Mémoire de stage

Master Systèmes et Applications Répartis

Optimisation, Déterminisme et Asynchronisme des Souches et Squelettes CORBA pour Systèmes Répartis Temps-Réel

Étudiant :

Bechir ZALILA

Responsables du stage :

Laurent PAUTET

Fabrice KORDON

15 Septembre 2005

Table des matières

Remerciements

Je tiens à exprimer mes remerciements les plus sincères à M. Laurent PAUTET pour m'avoir accueilli dans son équipe pendant ce stage et pour les compétences que j'ai acquises dans le domaine de l'informatique répartie et le développement des compilateurs.

Je remercie également Jérôme HUGUES pour les conseils qu'il m'a donnés et pour les réponses à toutes les questions que je lui posais très souvent qui m'ont fait gagner un temps précieux.

Mes vifs remerciement vont aussi à Thomas VERGNAUD et Khaled BARBARIA pour les aides et conseils qu'ils m'ont apportés.

Finalement, je remercie mes collègues stagiaires pour les agréables cinq mois et demi que j'ai passé avec eux.

Résumé

Dans le cadre de mon Master Systèmes et Applications Répartis à l'Université Pierre et Marie Curie, j'ai effectué un stage de six mois dans le département INFRES de Telecom Paris. Ce rapport décrit le travail effectué pendant le stage ainsi que les résultats obtenus.

Le stage, qui s'intitule : *Optimisation, Déterminisme et Asynchronisme des Souches et Squelettes CORBA pour Systèmes Répartis temps-réel*, consiste :

— Dans un premier temps à compléter l'implémentation de IAC (*Idl to Ada Compiler*), le nouveau compilateur IDL de l'intergiciel schizophrène PolyORB. IAC remplacera IDLAC, l'ancien compilateur de PolyORB en raison de la difficulté de la maintenance de ce dernier.
— Dans un second temps, à proposer puis mettre en œuvre des optimisations qui augmenteront les performances des souches et des squelettes générés par ce compilateur.

Chapitre I

Introduction

I.1 Autour de la problématique du temps réel

L'informatique temps-réel permet de développer des applications qui doivent respecter des contraintes temporelles plus ou moins strictes (temps réel dur vs. temps réel souple). Une des principales caractéristiques d'une application temps-réel est le déterminisme, ce qui veut dire que les temps d'exécution des différentes actions doivent être bornés.

Quand le besoin du temps-réel est exprimé par une application répartie, le déterminisme est plus difficile à obtenir parce que les communications en réseau introduisent beaucoup d'aléas dans les temps d'exécution des différentes actions entre les différentes entités de l'application répartie. Ajoutons à cela le fait que les entités d'une même application répartie pourraient s'exécuter sur des plates-formes différentes et le problème devient beaucoup plus difficile.

Pour pallier le problème de l'hétérogénéité, une nouvelle invention a vu le jour : les intergiciels (*Middleware*). Les intergiciels on été conçus pour pouvoir faire communiquer des entités s'exécutant sur des plates-formes différentes et éventuellement écrites dans des langages de programmation différents. Tout ceci en épargnant au développeur la programmation de la partie communication proprement-dite et en le laissant se concentrer sur ce que doit effectivement faire son application répartie. Dans le monde des intergiciels, CORBA (*Common Object Request Broker Architecture*) est un standard très connu. Il a été écrit par l'OMG (Object Management Group) [7] en 1991 pour les raisons et les besoins évoqués plus haut.

Le fait d'arriver à faire communiquer des entités hétérogènes diminue considérablement les performances de l'application répartie et rend les contraintes du temps réel beaucoup plus difficiles à respecter. C'est pourquoi tous les efforts sont fournis pour optimiser le fonctionnement des intergiciels.

I.2 Autour des optimisations dans les intergiciels

Le standard CORBA repose sur la philosophie suivante :

— Le client et le serveur peuvent être développés dans des langages de programmation différents et tourner sur des plates-formes différentes.

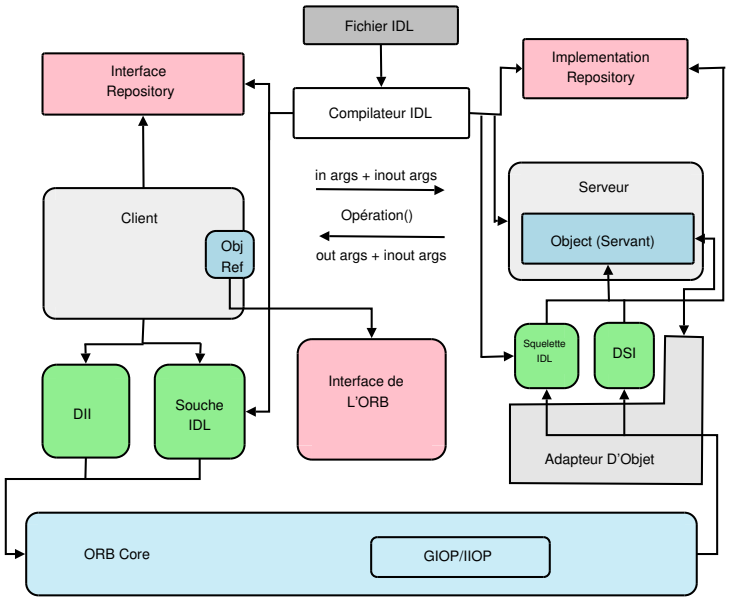

FIGURE I.1 – Architecture générale de CORBA

— La localisation de l'objet distant est faite de façon transparente par rapport au client. Ceci veut dire que dans le code du client développé par le programmeur, il n'y a aucune référence à l'adresse du serveur (qui peut d'ailleurs changer entre deux exécutions successives du client)
— La communication entre le client est le serveur est essentiellement faite de façon synchrone (bien que la spécification présente deux autres types de communication : semi-synchrone et asynchrone).

La figure I.1 donne un aperçu global de l'architecture CORBA. Pour construire une application répartie reposant sur ce standard, le développeur décrit les relations entre les différentes entités de l'application (les interfaces de l'application) dans un langage de description d'architectures (IDL). Généralement, chaque intergiciel dispose de son propre langage de description d'interfaces. Le standard CORBA décrit son langage IDL dans le chapitre 3 de la spécification [7].

Un compilateur IDL se charge de générer le code qui décrit la partie communication entre le client et le serveur (les *souches* et les *squelettes*).

Durant les dernières années, le débit du transfert des données dans les réseaux est devenu plus grand et ne constitue plus un goulot d'étranglement pour l'application répartie. Désormais, c'est le code de l'application elle-même qui doit être optimisé. En particulier, le code généré par les compilateurs IDL a besoin d'être très performant pour ne pas dégrader les performances de l'application répartie.

L'intergiciel utilisé dans le cadre de ce stage est l'intergiciel schizophrène PolyORB. La schizophrénie dans le domaine des intergiciels a été introduite pour permettre l'interopérabilité entre différents modèles de répartition (*Middleware To Middleware*). Ainsi, un intergiciel schizophrène instancie simultanément plusieurs personnalités et leur permet de cohabiter et d'interagir [11].

Comme toute implémentation classique de la spécification publiée par l'OMG, la personnalité CORBA de PolyORB souffre de quelques problèmes décrits longuement dans le mémoire bibliographique qui a précédé le stage [13]. Les principaux problèmes dont souffre une application

répartie conçue selon le standard CORBA sont :

— Le temps de communication long des requêtes qui est principalement dû au fait que le standard CORBA effectue une abstraction de l'hétérogénéité et que tous les traitements faits lors de l'exécution d'une requête ne sont pas toujours nécessaires (cas où le client et le serveur s'exécutent sur la même plate-forme). La personnalité CORBA ajoute à cela le fait qu'elle utilise un mécanisme dynamique de traitement des requêtes qui a, certes, l'avantage de pouvoir omettre de préciser le type des données échangées lors de la compilation mais qui introduit un délai supplémentaire non négligeable lors du traitement des requêtes.
— L'indéterminisme : en effet, le temps d'exécution des requêtes dépend fortement de la signature de l'opération correspondante à la requête et du nombre total d'opérations dans une même interface de l'application répartie.
— La grande empreinte mémoire de l'application répartie qui est aussi due à l'abstraction de l'hétérogénéité des différentes entités.
— Le synchronisme qui peut parfois être considéré comme une limite. Le traitement par défaut des requêtes dans CORBA est synchrone. Il existe des types d'opérations dites *oneway* qui permettent au client de poursuivre son exécution juste après l'envoi de la requête sans se bloquer, mais ce type d'asynchronisme ne peut s'appliquer qu'aux opérations qui ne retournent pas de résultat au client.
— La taille des messages échangés par les différentes entités de l'application répartie est grande et implique un temps d'exécution long des requêtes.

Pour chacun des problèmes de CORBA évoqués, des solutions (des optimisations) ont été proposées dans le mémoire bibliographique [13] en se basant sur différents travaux de recherches sur le standard CORBA.

La principale partie de l'intergiciel qui doit être optimisée étant le compilateur IDL, il faut que celui-ci soit très bien conçu au départ et qu'il soit, de plus, facilement maintenable pour pouvoir y intégrer les optimisations avec le moins de peine possible.

I.3 Autour des architectures de compilateurs

Le compilateur IDL de PolyORB s'appelle IDLAC (*IDL to Ada Compiler*). IDLAC possède l'architecture classique d'un compilateur :

— Une partie frontale "Parse" le fichier source IDL, vérifie sa syntaxe et génère un arbre syntaxique abstrait (*AST*). Cet arbre subit une expansion pour le rendre plus explicite et pour faciliter la génération de code à partir de cet arbre.
— Une partie dorsale qui prend en entrée l'*AST* et qui génère du code Ada conformément aux spécifications du mapping Ada publiées par l'OMG [6].

IDLAC a subi plusieurs modifications au fur et à mesure de l'évolution de l'intergiciel PolyORB. À chaque modification, le compilateur devient de plus en plus complexe. Actuellement, IDLAC est un compilateur très difficilement maintenable et souffre de plusieurs anomalies de fonctionnement. Ces propriétés de IDLAC se situent aux antipodes des propriétés attendues dans un compilateur pour faciliter son optimisation.

La meilleure solution pour avoir un compilateur flexible et optimisable était de construire un nouveau compilateur IDL en évitant de tomber dans les mêmes pièges de conception rencontrés lors

du développement de IDLAC. IAC, (*Idl to Ada Compiler*) est un nouveau compilateur IDL qui est censé remplacer IDLAC dans PolyORB à la fin de ce stage. La partie frontale de IAC a été développée pendant l'été 2003. Ses petites différences par rapport à la partie frontale de IDLAC seront détaillées dans le chapitre III. Le développement de la partie dorsale de IAC a commencé en 2004 dans le cadre d'un stage de DEA. L'architecture de cette partie est totalement différente de celle de IDLAC. Elle utilise le concept novateur de transformation d'arbre. Ce concept sera détaillé lors de la description de l'architecture de IAC dans le chapitre III.

L'objectif de ce stage est de :

1. Achever la conception de IAC en le mettant à niveau vers la version la plus récente de CORBA et en faisant en sorte qu'il offre toutes les fonctionnalités présentes dans IDLAC. Ce travail n'a pas été une tâche facile parce que la construction des compilateurs est différente de la construction des applications classiques (aussi complexes soient-elles). Il m'a fallu donc une période d'apprentissage pour m'approprier les sources de IAC et pour me familiariser avec les concepts utilisés pour ensuite pouvoir m'en servir durant le développement.
2. Proposer et implémenter des optimisations dans IAC en s'inspirant du travail de recherche effectué dans le mémoire bibliographique. A l'issue de cette phase, les applications réparties dont les souches et les squelettes ont été générés avec IAC devraient être nettement plus performantes que les applications développées à l'aide de IDLAC.

I.4 Description du mémoire

Le plan de ce mémoire s'articule en deux parties principales conformément à l'objectif du stage.

Dans le chapitre II, je décrirai l'architecture de départ de IAC, je présenterai ses avantages par rapport à l'architecture de IDLAC. Ensuite je présenterai les modifications et les ajouts que j'ai été amené à implémenter dans IAC pour le mettre à niveau vers la dernière version du standard CORBA notamment le nouveau schéma de fonctionnement de IAC qui intègre un expanseur d'arbre (II.2.1). Je listerai ensuite les nouvelles fonctionnalités ajoutées à IAC pour le mettre à niveau par rapport à IDLAC.

