REPUBLIQUE TUNISIENNE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR,
DE LA RECHERCHE SCIENTIFIQUE ET DE LA TECHNOLOGIE
UNIVERSITE DE TUNIS EL MANAR

INSTITUT SUPERIEUR INFORMATIQUE ISI
المعهد العالي للإعلامية

INSTITUT SUPERIEUR D'INFORMATIQUE

N° attribué par la bibliothèque

| E | / | 2 | 9 | 7 | / | C | R | I |

RAPPORT DE PROJET DE FIN D'ETUDES

Présenté en vue de l'obtention du
Diplôme National d'Ingénieur en Informatique
Option : Génie logiciel et systèmes d'information
Par

Amira Mensi

Analyse de flots de données

Encadré par : François Irigoin (CRI-ENSMP)
Corinne Ancourt (CRI-ENSMP)
Fabien Coelho (CRI-ENSMP)
Moncef Temani (ISI)

MINES ParisTech

Ecole Nationale Supérieure des Mines de Paris

Centre de Recherche en Informatique (CRI)

Année Universitaire 2007-2008

Remerciements

Au terme de ce stage, je tiens à adresser mes remerciements les plus sincères au professeur Robert Mahl, Directeur du Centre de Recherche en Informatique de l'école des Mines de Paris, pour m'avoir accueillie au sein de son centre.

Je remercie aussi monsieur François Irigoin Directeur-Adjoint du Centre de Recherche en Informatique de l'école des Mines de Paris, qui m'a accueillie au sein de son équipe de recherche, m'a encadré tout au long de ce stage. Il m'a consacré énormément de son temps et n'a cessé de me conseiller et de me faire profiter de son expérience et de son savoir faire.

Je souhaite exprimer ma gratitude à mon maître de stage madame Corinne Ancourt Enseignante-chercheur au Centre de Recherche en Informatique de l'Ecole des Mines de Paris, pour ses conseils judicieux, les conversations enrichissantes et particulièrement pour son soutien moral.

Je veux remercier également le professeur Moncef Temani, directeur de l'Institut Supérieur de l'Informatique pour la formation et l'encadrement dont j'ai bénéficiés au sein de son institut. Je le remercie aussi pour m'avoir encadré tout au long de ce stage. Il l'a dirigé, m'a conseillée et m'a apporté son aide pendant la rédaction de ce rapport.

Mes vifs remerciements s'adressent également à Fabien Coelho pour m'avoir initié à l'outil Subversion et Pierre Jouvelot pour avoir la partie concernant l'outil Newgen de ce rapport.

Je veux remercier toutes les personnes du CRI, qui ont contribué de près ou de loin au bon déroulement de ce travail par leurs encouragements, en particulier Claire Medrala pour sa disponibilité et Jacqueline Altimira pour son aide amicale.

Dédicaces

Je dédie ce mémoire à ceux que j'ai de plus cher, qui ont partagé mes joies et effacé mes peines, à mes très chers parents.

Je dédie ce mémoire à ceux qui ont toujours été à mes côtés ma sœur Emna et mon frére Mehdi.

Je dédie ce mémoire à ceux qui malgré la distance m'ont toujours accordé leur tendresse et soutien ; mes chers amis : Safa Ben Braik, Salima Landoulsi, Mohamed Aziz Nabli, Mohamed Mehdi Siddommou et oussama Ben Salem.

Je dédie ce mémoire à ceux qui m'ont accueillie parmi eux, m'ont offert leurs amitié et avec qui j'ai partagé des moments des plus agréables; mes amis : Sara Foccacia, Rima Ghazal, Sana Ben Jaafer, Brice Hoffmann, Gilles Pelfrene, Mario-Luis Rodriguez Chavez, Xavier Courtial, Salim Bensmina et particulièrement Cedric Taillandier.

Sommaire

Table des figures

Table des tableaux

Introduction

Introduction

I.1 Contexte

les grands problèmes peuvent être subdivisés en plusieurs sous petits problèmes qui peuvent être traités en parallèle. Ce traitement simultané des instructions est possible grâce à la programmation parallèle.

Il existe différentes formes de parallélisation de programmes comme la parallélisation au niveau des instructions, parallélisation au niveau des tâches et la parallélisation au niveau des données et plus précisément au niveau des nids de boucles qui nous concerne dans le cadre de ce stage.

La parallélisation au niveau des nids de boucles du programme agit sur la distribution des données à travers les différents nœuds qui vont être exécutées en parallèle. Elle s'adresse la plupart du temps aux programmes scientifiques de calcul numérique comme les applications de traitement de signal ou d'imagerie, où il devient intéressant de paralléliser les instructions internes des nids de boucles imbriquées. Cette parallélisation de programmes nécessite une phase d'analyse des différentes dépendances qui existent entre les instructions d'une même boucle et entre les instructions des boucles imbriquées.

La parallélisation de programmes n'est pas un domaine innovant mais l'intérêt qu'on lui porte a augmenté ces dernières années en raison des contraintes matérielles qui empêchent la mise à l'échelle de la fréquence des processeurs. En fait la programmation parallèle est devenue le paradigme dominant de la programmation des architectures, principalement pour les processeurs multi-cœurs. Et plusieurs logiciels se réclament performants dans le domaine de la parallélisation de programmes comme PIPS(parallélisation interprocédurale de programmes scientifiques) dont nous allons présenter les différentes analyses et transformations et SUIF (Stanford University Intermediate Format) que nous allons présenter dans le chapitre qui traite de l'état de l'art.

Motivation

Pendant des années, la parallélisation des programmes ne suscitait l'intérêt que de quelques universités sans pour autant susciter celui des industriels[1]. Ceci était dû au fait que les ordinateurs parallèles étaient chers et pas à la portée de tous. Mais dernièrement la situation a évolué avec l'apparition des processeurs multi-cœur qu'on intègre même dans les PC, les consoles de jeux et les cartes vidéos. Malgré cette explosion au niveau du hardware, les logiciels n'utilisent toujours pas efficacement ce que le matériel leur offre et ne produisent pas toujours des bonnes performances.

Prenons conscience de cette situation six entreprises majeures de l'industrie informatique : AMD, IBM, Intel, HP, Nvidia et Sun ont décidé d'investir $6 M pour soutenir un projet triennal d'exploration de nouveaux modèles de calcul parallèle à l'université de Standford afin de rapprocher le rythme de l'évolution logicielle de celui des processeurs multi-cœurs. En fait le laboratoire en charge du projet espère produire le matériel et le logiciel qui va permettre aux développeurs d'exposer le parallélisme dans leurs codes et d'exploiter plus efficacement le parallélisme que le matériel offre.

En effet le ralentissement de l'augmentation des performances pures des processeurs et de leur montée en fréquence va pousser les fabricants de microprocesseurs à utiliser des techniques augmentant le parallélisme des machines. Ainsi la multiplication du nombre de cœurs dans les microprocesseurs et l'utilisation de technologies comme l'HyperThreading (plusieurs tâches simultanées sur un seul cœur), vont augmenter le nombre de tâches que les processeurs peuvent exécuter en parallèle et concurrentiellement. Selon les prévisions d'Intel cette tendance ne semble pas prête de s'inverser[3].

Malheureusement, les techniques de programmation parallèle, dite de type « multithreadées », restent très complexes et sont pas encore suffisamment adaptées à ces changements. Tout d'abord, les bibliothèques de programmation parallèle sont assez nombreuses et ne sont pas assez standardisées. En outre, ces nouveaux paradigmes rendent la programmation plus complexe et introduisent de nouvelles sources d'erreurs possibles.

Afin de travailler sur ce sujet, Microsoft Research et Intel R&D ont conclu un partenariat de recherche avec l'université de Berkeley (Université de Californie) et l'université de l'Illinois à Urbana-Champaign (UIUC) afin de créer deux centres nommés « Universal Parallel Computing Research Centers » (UPCRC). Microsoft et Intel investiront 20 millions de dollars dans le projet sur 5 ans, et les deux universités financeront 15 autres millions dans le même laps de temps. Ces investissements de la part des industriels vont faire évoluer la

programmation parallèle, mais le vrai défi qui reste est d'appliquer ces acquis sur des programmes du monde réel.

Présentation de l'école

L'École des Mines appelée aujourd'hui Mines Paris Tech a été fondée en 1783, à l'époque où l'exploitation des mines était l'industrie de haute technologie par excellence et concentrait les problèmes de sécurité des personnels et de planification économique, voire les enjeux géopolitiques (l'accès aux matières premières rares ou stratégiques). Tout naturellement, les compétences de l'École ont suivi le développement de l'industrie et l'École des Mines étudie, développe et enseigne aujourd'hui l'ensemble des techniques utiles aux ingénieurs, y compris les sciences économiques et sociales.

Installée depuis 1816 au cœur du Quartier Latin, dans l'ancien Hôtel de Vendôme Mines Paris Tech s'est étendue en 1967 à Fontainebleau où sont installés plusieurs centres de recherche qui réalisent des recherches sous contrat. Le stage se déroule au sein du centre de recherche en l'informatique CRI dont les travaux s'articulent autour de deux axes :

Le premier concerne les analyses statiques et transformations de programme à des fins de développement rapide, d'optimisation, de maintenance et de réingénierie de codes scientifiques (équipe ATIP). Le stage s'inscrit dans cet axe.

Le second axe de travail concerne les applications d'Internet, du multimédia, des architectures documentaires et des outils collaboratifs (équipe ADM).

Par ailleurs, le CRI participe aussi activement à l'enseignement de tronc commun et aux cours de l'option informatique de l'école dont il assure l'organisation et l'encadrement.

