Rapport De Stage
Intégration de l'analyse RTI-D dans GCC

Nicolas COLLIN

25 août 2009

Table des matières

Table des figures

Chapitre 1

Remerciements

Je souhaite tout d'abord remercier M. Alain Esparel mon maître de stage pour toute l'aide qu'il m'a apportée lors de mon stage et pour tous les conseils qu'il m'a donnés. L'opportunité de voir ses méthodes de travail alors qu'il cherchait des solutions à mes problèmes m'ont beaucoup appris et beaucoup aidé pour la suite du développement et de la recherche des erreurs. Je voudrais bien entendu remercier M. Bernard Prissette de m'avoir donné la chance de faire ce stage qui m'aura beaucoup apporté et M. Jean-François Lenormand pour son accueil au sein de son département. J'adresse aussi mes remerciements à M. Dave Korn qui est la seule personne dans l'équipe de développement du programme GCC à bien avoir voulu répondre aussi bien qu'il pouvait à mes questions. Je tiens aussi à remercier M. Patrice Colzato qui a bien voulu prendre sur son temps afin de m'aider sur la compréhension du logiciel RTI-D.

Je remercie également toutes les personnes de chez Thales Systèmes Aéroportés qui ont contribué à la réussite de ce stage et qui m'ont permis de m'intégrer facilement dans l'entreprise.

Chapitre 2

Présentation de l'entreprise

2.1 Le groupe Thales

Thales, l'un des grands groupes mondiaux de l'électronique, est présent sur trois marchés : défense, aéronautique, sécurité et emploie 68 000 personnes dans plus de 50 pays (cf. Figure 2.1). Sa position de leader dans le domaine des hautes technologies est reconnue dans le monde entier. Thales est organisé en six grandes divisions : aéronautique, espace, systèmes aériens, systèmes terre et interarmées, naval, sécurité et services.

FIG. 2.1 – Implantation de Thales dans le monde.

2.2 La division Aéronautique

Leader européen et n° 3 dans le monde, la Division Aéronautique a pour mission de fournir aux fabricants de plates-formes aéronautiques, civiles ou

militaires, toute la gamme des équipements et systèmes embarqués. Elle intervient comme maître d'oeuvre, intégrateur de systèmes et de sous-systèmes, fournisseur d'équipements et de services à forte valeur ajoutée. Les systèmes aéronautiques de Thales sont aujourd'hui embarqués dans les plus grands programmes de plateformes : A380, A400M, Boeing 787, Rafale, Mirage 2000, Hélicoptères Lynx, Tigre, NH90, drone Watchkeeper, etc...

Chapitre 3

Cahier des charges

3.1 Contexte

3.1.1 L'application mission

Bien que le stage n'ai pas porté directement sur l'application mission, il est préférable d'en donner une brève description afin de comprendre la portée du projet confiée.

L'application mission est le logiciel qui gère tout ce qui est essentiel au vol de l'avion de chasse en vue d'une mission spécifique. Elle est le coeur du système de l'avion, par exemple elle fait le lien entre les commandes et les points d'emport (qui peuvent contenir des missiles, une caméra, etc ... selon la mission). Toutes les information vitales sur les systèmes de l'avions, sa position, sa vitesse, etc ... sont analysées par l'application mission qui effectue les traitements nécessaires, et affiche les informations nécessaires au pilote afin d'assurer la réussite de la mission.

[1] Les mécanismes temps réel de l'application mission sont, pour des raisons de fiabilité en vol (pas de sémaphore ou autre élément bloquant), organisés en tâches. En effet tous les traitements des applications missions sont divisés en tâches. Chaque tâche possède une priorité, qui indique notamment la fréquence de ré-execution de ses traitements (toutes les 20ms par exemple pour la tâche la plus prioritaire, toutes les 40ms pour la tâche qui est directement moins prioritaire à la tâche précédente, ...). A chaque instant, une seule tâche est en cours d'exécution. Les tâches moins prioritaires que la tâche en cours d'éxecution attendent la fin de son exécution.

Ainsi dans la figure 3.1, les traitements de la tâche C4 s'exécutent deux fois pendant une seule occurrence des traitements de la tâche C3. La tâche C4 reprend la main sur C3 pour sa seconde exécution, car elle est plus prioritaire.

