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 fournis 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 seul une partie des colonnes nous intéressent, 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 testé : 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 (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, ...

A	AQ	AR	AS	AT	AU	AV	AW	AX	AY
RB_Variable	Test_Summary	Test_Interface	Test_Condition	Test_Stimulus	Test_ExpectedBehavior	Test_Recorded	Test_LocalAlias	Test_Seve	Test_Response
Air_uRowTCACD0s									
Air_uRowTCACD0B1									
Air_uRowTCACD0B2	Test if the interface Air_uRowTCACD0B2 is correctly stub to 0	STUB	SUB_ECU_GO	SCENARIO Sub-SUB_SCENARIO_VS_0_50_0 END SCENARIO	EVAL(Air_uRowTCACD0B2 = 0, TOLRES(1))	HIL_VS_OUT, HIL_KEY_OUT, Air_uRowTCACD0B2		Low	D.Matichard (FSO ALTER09)
AirSys_uFil0									
EevT_outSendGL									
EevT_outSendPres									
EevT_outSens									
EevT_uRow									
FIFPSwt_outSendTstd					FIFPSwt_outSendTstd				
Mo_stinErrReadMsg					Mo_stinErrReadMsg				
FISys_cstFilvl	If fuel tank level become very low, anti air suction request must be set.	LOGICAL	SUB_ECU_GO	SCENARIO A1 FISys_cstFilvl = 0 CHECK(FISys_cstFilvl = 0) FISys_cstFilvl = 1 CHECK(FISys_cstFilvl = 1) CHECK(HIL_VS_OUT = 0, TOLRES(1))	IF (FISys_cstFilvl > C_FISys_cstFilvl) THEN EVALLV_REQ_ES_AAS = 1 EVALLV_AAS_ACT = 1 ELSE EVALLV_REQ_ES_AAS = 0	FISys_cstFilvl, C_FISys_cstFilvl, LV_REQ_ES_AAS, LV_AAS_ACT			
HwLJ4223_dDID_030A									

FIGURE 4.1 – Aperçu d'un fichier Walkthrough

4.1.2 Fonctionnalités principales

Le développement de *GreenT* va inclure un certains nombre de fonctionnalités attendu par le client et indispensable à son fonctionnement. D'autres fonctionnalités pourront apparaître plus tard, ainsi les fonctionnalités développées pourront être adaptées.

L'interaction entre les différents modules de *GreenT* est schématisée figure 4.3

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 enregistrés ainsi que les différentes données du test ².

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, ...

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 HIL³ 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 pensés un système d'alias : un alias possède un nom qui permet de 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.

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.

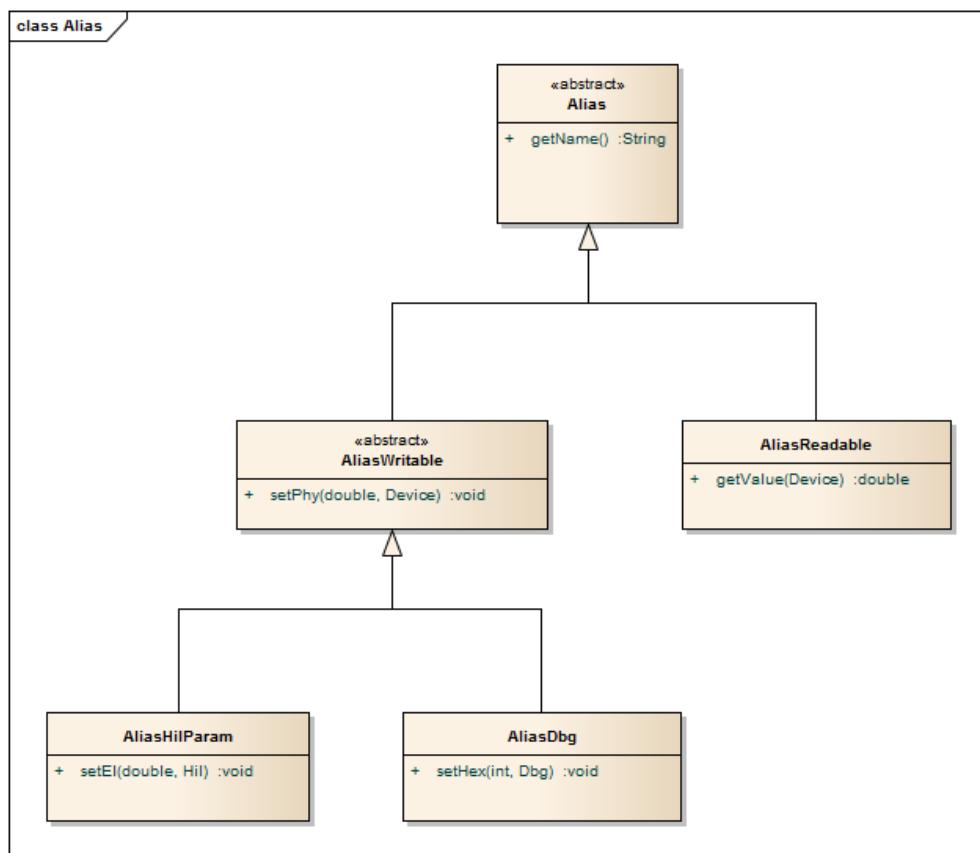


FIGURE 4.2 – Diagramme de classes des Alias

Les alias en écriture sont répartis en 2 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.