Organisation du rapport

L'organisation du rapport se présente comme suit :

Une première partie traite les différentes notions théoriques relatives au stage comme l'analyse de flots de données, les use-def chains, l'analyse interprocédurale et la parallélisation de boucles. Toujours dans la partie théorique, nous allons consacrer un chapitre pour l'état de l'art qui présente les différents logiciels de parallélisation actuellement sur le marché. Il sera suivi d'un chapitre consacré à l'introduction du logiciel PIPS du point de vue architecture et analyses effectuées sur les programmes comme le calcul des effets, des transformeurs et des préconditions.

Dans le chapitre qui suit nous présentrons l'environnement de développement qui inclut l'outil Newgen, Subversion, Emacs et gdb.

La deuxième partie traite de la mise en œuvre pratique relative au stage comme la validation et la mise au point de l'analyseur syntaxique et du prettyprinter. La mise au point du prettyprinter a nécessité l'extension de plusieurs fonctions qui seront détaillées sous la section « impression du code Fortran en C ». L'énumération des différentes fonctions sera suivie par la partie de création de nouveaux tests de non régression. Pour conclure la partie validation, nous résumerons les différents problèmes rencontrés. Par la suite nous présenterons le calcul des effets propres à la boucle « for ».

La dernière partie du rapport sera consacrée au projet Tera@ps : une présentation du projet, les premières analyses effectuées sur le code du projet au sein de PIPS, la découverte et l'isolation des bugs.

Dans le dernier chapitre « conclusion : contribution et perspectives », nous énoncerons les objectifs futurs de notre travail qui se basent sur les acquis et ce qui a été accompli lors de ce stage.

Chapitre II : Présentation analyse de flots de données et but de stage

Présentation analyse de flots de données et but de stage

Le but de l'analyse de flots de données est de comprendre comment les données sont crées et utilisées dans un programme pour pouvoir optimiser ce dernier et d'aboutir à un autre programme équivalent de point de vue sémantique.

Le stage comprend deux parties :

- Une première partie qui participe à l'extension du projet PIPS au langage C;
- Une deuxième partie qui utilise PIPS pour analyser le projet Tera@ps.

Le but est d'étendre le projet PIPS qui permet d'analyser, de transformer, d'optimiser et de paralléliser des programmes scientifiques écrits en Fortran aux applications écrites en C.

Avant la phase d'extension de PIPS nous avons procédé à des différentes phases de prétraitement dont :

- une phase de nettoyage du code;
- une phase d'extension sur les modes de passage des paramètres;
- une phase d'extension des effets à la boucle « for ».

tout en maintenant la validation des tests de non régression à jour.

Dans le cadre de la deuxième partie du stage nous avons utilisé PIPS pour analyser le code C du projet tera@ps afin d'étudier l'adaptabilité du code sur les machines parallèles. La mise en place des expériences pour le projet tera@ps se conclura par l'exploitation des résultats qui permettront la transformation et la parallélisation des programmes ainsi que la détection des bugs et donc la mise en place de nouveaux tests de non régression au niveau de PIPS .

Analyse de flots de données

Pour pouvoir optimiser du code et le paralléliser nous avons besoin de collecter des informations sur le programme tout entier. Puis de distribuer ces informations à chaque bloc du graphe de flots de données qui est une représentation sous forme de graphe de tous les chemins qui peuvent être traversés au cours de l'exécution d'un programme. Chaque nœud du graphe est une représentation d'un bloc basique du code tandis que les arcs expriment des flots de contrôle. Le graphe de flots de données sert de représentation abstraite du programme dont dépend étroitement l'analyse de flots de données.

En premier lieu nous allons introduire des classifications d'analyse de données et en deuxième lieu nous allons nous intéresser aux analyses effectuées au niveau du projet PIPS, qui utilisent d'autres représentations abstraites.

Afin de paralléliser automatiquement des programmes nous avons besoin d'étudier les dépendances entre les différentes instructions, ce qui n'est pas possible sans analyse complète des corps des procédures et sans la propagation des informations récoltées d'une procédure à une autre. À partir de cette constatation on peut déjà distinguer entre les analyses qui déterminent et propagent les résultats des analyses à l'intérieur des corps des procédures et celles qui propagent les résultats des analyses entre les différentes procédures d'un programme.

On parle alors d'analyse intraprocédurale(l'information circule dans le corps d'une même procédure) et d'analyse interprocédurale(l'information est propagée d'une procédure à une autre). À partir de cette classification on peut en dériver une autre, le sens de la propagation de l'information dans le graphe de contrôle de flots, on parle alors :

- d'analyse ascendante;
- d'analyse descendante.

Au niveau de PIPS, les analyses sont effectuées sur le graphe de contrôle hiérarchisé (HSCG : Hierarchical Structured Control-flow Graph) qui est une structure de donnée propre à PIPS. Inspiré du graphe de contrôle, le HSCG part du principe que le parallélisme est utile uniquement dans les parties structurées qui sont à la fois plus facile, plus rapide à manipuler et à analyser et qui sont très courantes dans les programmes scientifiques traités par PIPS.

La plupart des algorithmes d'analyse pour les codes structurés (sans instructions de saut) utilisent une description simple récursive sur l'arbre syntaxique abstrait, qui est une représentation sous forme d'arbre de la syntaxe d'un code source, où chaque nœud de l'arbre désigne une construction qui apparaît dans le code source. Mais la présence des branchements aléatoires ne permet pas d'exploiter cette propriété et impose le calcul des points fixes par itération sur le graphe de contrôle.

Lors de la phase d'analyse de PIPS, qui détermine le HSCG d'un programme, on essaie de masquer par rapport à l'extérieur, autant que possible, les influences des branchements de sorte que si, par exemple, une étiquette est utilisée uniquement à l'intérieur et à partir du corps d'une boucle, la boucle apparaisse comme une construction structurée de l'extérieur ; les aléas de contrôle sont masqués. Le HSCG est donc une structure de données en niveaux où chaque niveau est une construction structurée ou une description basée sur le graphe d'un morceau de

code non-structuré. Les algorithmes qui manipulent les HSCGs sont définis successivement par récursivité et itération du point fixe. En particulier, il est à noter que la présence des branchements locaux au niveau des boucles ne fait pas un obstacle à la parallélisation si les dépendances entre les données ne créent pas de conflits.

Use-def chains

Avant de définir les use-def chains, il faut d'abord introduire les « reaching definitions »[1] qui considèrent qu'une définition d'une variable x est une instruction qui affecte ou peut affecter une valeur à x. Les définitions les plus communes sont celles qui affectent une valeur à x, elles sont considérées comme des définitions non ambiguës. Par oppositions aux définitions non ambiguës, on trouve les définitions ambiguës qui peuvent affecter une valeur à x, comme les appels à des procédures avec x comme paramètre (autre que le passage par valeur) ou bien une procédure qui peut avoir accès à la valeur de x parce que x est dans la portée de la procédure.

Comme autre définition ambiguë, on peut citer les pointeurs qui font référence à x, par exemple l'affectation *q=y est une définition de x si q pointe sur x.

On dit qu'une définition d atteint un point p s'il y a un chemin à partir du point qui suit immédiatement d jusqu'à p, durant lequel d n'est pas « tuée ». On tue une définition d'une variable « a » si entre deux points il y a eu une lecture ou une affectation de la variable « a ».

Par la suite on a jugé plus judicieux de conserver les informations apportées par les « reaching definitions » comme des « use-def chains » qui sont des listes pour chaque utilisation d'une variable ou bien pour toutes les définitions qui atteignent cette utilisation.

Au niveau de PIPS, les « use-def chains » sont utilisées comme une première approximation du graphe de dépendance. L'algorithme utilisé pour générer les « use-def chains » est unique parce qu'il se base sur le graphe de contrôle hiérarchique propre à PIPS et non pas sur un unique graphe de contrôle.

Cet algorithme génère des dépendances inexistantes entre les indices de boucles. Ces dépendances apparaissent comme des arcs entre les entêtes des boucles DO et les boucles implicites DO dans les instructions IO, ou entre une entête d'une boucle DO dont les bornes ne sont pas exprimées utilisant cet index variable.

Analyse interprocédurale

L'analyse interprocédurale est la collecte d'informations sur le programme tout entier au lieu d'une seule procédure[4]. Des exemples de problèmes d'analyse interprocédurale est la

détermination des variables qui ont été modifiés par un appel à une procédure ou de trouver quelle paire de paramètres peut être confondue avec une autre en entré pour une procédure donnée.

L'utilité des procédures en programmation est qu'elles englobent des détails inutiles au programmeur. Le but de l'analyse interprocédurale est de découvrir le maximum de ces détails pour pouvoir mettre en œuvre des stratégies d'optimisation avancées.

Mais avant la mise en œuvre d'une stratégie d'optimisation, il faut d'abord faire face à l'un des problèmes majeurs de l'analyse interprocédurale qui est la gestion de la compilation du programme en entier. Dans le cadre de la compilation interprocédurale, le code de chaque procédure dépend étroitement du code du programme en entier. Ceci pose problème parce que cela implique que le moindre changement du code source nécessite une recompilation du programme en entier.

Pour remédier en partie à ce problème on peut subdiviser la phase de compilation en deux sous phases distinctes : une qui dépend de l'information interprocédurale et une qui ne l'est pas. La première phase qu'on peut appeler phase d'analyse locale inclut les tâches usuelles de compilation comme l'analyseur lexical, le parseur et l'analyse sémantique. La deuxième examine la procédure pour en extraire les entrées nécessaires pour l'analyse interprocédurale. Cette subdivision ne résoud pas le problème en entier mais quand la représentation intermédiaire est sauvegardée, l'analyse locale ne sera invoquée que si cette procédure est changée.