Ce type de mécanisme peut poser des problèmes d'incohérence de données. En effet, si la tâche C3 commence ses traitements avec une variable

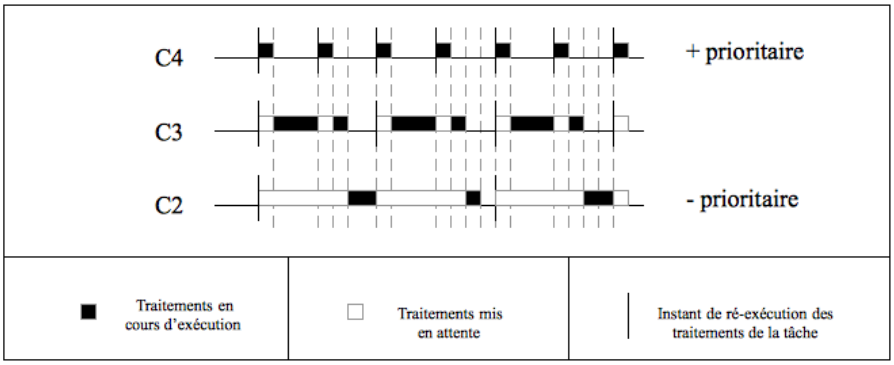

FIG. 3.1 – Plan général des calculs à effectuer [1].

"var" de valeur "10", elle n'est pas sûre de terminer ses traitements avec la même valeur, car la tâche C4 peut reprendre la main pour modifier cette même variable "var" avec une nouvelle valeur. Ainsi, les traitements débutés en C3 avec la valeur "10" pour la variable "var" peuvent se terminer avec la valeur "20".

Pour éviter ce type de problème, chaque donnée est dupliquée à chaque tâche et est mise à jour au début de chacune des tâches. Ce qui implique la mise en place de contrôles de règles d'exécution temps réel. Exemple : une donnée n'est modifiée que par une seule tâche, afin d'éviter des mises à jour croisées entre tâches.

3.1.2 RTI-D

[1] Après la mise en évidence des problèmes que peut poser les mécanismes temps réel choisis pour l'AM, il est possible de présenter le logiciel qui permet la résoltion de ces problèmes : RTI-D.

L'outil RTI-D est un outil d'analyse et de génération de code C++. Il a pour but d'analyser le code des applications missions afin d'extraire les informations d'utilisations des éléments temps réels de ces applications. Ces informations sont par la suite mises en forme pour générer des pièces de code de gestion des communications temps réels des application missions. Les éléments temps réels traités par l'outil sont :

- les **attributs**, éléments de mémorisation des informations temps réels applicatives.
- les **signaux**, éléments booléens indicateurs de la production d'informations à caractère particulier.
- les **requêtes**, éléments de mise à disposition de traitements communs utilisés par plusieurs classes des applications missions.
- les **messages applicatifs**, éléments émis ou reçus par l'AM

Ces éléments sont reconnus par analyse syntaxique du code C++ des applications missions et de marques supplémentaires rajoutées à l'intérieur de ce code.

L'ensemble des traitements de détection des tâches associées aux éléments RTI est donc le principal besoin auquel répond l'outil RTI-D. Grâce à cette détection, l'outil est en mesure de générer automatiquement le code de mise à jour inter-tâches des éléments RTI, appelés **gestion des communications temps réels**, et contrôle l'utilisation de ces éléments en fonction de règles d'éxecutions temps réels.

RTI-D possède deux modes de fonctionnements : RTI-D1 et RTI-D2. C'est une option qui permet de choisir les traitements à effectuer :

- **RTI-D1** analyse les fichiers d'interface (.h) et génère les fichiers Com.hpp qui sont sont inclus lors de la compilation de l'application mission.
- **RTI-D2** analyse le code C++ (.c) et génère les fichiers Com.cpp qui est la classe de contrôle inter-tâche des éléments RTI.

Le déroulement de ses traitements se font en deux parties : l'analyse et la synthèse. Durant l'analyse RTI-D (1 ou 2) lit le fichier et génère une fichier d'arbre dans lequel se trouve toutes les informations du code C++, durant la synthèse il se sert de fichier pour traiter les information et générer un fichier qui s'ajoute au code C++. C'est la phase d'analyse qui nous intéresse.