Le chapitre III présente les optimisations qui ont été retenues et implémentées dans IAC : L'utilisation des fonctions de hachage minimales parfaites pour garantir le déterminisme dans le temps d'exécution des requêtes et l'optimisation du traitement des requêtes en utilisant des méthodes statiques à la place des méthodes dynamiques qui étaient les seules utilisées par PolyORB. Pour chaque optimisation, je décrirai l'approche utilisée pour son implémentation et je donnerai les gains en performance obtenus.

Le chapitre IV conclue ce mémoire et présente les différentes perspectives qui s'ouvrent devant IAC. Il y aura notamment une listes de fonctionnalités utiles à implémenter dans le futur et une explication sur la manière de faire évoluer IAC.

À la fin de ce mémoire trois annexes contiennent des informations supplémentaires et utiles :

— Les annexes A et B décrivent les structures des arbres syntaxiques respectivement pour les langages IDL et Ada. Ces descriptions sont écrites en "PSEUDO IDL " et un programme

spécial (*mknodes*) génère les paquetages Ada nécessaires pour la construction et la manipulation de ces arbres.

— L'annexe C décrit le cas particulier du module CORBA existant dans le fichier `"orb.idl"` et les dispositions spéciales qui ont été prises dans IAC pour rendre possible l'analyse de ce module.

Chapitre II

Conformité au standard CORBA 3.0

IAC (Idl to Ada Compiler) est un générateur de code Ada à partir de descriptions IDL. Il est destiné à remplacer IDLAC car il est plus évolutif que lui. En effet, le développement de IAC a été fait de façon plus structurée que celle de IDLAC. Les séparations entre différentes parties du compilateur sont plus claires. De plus, IAC peut facilement être étendu pour générer des souches et des squelettes dans un langage autre que Ada à partir du langage IDL (mais le nom n'aurait plus une grande signification dans ce cas).

Bien entendu, IAC n'a pas été développé à partir de zéro : plusieurs parties bien faites de IDLAC ont été améliorées et intégrées dans IAC telles que la gestion des erreurs et des avertissements, le mécanisme de l'analyse de fichiers, la séparation de `Parser` et `Lexer`. Certaines parties de IAC, comme le mécanisme de construction de l'AST (*Abstract Syntax Tree*) ont été reprises ou inspirées des sources du compilateur Ada GNAT. La partie dorsale de IAC est différente de celle de IDLAC, elle permet d'avoir plus de flexibilité et de maintenabilité pour le compilateur.

Au début du stage, le niveau de IAC n'était pas aussi avancé que celui de IDLAC. Il implémentait la plupart du langage IDL, mais la version de CORBA dont il s'agissait n'était pas la dernière comme c'est le cas pour IDLAC. La version de départ de IAC implémente la grammaire IDL publiée dans la version 2.3 de la spécification de CORBA (règles $1 \Rightarrow 98$). Par exemple, IAC ne générait pas tous les sous-paquetages `Helper` qui contiennent des méthodes utiles pour la conversion de types. La même remarque est valable pour les paquetages `_IDL_File` qui regroupent les entité déclarées à l'extérieur de tout module et de toute interface. Pour tous ces paquetages, IAC se contentait de générer des squelettes (des paquetages vides) de fichiers.

La première phase du stage a consisté à mettre IAC à niveau pour qu'il implémente la génération de code pour la dernière version de la spécification CORBA (3.0.4). Hormis le fait que cette phase a mis IAC à niveau par rapport à IDLAC, elle m'a permis de m'approprier ses sources et de comprendre les mécanismes et les technologies de développement des compilateurs en lesquels je n'avais pas une grande expérience au début du stage.

II.1 Compilateur prototype IAC 1.0

Comme IDLAC et comme tout autre compilateur, IAC se compose d'une partie frontale (*frontend*) et d'une partie dorsale (*backend*). Le développement de ces deux parties dans IAC a été fait d'une

façon plus modulaire que dans IDLAC.

À titre d'exemple, la manipulation des chaînes de caractères n'est plus faite de façon directe comme dans le cas de IDLAC. Un niveau d'abstraction a été ajouté : la table de noms. Toutes les manipulations de chaînes de caractères en partant de l'analyse du fichier source IDL jusqu'à la génération de code Ada utilisent le paquetage `Namet` qui offre toutes les méthodes nécessaires. Ce niveau d'abstraction emprunté des sources du compilateur Ada GNAT permet au développeur de s'élever d'un cran dans la construction de IAC. Il est aussi à noter que la majorité écrasante des types de données utilisés dans IAC sont des types personnalisés ce qui constitue un autre niveau d'abstraction qui va assurer une dépendance vis-à-vis des différentes architectures et une flexibilité du compilateur.

II.1.1 Ancienne architecture de IAC

Avant de détailler l'architecture de IAC, il est important de parler d'une des phase les plus importantes de la compilation : le *Preprocessing*. Cette phase consiste à traiter des directives spéciales (commençant par le caractère "#" en début de ligne) qui ne font pas partie de la syntaxe du langage traité, en l'occurrence le langage IDL dans notre cas. Son principal effet est d'inclure des définitions à partir d'autres fichiers source (par le biais de la directive `#include`) et de supprimer ou garder des portions de code selon les options données en ligne de commande (directive `#ifdef` par exemple). Le *Preprocessing* peut être implémenté dans le compilateur IDL comme il peut être effectué en utilisant un programme indépendant. IDLAC et IAC utilisent tous les deux le pré-processeur de GCC pour traiter toutes les directives à l'exception de `#pragma` qui n'est pas reconnue par le pré-processeur C++.

À l'issue de la phase de *Preprocessing*, on obtient une suite de *Tokens* (c'est le terme utilisé dans la spécification de CORBA pour définir les différentes entités existant dans un fichier source IDL). Il existe cinq catégories de *Tokens* :

- Les identificateurs : qui servent à nommer les modules, les interfaces...
- Les mots-clés : qui sont la base du langage IDL.
- Les valeurs littérales d'entiers, de réels, de chaînes de caractères...
- Les opérateurs comme le "+", le "-"...
- Les autres séparateurs comme les accolades, les parenthèses, le point-virgule...

L'OMG définit pour certains types de *Tokens* des règles de construction (les identificateurs). Pour d'autres *Tokens*, une liste exhaustive de toutes les valeurs possibles est donnée (les mots-clés). C'est en respectant ces règles, et en s'appuyant sur la grammaire du langage IDL (chapitre 3 de [7]) que la partie frontale de IAC va traiter un fichier source IDL et produire une sortie correcte utilisable par la partie dorsale.

II.1.1.1 La partie frontale

Le fonctionnement de la partie frontale de IAC est similaire à celui de tout autre compilateur. En entrée, la partie frontale reçoit un fichier source IDL écrit conformément à la grammaire du langage spécifiée par l'OMG. La sortie principale de cette partie est un arbre représentant la structure du fichier : l'arbre syntaxique abstrait (*Abstract Syntax Tree*). Les sorties secondaires de la partie frontale consistent en d'éventuels avertissements ou messages d'erreurs. La figure II.1 donne un

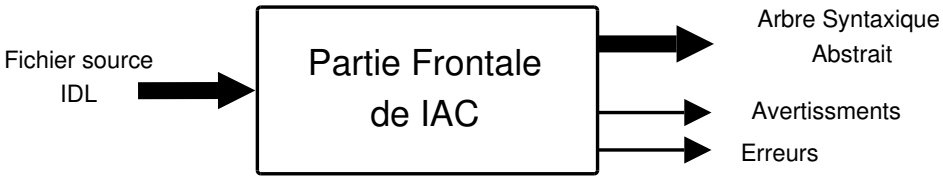

FIGURE II.1 – Partie frontale de IAC : fonctionnement global

aperçu global sur le fonctionnement de la partie frontale de IAC.

La partie frontale de IAC est décomposée en trois parties principales : Le lexeur (*Lexer*), le parseur (*Parser*) et l'analyseur (*Analyzer*). Le fonctionnement de ces trois parties n'est pas séquentiel, du moins pour le lexeur et le parseur. La figure II.2 montre la circulation du flux de données entre ces trois composantes de la partie frontale.

Le Lexeur : Cette partie parcourt le fichier source IDL caractère par caractère et essaie de reconnaître les différents *Tokens* rencontrés (identificateur, mot-clé, valeur littérale...). Chaque demande à cette partie d'avancer dans le traitement du fichier et de vérifier le prochain *token* met à jour un ensemble de variable indiquant le type du *token*, l'identificateur de son nom dans la table des noms et la valeur s'il s'agit d'une valeur littérale. Une des propriétés à la fois intéressante et utile de IAC est qu'on peut enregistrer à un moment donné l'état du lexeur, ensuite revenir vers cet état à un moment ultérieur. Cette fonctionnalité facilite beaucoup le traitement d'un fichier source IDL.

Le Parseur : Les routines présentes dans cette partie utilisent le lexeur et la grammaire IDL pour vérifier la syntaxe du fichier source IDL et construire l'AST. Globalement, pour chaque règle de la grammaire IDL [7], il existe une fonction `Parse_XXXX` qui appelle les routines du lexeur et construit une partie de l'AST. L'effet principal du parseur est de transformer les sources IDL d'une séquence linéaire de *Tokens* en une structure arborescente formée de plusieurs nœuds. La racine des source IDL étant la spécification, le point d'entrée vers le parseur de IAC est donc la fonction `Parse_Specification` qui, selon la nature des *Tokens* que le lexeur lui envoie, appelle les fonctions `Parse_XXXX` correspondantes. La fonction `Parse_Specification` retourne la racine de l'AST qui est un nœud de type `K_Specification`. La structure de l'arbre IDL ainsi que les mécanismes mis en œuvre pour sa génération sont expliqués dans l'annexe A de ce mémoire.

L'Analyseur : L'arbre IDL construit par le parseur vérifie certes la grammaire IDL mais la sémantique n'est pas nécessairement vérifiée. Il se peut par exemple que certaines constantes déclarées aient des valeurs qui se situent à l'extérieur de l'intervalle de leur type. L'analyseur vérifie la sémantique des sources IDL. Il complète aussi la construction de certaine parties de l'arbre qui ne peuvent être ajoutées que si le parseur a parcouru tout le fichier source (les *forward interface* et la résolution des noms par exemple).

À l'issue de l'analyse de la partie frontale, on obtient un arbre de syntaxe représentant la structure des sources IDL. Cet arbre va constituer l'entrée de la deuxième partie de IAC : la partie dorsale.

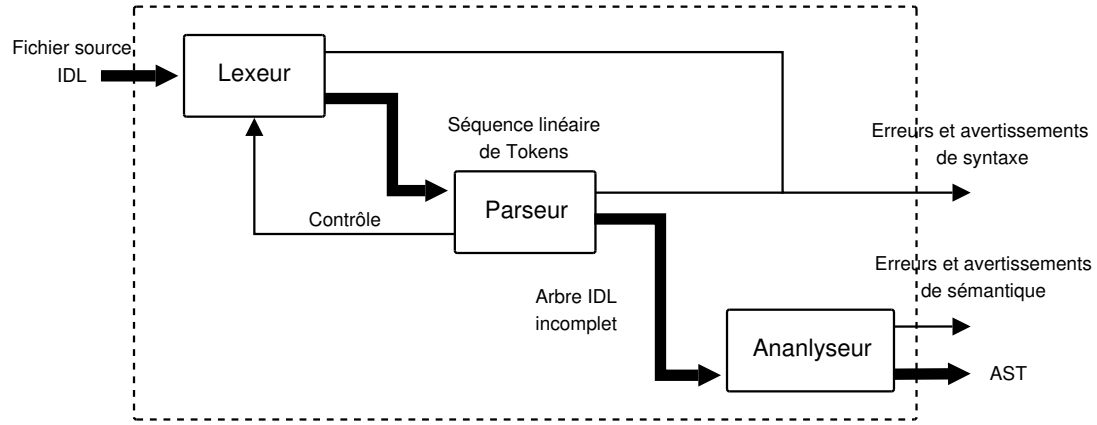

FIGURE II.2 – Partie frontale de IAC : détails

II.1.1.2 La partie dorsale

Comme il a été dit précédemment, IAC a été construit pour supporter plusieurs parties dorsales. Le choix du *backend* utilisé n'est donc pas figé dans les sources du compilateur comme c'est le cas pour IDLAC. Avec la ligne de commande, on choisi le *backend* à l'aide de l'option de compilation `-<lang>` dans laquelle `<lang>` désigne un langage de programmation supporté par IAC. Dans l'état actuel des choses, IAC dispose de deux parties dorsales :

1. Une partie dorsale IDL servant à générer un fichier source IDL à partir de l'arbre syntaxique construit dans la partie frontale. Ceci sert à vérifier le bon fonctionnement de la partie frontale.
2. Une partie dorsale Ada qui génère les souches et les squelettes CORBA conformément aux spécifications de mapping [6]. C'est cette partie qui a constitué l'objet principal du travail durant le stage. À partir de ce moment, et dans le reste du mémoire, le terme "Partie dorsale de IAC " désigne la partie dorsale relative au langage Ada.