Parallélisation de boucles

La parallélisation est une phase difficile du processus d'optimisation, surtout quand il s'agit de paralléliser du code séquentiel et de l'adapter à la machine cible et à sa hiérarchie de mémoire. Comme nous l'avons déjà mentionné, nous allons nous intéresser à la décomposition de données où plusieurs tâches exécutent le même traitement sur différents éléments des tableaux.

Pour le langage Fortran la possibilité d'exécuter différentes itérations d'une boucle Do en parallèle est vérifiée et le code est adapté dans ce but. Pour pouvoir réaliser de telles transformations, les conditions de Bernstein doivent être satisfaites ; deux itérations I1 et I2 peuvent être exécutées en parallèle si :

- Itération I_1 n'écrit pas dans une case lue par I_2 ;
- Itération I_2 n'écrit pas dans une case lue par I_1 ;
- Itération I_1 n'écrit pas dans une case dans laquelle I_2 écrit aussi.

Les exemples suivants illustrent les types de dépendances qui peuvent exister entre les différentes itérations. Nous allons commencer par illustrer la dépendance producteur-consommateur (RAW) :

```
pour  i = 1 ; on a : A[3] = …
                     …   =A[5]
pour  i = 3 ; on a : A[5] = …
                     …   =A[7]
```

Si on déroule les boucles :

```
For(i=0 ; i < 100 ; i ++)
        For (j=1; j< 10 ; j ++)
                V[ i +2] = …
                        =V[ i , j ]
```

Les itérations (1,1) et (3,2) sont en dépendance.

L'exemple suivant illustre la dépendance consommateur producteur(WAR) :

```
pour  i = 1 et j = 1 ; on a : V[3,2] = …
                                  … =V[1,1]
pour i = 3 et j = 2 ; on a : V[5,3] = …
                           …  =V[3,2]
```

Si on déroule les boucles :

```
For(i=1 ; i < n ; i ++)
                A[ i + 2] = …
                  … =A[ i + 4 ]
```

Les itérations (1) et (3) sont en dépendance.

L’étude des dépendances qui existent dans un programme permettent de déterminer s’il est possible de faire des transformations sur le programme en question. On parle de transformations sûres quand le programme transformé a la même sémantique que l’original. En d’autre termes on ne vérifie pas si le programme qu’on transforme est correct ou pas, mais plutôt si le programme transformé aboutit aux mêmes résultats auquel aboutit le programme original.

Conclusion

Nous avons présenter dans ce chapitre l’analyse de flots de données qui comporte le calcul des use-def chains. Les use-def chains seront utilisés par la suite dans lors de l’analyse interprocédurale des programmes.

Dans le prochain chapitre nous présenterons les différents projets de parallélisation de programmes dont SUIF.

État de l'art

État de l'art

Dans ce chapitre, nous exposeront les paralléliseurs existants, principalement le projet SUIF (Stanford University Intermediate Format) qui est le principal concurrent de PIPS. La conception de SUIF a pour but de développer un système extensible supportant un large nombre de sujets de recherche qui sont d'actualité incluant le domaine de la parallélisation, les langages de programmation orienté objet, l'optimisation scalaire et machine cible. SUIF tend à développer une architecture modulaire, facile à étendre, à maintenir et à réutiliser.

Il utilise la parallélisation interprocédurale qui inclut :

- l'analyse des dépendances des données qu'il s'agisse de données scalaires ou de tableaux;
- la privatisation des scalaires et des tableaux;
- l'analyse des pointeurs;
- la transformation des programmes pour le parallélisme;
- la distribution des boucles;
- la fusion des boucles;
- la réindexation des boucles.

Supporte aussi les langages orienté-objet pour permettre :

- la capture des définitions des objets;
- l'élimination des appels des fonctions virtuelles;
- la récupération automatique de mémoire.

SUIF effectue des analyses et des optimisations indépendantes des machines qui permettent :

- l'élimination du code mort;
- l'élimination partielle des redondances;
- la propagation conditionnelle des constantes.

Et des optimisations dépendantes des machines comme :

- l'ordonnancement des instructions;
- l'allocation des registres.

Parmi les autres paralléliseurs qui existent mais moins connus, on peut citer Bouclettes automatic parallelizer développé au sein du LIP : le laboratoire d'informatique de l' Ecole

Normale Supérieure de Lyon et PAF un paralléliseur automatique de Fortran basé sur un algorithme de normalisation de contrôle de flots qui facilite la transformation et la parallélisation des programmes. En effet l'algorithme normalise tous les cycles de contrôle de flots en une seule entrée, une seule sortie des boucles while et élimine tous les goto.

Conclusion

Dans ce chapitre nous avons présenté l'état de l'art des paralléliseurs, comme le projet SUIF, Bouclettes automatic parallelizer et PAF.

Dans le chapitre suivant, nous présenterons le paralléliseur PIPS : son architecture logicielle ainsi que les différentes analyses qu'il effectue.

Chapitre IV : Présentation PIPS

Présentation PIPS

Le but du projet PIPS est de développer un workbench libre ouvert et extensible pour analyser et transformer automatiquement les applications scientifiques et de traitement de signal. PIPS permet la compilation des programmes entiers, le reverse- engineering, la vérification des programmes, l'optimisation et la parallélisation source à source des programmes. Ses analyses interprocédurales permettent la compréhension des programmes et la vérification de la correction des transformations automatiques. Des transformations sont effectuées dans le but de réduire le coût et le temps d'exécution[5].

Les techniques développées pour PIPS peuvent être réutilisées pour les applications de traitement de signal écrits en C parce que les pointeurs, les structures de données et les allocations dynamiques ne sont pas beaucoup utilisés, elles peuvent être aussi appliquées pour l'optimisation du code Java.

PIPS effectue des analyses statiques qui produisent les graphes d'appel, les effets mémoires, les use-def chains, les graphes de dépendance, la vérification interprocédurale, les transformeurs, les préconditions, les conditions de continuation, l'estimation de complexité, la détection des réductions, les régions (read, write, in et out, may ou exact). Les résultats des analyses peuvent être affichés avec le code source, avec le graphe d'appel ou avec un graphe de flots de contrôle. Plus encore, les graphes de contrôle peuvent être affichés sous forme de texte ou de graphe.

Plusieurs algorithmes de parallélisation sont disponibles, y compris Allen&Kennedy et la distribution automatique de code et un compilateur prototype HPF, hpfc.

Quant aux transformations de programmes, elles incluent la distribution des boucles, la privatisation des scalaires et des tableaux, le déroulement des nids de boucles partiel et complet, la permutation de boucles, l'évaluation partielle, l'élimination du code mort, l'élimination des use-def et la normalisation de boucles.

Architecture logicielle

PIPS offre une architecture modulaire évolutive parce que c'est un projet qui fait intervenir plusieurs utilisateurs (des chercheurs, des étudiants, des analystes …) et qui doit

assurer une analyse interprocédurale et de l'interactivité. Ces exigences ont requis une structure en phases, où chaque phase manipule une représentation intermédiaire commune des programmes, qui a été soigneusement conçue et dont la mise à jour est supervisée. Mais le problème majeur de cette architecture reste la détermination de l'ordre qu'il faut adopter pour ordonnancer les différentes phases du processus de parallélisation tout en maintenant la consistance. PIPS adopte une approche innovante pour traiter ce problème : l'ordre est dirigé par la demande dynamique et automatiquement déduit des spécifications des dépendances qui existent entre les différentes phases, par un programme appelé pipsmake. Les utilisateurs peuvent envoyer des requêtes à pipsmake et obtenir des structures de données particulières ou bien appliquer des fonctions spécifiques.

Chaque structure de données utilisée, produite ou transformée par une phase dans PIPS, comme l'arbre syntaxique abstrait, le graphe de contrôle de flots ou les use-def chains est considérée comme une ressource sous le contrôle d'un gestionnaire de base de données appelé pipsdbm. L'emplacement où les données résident (mémoire ou fichier) est géré par pipsdbm qui optimise ces transferts selon l'espace mémoire disponible. Les structures de données effacées de la mémoire par effet de bord appelé aussi side effect (une fonction est dite à effet de bord si elle modifie un état autre que sa valeur de retour) peuvent être récupérées à partir du disque ainsi que les structures de données sauvegardées par des exécutions précédentes de PIPS, si elles sont encore consistantes.

Appelée par pipsmake, chaque phase commence par demander quelques ressources via la fonction db_get fournie par pipsdbm, effectue des traitements et déclare la disponibilité de son résultat via db_put. Chaque structure de données est attachée à son unité de programme. Pour le moment les unités connues par pipsmake sont le programme en entier, un module donné (une routine Fortran77 ou une fonction), les appelés (la liste des modules invoqués par un module donné), les appelants(la liste des modules qui invoquent un module donné). Via la notion d'appelés et d'appelants, pipsmake gère l'interprocéduralité de PIPS.