3. Hardware in the loop

Stimulation

Un test possèdera tout d'abord un scénario de pré condition : celui-ci est nécessaire afin d'initialiser les devices, en initialisant des valeurs, démarrant le debugger, etc... Ceci afin que les résultats des tests soit cohérents.

Celui-ci possédera également un certain nombre de scénarios de stimulations. Ceux-ci sont des scénarios destinés à effectuer des actions pour mettre le contrôleur en condition, afin de vérifier que le fonctionnement est bien celui attendu. Durant les scénarios de stimulations, des variables sont enregistrés, c'est une fois les stimulations exécutés que l'on peut vérifier que tout s'est passé correctement. Avant chaque execution de stimulation, le scénario de pré-condition est lancée afin de retourner dans un état cohérent.

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 représenté sous forme de courbe.

Une fois que la trace est complète, il est nécessaire de l'évaluer : le spécifieur a fournis une *Expected Behavior* détaillant dans quel cas le test est correct, ainsi cette expression va être transformé en arbre logique afin de l'évaluer à tout instant T de la trace. Une fonctionnalité de couverture de tests sera fournis afin de s'assurer qu'un test de sévérité *High* est correctement spécifié.

Le 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 appellés *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 : seuls leur *Expected Behavior* change, mais celles-ci pourront donc être évalués 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 existera plusieurs modes d'exécution du *TestManager* :

Check only Essaye de parser les différents fichiers, et vérifie que ceux-ci ne comporte 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 faux⁴, n'ayant pas pu être testé⁵ et étant correct⁶
- Le testeur aura à sa disposition les expressions concernés 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.

Schéma de fonctionnement global

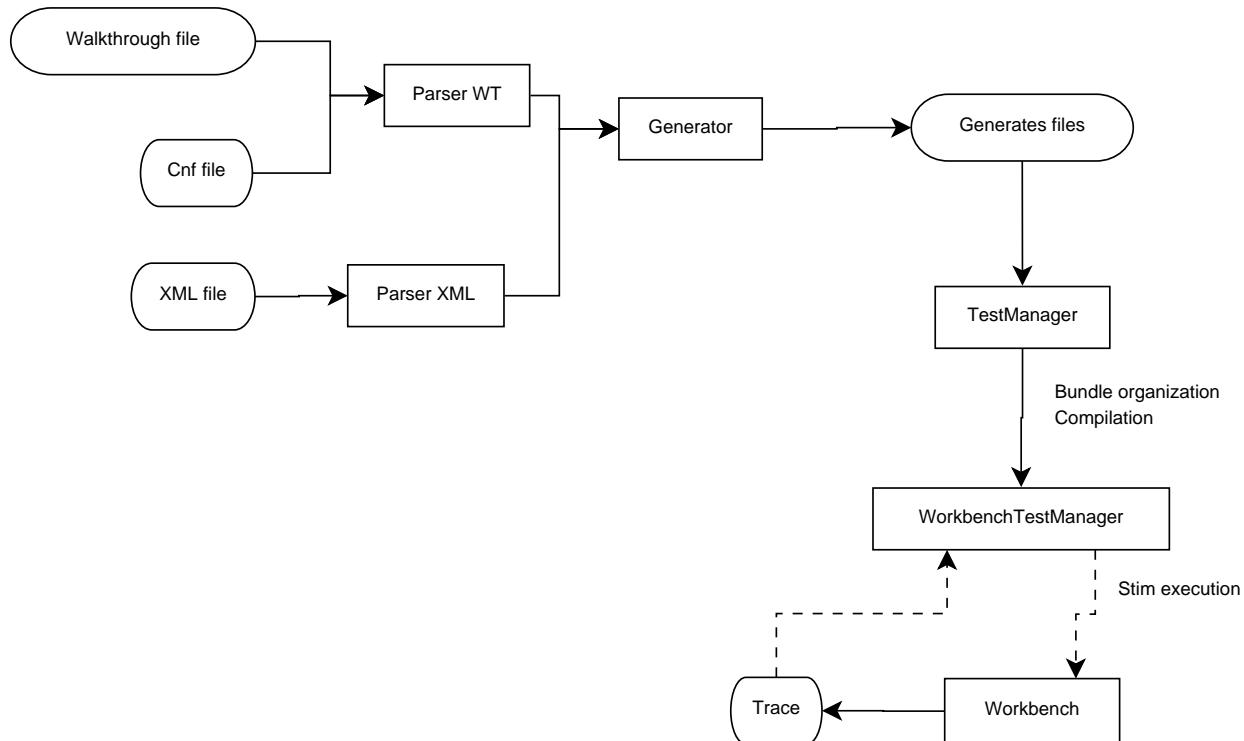


FIGURE 4.3 – Fonctionnement général du parser *GreenT*

4. Test « Rouge »
5. Test « Gris »
6. Test Vert

Dans ce schéma nous pouvons voir les différents modules détaillés précédemment : un type de parser prend en entrée un fichier, qu'il analyse, génère les fichiers Java, les donne au **TestManager**, qui les organise en *Bundle*, les compile, donne les binaires au **WorkbenchTestManager**, qui dialogue directement avec le banc, une fois le bundle exécuté, celui-ci retourne la trace. Il ne reste plus qu'à l'analyser et prononcer un verdict.