3.1.3 Génération d'un exécutable Application Mission

La génération de l'exécutable AM se déroule de la façon suivante (cf. Figure 3.2) :

1. Traitement RTI-D1
   - Analyse les fichiers d'interface (.hpp).
   - Génère les fichiers Com.hpp qui sont nécessaires à la compilation.
2. GCC
   - On compile les sources de l'AM (Application Mission).
3. Analyse RTI-D2
   - Analyse lexical et syntaxique du code C++ (Yacc, Lex).
   - Récupère les informations sur les éléments RTI grâce à des marques ou des types spéciaux.
   - Mémorise dans quelles tâches ils sont utilisés et générés.
   - Génère un arbre contenant toutes ces informations.
4. Synthèse RTI-D2
   - Par la suite lit l'arbre généré par l'outil d'analyse RTI-D2.
   - Traite toutes les données ainsi récupérées.
   - Génère le graphe d'appel des tâches.
   - Génère les fichiers Com.cpp afin d'assurer la cohérence des actions en temps réel.
5. Edition des liens de tous les fichiers objets ainsi obtenus

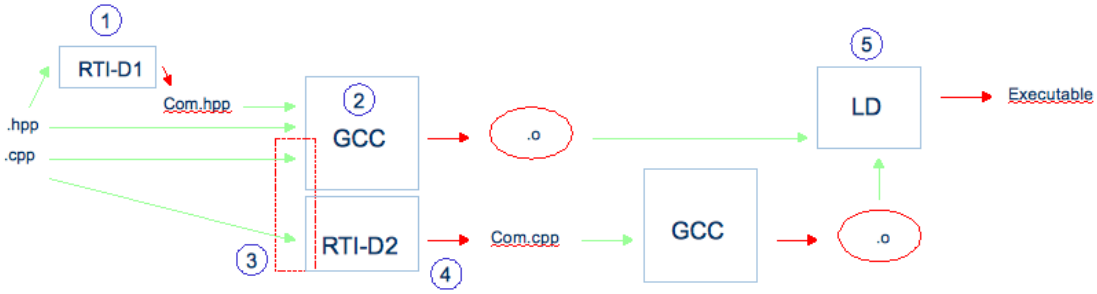

FIG. 3.2 – Génération de l'exécutable AM.

On voit bien sur la figure 3.2 (partie en pointillés rouges) que l'on effectue une analyse syntaxique et lexicale deux fois sur les mêmes fichiers .cpp. La première est effectuée par GCC lors de la compilation des sources, la deuxième est effectuée par RTI-D2. Cette redondances est une perte de temps lors de la génération. Ce qui nous amène à l'objectif du stage.

3.2 L'objectif du stage

Jusqu'à présent la phase d'analyse lexicale et syntaxique était à la charge de RTI-D, or cette même analyse était effectuée dans un même temps par GCC. La mission qui m'a été confiée était de trouver un moyen efficace d'intégrer les traitements de l'analyseur RTI-D2 au compilateur GCC. Ainsi une seule analyse des fichiers sources serait effectuée lors de la génération de l'AM. Les gains attendus par cette intégration :

- Gain de temps lors de la production : environ 10%.
- Facilité de maintenance et d'optimisation, la phase d'analyse du code source n'est plus à la charge de RTI-D.

3.3 Spécifications

Le stage avait pour but de savoir si l'intégration de RTI-D à GCC était possible et de trouver un moyen efficace de le réaliser, il ne suffisait pas de coder des fonctions données. Les spécifications au début étaient donc très larges et vagues puisque le domaine était totalement inconnu : le coeur de GCC n'est connu que d'une poignée. Voici donc les rares spécifications qui étaient données au début. Le reste dépendra de la suite.

Tout d'abord la version utilisée de GCC lors de la génération : GCC version 2.95, année de sortie : 1998. La version actuelle datant de 2009 était la 4.4. Les tests ayant été validés sur la version 2.95 et ceux-ci étant très coûteux il n'est pas envisageable de passer à la version plus récente (même si cela aurait facilité les choses et permis plus de possiblités comme je l'expliquerai

plus tard). Les fichiers objets générés par GCC doivent être identiques à ceux générés par la version non modifiée, il ne faut donc pas toucher au fonctionnement interne de GCC. Le fichier d'arbre généré par GCC après analyse doit être identique au fichier généré par la phase d'anlayse de RTI-D2, ainsi la phase de synthèse de RTI-D2 pourra s'en servir afin d'effectuer les traitements nécessaires et générer des fichiers exacts. Il faudra donc comparer à la fin les fichiers générés par les 2 exécutables. Il faut aussi que GCC soit configuré de façon à effectuer une compilation croisée, c'est à dire qu'il faudra que l'on compile sur Solaris et que l'on génère un exécutable tournant sur PowerPC. Cela n'est pas un problème en soit puisque c'est lors de la configuration de GCC qu'il faut le spécifier.