La partie dorsale de IAC est différente de celle de IDLAC. Alors que dans IDLAC, la génération de code se résume à des `Put_Line`, IAC adopte une autre stratégie : à partir de l'arbre IDL généré dans la phase frontale de la compilation, le compilateur génère un arbre Ada qui contient la description des souches et squelettes à implémenter.

La partie dorsale est donc découpée en deux parties principales (cf figure II.3) :

II.1.1.2.1 Construction de l'arbre Ada Dans cette partie, l'arbre IDL est parcouru un nombre de fois égal au nombre de types de paquetages Ada qui doivent être générés. Pour générer les souches, les *Helpers*, les squelettes et les implémentations, il faut donc parcourir l'arbre IDL 4 fois. Il faut noter que l'arbre Ada n'a pas une seule racine comme il est le cas pour l'arbre IDL, mais il dispose d'une liste de racines (4 si on ne génère que les souches, les *Helpers*, les squelettes et les implémentations). Chacune de ces racines est générée lors d'un parcours de l'AST. Pour plus de détails concernant la structure de l'arbre Ada et la manière dont il est construit, voir l'Annexe B de ce mémoire.

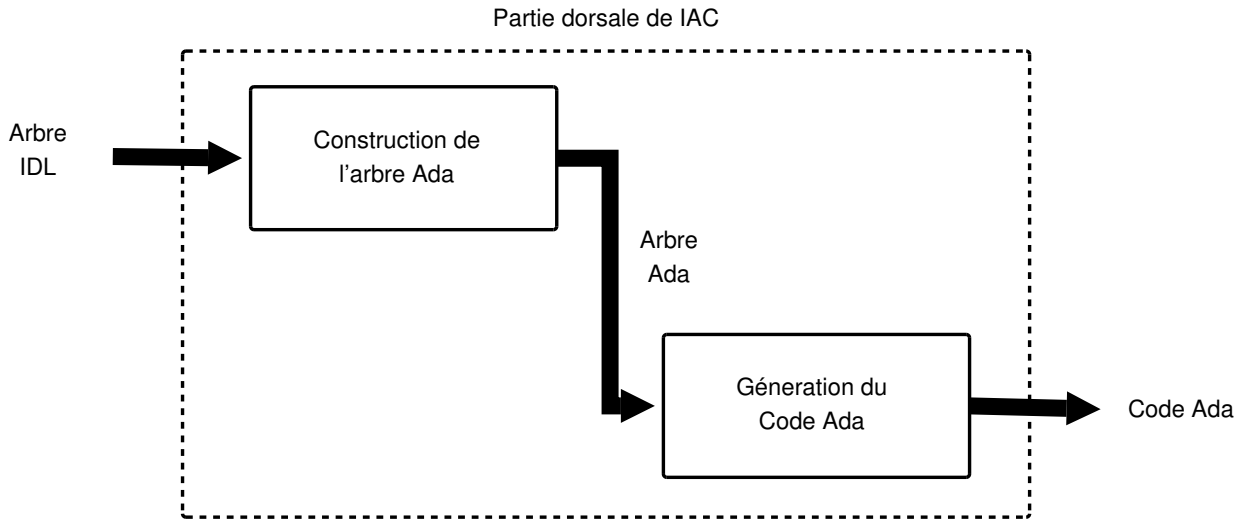

FIGURE II.3 – Partie dorsale de IAC : fonctionnement global

L'ordre de la construction des paquetages est crucial. Tout d'abord il faut construire toutes les spécifications de paquetages avant les corps de ces paquetages parce que les branches correspondant aux corps utilisent des nœuds de la branche correspondant aux spécifications. Par exemple pour construire le corps d'un sous-programme, on ne construit pas une nouvelle spécification mais on utilise celle créée lors de la construction de la partie spécification. Ceci va diminuer le nombre total de nœuds dans l'arbre Ada.

De plus, certains types de paquetages utilisent des entités déclarées dans d'autres types. Par exemple, les *Helpers* utilisent les entités déclarées dans les souches, les corps des squelettes utilisent des entités des spécifications des *Helpers* et des implémentations. L'ordre de construction qui respecte ces dépendances est le suivant :

1. Construction des spécifications des Souches
2. Construction des spécifications des *Helpers*
3. Construction des spécifications des Implémentations
4. Construction des spécifications des Squelettes
5. Construction des corps des Souches
6. Construction des corps des *Helpers*
7. Construction des corps des Squelettes
8. Construction des corps des Implémentations

Pour pouvoir accéder aux nœuds crées lors de la génération d'un type de paquetage donné, les deux arbres IDL est Ada sont liés et les *Bindings* nécessaires sont créés au fur et à mesure de la construction de l'arbre Ada (voir l'annexe B pour plus de détails sur les *Bindings* entre les deux arbres).

Une deuxième amélioration consiste à gérer de façon abstraite les noms des paquetages de l'API CORBA. Toutes les entités sont déclarées dans un paquetage séparé (`Backend.BE_Ada.Runtime`).

Si une entité change de lieu dans l'API, il suffit d'effectuer la modification dans ce paquetage. De plus la gestion de la clause `with` est effectuée de façon tout à fait transparente pour l'utilisateur : la fonction `Expand_Designator` du paquetage `Backend.BE_Ada.Expand` qui sert à générer des identificateurs complètement qualifiés ajoute les clauses `with` nécessaires au paquetage courant tout en s'assurant de la non duplication d'une même clause.

II.1.1.2.2 Génération du code Ada Une fois l'arbre Ada construit, la génération du code s'effectue en appelant la méthode `Generate` du paquetage `Backend.BE_Ada.Generator` sur la liste-racine de l'arbre Ada. Cette méthode effectue des appels *dispatchant* vers les différentes méthodes `Generate_XXXX` selon les types de nœuds rencontrés.

II.1.2 Avantages par rapport à IDLAC

Pour pouvoir comparer les parties dorsales des deux compilateurs, il est nécessaire de décrire leur fonctionnement. Celle de IAC étant décrite en II.1.1.1, voici une brève description de la partie dorsale de IDLAC :

Avant de générer le code à partir de l'arbre IDL construit dans la partie frontale, il faut savoir que les informations contenues dans cet arbre sont incomplètes. En effet, il existe plusieurs données implicites qui n'ont pas été représentées dans l'arbre comme les opérations et les attributs hérités par une interface, les membres d'une exception... La phase de l'expansion consiste à ajouter les données manquantes à l'arbre pour qu'il soit prêt à la génération de code. Il faut noter que cette phase aurait pu être omise ou plus précisément intégrée dans la phase de génération de code mais ceci va beaucoup compliquer cette dernière phase.

Après la phase d'expansion, arrive la dernière phase qui est la phase de génération de code. Lors de cette phase, l'arbre IDL est parcouru en profondeur et le code nécessaire pour chaque nœud est généré en respectant les spécifications du mapping Ada publiées par l'OMG [6]. Cette manière très intuitive de générer le code présente plusieurs inconvénients :

— Il n'y a pas de possibilité de retour en arrière, ce qui veut dire que pour générer le corps d'une procédure, il faut impérativement générer toute la partie déclarative avant la partie instructions, ou encore, dans un paquetage, il faut ajouter toutes les clauses `with` nécessaires manuellement au début. Ceci pose beaucoup de contraintes si on veut implémenter des optimisations dans le compilateur : l'endroit où implémenter ces optimisations sera imposé par le fait que le code est généré de façon linéaire. Ce qui va éventuellement rendre les sources du compilateur très difficiles à comprendre.
— Puisque la plus grande partie du code est générée en utilisant des appels à la procédure `Put_Line`, la maintenabilité du compilateur est mise en question. Par exemple, si l'API pour laquelle le code est généré est modifiée, plusieurs modifications doivent être effectuées sur IDLAC.

Le fait de séparer dans IAC la partie traitement des données de la partie génération de code apporte une grande flexibilité à ce compilateur. Le développeur ne manipule plus des chaînes de caractères qui, une fois générées dans un fichier, ne peuvent plus être modifiées. Il manipules des nœuds et des listes qu'il peut modifier à volonté tant que la phase génération de code n'a pas été entamée. Le mécanisme d'ajout des clauses `with` profite pleinement de ce concept.

De plus, la liaison (*Binding*) effectuée entre l'arbre IDL et l'arbre Ada permet d'accéder à des nœuds déjà crées et de les utiliser sans devoir créer de nouveaux nœuds. Ceci contribue à diminuer le nombre de nœuds dans l'arbre Ada.

II.1.3 Limitations

Les limitations de IAC peuvent être divisées en deux catégories, les fonctionnalités manquantes et les défauts d'architecture.

II.1.3.1 Les fonctionnalités manquantes

Au début du stage, il y avait tellement de fonctionnalités manquantes dans la partie dorsale de IAC qu'il n'arrivait pas à générer du code Ada correct pour le simple fichier IDL suivant :

```
module Tpcorba {

  exception divisionParZero {};

  interface Calcul  {
    void Bid_Rac(inout long Data);
    void Bid_Carre(inout long Data);
  };
};
```

Ceci m'a empêché d'entamer directement la partie optimisation dès le début du stage car il fallait un compilateur capable de compiler la plupart du langage IDL. Il a fallu alors une période relativement longue (la moitié de la durée du stage à temps plein, ensuite en parallèle avec les optimisations) pour implémenter toutes les fonctionnalités manquantes dans la partie dorsale de IAC. Les fonctionnalités qui ont été implémentées dans IAC sont détaillées dans II.2.2 et dans II.2.3.

II.1.3.2 Les défauts d'architecture

Au fur et à mesure de l'ajout de nouvelles fonctionnalités dans IAC je me suis aperçu que la partie expansion, jusque là inexistante, était nécessaire pour résoudre plus simplement certains problèmes rencontrés, notamment pour traiter certains cas particuliers des *forward* d'interfaces. J'ai constaté aussi de la réplication de code à plusieurs endroits dans les source de IAC chose qui rend très fastidieuse la correction des bogues et la modification des sources.

II.2 Nouvelle architecture

Dans la partie II.2.1, je décrirai la nouvelle architecture de la partie dorsale de IAC. Cette nouvelle architecture a rendu IAC plus flexible et a facilité considérablement l'implémentation des optimisations décrites dans III.

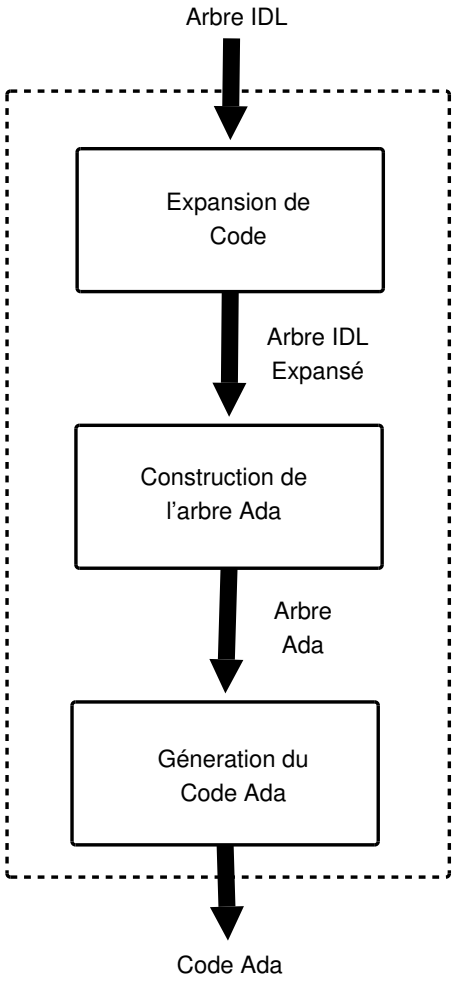

FIGURE II.4 – Partie dorsale de IAC : nouvelle architecture

II.2.1 Nouveau schéma de fonctionnement

II.2.1.1 Expansion de code

Comme il a été précisé dans II.1.3.2, une partie expansion s'est avérée nécessaire pour rendre la génération de code plus simple et pour garder une structure flexible pour IAC. Plusieurs fonctionnalités seront implémentées directement dans la partie expansion, comme l'ajout des *forward* implicites d'interfaces imposé par la nature du langage Ada. D'autres fonctionnalités existant déjà dans le cœur de la partie dorsale devraient être déplacées dans la partie expansion pour alléger la construction de l'arbre Ada.