La figure suivante illustre l'architecture logicielle de PIPS[7] :

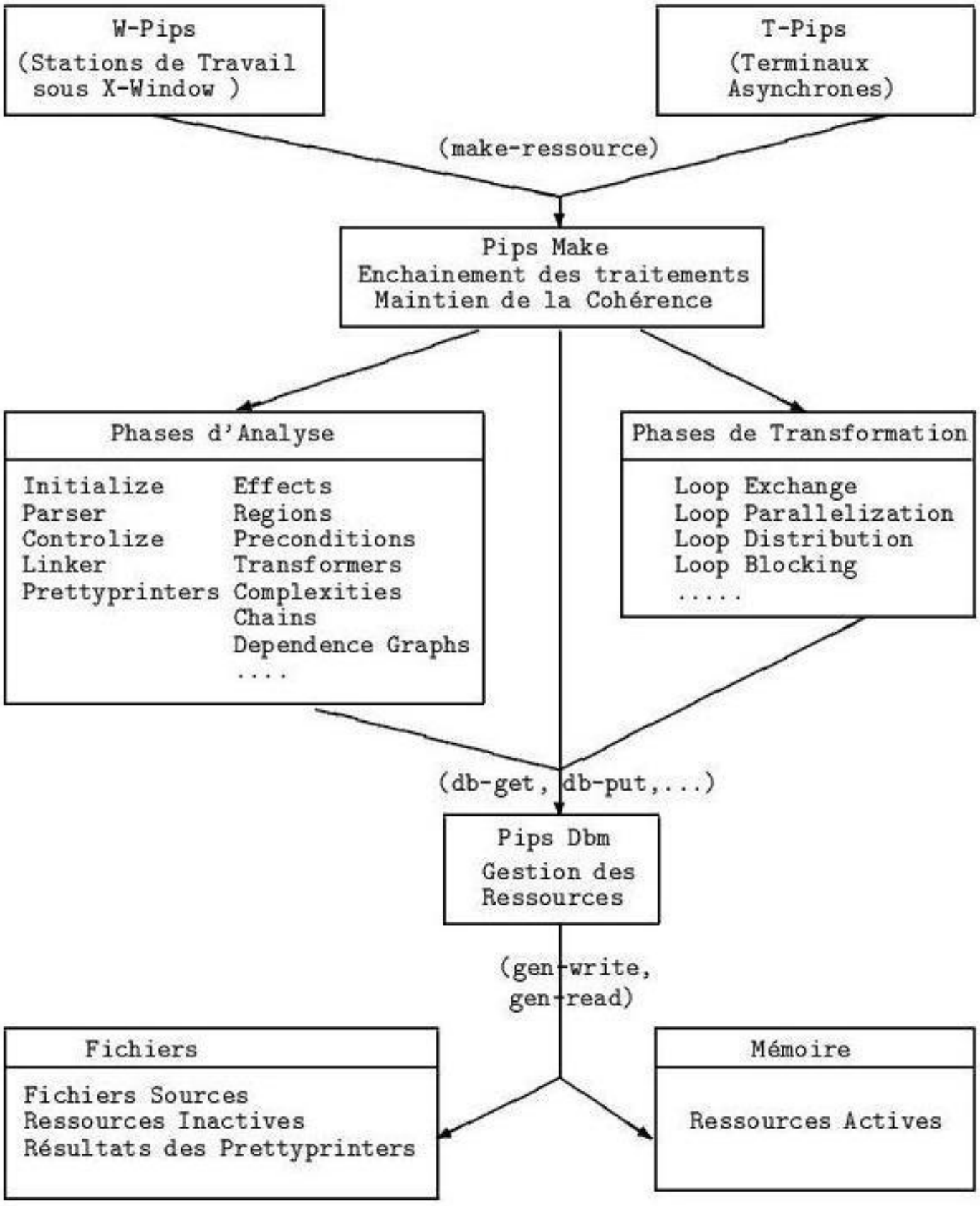


Figure 1 : Architecture logicielle de PIPS

PIPS offre plusieurs interfaces utilisateurs, mais la plus stable reste l'interface Shell. Ces interfaces permettent à l'utilisateur de choisir les programmes qu'il veut transformer ainsi que les options à appliquer. Bien sûr toutes les transformations et les analyses restent sous le contrôle de pipsmake qui tient aussi compte des dépendances indirectes qui sont dues à l'interprocéduralité. Pour cela, pipsmake a toujours recours au graphe des appels du programme en cours d'analyse.

Chaque fois que l'utilisateur spécifie les fichiers sources de son programme, PIPS calcule le graphe des appels. L'enchainement des phases dans PIPS est assuré par l'absence de récursivité dans le langage Fortran[6]. Quant à pipsdbm, comme nous pouvons le voir dans la figure ci dessous, il gère les ressources qui doivent soit résider dans la mémoire (les entités), soit résider sous forme de fichiers, soit migrer entre les deux formes : les ressources pouvant migrer sont dites actives lorsqu'elles sont en mémoire sinon elles sont dites inactives, l'utilisation d'une ressource nécessite que celle-ci soit active.

Analyse

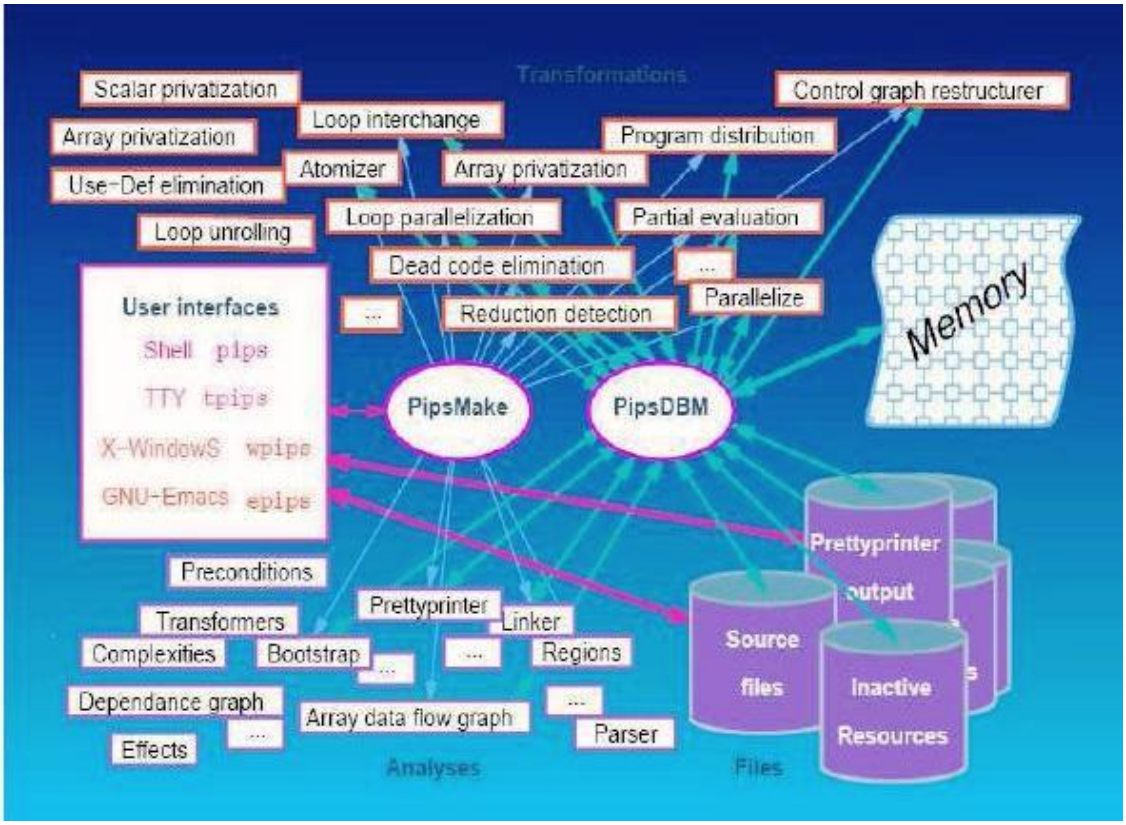


Figure 2 : Vue globale de PIPS

Chaque phase dans PIPS, comme l'analyse d'un code source d'un programme ou le processus de privatisation désigne une fonction, il peut utiliser et/ou produire des ressources. Chaque phase est déclarée via des règles de production ; qui sont stockées dans un fichier de

configuration qui est utilisé par pipsmake pour ordonnancer l'exécution de chaque fonction selon les besoins de l'utilisateur et de phase. L'exemple suivant est tiré du fichier de configuration :

```
proper_effects  > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects
cumulated_effects       > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code MODULE.proper_effects
summary_effects > MODULE.summary_effects
        < PROGRAM.entities
        <MODULE.code MODULE.cumulated_effects
```

Dans l'exemple ci –dessus, la phase `cumulated_effects` traite les effets cumulés de toutes les instructions d'un `MODULE`. Pour cela nous avons eu besoin de la définition de toutes les entités du `PROGRAM`. On rappelle ici que PIPS est interprocédurale, on tient compte du code du module et ses propres effets et les effets des appels des routines atomiques. Dans la section suivante nous détaillerons les différents effets calculés au niveau de PIPS, ainsi que les transformeurs et les préconditions.

Les effets

Au niveau de PIPS, on s'intéresse aux effets des instructions sur la mémoire. C'est dans ce but qu'on calcule les effets propres, cumulés et résumés. Tout en faisant la distinction entre les effets en lecture (READ), les effets en écriture (WRITE), les effets qui ont toujours lieu (MUST) et les effets qui peuvent avoir lieu (MAY).

Effets propres

Les effets propres d'un bloc de code sont la liste des variables lues ou écrites par le bloc. Ils sont utilisés pour construire les use-def chains et le graphe de dépendance par la suite. On désigne les effets par effets propres lorsque un bloc composé ne prend pas en compte les effets des blocs inférieurs, comme par exemple le corps d'une boucle, les branchements vrais

et faux d'un bloc de test et les nœuds de contrôle dans les blocs non structurés qui sont ignorés lors du calcul des effets propres d'une boucle, d'un test ou bloc non structuré.

Effets cumulés

Les effets cumulés sont aussi des listes de variables lues et écrites. On parle d'effets cumulés lorsque les effets d'un bloc composé, d'une boucle Do, d'un test ou d'un bloc non structuré, incluent les effets des blocs inferieurs comme le corps d'une boucle ou une branche d'un test.