Chapitre 4

Présentation du stage

Comme dit précédemment le stage avait pour but d'étudier la possibilité d'un intégration et si possible la réalisation d'une maquette. Il ne s'agit pas d'un simple stage de codage. C'est pourquoi le stage ne s'est pas déroulé d'une façon normale de conception (cycle en V, etc ...). Le développement a été une suite de phase de recherche, de contournement des limitations, de codage, et de débogage. Ce chapitre va présenter les points importants du stage.

4.1 Découverte du projet

Au début, découverte du logiciel RTI-D : ce qu'il fait, à quoi il sert, comment il le fait. Une première idée sur la façon de procéder émerge, déplacer complètement le code de RTI-D dans GCC et l'intégrer de façon brute : c'est-à-dire modifier les fichier Lex et Yacc afin d'intervenir durant l'analyse lexicale et syntaxique du fichier. Par la suite, analyse plus en profondeur des logiciels utilisés afin de voir comment réaliser l'intégration brute prévue et surtout si elle est réalisable sans risque d'influence sur le déroulement normal de GCC. Ceci se traduit par une étude du code de RTI-D et de la documentation de GCC ainsi que de ses sources.

Grâce à mon maître de stage j'ai pu voir à la fin de la première semaine une personne ayant travaillé sur la réalisation de RTI-D. Il m'a présenteé les grandes lignes du travail du logiciel et le contexte de celui-ci. Grâce à cette intervention ma compréhension du sujet a beaucoup avancé. Cette entrevue m'a guidé dans mes recherches sur l'intégration dans GCC et après de longues recherches sur la documentation, il est apparu que la première méthode d'intégration envisagée n'est pas en fait la meilleure ; en effet GCC utilise en interne un arbre qui contient toutes les informations sur le code source analysé et c'est celui-ci qu'il utilise afin de générer le code machine. Cet arbre est un très bon point de départ pour la création de l'architecture interne à RTI-D, en effet il a besoin de beaucoup d'informations sur le code

source et cet arbre interne pourra les lui fournir.

Après avoir trouvé le fil conducteur de la réalisation il faut maintenant trouver où implanter toutes ces modifications dans GCC. Après lecture d'un article sur internet [1], il devient évident que l'arbre n'est présent que dans le front-end [2].

Un problème se pose : la version sur laquelle on travaille est la 2.95, la version actuelle est 4.4, celle que nous avons date de 1998 et les documents n'y font pas référence. Beaucoup de fonctions décrites dans le document n'étaient pas encore implémentées à ce moment là. Nécessité de lire le code de tree.h et tree.c afin de comprendre les macros puisque les documents sont aussi très réduits.

4.2 L'emplacement des fichiers

Nous avons introduit deux nouveaux fichiers dans GCC, et il faut que celui-ci sache qu'il faut les compiler en même temps que le reste du programme, il faut donc ajouter des règles dans les Makefile. Cette partie pourra être utilisée comme base pour introduire de nouveaux fichiers dans le logiciel par la suite. Tout d'abord une petite explication sur le choix du placement des fichiers dans le sous-répertoire gcc/cp. Ce sous-répertoire contient tous les fichiers relatifs au traitement du C++, ces fichiers ne font pas partis du code source de GCC mais d'un autre exécutable : «cc1plus», qui est appelé par gcc lors de l'analyse d'un fichier .cpp. Les fonctions et méthodes dont nous avons besoin pour parcourir l' arbre relatif au code C++ se trouvent dans cet exécutable c'est pour cette raison que nous avons décidé d'intégrer «cc1plus» plutôt que GCC directement.