La nouvelle partie dorsale de IAC décrite par la figure II.4 se compose désormais de 3 parties :

La partie expansion : qui prend en entrée l'arbre IDL construit par la partie frontale (AST), le modifie pour générer en sortie un arbre IDL qui donne lieu à une génération de code Ada beaucoup plus simplement.

La construction de l'arbre Ada : qui, en parcourant l'arbre IDL expansé, génère un arbre

Ada qui décrit la structure des souches et des squelettes CORBA de l'application répartie en cours de développement. Le parcours de l'arbre IDL s'effectue à l'aide des procédures `Visit_<Frontend_Node>` où `<Frontend_Node>` désigne un type de nœud de l'arbre IDL. La construction de l'arbre Ada se fait en utilisant les fonctions `Make_<Backend_Node>` du paquetage `Backend.BE_Ada.Nutils` où `<Backend_Node>` désigne un type de nœud de l'arbre Ada.

La génération de code Ada : Une fois l'arbre Ada construit, la génération de code se fait à l'aide des procédures `Generate_<Backend_Node>` du paquetage `Backend.BE_Ada.Generator`.

II.2.1.2 Factorisation de code

Lors de la construction de l'arbre Ada, la création de certains types de nœuds nécessite l'accès à d'autres nœuds déjà crées. Par exemple pour pouvoir créer un appel vers une fonction `To_Any` d'un certain type, il faut connaître le type du paramètre pour créer l'appel vers la bonne fonction Au fur et à mesure de l'évolution de IAC, de nouveaux cas de figure pour l'accès au nœuds sont apparus et la duplication de code allait rendre le développement fastidieux. Le code a donc été factorisé en sous-programmes. Pour accéder aux fonctions `To_Any` et `From_Any` d'un type donné, le développeur doit utiliser les fonctions `Get_From_Any_Node` et `Get_To_Any_Node` du paquetage `Backend.BE_Ada.Nutils`. Ces deux fonctions effectuent les tests de tous les cas de figure possibles et retournent le bon nœud.

II.2.2 Fonctionnalités implémentées dans la partie frontale

La mise à niveau de la partie frontale vers la version la plus récente de CORBA 3.0.4 est quasiment achevée. Il ne manque plus que l'analyse des *import* qui impose que le parseur de IAC doive être repensé et qui, pour des contraintes temporelles n'a pu être achevée par manque de temps. Les *typeprefix* et les *typeid* ont été implémentés mais ne sont pas complètement testés (notamment tous les cas particuliers donnés dans la specification CORBA [7]).

Les fonctionnalités qui ne sont pas implémentées dans PolyORB (les *valuetypes* par exemple) ne sont implémentées ni par IAC ni par IDLAC.

Certaines anomalies de fonctionnement trouvées dans la partie frontale ont été corrigés :

— Dans une opération, une structure ou une union, il n'était pas permis d'utiliser un paramètre (ou membre) de même nom que son type, ce qui est légal. Bien entendu, le type dont il est question est un type interface, structure ou union ou un type qui redéfinit un type de base car il est interdit d'utiliser les mots-clés comme des identificateurs.

— Une des optimisations apportées par la partie frontale de IAC consiste à évaluer les expressions littérales avant de les intégrer dans l'arbre IDL le type de l'expression n'est pas changé en fonction de sa nouvelle valeur. Exemple :

```
const short a = 1 - 2;
```

La constante `a` vaut `-1`, mais le type de la valeur littérale `-1` reste toujours non signé (car le type des constantes `1` et `2` est non signé) ce qui génère une erreur lors de la vérification des bornes à la phase d'analyse.

II.2.3 Fonctionnalités implémentées dans la partie dorsale

II.2.3.1 Fonctionnalités complétées

II.2.3.1.1 La gestion des exceptions : Au début du stage, IAC ne générait pas la constante `<Nom_Exception>_Repository_Id`. Il ne générait pas non plus le corps de la fonction `Get_Members` imposée par les spécification du mapping Ada [6]. Il manquait aussi les fonctions `From_Any`, `To_Any` et la constante `TC_<Nom_Exception>` qui sont nécessaires pour la conversion des *members* de l'exception ainsi que la procédure `Raise_Exception`.
À la fin du stage, les exceptions d'utilisateurs, les exceptions système et les exceptions inconnues sont parfaitement gérées par IAC.

II.2.3.1.2 Les interfaces : Il manquait la constante `<Nom_Operation>_Repository_Id`. Les corps des différents sous programmes déclarés dans la *spec* (les souches) (construction des requêtes, empaquetages des paramètres...) sont incomplets, il manque la gestion des paramètres de mode `out` et `inout`. Toutes ces fonctionnalités manquantes ont été ajoutées et sont opérationnelles.

II.2.3.1.3 Le squelette : J'ai implémenté la gestion des exceptions et la mise à jour des paramètres `out` et `inout` d'une opération.

II.2.3.2 Fonctionnalités Ajoutées

II.2.3.2.1 L'initialisation des paquetages *Helper* et des squelettes : Cette partie est nécessaire pour le bon fonctionnement de l'application répartie et pour que PolyORB puisse initialiser les différents paquetages dans le bon ordre.

II.2.3.2.2 Les séquences : La génération de code pour les séquences bornées et non bornées qui est une fonctionnalité vitale pour un compilateur IDL a été implémentée et testée avec succès.

II.2.3.2.3 L'héritage des interfaces : le langage IDL, permet à une interface d'hériter d'une ou plusieurs autres interfaces. Dans ce cas, l'interface fille possède toutes les opérations et les attributs des interfaces parentes. De plus, toutes les définitions de types et les déclarations de constantes et d'exceptions qui existent dans les interfaces parentes doivent être visibles dans l'interface fille. Pour cela les spécifications du mapping Ada [6] imposent de "renommer" (au sens Ada du terme) les constantes et les exceptions ou de créer des sous-types dans le mapping de l'interface fille. Visiblement, IDLAC n'a pas choisi cette voie. Il utilise directement les constantes, exceptions ou types à partir du mapping de l'interface parente.
La partie frontale de IAC semble s'être engagée sur la même voie que IDLAC. En partant de l'exemple ci dessous :

```
module m {
  interface int1 {
    typedef long T1;
    typedef short T2;
```

```
    long op1();
    attribute long attr1;
    exception myException {};
  };

  interface int2 : int1 {
    typedef string T2;
    string op3() raises (myException);
    attribute string attr3;
    T1       op31();
    T2       op32();
    int1::T2 op33();
  };
};
```

L'interface `int2` hérite de l'interface `int1`. Elle doit donc pouvoir accéder au type `T1` défini par `int1` comme s'il était définit par elle.
Par contre, dans la ligne "`T2 op32();`", le type `T2` désigne le type défini dans `int2`. La ligne "`int1::T2 op32();`" ne présente aucune ambiguïté. En utilisant le backend IDL de IAC, le résultat est le suivant :

```
module m {
  interface int1 {
    typedef long T1;
    typedef short T2;
    long op1();
    attribute long attr1;
    exception myException {
    };
  };
  interface int2 : m::int1 {
    typedef string T2;
    string op3() raises (m::int1::myException);
    attribute string attr3;
    m::int1::T1 op31();
    m::int2::T2 op32();
    m::int1::T2 op33();
  };
};
```

On voit bien donc que :

- `int2` utilise les types et les exceptions de `int1`.
- Le type `T2` a été redéfini dans `int2`.
- L'accès à la version de `T2` définie dans `int1` demeure possible en indiquant les noms complets.

La génération de code pour l'héritage multiple d'interfaces a été implémentée dans la partie dorsale de IAC. Les constantes et les définitions de types et d'exceptions dans les interfaces parentes sont

renommées dans l'interface fille pour permettre au client d'avoir accès à ces entités. Si un conflit est détecté, par exemple, si deux parents contiennent des définitions de mêmes noms, seule la première est renommée. Les autres seront ignorées.

II.2.3.2.4 Les tableaux multidimensionnels et imbriqués : Cette fonctionnalité était absente au début du stage. Elle était parmi les premières fonctionnalités que j'ai implémenté dans IAC. Cependant suite à une modification de la façon dont PolyORB encode les tableaux, j'ai été amené à la réimplementer conformément à la nouvelle spécification de PolyORB. Actuellement, le code généré pour les tableaux multidimensionnels et imbriqués est correct.

II.2.3.2.5 L'utilisation de types crées par des interfaces : les interfaces IDL sont des constructeurs de types, on doit donc pouvoir utiliser le nom d'une interface comme type de paramètre dans une opération par exemple.

II.2.3.2.6 Les interfaces locales : Ces types d'interfaces diffèrent des interfaces classiques par le fait qu'elles ne peuvent faire l'objet d'une communication entre deux entités différentes (client et serveur). Ceci a pour conséquence la non génération de squelettes et de fonction de conversion à partir et vers le type `Any` pour ce type d'interfaces.

II.2.3.2.7 Les interfaces Abstraites : Ce sont des interfaces qui ne peuvent être utilisées directement mais doivent être héritées par des interfaces ou des *valuetypes*. La partie concernant les interfaces dans les interfaces abstraites à été implémentée avec succès.

II.2.3.2.8 Les unions : La génération de code pour les types *union* est maintenant implémentées et testée avec succès. Ces types ont l'avantage d'économiser l'empreinte mémoire par rapport aux simples structures.

II.2.3.2.9 La *forward declaration* des interfaces : C'est une fonctionnalité vitale pour un compilateur IDL. Cette fonctionnalité a été implémentée avec succès dans la partie dorsale de IAC. Certains cas particuliers où on doit ajouter des *forward declarations* pour éviter le problème de la dépendance mutuelle de deux paquetages Ada ont été résolus par la partie expansion de code.

II.2.3.2.10 La génération de commentaires : Pour rendre le code généré plus lisible, des commentaires Ada ont été générés à différents endroits des souches et des squelettes. Des entêtes de commentaires ont été générés pour préciser que les fichiers sont générés automatiquement et qu'il ne faut pas les modifier à la main (sauf pour les fichiers `Impl`).

II.2.3.2.11 Traitement des fonctions implicites de CORBA : dans CORBA le client peut invoquer des méthodes implicites. Pour traiter ces méthodes, un nouveau paquetage est ajouté de la personnalité CORBA de PolyORB : `PolyORB.CORBA_P.Implicit_CORBA_Methods`.

II.2.3.2.12 Analyse du module CORBA : D'après le chapitre 5 de [6], le fichier `orb.idl` contient un module particulier, le module CORBA. Pour des raisons relatives à la nature du langage Ada, l'arbre IDL relatif au fichier `orb.idl` doit être restructuré. Après cette modification, deux nouveaux sous modules du module CORBA sont crées :

1. `CORBA::Repository_Root` qui contient toutes les *forward* interfaces et les entités relatives à l'*Interface Repository*
2. `CORBA::IDL_Sequences` qui contient la déclaration de types séquences définis à partir des type de base CORBA

La raison principale de cette restructuration est que les entités des deux sous-paquetages ne sont pas utilisées par la plupart des applications réparties et, étant donné que le *forward* d'interface et les types *sequence* impliquent des instanciations de paquetages génériques en Ada, la taille du paquetage CORBA de l'intergiciel deviendrait énorme ce qui augmenterait l'empreinte mémoire de l'application répartie.

L'analyse du fichier `orb.idl` sert dans la plupart des cas uniquement à faire entrer le *scope* du module CORBA dans le *global scope* et ne doit généralement pas donner lieu a une génération de code Ada. Il se trouve que certains paquetages dont le mapping Ada [6] recommande l'existence sont des souches et des squelettes d'entités se trouvant dans le fichier `orb.idl` et leur intégration dans le dépôt contraint le principe disant qu'il ne faut pas intégrer du code généré automatiquement dans un dépôt. Ces paquetages sont générés automatiquement lors de la compilation de PolyORB. Pour plus de détails sur l'analyse du module CORBA, voir l'annexe C.