Dans des objets ovales sont représentés des fichiers, les carrés représentent des modules de la plateforme et les flèches en pointillés un transfert réseau.

Pour ma part, je me suis occupé de la partie du parser, du générateur, et j'ai aidé à la conception du système d'évaluation et d'analyses des traces.

4.2 Le parser

Avant mon arrivée, le client et l'équipe avait conçu une grammaire de langage permettant de spécifier le fonctionnement du test, afin d'effectuer un scénario de stimulation où 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

Un *precondstim* peut contenir des affectations, des *check* et des actions débugger :

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 près. 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_G0** et **T32_CPU_STOP**.

Les scénarios de stimulations 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 textes dans la même cellule, il paraît 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 nom⁷ : 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 :

```
1 // Section for group of parameters
SECTION SUB
3   SUB_ECU_GO : {
```

7. Cela pourra s'approcher du fonctionnement de sous-programmes

```

5   HIL_VB = 13;
6   HIL_KEY = 1;
7   CHECK(HIL_KEY_OUT = 1);
8   T32_G0;
9 }

10  SUB_ECU_OFF : {
11    T32_CPU_STOP;
12    HIL_KEY = 0;
13    HIL_VB = 0;
14 }

15  SUB_SCENARIO_VS_0_50_0 : {
16    HIL_VS = 0;
17    CHECK(HIL_VS_OUT = 0, TOLRES(1)) ;
18    HIL_VS = 0;
19    CHECK(HIL_VS_OUT = 50, TOLPER(1)) ;
20    HIL_VS = 0;
21    CHECK(HIL_VS_OUT = 0, TOLRES(1));
22 }

23 END SUB

```

Listing 4.1 – Exemple de fichier cnf



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 indiquée, 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 :

```

1 if tco > c1 then
2   EVAL(lv_cfa = 1)
3 else
4   if(tco < c2) then
5     EVAL(lv_cfa = 0)
6   else
7     EVAL(lv_cfa = PRE(lv_cfa))
8   endif
9 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**⁸ 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 changé.

8. 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 | F1Sys_stF1Lvl, C_F1Sys_stF1Lvl, 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. 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 Antlr⁹.



Antlr est un framework libre de construction de compilateurs, 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 effectuons 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 clé tel que `if`, `check`, mais aussi les caractères utilisés dans le langage tel que `(`, `)`, `,`, `;` où simplement le retour chariot¹⁰. 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

9. ANother Tool for Laguage Recognition

10. 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 précédent est facultative, quant-àu mots commençant par une majuscule, 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 :

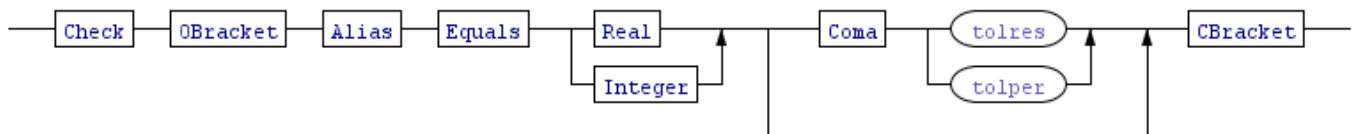


FIGURE 4.4 – Diagramme syntaxique du Check

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 quel grammaire avec ces caractères de base. Comme nous pouvons le voir, la grammaire est récursive. Son diagramme syntaxique est présent figure 4.5.

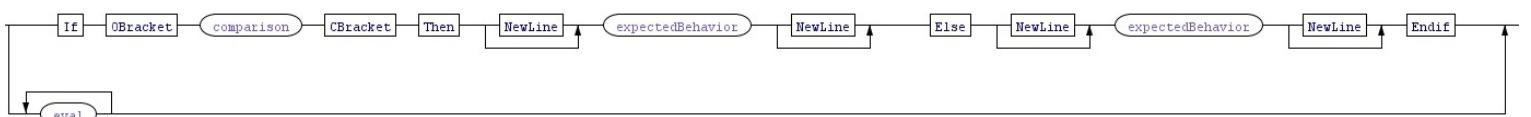


FIGURE 4.5 – Diagramme syntaxique du Check



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 noeuds 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`¹¹, 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 ont pour convention de commencer par `visit` suivis par le nom du token. Ces méthodes sont appelés à chaque fois que nous passons dans ce noeud de l'arbre. Celles-ci contiennent en paramètre le noeud où l'on se trouve, il est donc possible de chercher des noeuds fils si-besoin, et ensuite d'appeler la suite de l'évaluation de l'arbre.

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

```

2   @Override
3   public String visitCheckFunc(CheckFuncContext ctx) {
4       String aliasName = ctx.getChild(2).toString(); // The alias name to check
5       if(ctx.getChildCount() > 6) { // If tolerance is present
6           // It's private method who call the generator, it will be explained after
7           addCheckParam(aliasName, ctx.getChild(4).toString(),
8                           (ctx.getChild(6).getChild(0).getText().equals("TOLRES") ? ←
9                             TolType.TOLRES : TolType.TOLPER),
10                          ctx.getChild(6).getChild(2).getText());
11     } else {
12         addCheckParam(aliasName, ctx.getChild(4).toString());
13     }
14     // Generator too
15     aliasReadableRequired.add(new AliasReadable(aliasName));
16
17     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 tests¹², il était nécessaire d'effectuer le maximum de vérification avant la phase d'exécution afin de ne pas avoir une erreur qui remonte au bout de plusieurs heures d'exécutions.