4.3 La version

Vers le milieu du stage je me suis rendu compte que le projet ne pourrait pas être terminé à moins de changer de version. En effet la partie de l'arbre décrivant le corps des fonctions n'existait pas dans cette version. Il semble que GCC traduise directement les instructions présentes dans le corps des fonctions en RTL [3]. Or nous avons besoin de cette information pour que le projet soit opérationnel. Les modifications apportées à GCC ont alors été portées sur la version 3.4.6 (sortie en 2006 et utilisée par un autre département dans Thales), ceci a nécessité beaucoup de modifications, l'architecture étant particulièrement différente entre ces deux versions. Dans la suite du

[1] http ://www.redhat.com/magazine/002dec04/features/gcc/

[2] Partie se chargeant de l'analyse du code source dans GCC, il transmet les informations importantes au middle-end dans un format indépendant du langage de programmation

[3] Langage assembleur propre à GCC, il est indépendant de l'architecture pour laquelle on souhaite compiler

rapport nous parlerons uniquement des implémentations réalisées dans cette dernière version.

4.4 L'exploration de l'arbre et la génération du fichier

J'ai trouvé l'endroit où placer mon code afin d'explorer l'arbre interne à GCC : le fichier decl2.c contient toutes les fonctions appelées lors de l'analyse syntaxique et, à l'intérieur, la fonction finish_file qui est appelée lorsque l'analyse syntaxique est terminée. Grâce à l'option -funit-at-a-time l'arbre est complet à la fin de l'analyse syntaxique (sinon GCC génère le code assembleur au fur et à mesure de l'analyse et supprime les parties de l'arbre inutilisées par la suite). La génération du fichier, initialement prévue pour être effectuée par les traitements déjà mis en place dans RTI-D2, a été implémentée à l'intérieur de l'exploration des graphes afin de gagner en temps de calcul et en place. En effet RTI-D2 créé une arborescences complète d'instances avant de générer le fichier, alors que notre méthode implique juste l'écriture d'un fichier sans classe ni allocation mémoire. A la fin tout le code de RTI-D2 a été abandonné et nous ne nous en sommes pas servi.

Les traitements se déroulent d'une façon très hiérarchique, suivant l'architecture en arbre de la représentation interne à GCC (cf Fig. 4.1). Tout d'abord nous entrons dans une première fonction treeExplore ouvrant le fichier (son nom est le même que le fichier C++ lu avec l'extension .arbD2 à la place de .cpp) et lançant l'analyse sur le namespace principal (c'est là où se trouvent les variables, fonctions, structures, classes,... déclarées de façon globale). Durant cette analyse qui se déroule dans la fonction namespaceExplore nous récupérons, grâce à une fonction que nous avons implémenté (pour ce faire nous nous sommes inspirés d'une fonction déjà présente dans GCC), la liste des structures et classes définies. Alors nous parcourons cette liste ne contenant normalement que la classe correspondante au fichier (en effet de manière générale un fichier source de l'AM ne contient qu'une seule classe). Nous lançons alors l'analyse sur cette classe. La fonction classExplore met en place la première partie du fichier (contenant des informations sur la classe comme par exemple son nom, ...) et lance le traitement de deux listes : la première contient les champs et la deuxième les méthodes. Le traitement de ces listes se fait par l'intermédiaire d'une fonction que nous avons implémenté et qui applique une fonction passée en paramètre sur tous les éléments de la liste récupérés via la macro TREE_CHAIN (qui sert à parcourir les éléments des listes telles qu'elles sont implémentées dans la représentation interne). Pour le traitement des champs nous distinguons deux cas, les typedef et les variables, ceux–ci sont écrits différemment dans le fichier. Pour les variables nous vérifions si un attribut y est lié, au cas où ce serait un RTIattribut.

Pour le traitement des méthodes elle se déroule en deux phases, la méthode elle-même, ses arguments et son type de retour, puis l'analyse de son corps (cf Fig. 4.2) : les fonctions appelées, les attributs utilisés.