II.2.4 Validation de la conformité

La validation des différentes fonctionnalités ajoutées s'est effectuée en utilisant une suite très riche de tests de régression. Les différentes catégories de tests pour IAC sont :

1. **ada0001 → ada0009 :** Les tests de la bonne génération des spécifications des paquetages pour les souches et les squelettes. Les spécifications générées sont compilées avec des corps de paquetages générés par IDLAC. Ces tests couvrent une large partie des la génération de code, tels que les types, les unions, les structures, les exceptions...
2. **ada0010 → ada0018 :** Les tests de la bonne génération des corps des paquetages pour les souches et les squelettes. Ce sont les mêmes tests que la suite précédente mais là, tout le code Ada est généré par IAC.
3. **ada0019 → ada0022 :** Les tests pour la redéfinition de types de base CORBA.
4. **chicken-egg, circular et forward01 → forward01 :** Les tests pour les *forward declarations* des interfaces. L'exemple est tiré de la spécification CORBA [7]. Il y a aussi les tests pour les cas particuliers où le compilateur doit générer des *forward declarations* implicites pour les besoins du langage Ada.
5. **inherit001 → inherit005 :** Les tests pour l'héritage des interfaces. La complexité de ces tests augmente avec leur numéro, allant du simple héritage jusqu'à arriver vers l'héritage multiple avec redéfinition des constantes, types et exceptions.
6. **local001 → local002 :** Les tests pour les interfaces locales.
7. **abstract001 :** Test pour les interfaces abstraites.
8. **idl* :** Les tests vérifiant le bon fonctionnement du parseur, lexeur et analyseur et la bonne génération des messages d'erreur.

9. **test001 → test054 :** Tests vérifiant la bonne construction de l'AST.
10. **corba-idl :** Test vérifiant que le module CORBA est bien analysé.

En plus de tous les tests cités ci-dessus et qui sont passés avec succès, trois exemples d'application répartie sont fournis pour tester le bon fonctionnent du code généré par IAC :

all_functions : teste tous les modes de passages de paramètres possibles dans une opération. Ce test vérifie aussi le bon fonctionnement de l'asynchronisme (opérations *oneway*).

all_types : teste tous les types de données qu'on peut déclarer dans un fichier IDL (séquences, unions, tableaux, types de base...)

all_types_local : exactement le même que all_types mais avec une interface locale.

Chapitre III

Vers un intergiciel optimisé et déterministe

La partie mise à niveau de IAC à pris un peu plus de temps que prévu (un peu plus de la moitié du stage) à cause du nombre important de fonctionnalités manquantes. Le début de la phase d'optimisation ne pouvait se faire sans la présence d'un compilateur parfaitement opérationnel qui compile une large variété de fichiers source IDL.

Il a fallu donc choisir les optimisations les plus importantes parmi celles que j'ai proposé dans le mémoire bibliographique [13] et rejeter les autres. Le deux optimisations qui ont été jugées les plus importantes sont :

1. L'apport du déterminisme dans le traitement des requêtes par les squelettes en utilisant les tables de hachage minimales parfaites.
2. L'implémentation de la SII.

III.1 Utilisation des tables de hachage minimales parfaites

Quand le squelette d'une application répartie reçoit une requête à traiter, il compare le nom de la requête avec les noms de toutes les opérations de l'interface jusqu'à trouver la bonne opération et appeler son implémentation. Cette méthode de recherche d'opérations est adoptée par IDLAC et a été reprise dans IAC. Par exemple si une interface IDL contient dix opérations :

```
long echoLong01 (in long data) ;
long echoLong02 (in long data) ;
...
long echoLong10 (in long data) ;
```

La structure du squelette va avoir cette allure :

```
if (Operation = "echoLong01") then
   <Traitement>;
```

```
elsif (Operation = "echoLong02") then
   <Traitement>;
...
elsif (Operation = "echoLong10") then
   <Traitement>;
else
   <Erreur>;
end if;
```

Cette manière très intuitive de traiter les requêtes à un inconvénient principal : le temps que met le serveur pour trouver une opération est directement lié à la place où cette opération est déclarée (au début ou à la fin de l'interface).

L'optimisation consiste à rendre le temps de recherche du nom de l'opération constant quelque soit l'endroit où elle est déclarée. Ceci est effectué en utilisant les tables de hachages minimales parfaites.

Une fonction de hachage h est une fonction qui associe les éléments d'un ensemble de mots W de taille m aux éléments d'un intervalle d'entiers $[0, k-1]$. La fonction $h(w)$ calcule généralement une adresse ou un indice dans un tableau où le mot w est stocké. Ce tableau est connu aussi sous le nom de "Table de Hachage". On dit qu'il y a une collision si deux ou plusieurs mots ont le même code de hachage. Une fonction de hachage *parfaite* est une fonction injective. Une fonction de hachage minimale parfaite est une bijection de l'ensemble des mot W vers l'intervalle $[0, k-1]$ ce qui implique que les tailles des deux ensembles soient égales.

Le compilateur Ada GNAT implémente un algorithme de génération de fonctions de hachage minimales parfaites. C'est un algorithme probabiliste écrit par Czech, Havas et Majewski en 1992 [3]. La génération statique de fonctions de hachage minimales parfaites impose la contrainte de connaître tous les mots de l'alphabet avant l'exécution de l'algorithme. Il se trouve que c'est le cas des squelettes : à la compilation d'un fichier source IDL, on a accès aux noms de toutes les opérations. L'algorithme implémenté peut fonctionner en deux modes : un mode optimisant l'empreinte mémoire des exécutables et un autre mode optimisant le temps de calcul. Les deux modes sont supportés par IAC et l'utilisateur peut choisir l'optimisation qui lui convient.

III.1.1 Approche

III.1.1.1 Première approche

Une première approche a consisté à construire pour chaque opération, une procédure `Invoke_<Nom_Operation>`. La fonction de hachage est générée en même temps que les souches et les squelettes. Les *hash codes* sont des indices dans un tableau de pointeurs sur les procédures `Invoke_<Nom_Operation>`. Le rôle de la procédure `Invoke` est de calculer le *hash code* d'une opération et, ensuite, d'invoquer la bonne procédure ou de générer un erreur si l'opération n'existe pas.

Cette approche a très vite été écartée car les indirections induites par l'utilisation de pointeurs sur des procédures n'améliorent les performances que lorsque le nombre d'opérations est très grand (voir les résultats des tests en III.1.2).

III.1.1.2 Deuxième approche

La deuxième approche qui a été retenue est très proche de la première mais dans ce cas, le table de hachage ne contient plus des pointeurs sur des procédures mais les chaînes de caractères elles-mêmes pour des raisons de vérification. La structure conditionnelle dans la procédure `Invoke` est remplacée par un *switch case* dont les paramètres sont les *hash codes* (qui appartiennent à un type scalaire et peuvent donc être utilisés dans un *switch case*). Cette approche fait augmenter considérablement les performances vu que le *switch case* effectue des sauts directs et ne teste pas toutes les conditions (cf III.1.2).

III.1.2 Gains obtenus

Tous les tests ont été effectués sur un PC compatible IBM qui possède les caractéristiques suivantes :

Processeur : Intel(R) Pentium(R) 4 3.00 GHz

Mémoire vive : 1 Go

OS : Debian GNU Linux (unstable) avec un noyau 2.6.8

Compilateur Ada : GNAT Pro 5.03a (20050114-34)

Le test consiste en une application répartie contenant une centaine d'opérations :

```
interface test_hash {
   long echoLong00 (in long data) ;
   long echoLong01 (in long data) ;
   ...
   long echoLong99 (in long data) ;
}
```

Le script du test automatique compile trois fois l'application répartie : une première fois sans utiliser l'optimisation des tables de hachage minimales parfaites, une deuxième fois en optimisant le temps de calcul et une troisième fois en optimisant l'empreinte mémoire. Après chaque compilation, le serveur et le client sont exécutés. Le client effectue 4 types d'invocations un nombre donné de fois (donné en ligne de commande) :

1. En exécutant seulement la première opération
2. En exécutant seulement la dernière opération
3. En exécutant toutes les opérations dans l'ordre de façon circulaire
4. En exécutant toutes les opérations dans un ordre aléatoire

Type d'optimisation	Pas d'optimisation	Temps CPU	Empreinte mémoire
Client	2165	2165	2165
Serveur	2979.55	2979.64	2979.27

TABLE III.1 – Taille des exécutables (Kilo-Octets) en fonction du type d'optimisation

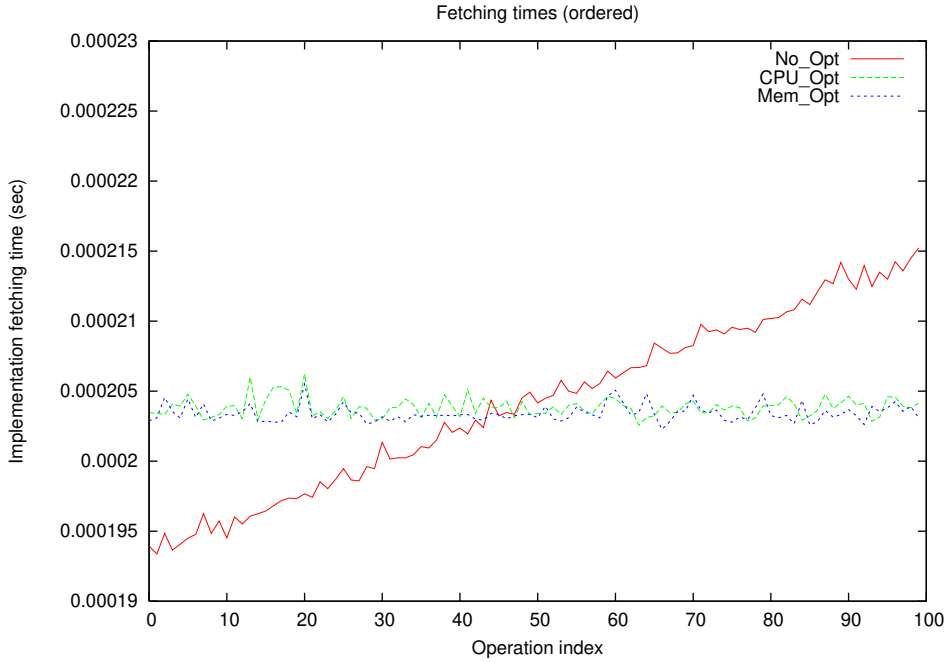

FIGURE III.1 – Première approche d'utilisation des tables de hachage

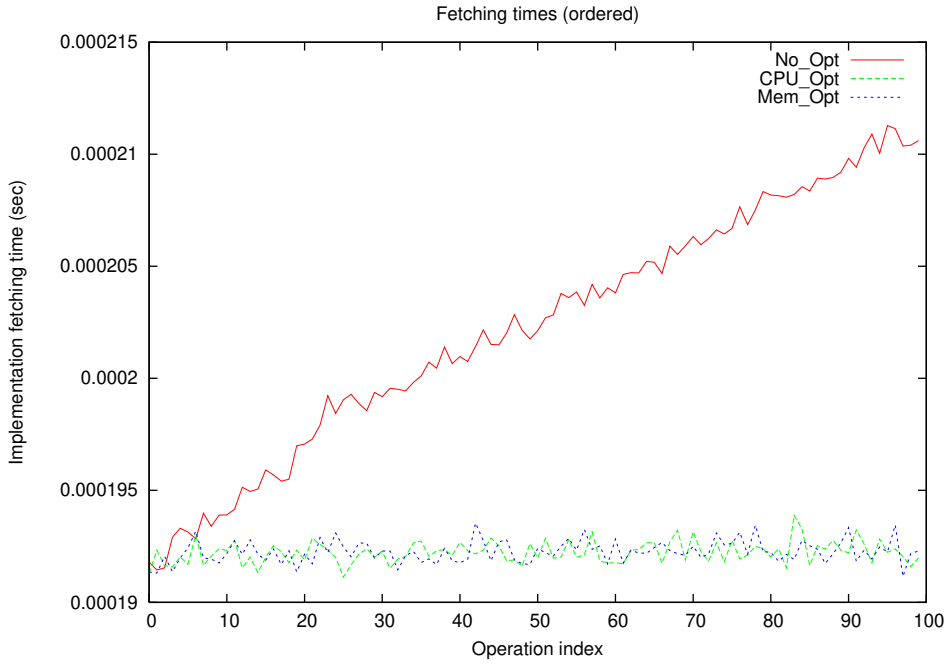

FIGURE III.2 – Deuxième approche d'utilisation des tables de hachage

Les temps d'exécution sont récupérés dans des fichiers qui donnent lieu ensuite à un graphique grâce à un script GNU Plot. La figure III.1 montre les temps d'exécution en fonction de l'indice de l'opération en utilisant la première approche (celle des pointeurs). La figure III.2 montre les mêmes résultats mais en utilisant la deuxième approche (celle du *switch case*). On voit bien l'amélioration de performance apportée par le seconde approche : on gagne dès l'utilisation de la première opération. Le tableau III.1 montre les tailles des exécutables pour les trois cas de figure. Bien entendu, puisque le client n'utilise pas le squelette de l'application répartie, sa taille reste inchangée. Quant au serveur, sa taille lors de l'optimisation CPU est un peu plus grande que celle du cas sans optimisation. Ceci est dû à l'utilisation de tables relativement grandes pour les calcul des *hash codes*. L'optimisation mémoire permet d'obtenir la plus petite taille mémoire. Étant donné que les performances sont quasiment les mêmes qu'avec l'optimisation CPU (au contraire de ce qui est affirmé dans l'article [3]), c'est cette optimisation qu'il est conseillé d'utiliser.