Pour cela, nous avons utilisés 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 Java qui en cas d'erreur de type ne compilerait pas¹³, mais l'autre approche était l'utilisation des

11. Walkthrough étant le nom du fichier contenant la grammaire antlr

12. Environ une quinzaine d'heures

13. Comme l'écriture sur un alias en lecture

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 retourné contenant le message de toutes les erreurs trouvés durant ce parsing, toutes les lignes n'ayant pas obtenu d'erreur sont tout de même générées.

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 affiché 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 clair, 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'envoie d'une exception permet de traiter le problème plus haut.

```
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'exécution 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 4.3, une fois le walkthrough parsé, il faut générer les fichiers nécessaires 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.

4.3.1 Génération des tests

Je dois générer 3 types de fichiers : un `precondstim`, des `stimsenarios` et un `GreenTTest` pour chaque test. Chacune des classes que je génères hérite de 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 :

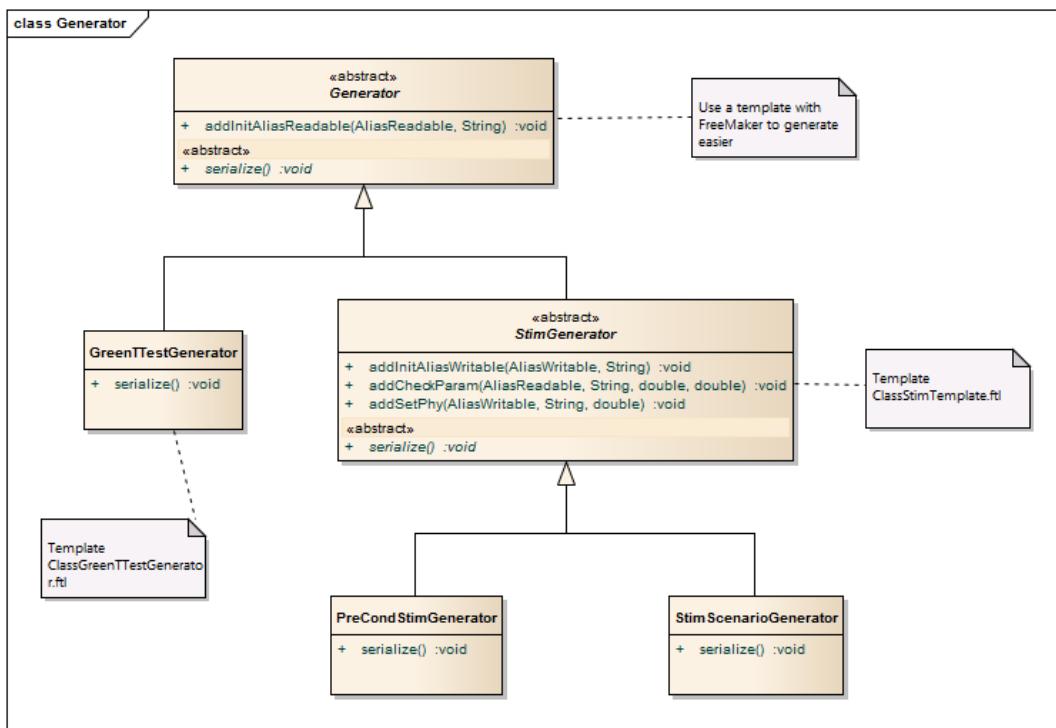


FIGURE 4.6 – Diagramme de classes du générateur

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 génères ont un gabarit commun, seul l'implémentation des méthodes changent, ainsi plutôt que réécrire systématiquement tout le corps de la classe, il paraissait intéressant d'avoir un moteur de *template*, j'ai choisi FreeMarker.

J'ai un fichier gabarit utilisant le format FreeMarker, avec dedans des appels d'objet, lors de la sérialisation, je fournis à FreeMarker les objets concernés, l'adresse du gabarit et c'est lui qui écrira le fichier `.java`.