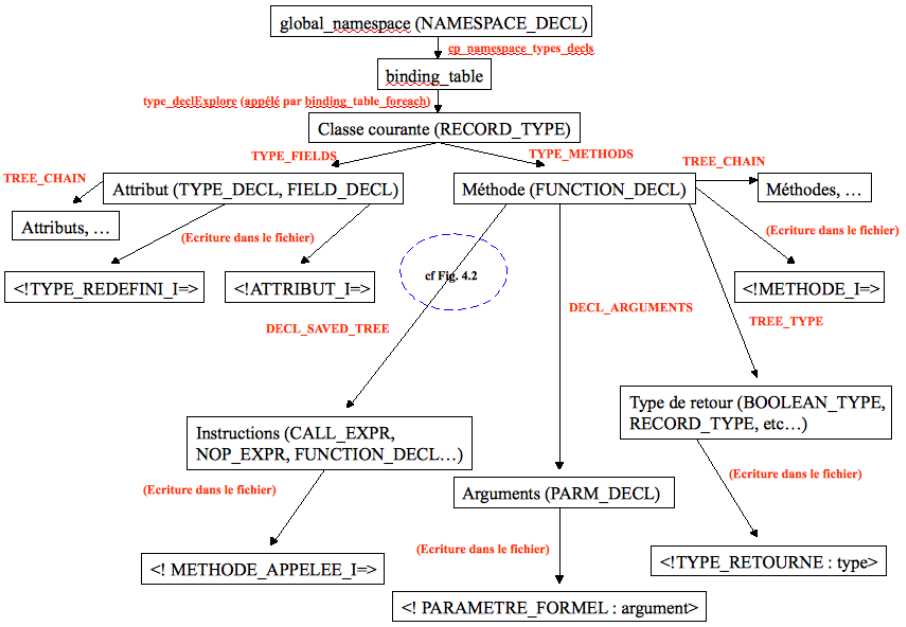

FIG. 4.1 – Le parcours du fichier en général.

4.5 L'option "-frtid" qui active les traitements

La mise en place de l'option fut assez laborieuse puisque dans la version 3.4.6 les options se trouvent dans un fichier auto-généré qui suit des informations qui sont éparpillées. Comprendre l'organisation fut ce qui demanda le plus de temps. La méthode décrite ci-dessous permet d'intégrer une option qui sera précédée automatiquement par –f, en effet les options portant aux fonctionnalités dans gcc sont précédées par ce préfixe, par exemple pour «rtid» l'option est «–frtid». Dans le fichier gcc/opts.c il faut ajouter un case au switch, précédé par «opt_f», par exemple pour «rtid» cela donne : «OPT_frtid». Il faut aussi ajouter à cet endroit une instruction qui active l'option -funit-at-a-time dont nous avons parlé précédememnt (cf. 4.4) et qui est nécessaire. Dans le fichier gcc/toplev.c il faut rajouter une entrée dans le tableau «const lang_independent_options f_options» avec en premier élément le nom de la fonction (exemple «rtid») puis l'adresse de la variable à modifier (exemple &flag_rtid) et pour finir la valeur que l'on veut que la variable (préalablement déclarée dans gcc/flags.h) prenne si l'option est don-

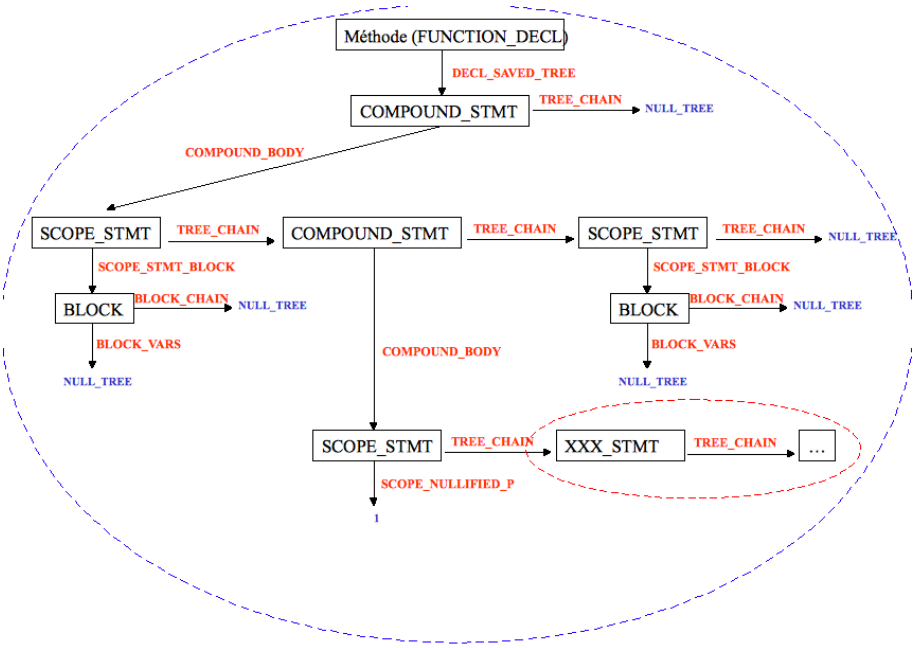

FIG. 4.2 – Le parcours du corps des fonctions et/ou méthodes.

née (exemple 0 ou 1). Dans le fichier gcc/common.opt il faut ajouter l'option ainsi qu'une description.