III.2 L'implémentation de la SII

L'intergiciel PolyORB traite toutes les requêtes d'une manière dynamique. Ce mode de traitement s'appelle la DII (*Dynamic Invocation Interface*). Il a l'avantage de permettre le traitement d'une requête même si la signature exacte de l'opération n'est pas connue à la compilation de l'application répartie. Dans ce type de traitement, le client construit une liste des paramètres convertis au type `Any` (la *NVList*). Cette liste est passée à la couche protocolaire qui se charge de faire l'empaquetage des paramètres après les avoir convertis une deuxième fois vers leur types d'origine. Du coté serveur, le processus inverse est effectué : les paramètres sont désempaquetés du *buffer*, convertis au type `Any` et mis dans une *NVList* par la couche protocolaire de l'intergiciel. Le squelette de l'application répartie se charge de convertir les paramètres vers leur type d'origine avant d'appeler l'implémentation. Pour envoyer le résultat d'une requête et les différent paramètres `out` et `inout`, le même travail est effectué. La figure III.3 décrit le fonctionnement de la DII dans PolyORB.

On voit bien d'après la figure III.3 qu'il existe plusieurs conversions vers et à partir du type `Any`. Si la signature de l'opération est connue à la compilation, il serait judicieux d'empaqueter les paramètres directement sans les convertir vers le type `Any`. C'est le principe de la SII (*Static Invocation Interface*). La figure III.4 décrit le fonctionnement de la SII. Au contraire de la DII, la couche applicative du client ne remplit pas une *NVList* mais met à jour les champ d'un enregistrement. Ces champs représentent les différents paramètres de l'opération en plus du résultat s'il s'agit d'une fonction. Ainsi, on arrive à éviter l'utilisation inutile des types `Any`.

III.2.1 Approche

Pour pouvoir implémenter la SII, il faut que chaque opération dispose de ses propres fonctions d'empaquetage (*marshalling*) et de désempaquetage (*unmarshalling*) des paramètres. Ces deux fonctions sont générées automatiquement par IAC dans un sous-paquetage séparé, de même niveau que les `Helpers` et qui est appelé `CDRs` pour (*Common Data Represenatation*). En plus de ces deux sous programmes, le paquetage `CDRs` contient la déclaration du type enregistrement qui contient les paramètres et le résultat. Ce type hérite du type abstrait `Request_Args` décrit dans le paquetage `PolyORB.Requests` :

```
type Request_Args is abstract tagged null record;
```

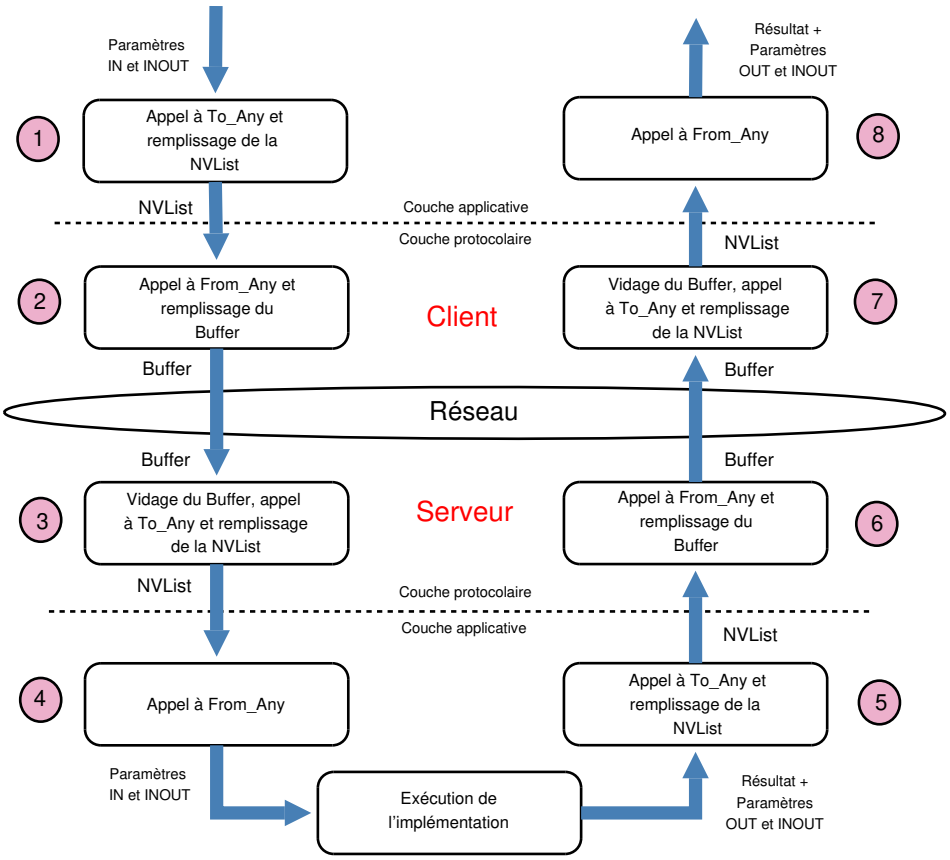

FIGURE III.3 – Fonctionnement de la DII

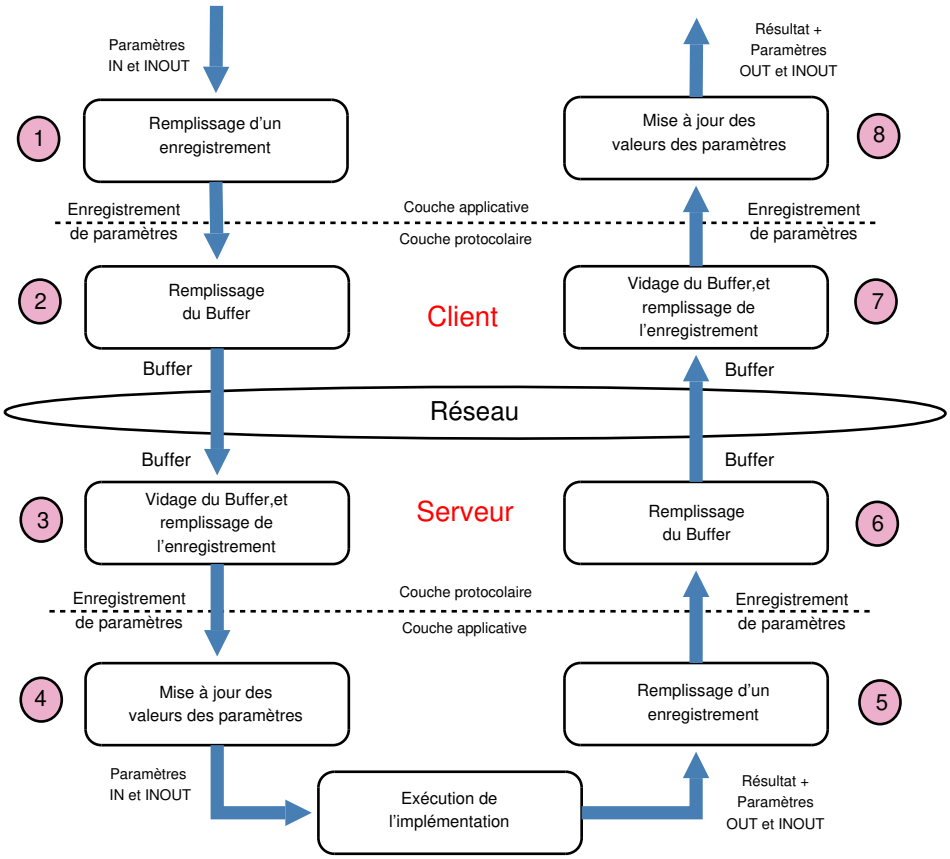

FIGURE III.4 – Fonctionnement de la SII

```
--  This type should be extended in order to contain the request arguments

type Request_Args_Access is access all Request_Args'Class;
```

Un troisième sous programme est généré automatiquement pour mettre à jour le nouveau champ qui a été ajouté à la structure de la requête : c'est le champ `Payload` qui est un pointeur vers une variable de type `Request_Payload`. Ce dernier type est un enregistrement qui contient un champ de type `Request_Args_Access`. Il est aussi déclaré dans `PolyORB.Requests` :

```
type Request_Payload is tagged record
   Args : Request_Args_Access;
end record;
--  This type may be extended to add accesses to subprograms that handle the
--  arguments

type Request_Payload_Access is access all Request_Payload'Class;
```

La personnalité protocolaire GIOP étend ce type pour ajouter deux pointeurs vers les fonctions d'empaquetage et de désempaquetage :

```
--  The Payload-derived type which contains the accesses

type Operation_Payload is new PolyORB.Requests.Request_Payload with
   record
      From_CDR : CDR_Subprogram_Type;
      To_CDR   : CDR_Subprogram_Type;
   end record;
```

Il existe quatre endroits dans lesquels les paramètres de l'opération sont traités :

1. L'empaquetage des paramètres IN et INOUT par le client
2. Le désempaquetage des paramètres IN et INOUT par le serveur
3. L'empaquetage des paramètres OUT et INOUT par le serveur
4. Le désempaquetage des paramètres OUT et INOUT par le client

Dans ces quatre endroits, un test est ajouté. On vérifie si le champ `Payload` de la requête a bien été mis à jour. Si oui, on utilise la SII on appelant les fonctions générées automatiquement. Sinon, on utilise la DII exactement comme si aucune modification n'avait été apportée.

III.2.2 Gains obtenus

La génération automatique des fonctions d'empaquetage et de désempaquetage est opérationnelle pour les types suivants :

— Les long
— Les unsigned long
— Les short

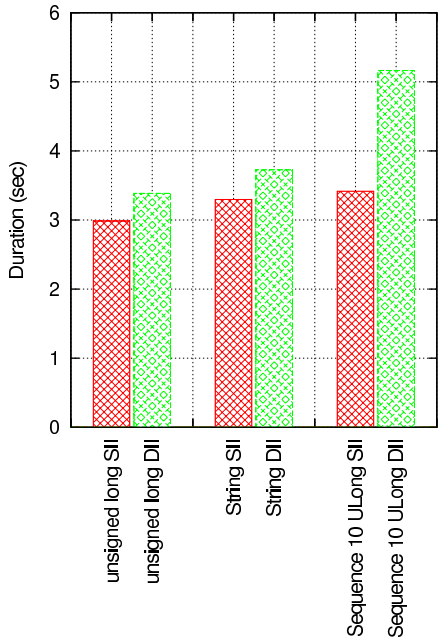

FIGURE III.5 – Comparaison de la SII et la DII pour certains types (10000 itérations)

— Les string
— Les séquences de types supportés (bornées et non bornées)

Le principe du test de performance est le suivant : effectuer un grand nombre de fois des appels vers des fonctions `echo<Type>` qui prennent comme paramètre une variable d'un certain type et se contentent de retourner la même variable. Les tests sont effectués sur une plate-forme identique à celle de III.1.2.