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

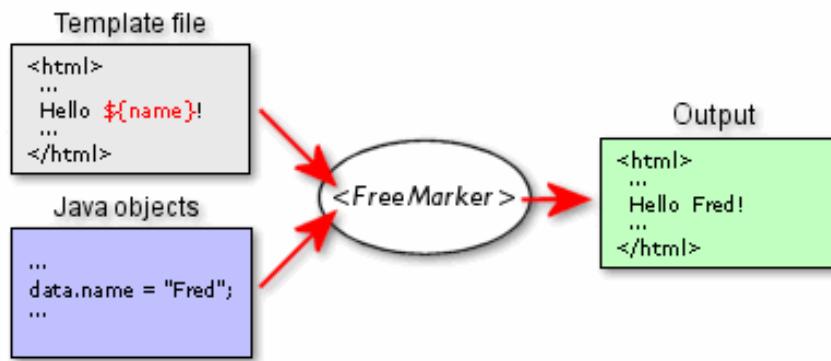


FIGURE 4.7 – Schéma de fonctionnement de FreeMarker



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, et 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 choisis 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 conditions de l'*Expected behavior* peut être transformée en une *implication logique*, plusieurs evals à la suite correspondent à un *et 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 notifié de cette modification à l'aide du design pattern *Observer*. Une variable peut être spécifique en fonction du type de la variable, ceci afin d'aider les conversions.

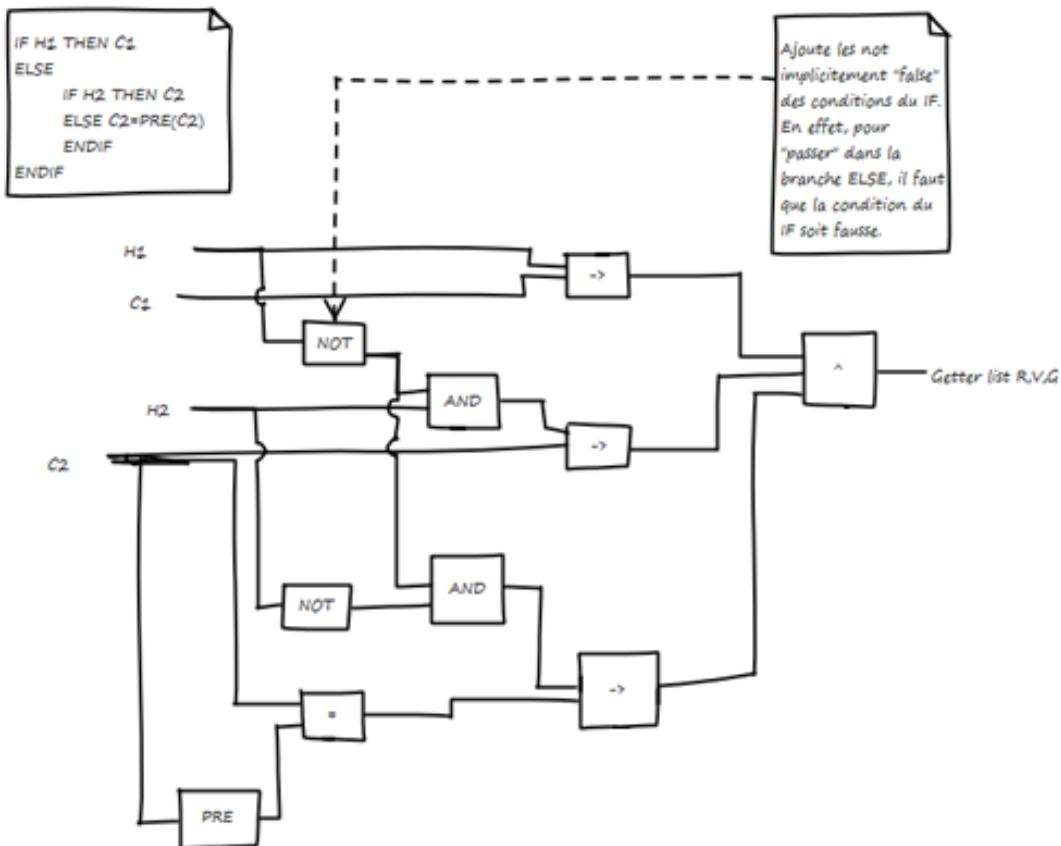


FIGURE 4.8 – Transformation d'une expectedBehavior en arbre logique

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

Rouge Au moins un eval à renvoyé faux

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

Vert Toutes les branches ont été testé, 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 une Map ayant pour clé le timestamp et pour valeur la liste des variables ayant subis une modification, ainsi nous allons avoir une boucle itérant sur cette Map qui à chaque itération modifiera les variables nécessaire, lors de cette modification une notification va être envoyé 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.

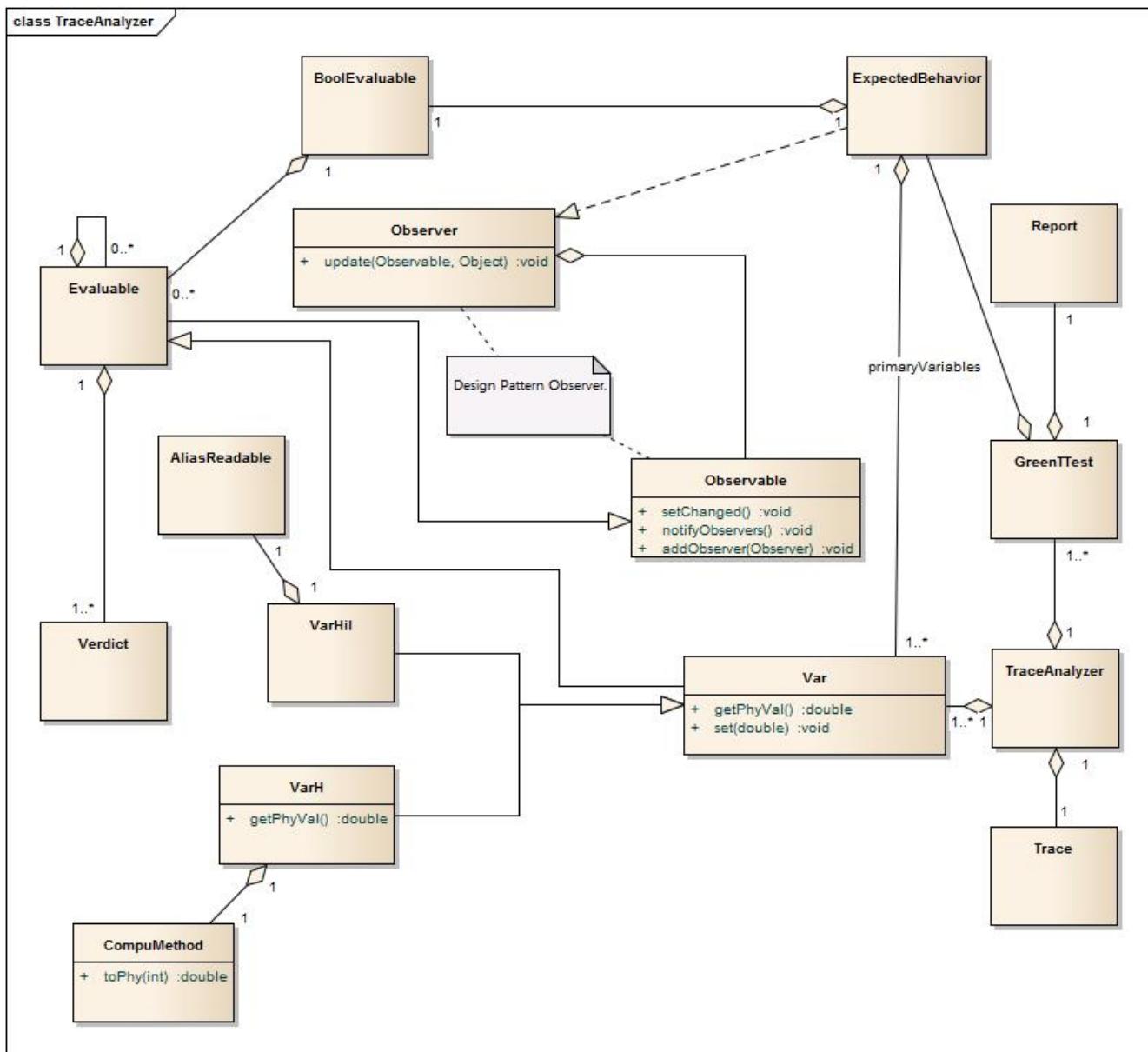


FIGURE 4.9 – Diagramme de classe de l'analyse de trace

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.

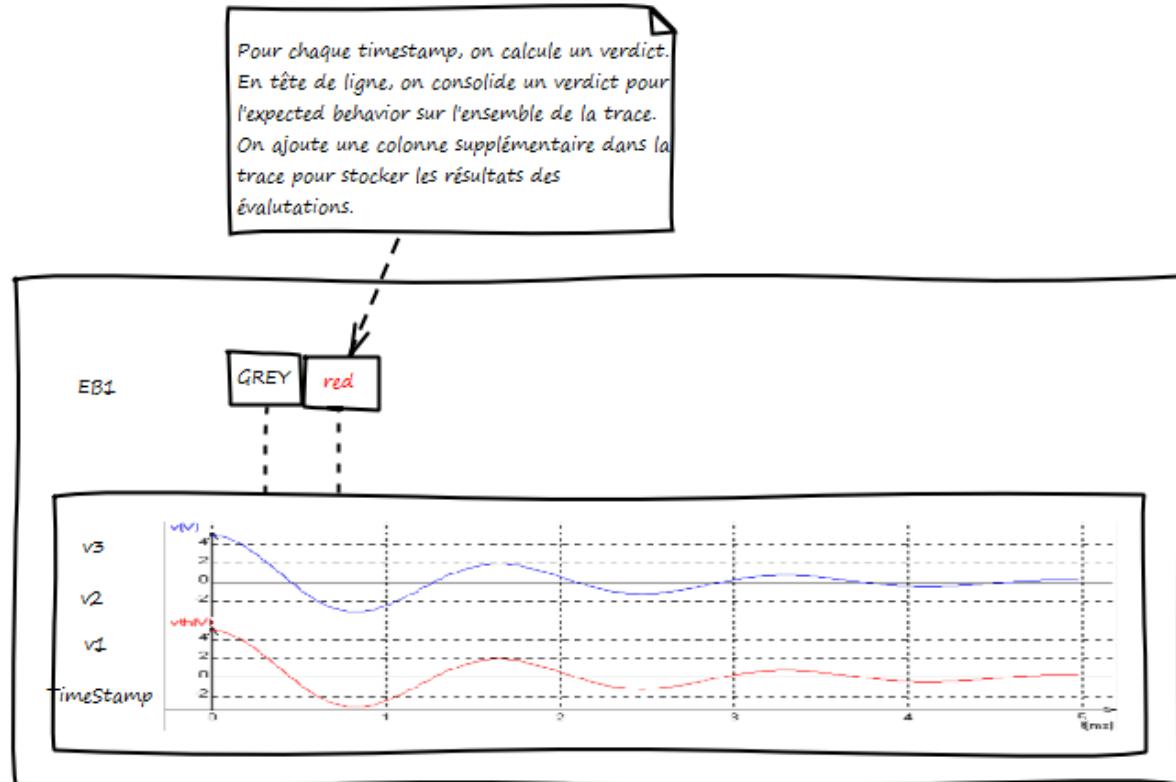


FIGURE 4.10 – Exemple de Trace sous forme de courbes

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.

A

Glossaire

B