Par contre les effets cumulés ne prennent pas en compte les effets locaux des variables privées, comme les variables déclarées dans les blocs C ou dans les boucles parallèles DO de Fortran. Quand il s'agit d'analyse interprocédurale, les effets mémoires sur les variables dynamiques sont ignorées au niveau des effets résumés parce qu'ils ne peuvent pas être observés par les appelants.

Les effets cumulés du module en entier sont utilisés pour calculer les effets propres des sites d'appels correspondants. Ils sont transmis de la portée de l'appelé vers la portée de l'appelant.

Effets résumés

Le résumé du flots des données est l'information interprocédurale la plus simple dont on a besoin pour être prise en compte dans un paralléliseur.

Les effets résumés d'un module sont les effets cumulés de son bloc supérieur, mais les effets sur les variables dynamiques locales sont ignorés et les expressions d'indice des autres effets sont éliminés.

Les effets propres d'un site d'appel sont calculés par traduction des effets résumés de l'appelé.

Les transformeurs

Un transformeur est une relation approximative entre les valeurs initiales d'une variable scalaire et ses valeurs après l'exécution d'un bloc d'instructions qu'il soit simple ou composé. Dans une terminologie d'interprétation abstraite, un transformeur est une commande abstraite liant l'état abstrait d'entrée d'un bloc et son état de sortie.

Les transformeurs peuvent être calculés de manière intraprocédurale indépendamment au niveau de chaque fonction, ou peuvent être calculés de manière interprocédurale en commençant par les feuilles de l'arbre des appels.

Les algorithmes intraprocéduraux utilisent cumulated_effects pour supporter les appels de procédure correctement. D'une certaine façon, ils sont interprocéduraux puisque les appels de blocs sont acceptés. En effet, les algorithmes interprocédurax utilisent `summary_transformers` pour les procédures appelées.

Pour mieux illustrer la notion de transformeurs avec ou sans propagation interprocédurale, prenons l'exemple suivant, où la procédure INC1(I) incrémente la variable K de 1 :

Si on appli

```
Subroutine INC1(I)
I=I+1
End
```

:

Par contre s

```
T(K) {}
Call INC1(K)
```

des paramètres formels en para

```
T(K) {K=K#INIT+1}
Call INC1(K)
```

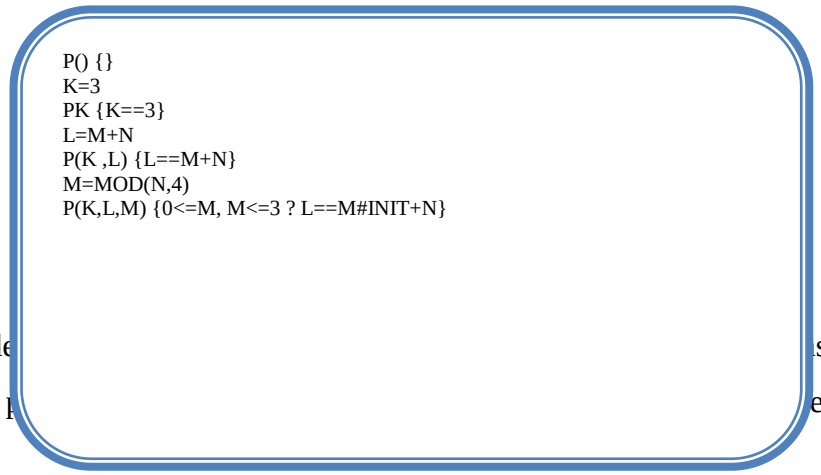
Préconditions

Pour un blo prédicat vrai pour chaque état qu'on peut atteindre à partir de l'état initial du module dans lequel le bloc d'instructions est exécuté. Au niveau de PIPS il faut activer l'option preconditions_intra pour associer une précondition à chaque bloc d'instruction, tout en supposant qu'il n'y a pas d'information disponible au niveau de point d'entré du module.

Les préconditions interprocédurales peuvent être calculées avec les transformeurs intraprocéduraux. Les préconditions intraprocédurales peuvent être calculées avec les transformeurs interprocéduraux. Ceci présente l'avantage d'être plus rapide qu'une analyse

interprocédurale complète parce qu'il n'y pas besoin d'une propagation descendante du résumé des préconditions.

L'exemple suivant illustre le calcul des préconditions :



À partir de sformeurs et des préconditions es au niveau de PIPS dont :

- le déroulage de boucle;
- l'échange de l'ordre des boucles;
- la distribution des boucles;
- l'évaluation partielle;
- l'élimination du code mort;
- la détection des réductions.

L'application de ces transformations dépend du choix de l'utilisateur qui choisit grâce à l'interactivité de PIPS, les options qu'il veut activer ou désactiver.

Conclusion

Dans ce chapitre nous avons présenté les différentes analyses des programmes réalisées par PIPS. Ces analyses comporte le calcul des effets mémoires, les préconditions ainsi que les transformeurs. Par la suite nous avons présenté les transformations des programmes comme le déroulage de boucle et l'élimination du code mort.

Dans le chapitre suivant nous présenterons l'environnement de travail qui comporte les outils Newgen, SVN, Emacs et gdb.

Chapitre V : Présentation de l'environnement de développement

Présentation de l'environnement de développement

Dans le cadre du stage, nous avons eu l'occasion de nous initier à plusieurs outils et logiciels, dont Newgen que PIPS utilise pour la représentation interne des structures de données. Subversion comme système de gestion de versions, Emacs comme éditeur de texte et gdb comme debugger. Tous ces outils et logiciels seront présentés dans les sections suivantes.

Newgen

Définition :

À partir d'une spécification haut niveau des types de données utilisateur, Newgen permet la génération de fonctions de création, d'accès et de modification de ces type de données, perçus comme des types abstraits. En particulier, les fonctions pour lire et écrire les valeurs définis au niveau de Newgen dans des fichiers sont supportées, fournissant ainsi un inter langage compatible via les fichiers ou les flux. À titre d'exemple, les structures de données crées par un programme C qui utilisent Newgen-C et qui écrivent dans un fichier peuvent être lues par un programme CommonLISP qui utilise Newgen-Lisp.

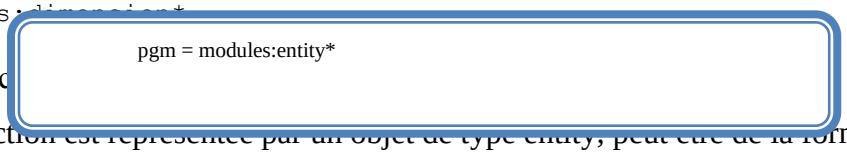
Newgen permet la définition des domaines basiques, qui sont le cœur à partir duquel des types plus élaborés peuvent être définis. Par exemple Newgen fournit les domaines basiques int et float.

À partir de ces domaines prédéfinis, Newgen permet la définition des domaines construits ; les domaines produits permettent la manipulation des n-uplets, les domaines listes de valeurs et les domaines tableaux multidimensionnels accélèrent l'accès direct à un ensemble de valeurs. Pour permettre un usage plus compatible de Newgen, les domaines externes qui permettent l'introduction de valeurs à partir de domaines préexistants dans structures de

```
array  = basic x dimensions:dimension* ;
```

Les exe domaines ;

Dans la définition du type array; nous avons une variable Fortran qui est représentée par un objet de type `array`, qui se compose d'un type de base `basic` et d'une liste de dimensions `dimensions:dimension*`.



La desc de fonctions, où chaque fonction est représentée par un objet de type entity, peut être de la forme suivante :

À partir de telles définitions, Newgen génère pour chaque langage supporté, un ensemble de fonctions qui :

- Créent, qu'elles soient initialisées ou pas, des valeurs de donnés;
- Accèdent aux parties pertinentes des valeurs construites;
- Modifient certaines parties des valeurs construites;
- Ecrivent et lisent des valeurs à partir et dans des fichiers;
- Libèrent récursivement les valeurs des données.

Indépendamment du langage de développement Newgen l'enrichit avec les fonctions citées ci dessus, cet enrichissement a plusieurs conséquences. Tout d'abord, tous les membres d'une équipe de développement utilisent les mêmes fonctions et les mêmes types de données, ce qui a pour effet d'uniformiser la façon de programmer. Ensuite, la disponibilité de ces fonctions permet à chacun de programmer plus vite, de comprendre plus facilement le code des autres, et de diminuer la quantité de commentaires nécessaires.

Newgen offre aussi la vérification dynamique via ses fonctions. Par exemple, un appel à la fonction employe_nom sur un objet «e » vérifie d'abord que « e » est un objet de type employe, si oui renvoie le nom de cet employé et si non un message d'erreur est renvoyé. Cette vérification reste néanmoins optionnelle dans le but de ne pas ralentir les programmes corrects.

La représentation interne de PIPS

Au niveau de PIPS, pour la représentation intermédiaire des programmes, le méta-langage de description des structures des données utilise le langage de définition de Newgen. Les fonctions qui manipulent ces structures de données sont disponibles et regroupées au niveau de la librairie `$PIPS_ROOT/src/Libs/ri-util`.

Les deux constructeurs + et x, les deux itérateurs [] et * et les domaines prédéfinis proposés par Newgen permettent de définir simplement toutes les structures de données. La représentation intermédiaire des programmes Fortran pour PIPS est disponible en annexe.

Subversion(SVN)

Définition

Subversion est un système de gestion de version distribué sous licence Apache et BSD(la licence Apache est une licence de logiciel libre et open source, la licence BSD (Berkeley software distribution license) est une licence libre utilisée pour la distribution de logiciels). C'est aussi un système centralisé pour le partage d'information, dont le cœur est le dépôt qui est un magasin centralisé de données. Le dépôt sauvegarde les informations sous la forme d'un arbre hiérarchique de fichiers et de dossiers. Les clients se connectent au dépôt, lisent ou écrivent ces fichiers. En écrivant des données, un client met l'information à la disponibilité des autres et en lisant il la reçoit de leur part.