Toutes ces informations servent lors de la génération du fichier <rep. generation>/gcc/options.c.

4.6 Les difficultées rencontrées

4.6.1 Le type de retour des méthodes

En théorie la suite de macros TREE_TYPE(TREE_TYPE(methode)) devrait renvoyer le type de retour de la méthode mais en pratique il n'en est rien. La valeur renvoyée est en fait un type n'ayant rien à voir avec le véritable type : il est soit un ARRAY_TYPE, soit METHOD_TYPE, soit REFERENCE_TYPE, etc... En fait l'arbre est une suite de types incohérents avec à la fin le véritable type. C'est pour cette raison que nous avons mis en place la fonction getActualType qui nous évite de nous encombrer avec ces types et qui nous renvoie directement le véritable type.

4.6.2 Le corps des méthodes

La Figure 4.2 illustre ce problème.

Sans doute la partie la plus complexe de l'arbre. En effet l'arbre représentant le corps de la fonction se présente en un tronc se divisant en 3 branches, ces 3 branches sont terminées par des valeurs NULL_TREE. Or aucune information ne se trouve dans les feuilles ni dans les nœuds ainsi parcourus, l'arbre semble vide. En vérité la 2ème branche se divise elle aussi en deux branches à un certain endroit et c'est ici qu'on peut récupérer les données qui nous intéressent. Mis à part cette minuscule partie, l'arbre en entier n'est qu'un regroupement de valeurs nulles et ne contient aucune information. Arriver jusqu'à ce point d'entrée n'est en fait que le début. En effet cet arbre peut s'avérer être très complexe : chaque instruction est représentée par un nœud qui peut lui-même être divisé en plusieurs nœuds s'il définit un regroupement d'instructions. Pour ce parcours nous avons mis en place une boucle incluant un switch. Le traitement s'effectue de cette façon : nous obtenons un nœud, nous regardons son type, selon celui-ci nous savons dans quelle direction aller (grâce au switch), nous rentrons dans ce nœud et nous continuons jusqu'à trouver un nœud d'appel de méthode. Détail important : il se peut qu'à un moment l'exploration soit divisée en deux parties (exemple : if (instr1)else(instr2), il faut explorer les deux arbres instr1 et instr2), nous utilisons une pile afin de mémoriser les branches restantes à explorer. Quand nous arrivons en bas de l'arbre nous recommençons tant qu'il reste des branches à explorer. S'il apparaît lors des tests que des appels de méthodes ne sont pas reconnus il est certain que c'est dans cette partie qu'il faut agir, il faudra certainement ajouter un cas de type de nœud rencontré (ex if, else, switch, etc...). Les cas sont nombreux et il est possible que l'on en ai oublié, c'est le rôle des tests de trouver ces cas spéciaux.

4.6.3 Les instances

Cette partie fut très longue malgré le peu de choses à faire. En effet lorsqu'une méthode est appelée sur une instance nous devons récupérer le nom de cette instance, après de longues recherches infructueuses il était apparu que cette donnée n'était pas gardée par GCC, on ne sait pas pour quelle raison. Cette donnée est pourtant indispensable et le fait qu'elle soit introuvable rendrait le projet inutile. Pour la trouver je suis une idée : GCC doit supprimer cette donnée lors de l'optimisation. Je vais alors aller chercher cette donnée avant qu'il ne la supprime, aux environs du parser. Je trouve le fichier gcc/cp/parser.c où est appelée la fonction build_new_method_call lors de la reconnaisance d'un appel à une méthode. Cette fonction est définie dans gcc/cp/call.c. En paramètre est passé un nœud appelé «instance», en l'explorant je trouve enfin la donnée manquante. Effectivement, GCC la supprimait pour une raison inconnue. La meilleure solution serait de stocker la donnée dans l'arbre, à un endroit caché où GCC ne pourrait aller et ainsi eviter toute erreur, mais après un certain nombre de tests il apparu évident que GCC regénérait l'arbre à un endroit et modifiait tous les nœuds, ainsi