Templates

B.1 Template des stimulations

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

6
7 import java.util.List;

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

13
14 import com.continental.gt.devices.Device;
15 import com.continental.gt.exception.CheckFailedGreenTException;
16
17 <#foreach import in imports>
18 import ${import};
19 </#foreach>
20
21 import org.apache.thrift.TException;
22
23 /**
24 * Test of stubbed class generated
25 * Generated by GreenT
26 */
27 public class ${nameClass}_${nameTest} extends ${nameClass} {
28
29   public ${nameClass}_${nameTest}(){
30     super("Anonymous ${nameClass}_${nameTest}");
31   }
32
33   public ${nameClass}_${nameTest}(String name) {
34     super(name);
35   }
36
37   @Override
38   public void addAllRequiredAlias() {
39     <#foreach alias in initAlias>
40       ${alias};
41     </#foreach>
42   }
43   /*
44    * @see com.continental.gt.test.stim.${nameClass}#exec()
45  }
```

```
47     * Generated by GreenT.  
48     */  
49     @Override  
50     public void exec(List<Device> devices) throws CheckFailedGreenTException {  
51         showMsg(".exec() : executing stimulation code of ${nameClass}_${nameTest} ←  
52             class...");  
53         try {  
54             double n;  
55             Thread.sleep(500); // TODO remove me  
56             <#foreach device in deviceDeclarationList>  
57                 ${device};  
58             </#foreach>  
59             <#foreach exec in instructionsExec>  
60                 ${exec};  
61             </#foreach>  
62         } catch (InterruptedException e) {  
63             e.printStackTrace();  
64         } catch (TException e) {  
65             e.printStackTrace();  
66         }  
67         showMsg("... complete ok!");  
68     }  
69 }
```