Sous SVN on peut gérer les changements que les dossiers et les fichiers subissent au fil du temps. Cette gestion qui prend en compte les changements passés qui ont été faits permet la récupération des vieilles versions des données et de garder toujours un historique des changements.

La disponibilité de SVN en réseaux permet à plusieurs personnes de travailler à partir de différents ordinateurs. Cette disponibilité sur le réseau améliore la collaboration entre les membres d'une équipe, et la gestion des versions permet de récupérer toujours des données cohérentes tout en offrant la possibilité d'annuler les changements erronés.

Apports de SVN :

Par rapport aux autres systèmes de gestion de version, principalement CVS, SVN supporte plusieurs autres fonctions dont :

La gestion des versions de dossiers : alors que CVS ne gère que l'historique des fichiers individuels, SVN implémente un système de fichiers virtuel qui traque tous les changements que subit l'arbre des dossiers au fil du temps. En conséquence les fichiers et les répertoires sont versionnés.

Une historique des versions fiable : Puisque CVS se limite seulement aux versions des fichiers, les opérations comme les copies et le renommage, qui sont considérées comme des changements du contenu des dossiers ne sont pas pris en compte. Avec SVN, il est possible

d'ajouter, renommer, supprimer les fichiers et les dossiers, en ayant pour chaque nouveau fichier une nouvelle historique propre à lui.

Des validations atomiques : une collection de modifications est soit validée en entier au niveau du dépôt ou pas du tout, ceci prévient des problèmes qui peuvent avoir lieu si seulement une partie des changements est sauvegardée.

Des métadonnées versionnées : chaque fichier et dossier a un ensemble de propriétés, les clés et leurs valeurs, qui lui sont associés. On peut créer et sauvegarder la paire de clé/valeur qu'on veut. Ces propriétés sont versionnées au fil du temps, tout comme le contenu du fichier.

Le choix du réseau : SVN a une notion abstraite de l'accès au dépôt, qui permet une implémentation facile d'un nouveau mécanisme de réseau. Un plugin pour le serveur Apache http est disponible, il permet à SVN de profiter des fonctionnalités qu'offre le serveur comme l'authentification et l'autorisation.

Le Support de données consistant : SVN utilise un algorithme qui permet d'identifier les différences au niveau des fichiers texte et binaire.

Maintenabilité : SVN est implémenté comme une collection de librairies partagées C, avec des API bien définies. Ceci le rend extrêmement facile à maintenir et utilisable par d'autres applications et langages.

PIPS sous SVN

SVN utilise le modèle fusion des copies modifiées(copy-modify-merge) où chaque utilisateur crée sa copie de travail personnelle, qui est une copie locale des fichiers et dossiers du dépôt. Les utilisateurs travaillent par la suite simultanément et indépendamment sur leurs copies privées. Par la suite les copies privées sont fusionnées ensemble dans une version nouvelle. Le système assiste la fusion, mais seule une intervention humaine peut le réaliser correctement.

La politique de développement au sein de PIPS stipule qu'une version trunk du projet doit assurer une qualité qui lui fait passer tous les tests de non régression qui s'exécutent la nuit au CRI. Mais les développements individuels doivent se faire indépendamment sous des branches (copie du projet en entier) qui doivent être validées avant d'être fusionnées dans le trunk.

Le schéma suivant illustre les copies de travail qu'il fallait mettre en place pour pouvoir implémenter au niveau de PIPS .

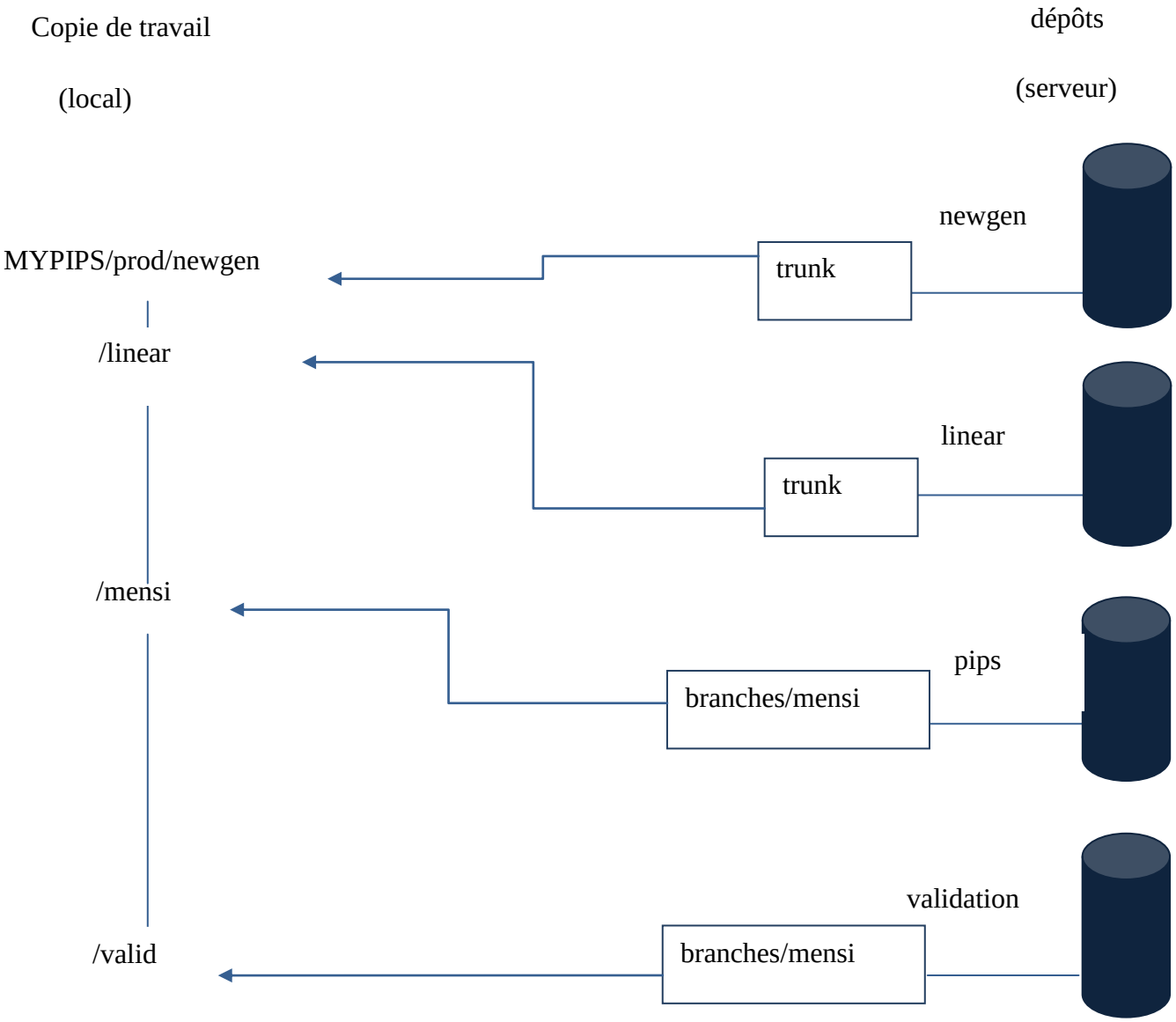


Figure 3 : Développement PIPS sous SVN

Pour pouvoir synchroniser la branche avec le trunk, on définit deux notions :

- Le pull : récupération des modifications qui ont été faites au niveau du trunk dans la branche de développement;
- Le push : injection des modifications qui ont été faites au niveau de la branche dans le trunk.

```
Svn log –stop-on-copy- http://svn.cri.ensmp.fr/svn/pips/branches/mensi
```

l'URL cible selon l'opération qu'on désire effectuer. Dans l'exemple suivant on effectue un pull pour récupérer les modifications qui ont été faites au niveau du trunk dans la branche.

```
Svn merge –revision 12820 :HEAD http://svn.cri.ensmp.fr/svn/pips/trunk
```

Décrémenter de un le numéro de révision, puis taper la commande merge :

Après avoir compiler et valider dans le cas où il n'y a pas de conflit, terminer avec un commit. La figure suivante résume la synchronisation des copies de travail de la branche et du trunk, de la branche et du trunk.