La figure III.5 donne une comparaisons des temps d'exécution de 10000 requêtes pour les *unsigned long*, *string* et séquences de 10 *unsigned long*. Les gains en performance sont très clairs :

— Pour les *unsigned long* : 11%
— Pour les *string* : 11%
— Pour les séquences : 33%

Test	Unsigned Long		String		Séquence	
Entité	**Client**	**Serveur**	**Client**	**Serveur**	**Client**	**Serveur**
DII	1811.57	2497.05	1813.21	2495.85	2050.73	2774.70
SII	1811.46	2496.06	1819.59	2501.67	2053.74	2767.87

TABLE III.2 – Taille des exécutables (Kilo-Octets) en SII et DII

Le tableau III.2 donne les tailles des exécutables utilisés dans le dernier test. On remarque qu'il n'existe pas de règle générale pour la modification de l'empreinte mémoire de ces exécutables en passant de la DII à la SII. Dans le cas des `long`, la taille mémoire du client et du serveur diminue en passant à la SII. Elle augmente dans le cas des `string`. Dans le cas des séquences, cette taille augmente pour le client et diminue pour le serveur. Ceci est dû au fait que le passage à la SII nécessite l'ajout d'un code supplémentaire à l'application (la structure contenant les paramètres et les deux procédures de stockages des paramètres). Il implique aussi la non utilisation des fonctions

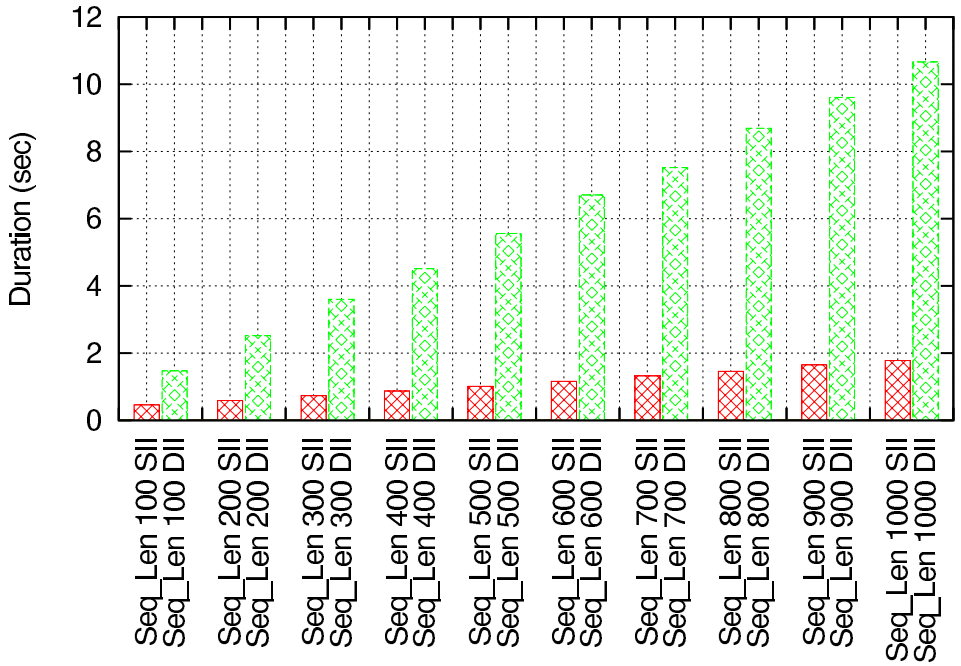

FIGURE III.6 – Comparaison de la SII et la DII pour les séquences (1000 itérations)

de stockage de PolyORB utilisant la DII. Toutefois, les tailles des exécutables sont très proches et les différences ne dépassent jamais la dizaine de Kilo-Octets.

La figure III.6 montre l'évolution des gains en performance pour l'exécution de 1000 itérations pour les séquences en changeant leur longueur. On remarque que plus les séquences sont longues, plus le gain en performance est important. Dans le cas de la DII, le temps d'exécution augmente très rapidement en fonction de la longueur de la séquence tandis que dans le cas de la SII, cette croissance est lente.

Chapitre IV

Conclusions et perspectives

IV.1 Conclusions

À la fin de ce stage, je peux dire que les objectifs qui ont été fixés au départ ont été atteints : le compilateur IAC est maintenant au niveau de IDLAC. Le remplacement de IDLAC par IAC dans les sources de PolyORB ne nécessite plus qu'un minime effort. IAC est un compilateur flexible et se prête à la maintenance beaucoup plus facilement que IDLAC.

Les deux optimisations qui ont été implémentées dans IAC ont beaucoup augmenté les performances des applications réparties générées par ce compilateur aussi bien en ajoutant le déterminisme (tables de hachage minimales parfaites) qu'en diminuant considérablement le temps d'exécution des requêtes (SII).

Maintenant que les tests ont montré le grand apport en performance de la SII, la poursuite de la génération de code pour le reste des types (structures, unions, tableaux) qui est une tâche facile mais qui va prendre un temps considérable peut être entamée.

IV.2 Perspectives

Certaines fonctionnalités n'ont pu être ajoutées dans IAC par manque de temps, il serait intéressant de commencer par les implémenter avant de poursuivre l'évolution de IAC et pour pouvoir remplacer définitivement IDLAC par IAC :

1. L'achèvement des *imports* qui va nécessiter l'apport de modifications profondes dans la structure du *Parser*.
2. L'implémentation de la génération de code pour les types anonymes (cf 3.11.6 de [7]). Bien que l'utilisation de tels types soit obsolète dans CORBA 3.0, certains fichiers source IDL de PolyORB les contiennent. L'implémentation de cette fonctionnalité devrait être effectuée dans la partie expansion de la partie dorsale.
3. Les chaînes de caractères bornées.

Deux fonctionnalités qui sont très utiles dans un compilateur IDL, et qui constitueraient des enri-

chissements de IAC par rapport à IDLAC (avec l'optimisation des squelettes et l'implémentation de la SII) seraient :

1. La génération de code pour les *forward declarations* de structures et d'unions. Une telle implémentation ne serait pas aussi facile que la *forward declaration* des interfaces. En effet, dans la spécification CORBA, la possibilité d'une telle déclaration est précisée. Cependant, dans les spécifications du mapping Ada [6], la marche à suivre n'est pas donnée.
2. L'implémentation de la génération de code pour les *valuetypes* dans IAC et dans PolyORB.

Une optimisation très intéressante qui pourrait aussi être implémentée dans IAC dans la continuité de la SII, est l'allocation statique des *buffers* en calculant à l'avance leur taille. Ce n'est pas une tâche très simple vu que cette taille dépend, en plus du nombre et de la nature des paramètres, de la version du protocole GIOP. Il faudra aussi implémenter les mécanismes d'allocation statique dans le paquetage `PolyORB.Buffers`. On pourrait aussi envisager d'utiliser les mêmes procédures d'empaquetage et de désempaquetage pour les opérations ayant les mêmes signatures. Ceci apportera un gain en taille mémoire et peut s'avérer utile pour les applications embarquées.

Enfin, voici quelques conseils que je donne à tous ceux qui veulent apporter des modifications ou des optimisations à IAC :

— Pour les liaisons entre les arbres IDL et Ada, utiliser les fonctions `Get_XXXX_Node` déclarées dans le paquetage `Backend.BE_Ada.Nutils` en priorité sur les fonctions `XXXX_Node` du paquetage `Backend.BE_Ada.Nodes`. Car les premières contiennent beaucoup de tests et évitent la réplication de code.
— Préférer la fonction `Get_Correct_Parent_Unit_Name` du paquetage `Backend.BE_Ada.Nutils` à la fonction `Get_Parent_Unit_Name` du paquetage `Backend.BE_Ada.Nodes` quand cela est possible.
— Ne pas hésiter à créer ses propres liens entre les deux arbres en suivant la même démarche que pour les liens déjà créés.

Bibliographie

[1] Imran Ahmad and Shikharesh Majumdar. Achieving High Performance in CORBA-Based Systems with Limited Heterogeneity. In *ISORC*, pages 350–359, 2001.

[2] John Barnes. *Programmer en Ada 95*. Traduction de Hugues Foconnier, 2éme édition, Vuibert, 2000.

[3] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An Optimal Algorithm for Generating Minimal Perfect Hash Functions. In *Information Processing Letters*, pages 257–264, Oct 1992.

[4] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick : A Flexible, Optimizing IDL Compiler. In *Proceedings of the ACM SIGPLAN'97 Conferance PLDI*, Las Vegas, NV, June 1997.

[5] Aniruddha Gokhale and Douglas C. Schmidt. Optimizing a CORBA Inter-ORB Protocol (IIOP) Engine for Minimal Footprint Embedded Multimedia Systems.

[6] The Object Management Group. *Ada Language Mapping Specification*. Version 1.2, 2001.

[7] The Object Management Group. *The Common Object Request Broker : Core Specification*. Version 3.0.3, 2004.

[8] Andread Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-Code Performance is Becoming Important. In *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software*, San Diego, CA, October 2000.

[9] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[10] Shivakant Mishra and Nija Shi. Improving the Performance of Distributed CORBA Applications. In *IPDPS '02 : Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 184. IEEE Computer Society, 2002.

[11] Laurent Pautet. *Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition*. Habilitation à diriger des recherches, Université Pierre et Marie Curie – Paris VI, December 2001.

[12] Thomas Vergnaud, Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. PolyORB : A Schizophrenic Middleware to Build Versatile Reliable Distributed Applications. In *Ada-Europe*, pages 106–119, 2004.

[13] Bechir Zalila. *Optimisation Déterminisme et Asynchronisme des Souches et Squelettes CORBA pour Systèmes Répartis Temps-réel*. Mémoire bibliographique du stage de Master 2 SAR, April 2005.

Annexe A : Description de l'arbre IDL

1 Description générale

La partie frontale de IAC génère un arbre syntaxique abstrait qui est une représentation plus élaborée de la séquence linéaire de *Tokens* que sont les sources IDL à compiler.

Durant le développement de IAC la structure de cet arbre subit des évolutions, ne serait ce que pour mettre à jour le compilateur vers une version plus récente de la spécification CORBA. Il est donc nécessaire que cette mise à jour de la structure de l'arbre s'effectue avec le moins de peine possible pour le développeur. Pour ce faire, seuls les différents types de nœuds sont donnés par le développeurs. Ensuite un programme nommé *mknodes* génère automatiquement les différents sous-programmes qui contrôlent l'AST et qui permettent de l'enregistrer en mémoire. Cette structure est décrite dans un fichier pseudo-IDL où chaque interface représente une entité IDL (interface, module, opération...) et les champs déclarés dans chaque interface correspondent aux entités qu'on peut trouver dans cette entité et aux propriétés de cette entité. Généralement, chaque interface dans le fichier décrivant l'arbre correspond à une règle de la grammaire IDL [7]. Par exemple, les règles de la grammaire définissant la déclaration d'une interface sont :

```
(5) <interface_dcl> ::= <interface_header> "{" <interface_body> "}"
(7) <interface_header> ::= [ "abstract" | "local" ] "interface" <identifier>
                           [ <interface_inheritance_spec> ]
(8) <interface_body> ::= <export>*
(9) <export> ::= <type_dcl> ";"
             | <const_dcl> ";"
             | <except_dcl> ";"
             | <attr_dcl> ";"
             | <op_dcl> ";"
             | <type_id_dcl> ";"
             | <type_prefix_dcl> ";"
```

Dans la description de l'arbre IDL on trouve une description très similaire pour la déclaration d'interface :

```
interface Interface_Body : List_Id {};
interface Interface_Declaration : Scope_Definition {
  boolean     Is_Abstract_Interface;
  boolean     Is_Local_Interface;
```

```
  List_Id     Interface_Spec;
  List_Id     Interface_Body;
  List_Id     Type_Prefixes;
};
```

L'interface `Scope_Definition` qui est le parent de `Interface_Declaration` est déclarée aussi dans le même fichier et décrit le nom des entités IDL. On remarque la similarité entre les deux types de déclarations à l'exception de l'ordre car en IDL on ne peut utiliser que des entités déclarées auparavant.