Listing B.1 – Template des stimulations

B.2 Template d'un GreenTTest

```

1 package com.continental.gt.generation.test;
2
3 import com.continental.gt.test.GreenTTest;
4 import com.continental.gt.test.report.Severity;
5 <#foreach import in imports>
6 import ${import};
7 </#foreach>
8
9 import org.apache.thrift.TException;
10
11 /**
12 * Test of stubbed class generated
13 * Generated by GreenT
14 */
15 public class GreenTTest_${nameTest} extends GreenTTest {
16
17     public GreenTTest_${nameTest}() {
18         super("Anonymous GreenTTest_${nameTest}");
19     }
20
21     public GreenTTest_${nameTest}(String name) {
22         super(name);
23     }
24
25     @Override
26     public void addAllRequiredContextualData() {
27         <#foreach alias in recordedVars>
28             ${alias};
29         </#foreach>
30     }
31
32     @Override
33     public void createReport() {
34         report.setVariableLongName("${variableLongName}");
35         report.setVariableName("${variableName}");
36         report.setResponsible("${responsible}");
37         report.setSeverity(${severity});
38         report.setTestSummary("${testSummary}");
39
40         <#foreach comment in comments>
41             report.addComment("${comment}");
42         </#foreach>
43     }
44
45     @Override
46     public void verdict() {
47         // TODO Auto-generated method stub
48     }
49 }