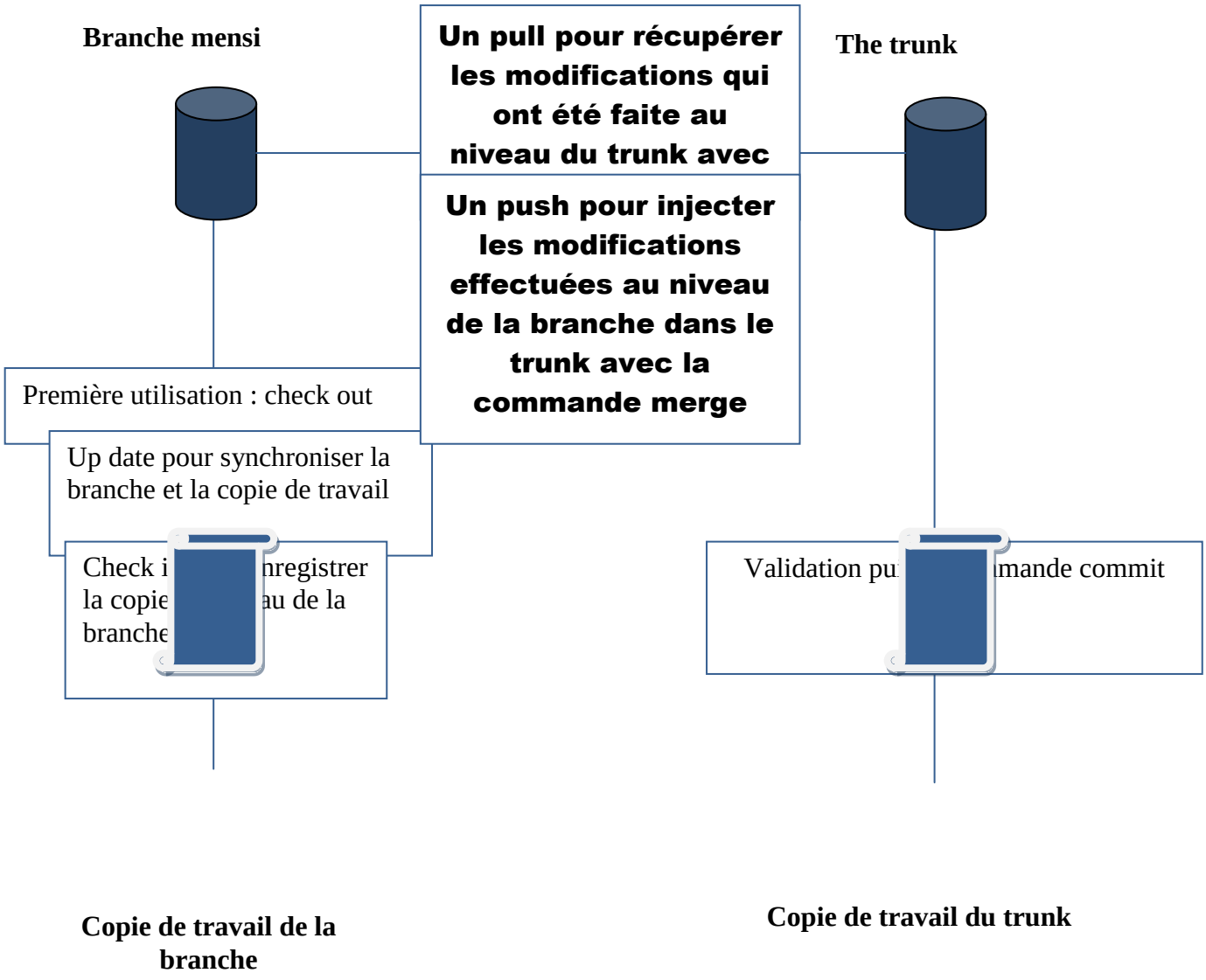


Figure 4 : La synchronisation de la branche

Pour gérer les données au quotidien, on a eu recours aux commandes suivantes pendant la phase de développement du stage :

- mise à jour de la copie de travail : svn update;
- effectuer des changements : svn add, svn delete, svn copy, svn move;
- examiner les changements : svn status, svn diff;
- annuler certains changements : svn revert;
- résoudre des conflits : svn update, svn resolved;
- valider les changements : svn commit.

Emacs

Définition

Emacs est une famille d'éditeurs de texte disposant d'un ensemble extensible de fonctionnalités, il doit sa puissance au langage d'extension, Emacs Lisp, qui permet la prise en charge de tâches évoluées, telles que l'écriture et la compilation de programmes, la navigation sur le WEB, la lecture des forums de discussion ou du courrier électronique.

Emacs inclut un grand nombre de bibliothèques Emacs Lisp, et autres, disponibles sur Internet. Beaucoup de bibliothèques proposent des facilités pour les programmeurs, reflétant la popularité d'Emacs parmi les informaticiens. Emacs peut être utilisé comme un environnement de développement intégré (EDI), permettant aux programmeurs de modifier, compiler et débuguer leur code depuis une unique interface. D'autres bibliothèques ont des fonctions moins habituelles comme :

- Calc, une calculatrice numérique performante;
- Calendar-mode, pour gérer son emploi du temps;
- Dunnet, un jeu d'aventure en mode texte;
- Ediff, pour travailler de manière interactive avec les fichiers de différences;
- Emerge, pour comparer des fichiers et les fusionner;
- Emacs/W3, un navigateur web;
- ERC (Emacs), un client IRC;
- Gnus, un client mail et news complet;
- MULE, MUltiLingual extensions to Emacs, permettant l'édition de texte écrit dans plusieurs alphabets et plusieurs langues, semblable à Unicode;

- Info, un navigateur hypertexte pour l'aide en ligne;
- Tetris.

Développement PIPS sous Emacs

La majorité des contributeurs au développement de PIPS, utilisent Emacs. Pour compiler les codes sources, il suffit de choisir le menu Tools/Compile... Pour visualiser les erreurs colorées en rouge par un simple click aller directement à l'emplacement exact dans le code source.

L'utilisation des tags permet l'indexation des fichiers sources et par conséquent une navigation rapide dans le code des fonctions et des macros.

Gdb

Définition

Au cours du développement sous PIPS, il fallait avoir recours à un débuggeur pour pouvoir examiner le code et remonter jusqu'à l'erreur et pouvoir par la suite créer des tests de non régression. C'est dans ce but qu'on a eu recours à GNU Debugger également appelé aussi gdb qui est le débogueur standard du projet GNU. Il permet de déboguer un programme en cours d'exécution, en le déroulant instruction par instruction ou en examinant et modifiant ses données.

PIPS sous gdb

Nous listons ici les différentes commandes nécessaires au débogage de programmes PIPS :

- le placement des points d'arrêt, les breakpoints : break nom_fonction;
- le démarrage d'un programme : run nom_programme;
- l'exécution de la ligne de commande suivante, en entrant à l'intérieur du code des fonctions : step;
- l'exécution de la ligne de commande suivante, sans entrer à l'intérieur du code des fonctions : next;
- la reprise de l'exécution du programme jusqu'au point d'arrêt suivant : continue;
- l'affichage de la valeur d'une variable : print nom_variable.

Conclusion

Dans ce chapitre nous avons présenté l'environnement de développement. Dans le prochain chapitre nous présenterons la mise à jour de l'analyseur syntaxique et du prettyprinter.

Validation et mise au point de l'analyseur syntaxique et du prettyprinter

Validation des tests de non régression de l'analyseur syntaxique C

Dans le cadre de la validation, nous avons modifié des fichiers sous le répertoire valid qui est une copie de travail des tests de non régression de PIPS .

Les tests de non régression de PIPS sont stockés sous le répertoire validation, dans lequel pour chaque fichier source(en fortran ou en C) on associe un fichier dont l'extension est .tpips et un sous répertoire result qui contient un fichier test dont le contenu est le résultat de l'exécution .

Pour lancer la validation, il faut lancer la commande suivante :

```
mensi@delhi:/home/mensi/MYPIPS/valid$ make clean
mensi@delhi:/home/mensi/MYPIPS/valid$ make TARGET=C_syntax validate
```

La sortie prévue suite à cette commande, ressemble à ce qui suit :

```
mensi@delhi:/home/mensi/MYPIPS/valid$ make TARGET=C_syntax validate
PIPS_MORE=cat pips_validate -v -V /home/mensi/MYPIPS/valid -O RESULTS
C_syntax
# considering 'C_syntax' directory
/home/mensi/MYPIPS/mensi/utils/pips_validate: line 129: 14467 Aborted
$tpips $full_name.tpips 2>$out.err
/home/mensi/MYPIPS/mensi/utils/pips_validate: line 129: 14477 Aborted
$tpips $full_name.tpips 2>$out.err
validation FAILED 10/137 (C_syntax )
validation for C_syntax
in directory /home/mensi/MYPIPS/valid/
 http://svn.cri.ensmp.fr/svn/validation/branches/mensi@103:117M
for LINUX_x86_64_LL architecture
output in directory RESULTS/
with pips=/home/mensi/MYPIPS/mensi/bin/pips
tpips: (pips)
ARCH=LINUX_x86_64_LL
REVS=
```

```
newgen:  http://svn.cri.ensmp.fr/svn/newgen/trunk@903
 linear:  http://svn.cri.ensmp.fr/svn/linear/trunk@1259
 pips:    http://svn.cri.ensmp.fr/svn/pips/branches/mensi@12937:12946M
 nlpmake: http://svn.cri.ensmp.fr/svn/nlpmake/trunk/makes@924
DATE=Mon Jul 21 09:28:03 UTC 2008
and tpips=/home/mensi/MYPIPS/mensi/bin/tpips
tpips: (tpips)
ARCH=LINUX_x86_64_LL
REVS=
 newgen:  http://svn.cri.ensmp.fr/svn/newgen/trunk@903
 linear:  http://svn.cri.ensmp.fr/svn/linear/trunk@1259
 pips:    http://svn.cri.ensmp.fr/svn/pips/branches/mensi@12937:12946M
 nlpmake: http://svn.cri.ensmp.fr/svn/nlpmake/trunk/makes@924
DATE=Mon Jul 21 09:28:04 UTC 2008
running in directory ./
on delhi
by mensi at Mon Jul 21 19:17:48 CEST 2008
changed: C_syntax/adi.c
failed: C_syntax/block_scope02.c
failed: C_syntax/block_scope03.c
changed: C_syntax/continue.f
changed: C_syntax/goto.c
changed: C_syntax/include.c
changed: C_syntax/main_hello_world.c
changed: C_syntax/multideclsinfile.c
changed: C_syntax/one_liner_06.c
changed: C_syntax/stop_pause.f
10 failed out of 137 on Mon Jul 21 19:17:57 CEST 2008
```

Suite à une telle sortie, nous devons récupérer pour chaque fichier la différence entre l'exécution en cours et celle déjà mise en place dans le fichier test avec la commande suivante :

```
mensi@delhi:/home/mensi/MYPIPS/valid$ more RESULTS/C_syntax_adi.diff
```

dont la sortie est comme suit:

```
--- /home/mensi/MYPIPS/valid/C_syntax/adi.result/test   2008-07-15
11:47:18.000000000 +0200
+++ RESULTS/C_syntax_adi.out    2008-07-21 19:17:49.000000000 +0200
@@ -42,9 +42,9 @@
void free_dvector();
  void nrerror();
  void tridag();
-
  if (jmax>50)
    nrerror("in ADI, increase JJ");
+
  if (k>6-1)
    nrerror("in ADI, increase KK");
  psi = dmatrix(1, 50, 1, 50);
@@ -148,6 +148,7 @@
  double *dvector();
  void nrerror();
  void free_dvector();
+
  gam = dvector(1, n);
  if (b[1]==0.0)
    nrerror("error 1 in TRIDAG");
```

Le signe « + » désigne les instructions en plus qui apparaissent et le signe « - » qui ont disparues par rapport à l'exécution originale.