2 Avantages par rapport à l'AST de IDLAC

Dans IDLAC, la construction des paquetages relatifs à l'arbre IDL est elle aussi faite de façon automatique mais un peu plus primitive que dans IAC. Dans les deux compilateurs, le type représentant un nœud de l'arbre est un enregistrement et l'arbre est une table d'enregistrement instanciée à partir du paquetage générique `GNAT.Table`. Dans le cas de IDLAC, l'ensemble des champs contenus dans l'enregistrement est l'union (au sens ensembliste du terme) des champs de tous les types de nœuds. Par exemple, le nœud correspondant à un *attribute* contient un champ booléen `Is_Readonly` et le nœud correspondant à une *interface* contient un champ booléen `Local`. Dans l'enregistrement représentant un nœud de l'arbre IDL, on verra :

```
type Node_Type is record
   ...
   Is_Readonly              : Boolean;
   ...
   Local                    : Boolean;
   ...
end record;
```

En appliquant ce procédé à tous les types de nœuds possibles, on obtient un enregistrement qui contient 77 champs ce qui est très pénalisant et très coûteux en taille mémoire.

Dans IAC, une optimisation a été faite pour diminuer considérablement le nombre de champs dans l'enregistrement représentant un nœud de l'arbre IDL. L'idée est la suivante : pour deux types de nœuds contenant un champ de même type mais de noms différents, il serait intéressant de pouvoir confondre les deux champs dans l'enregistrement qui contient désormais des tableaux représentant le nombre maximal de champs d'un type donné présents dans un type de nœud donné. On obtient un enregistrement qui est beaucoup plus compact et qui ne contient que 5 champs dont trois tableaux dont les longueurs sont calculées à la génération des paquetages pour l'arbre IDL :

```
type Boolean_Array is array (1 .. 2) of Boolean;
type Byte_Array is array (1 .. 1) of Byte;
type Int_Array is array (1 .. 10) of Int;

type Node_Entry is record
   Kind : Node_Kind;
```

```
   B    : Boolean_Array;
   O    : Byte_Array;
   L    : Int_Array;
   Loc  : Location;
end record;
```

Ceci réduit considérablement le nombre de champs présents dans l'enregistrement à 15, soit 5 fois plus petit que IDLAC. Tout ceci est bien entendu fait de façon automatique et transparente et l'utilisateur n'aura pas à gérer les index dans les tableaux : des "accesseurs" pour chaque entité d'un type de nœud sont générés. C'est l'objet de la partie qui suit.

3 Entités générées

Le programme *mknodes* Parse le fichier `frontend-nodes.idl` (pseudo-IDL) en utilisant le lexeur et le parseur de IAC (cf II.1.1.1). Le résultat est un paquetage Ada appelé `Frontend.Nodes` qui contient, pour chaque champ déclaré dans une interface du fichier IDL, deux "accesseurs" pour pouvoir lire et modifier la valeur de ce champ. C'est dans ces "accesseurs" qu'est faite la résolution des index des tableaux. Par exemple, pour le champ booléen `Is_Abstract_Interface` indiquant si une interface est abstraite, deux sous programmes sont générés :

```
function Is_Abstract_Interface (N : Node_Id) return Boolean;
procedure Set_Is_Abstract_Interface (N : Node_Id; V : Boolean);
```

De plus, pour permettre un débogage facile de IAC chaque sous programme contient un ensemble d'assertions au début pour vérifier si le champ auquel on veut accéder existe vraiment dans le type de nœud donné en paramètre du sous-programme :

```
function Is_Abstract_Interface (N : Node_Id) return Boolean is
begin
   pragma Assert (False
     or else Table (Node_Id (N)).Kind = K_Forward_Interface_Declaration
     or else Table (Node_Id (N)).Kind = K_Interface_Declaration);

   return Boolean (Table (Node_Id (N)).B (1));
end Is_Abstract_Interface;
```

Le paquetage `Frontend.Nodes` contient aussi des sous programmes qui servent à écrire chacun des nœuds si le développeur désire regarder l'arbre IDL généré.

Annexe B : Description de l'arbre Ada

1 Description générale

Avant la génération de code Ada, et pour rendre le compilateur IAC plus flexible, il y a une phase de transformation de l'arbre IDL en un arbre Ada.

La structure de l'arbre est fortement inspirée de la structure de l'AST de GNAT décrite dans le fichier `sinfo.ads` dans les sources du compilateur Ada mais seulement une partie de cette structure est utilisée dans la partie dorsale de IAC (suffisamment pour pouvoir générer les structures de données des souches et des squelettes).

Comme pour l'arbre IDL, la structure de l'arbre Ada est décrite dans un fichier pseudo-IDL où chaque interface représente une entité Ada (appel à un sous-programme, déclaration d'objet, spécification d'un paquetage...) et les champs déclarés dans chaque interface correspondent aux entités qu'on peut trouver dans cette entité et aux propriétés de cette entité. Les paquetages Ada qui permettent de construire et de manipuler l'arbre Ada sont générés automatiquement par le même programme *mknodes* qui génère les fichiers relatifs à l'arbre IDL.

Ci-dessous, un petit exemple présente comment est décrite la structure conditionnelle `if` dans le fichier `backend-be_ada-nodes.idl` :

```
interface If_Statement : Node_Id {
    Node_Id    Condition;
    List_Id    Then_Statements;
    List_Id    Elsif_Statements;
    List_Id    Else_Statements;
};
```

Ensuite, pour créer une telle structure, on commence par créer les différentes parties qui la composent, et finalement on appelle la fonction :

```
function Make_If_Statement
  (Condition        : Node_Id;
   Then_Statements  : List_Id;
   Elsif_Statements : List_Id := No_List;
   Else_Statements  : List_Id := No_List)
  return Node_Id;
```

Le nœud retourné par cette fonction est de type `K_If_Statement` et il représente la structure conditionnelle créée. On pourra l'ajouter dans l'arbre Ada à l'aide des différentes méthodes `Make_XXXX` du paquetage `Backend.BE_Ada.Nutils` ou directement en utilisant la procédure `Append_Node_To_List` du même paquetage si on sait exactement où ajouter le nœud.

2 Liaisons entre les deux arbres

Comme il a été précisé dans II.1.1.2, la construction de l'arbre Ada s'effectue en parcourant l'arbre IDL un certain nombre de fois, et il arrive très souvent qu'on utilise des nœuds déjà créés. Cette utilisation est rendue possible grâce à des liaisons créées entre les deux arbres. Prenons l'exemple IDL suivant :

```
interface myInt {
   typedef MyStr string;
   MyStr echoString (in MyStr data);
};
```

Lors du parcours pour générer le paquetage `Helper` de l'interface `myInt`, IAC génère les fonction `To_Any` et `From_Any` pour le type `MyStr`. Ces deux fonctions sont utilisées dans la souche et dans le squelette. Pour pouvoir accéder à ces fonctions et à plusieurs autres entités, IAC crée une liaison entre le déclarateur `MyStr` et les deux nœuds des deux fonctions. Cette liaison est rendue possible en ajoutant un nouveau champs `BE_Node` pour le nœud `Identifier` de l'arbre IDL :

```
interface Identifier : Node_Id {
  ...
  Node_Id    BE_Node;
  // Links nodes together
};
```

Ce nouveau champ pointe vers un nœud de type `BE_Ada` de l'arbre Ada qui contient comme champs, les différents nœuds vers lesquels il peut y avoir une liaison :

```
interface BE_Ada : Node_Id {
  Node_Id    Stub_Node;
  Node_Id    Helper_Node;
  Node_Id    Skel_Node;
  Node_Id    Impl_Node;
  Node_Id    TC_Node;
  Node_Id    From_Any_Node;
  Node_Id    To_Any_Node;
  Node_Id    To_Ref_Node;
  Node_Id    U_To_Ref_Node;
  Node_Id    Type_Def_Node;
  Node_Id    Forward_Node;
  Node_Id    BE_Ada_Instanciations;
```

```
  Node_Id    From_CDR_Node;
  Node_Id    To_CDR_Node;
  Node_Id    Set_Args_Node;
};
```

Le fait de passer par un nœud intermédiaire permet de simplifier la structure de l'arbre IDL. Le sens inverse de la liaison existe :

```
interface Node_Id {
  Node_Id    Next_Node;
  Node_Id    FE_Node;
};
```

Cette façon de lier les deux arbres simplifie beaucoup la construction de l'arbre Ada et évite d'y créer des nœuds identiques. Il existe cependant des situations où on est obligé de créer des nœuds identiques dans l'arbre Ada ; par exemple, un même nœud ne peut être présent dans deux listes différentes car la structure de la liste repose sur le champ `Next_Node` et un nœud ne possède qu'un seul champ `Next_Node`. Pour cette raison, la fonction `Copy_Node` est utilisée pour créer des copies identiques de certains types de nœuds.

Annexe C : Cas particulier du module CORBA

La description du module particulier `CORBA` existe dans le chapitre 5 des spécifications du mapping Ada de CORBA [6]. Ce module contient les déclarations de plusieurs entités vitales pour le fonctionnement de l'intergiciel (création de requêtes, DII, *Repository_Id*, exceptions système...). Il contient aussi la définition de types séquences pour la plupart des types de base CORBA. Toutes ces entités sont décrites dans un fichier particulier appelé `orb.idl`.

Les spécifications du mapping donnent la plupart des spec Ada à générer. La plupart du temps, le code qui doit être généré est différent du code généré pour un fichier IDL classique. Pour ces parties, le code Ada a été écrit manuellement (et non généré automatiquement) lors du développement de la personnalité CORBA de PolyORB. Cependant il peut se trouver qu'un fichier source IDL inclue le fichier `orb.idl` pour pouvoir utiliser les entités qui y sont décrites. Il faut donc empêcher le compilateur IDL de générer du code Ada pour les parties écrites manuellement, tout en générant le bon code pour le fichier source classique. Ajoutons à cela que certaines partie du modules `CORBA` sont écrites en pseudo-IDL et certaines autres ne respectent pas la grammaire et nous nous trouvons devant un problème très difficile. Dans ce qui suit, je donnerai la description de la manière dont le module `CORBA` est analysé et des différentes anomalies de syntaxe et de sémantique présentes dans le fichier `orb.idl`.

1 Modification de l'arbre IDL

Comme il a été dit précédemment, des types séquences pour les types de bases sont déclarés dans le module `CORBA`. L'utilisation de ces types est très rare par les applications réparties. Sachant que toutes les applications réparties ont besoin du module `CORBA` pour fonctionner (plus précisément du paquetage Ada `CORBA`), et sachant que la déclaration des types séquence implique plusieurs instanciations de paquetages génériques, les empreintes mémoires des applications réparties vont augmenter inutilement. Pour cette raison il a été décidé lors du développement de la personnalité CORBA de PolyORB que les types séquence doivent être déplacés dans un module fils du module `CORBA` appelé `CORBA::IDL_Sequences`. De cette manière, les applications qui n'utilisent pas ces types ne devront pas les inclure dans leur code. Les autres entités du module `CORBA` pour lesquelles une génération automatique de code est nécessaire contiennent essentiellement des *forward declarations* d'interfaces et sont aussi déplacées dans un sous-module appelé `CORBA::Repository_Root`.

Lors de l'analyse du fichier `orb.idl`, il faut modifier la branche de l'arbre IDL relative au module

CORBA en supprimant les parties pour lesquelles aucune génération de code n'est demandée et en déplaçant le reste conformément à ce qui a été dit dans le paragraphe précédent.

2 Anomalies dans `orb.idl`

Deux anomalies existent dans le fichier `orb.idl` et rendent son analyse un peu délicate.

L'entête du fichier contient l'instruction suivante qui permet de spécifier un préfixe pour toutes les constantes de l'*interface Repository* :

```
typeprefix CORBA "omg.org"
```

Dans la grammaire IDL [7] la règle 2 est très claire :

```
(2) <definition> ::= <type_dcl> ";"
   ...
   | <type_prefix_dcl> ";"
   ...
```

Ce qui veut dire qu'il faut mettre un ";" après un *typeprefix*. D'une part IAC doit être rigoureux vis-à-vis le respect de la grammaire par les sources IDL et d'une autre part, il doit arriver à parser le fichier `orb.idl`. La solution adoptée était d'accepter les deux types de déclarations pour les *typeprefix* (et pour les *typeid* qui souffrent de la même anomalie), avec ";" et sans ";" et d'émettre un *Warning* pour la seconde.

La deuxième anomalie que j'ai rencontré était l'utilisation d'un mot-clé comme identificateur d'interface, c'est le cas de l'interface `Object`. Dans ce cas, la parade qui existait déjà dans IDLAC, était de mettre un :

```
#define Object OObject
```

pour ne pas générer une erreur de syntaxe. Ceci est sans conséquence majeure puisque aucun code ne doit être généré automatiquement pour cette interface.