```

Listing B.2 – Template d'un GreenTTest

C

Exemples de fichiers générés

C.1 Exemple de StimScenario

```
1 package com.continental.gt.generation.test.stim;
2
3 import java.util.List;
4
5 import com.continental.gt.test.stim.StimScenario;
6
7 import com.continental.gt.devices.Device;
8 import com.continental.gt.exception.CheckFailedGreenTException;
9
10 import com.continental.gt.devices.Hil;
11 import com.continental.gt.test.alias.AliasHilParam;
12 import com.continental.gt.test.alias.AliasReadable;
13
14 import org.apache.thrift.TException;
15
16 /**
17  * Test of stubbed class generated
18  * Generated by GreenT
19  */
20 public class StimScenario_Stub_1 extends StimScenario {
21
22     public StimScenario_Stub_1(){
23         super("Anonymous StimScenario_Stub_1");
24     }
25
26     public StimScenario_Stub_1(String name) {
27         super(name);
28     }
29
30     @Override
31     public void addAllRequiredAlias() {
32         addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
33         addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
34         addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
35         addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
36         addAliasWritable(Hil.class, new AliasHilParam("HIL_VS"));
37         addAliasReadable(Hil.class, new AliasReadable("HIL_VS_OUT"));
38         addAliasWritable(Hil.class, new AliasHilParam("HIL_KEY"));
39         addAliasWritable(Hil.class, new AliasHilParam("HIL_VB"));
40     }
41     /*
42      * @see com.continental.gt.test.stim.StimScenario#exec()
43      * Generated by GreenT.
44      */
45     @Override
```

```

46     public void exec(List<Device> devices) throws CheckFailedGreenTException {
47         showMsg(".exec() : executing stimulation code of StimScenario_Stub_1 ←
48             class...");  

49         try {
50             double n;
51             Thread.sleep(500); // TODO remove me
52             Hil hil = (Hil) getDeviceByClass(devices, Hil.class);
53             Dbg dbg = (Dbg) getDeviceByClass(devices, Dbg.class);
54
55             ((AliasHilParam) (getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);
56
57             n = getAliasReadable(Hil.class, "HIL_VS_OUT").getValue(hil);
58             if (!(n >= -0.2 && n <= 0.2)) {
59                 throw new ←
60                     CheckFailedGreenTException("CHECK(HIL_VS_OUT,0.0,TOLRES(1.0))");
61             };
62             ((AliasHilParam) (getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);
63
64             n = getAliasReadable(Hil.class, "HIL_VS_OUT").getValue(hil);
65             if (!(n >= 49.5 && n <= 50.5)) {
66                 throw new ←
67                     CheckFailedGreenTException("CHECK(HIL_VS_OUT,50.0,TOLPER(1.0))");
68             };
69             ((AliasHilParam) (getAliasWritable(Hil.class, "HIL_VS"))).setPhy(0, hil);
70
71             n = getAliasReadable(Hil.class, "HIL_VS_OUT").getValue(hil);
72             if (!(n >= -0.2 && n <= 0.2)) {
73                 throw new ←
74                     CheckFailedGreenTException("CHECK(HIL_VS_OUT,0.0,TOLRES(1.0))");
75             };
76             dbg.stop();
77             ((AliasHilParam) (getAliasWritable(Hil.class, "HIL_KEY"))).setPhy(0, hil);
78             ((AliasHilParam) (getAliasWritable(Hil.class, "HIL_VB"))).setPhy(0, hil);
79         } catch (InterruptedException e) {
80             e.printStackTrace();
81         } catch (TException e) {
82             e.printStackTrace();
83         }
84
85         showMsg("... complete ok!");
86     }
87 }

```

Listing C.1 – Exemple de StimScenario

C.2 Exemple de GreenTTest

```

1 package com.continental.gt.generation.test;
2
3 import com.continental.gt.test.GreenTTest;
4 import com.continental.gt.test.report.Severity;
5 import com.continental.gt.test.alias.AliasReadable;
6
7 import org.apache.thrift.TException;
8
9 /**
10 * Test of stubbed class generated
11 * Generated by GreenT
12 */
13 public class GreenTTest_AIRT_Air_uRawTCACDsB2 extends GreenTTest {
14
15     public GreenTTest_AIRT_Air_uRawTCACDsB2() {
16         super("Anonymous GreenTTest_AIRT_Air_uRawTCACDsB2");
17     }
18
19     public GreenTTest_AIRT_Air_uRawTCACDsB2(String name) {
20         super(name);
21     }
22
23     @Override
24     public void addAllRequiredContextualData() {
25         addRecordedVariable(new AliasReadable("Air_uRawTCACDsB2"));
26         addRecordedVariable(new AliasReadable("HIL_VB_OUT"));
27         addRecordedVariable(new AliasReadable("HIL_KEY_OUT"));
28     }
29
30     @Override
31     public void createReport() {
32         report.setVariableLongName("Sensed value of down stream charged air ←
33             temperature (bank-2)");
34         report.setVariableName("Air_uRawTCACDsB2");
35         report.setResponsible("D.Matichard (FSD ALTEN09)");
36         report.setSeverity(Severity.LOW);
37         report.setTestSummary("Test if the interface Air_uRawTCACDsB2 is correctly ←
38             stub to 0");
39
39         report.addComment("");
40         report.addComment("");
41         report.addComment("");
42     }
43
44     @Override
45     public void verdict() {
46         // TODO Auto-generated method stub
47     }
48 }

```

Listing C.2 – Exemple de GreenTTest

D

Liste des codes sources

4.1	Exemple de fichier cnf	24
4.2	Exemple d'expected Behavior	25
4.3	Exemple de définition d'alias local	26
4.4	Exemple de définition d'alias à enregistrer	26
4.5	Grammaire Check	27
4.6	Grammaire Check	27
4.7	Surcharge de <code>visitCheckFunc</code>	28
4.8	Affichage d'une erreur de syntaxe	29
4.9	Affichage d'une erreur d'alias	29
B.1	Template des stimulations	41
B.2	Template d'un GreenTTest	43
C.1	Exemple de StimScenario	45
C.2	Exemple de GreenTTest	47

E

Table des figures

1.1	Chiffre d'affaire et nombre d'employés	9
1.2	Répartition du groupe continental dans le monde	10
1.3	Logo de Continental	10
1.4	Structure de continental	11
3.1	Exemple de banc de tests à Continental – HIL DSpace	16
3.2	Exemple de Debugger à continental – Trace32	16
4.1	Aperçu d'un fichier Walkthrough	20
4.2	Diagramme de classes des Alias	21
4.3	Fonctionnement général du parser <i>GreenT</i>	23
4.4	Diagramme syntaxique du Check	27
4.5	Diagramme syntaxique du Check	27
4.6	Diagramme de classes du générateur	30
4.7	Schéma de fonctionnement de FreeMarker	31
4.8	Transformation d'une expectedBehavior en arbre logique	32
4.9	Diagramme de classe de l'analyse de trace	33
4.10	Exemple de Trace sous forme de courbes	34