les modification étaient perdues et les données toujours introuvables. Nous avons alors réalisé la seule solution restante : nous avons déclaré un tableau de «tree» dans tree.h en extern afin qu'il soit visible de partout. Dans la fonction build_new_method_call nous introduisons grâce à un compteur tous les nœuds d'instance dont nous avons besoin. Nous avons remarqué que l'ordre dans lequel le parser lit les méthodes est le même que celui que nous retrouvons dans l'arbre ainsi il nous suffit de reproduire un même compteur dans la fonction exprExplore du fichier gcc/cp/rtid-analyse.c et récupérer les instances à chaque appel de méthode atteint. Détail : le tableau est alloué de manière dynamique et on l'agrandit quand il est plein.

4.7 Les tests

Les tests que nous avons menés se résument à la comparaison entre les fichiers générés par GCC après modification et ceux générés par RTI-D2. Pour repérer d'éventuel différences, nous avons utilisé le logiciel GVIM avec l'option -d qui met en valeur les différences.

4.8 Les résultats du projets

La seule différence qui se trouve entre les deux fichiers apparaît dans les types de certains champs, lorsque ceux-ci on un type défini par typedef. En effet dans ce cas là GCC ayant déjà efectué une optimisation, le type que nous renvoyons est le véritable type (traduit à partir du typedef) alors que RTI-D2 se contente de renvoyer le nom du type redéfini. Après avoir parlé de ce problème avec la personne ayant participé au développement de RTI-D2 il apparait que ceci n'a pas d'impact lors de la synthèse donc cette différence peut être ignorée.

A la fin du stage, le programme est fonctionnel et il ne reste plus qu'à finir les tests. En effet les problèmes qui se sont posés lors du changement de version (compilation et configuration difficiles à réaliser et nécessitent des modifications) m'ont fait perdre beaucoup de temps ce qui explique que les tests n'ont pas pu être terminés.

Au final la méthode choisie afin d'intégrer RTI-D dans GCC m'a permis d'atteindre les objectifs fixés même si elle m'a limité sur certains points : les types différents et l'option -funit-at-a-time obligatoire qui est susceptible de modifier le code machine généré par GCC (optimisation du code). Ces deux points seront à vérifier malgré le peu de chance qu'ils posent effectivement des problèmes. Malgré tout si je n'avais pas choisi cette voie, il y a de fortes chances que les délais n'auraient pas pu être tenus.

Chapitre 5

Conclusion

Ce stage a été très enrichissant pour moi, d'un point de vue programmation et d'un point de vu professionel.

J'ai tout d'abord pu voir l'organisation d'une grande entreprise comme Thales, comment le travail y était organisé et comment la vie se déroule dans l'entreprise. Mais bien sûr la partie la plus enrichissante fut le projet en lui-même : il m'a fait découvrir bien des facettes de la programmation. En effet j'ai pu entrer dans le sujet de la compilation et j'ai découvert les mécanismes intérieurs de GCC, j'ai acquis une grande expérience en terme de lecture de code et de documentation, et cela m'a particulièrement intéressé d'étudier l'architecture de GCC qui est un logiciel particulièrement important et complexe. Cela m'a permis de lire du code source clair et particulièrement soigné. Ce stage m'a permis aussi de me surpasser en matière de recherche et de programmation, il m'a fallu à maintes reprises aller modifier des choses dans le code même de GCC, de trouver des solutions aux problèmes qui se posaient, et d'avancer malgré les contraintes qu'il fallait contourner. Il m'a permis de voir ce que je valais en étant "lâché" sur un sujet assez ouvert où il m'incombait de trouver les idées directrices et de réfléchir sur les meilleures façon de réaliser la tâche confiée.

Il m'a aussi confirmé que j'aimais aller au coeur des choses (je n'aurais pas pu finir le projet si je ne m'étais pas intéressé de près aux mécanismes internes de GCC) et que le parcours recherche pourrait particulièrement m'attirer dans la suite de mes études.

Bibliographie

[1] Equipe OGC. Description de conception du logiciel - sdd - outil de generation de code outil rti-d. Technical Report 61063770-549C, Thales Systemes Aeroportes, September 2003.