Pour avoir plus de détails sur l'exécution originale que nous avons récupérés au niveau du fichier nom_fichier.result/test et qui sert de référence, on peut toujours avoir recours à la commande suivante qui affiche le fichier test :

```
mensi@delhi:/home/mensi/MYPIPS/valid$ more C_syntax/adi.result/test
```

Impression du code Fortran en code C

PIPS permet la visualisation des analyses et des transformations sous différents formats. Les vues utilisateurs sont les plus fidèles au code source initial. Les vues séquentielles sont obtenues par impression de la représentation interne de PIPS des modules. Le code peut être aussi affiché graphiquement ou à l'aide des fonctionnalités de Emacs. Les versions parallélisées restent disponibles, et à ce niveau de programmes les graphes d'appels et de contrôle de flots interprocédural fournissent des résumés intéressants.

Lors de la phase de la validation et afin d'étendre PIPS au langage C, nous avons été amenés à effectuer des changements pour supporter l'impression du code C. Ces changements ont été effectués au niveau du fichier prettyprint.c qui implémente les fonctions d'affichage.

Extension de la fonction words_nullary_op

Lors de sa première implémentation, la fonction words_nullary_op ne prenait en compte que l'affichage du code Fortran des instructions RETURN,PAUSE et STOP, concernant le langage C ; seule l'instruction RETURN était traitée.

Par souci de modularité, la fonction words_nullary_op a été divisé en deux fonctions :

- words_nullary_op_fortran pour traiter le langage Fortran;
- words_nullary_op_c pour traiter le langage C.

Tout en préservant la signature de la fonction d'origine qui selon le langage fait appel à la fonction adéquate.

```
static list
words_nullary_op(call obj,
                 int precedence,
                 bool __attribute__ ((unused)) leftmost)
{
 return is_fortran? words_nullary_op_fortran(obj, precedence, leftmost) : words_nullary_op_c(obj, precedence,
leftmost);
}
```

Pour être conforme à la nome ISO, nous avons pris en considération le nombre d'arguments que les instructions RETURN, PAUSE et STOP écrites en Fortran peuvent avoir.

L'impression du code Fortran en C a nécessité la mise en place de la fonction words_nullary_op_c. La traduction des instructions RETURN, PAUSE et STOP du Fortran en C n'étant toujours pas possible directement ; il a fallu mettre en place des fonctions intrinsèques . En l'occurrence la fonction _f77_intrinsics_pause_() qui traite l'instruction pause, inexistante au niveau du langage C. Aussi en cas de nombre d'arguments supérieur à zéro, il a fallu gérer les parenthèses par la mise en place de flags qui indiquent s'il s'agit d'expression basique ou non et si des parenthèses s'imposent ou non.

Extension de la fonction words_infix_binary_op

Pour gérer l'impression en code C des opérateurs infixes LT,GT,LE,GE,EQ et l'opérateur préfixe NOT du langage Fortran nous avons eu à modifier les fonctions suivants :

Pour les opérateurs infixes on a modifié la fonction words_infix_binary_op :

```
static list
words_infix_binary_op(call obj, int precedence, bool leftmost)
{ … }
```

Lors de l'implémentation de cette fonction, nous avons fait appel aux macros qui définissent les opérateurs infixés qui appartiennent au langage Fortran ou au langage C. Ceci permet une meilleure évolutivité et maintenance du code et permet aussi d'éviter d'éventuels erreurs comme celles que nous allons mettre en évidence dans le chapitre VII.

Extension de la fonction words_prefix_unary_op

Pour gérer l'impression de l'opérateur NOT du Fortran en C, nous avons eu à modifier la fonction words_prefix_unary_op :

```
static list
words_prefix_unary_op(call obj,
                      int __attribute__ ((unused)) precedence,
                      bool __attribute__ ((unused)) leftmost)
{ … }
```

Cette fonction traite les opérateurs préfixés comme ceux d'incrémentation ou de décrémentation, ainsi que les signes « + » et « - ».

Extension de la fonction words_io_inst

Afin de gérer l'impression de l'instruction « READ *, x » de Fortran en code C il a fallu récupérer les variables lues et mettre en œuvre un appel à une fonction_f77_intrinsics_read qui prend comme arguments les variables récupérées précédemment.

```
static list
words_io_inst(call obj,
              int precedence, bool leftmost)
{ … }
```

Implémentation de la fonction C_comment_p

PIPS est un compilateur source à source qui permet particulièrement la conservation des commentaires. Afin de l'étendre au langage C et pour faciliter la manipulation et l'impression des commentaires en C, on a mis en œuvre une fonction dont on a modélisé le comportement par l'automate suivant qui vérifie s'il s'agit bien ou non d'un commentaire C :

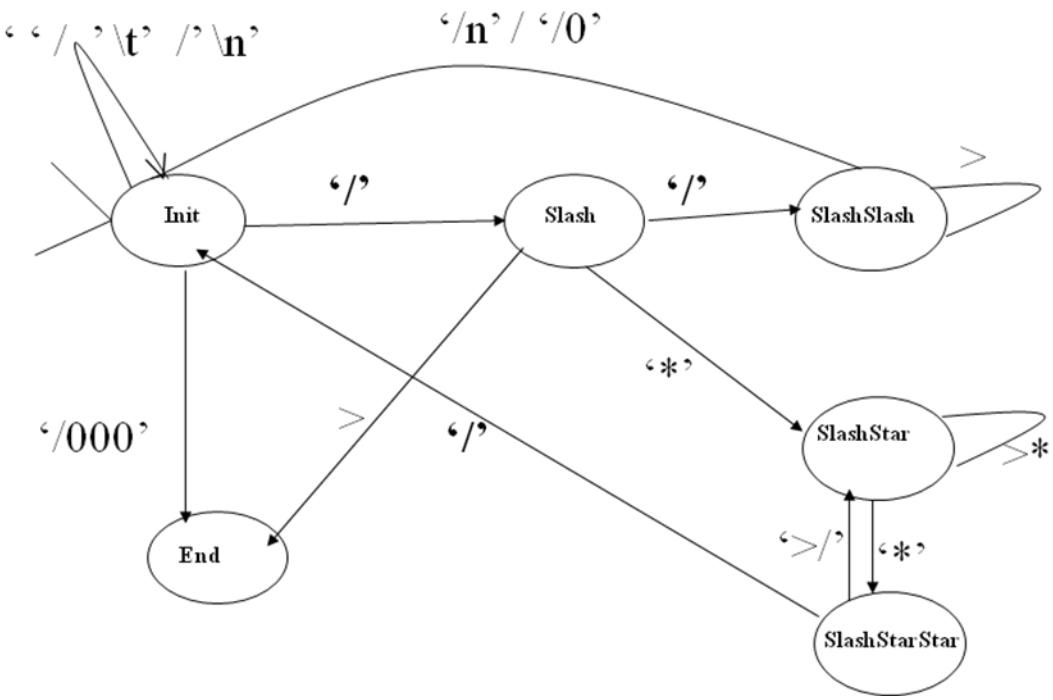


Figure 5 : Automate modélisant la fonction c_comment_p

Après avoir modéliser le comportement de la fonction, l'implémentation de la fonction a aboutit au code suivant :

```
/* In case the input code is not C code, non-standard comments have to
be detected */
bool  C_comment_p(string c){
 bool is_C_comment=TRUE;
 char * ccp=c;
 char cc=' ';
init:
 cc=*ccp++;
 if(cc==' '|| cc=='\t' || cc=='\n')
  goto init;
else if( cc=='/')
  goto slash;
else if(cc=='\000')
  goto end;
 else {
  is_C_comment=FALSE;
  goto end;
 }
slash:
 cc=*ccp++;
 if(cc=='*')
  goto slash_star;
 else if(cc=='/')
goto slash_slash;
else{
   is_C_comment=FALSE;
  goto end;
 }
slash_star:
  cc=*ccp++;
if(cc=='*')
```

```
goto slash_star_star;
 else
  goto slash_star;
slash_slash:
 cc=*ccp++;
 if(cc=='\n'|| cc=='\0')
  goto init;
 else
  goto slash_slash;
 slash_star_star:
cc=*ccp++;
 if(cc=='/')
  goto init;
 else
  goto slash_star;
 end : return is_C_comment;
}
```

Extension de la fonction text_block_elseif

Le traitement de l'impression des blocs de tests et des boucles nécessitent plus d'informations que les instructions simples ; surtout l'information sur l'unicité des instructions dans le bloc dont va dépendre la gestion de l'affichage des « { }» .

L'information sur l'unicité de l'instruction d'un bloc a nécessité la mise en œuvre d'une fonction one_liner_p() dont dépendent plusieurs autres fonctions comme le montre le graphe de dépendances décrit dans la figure 6 :

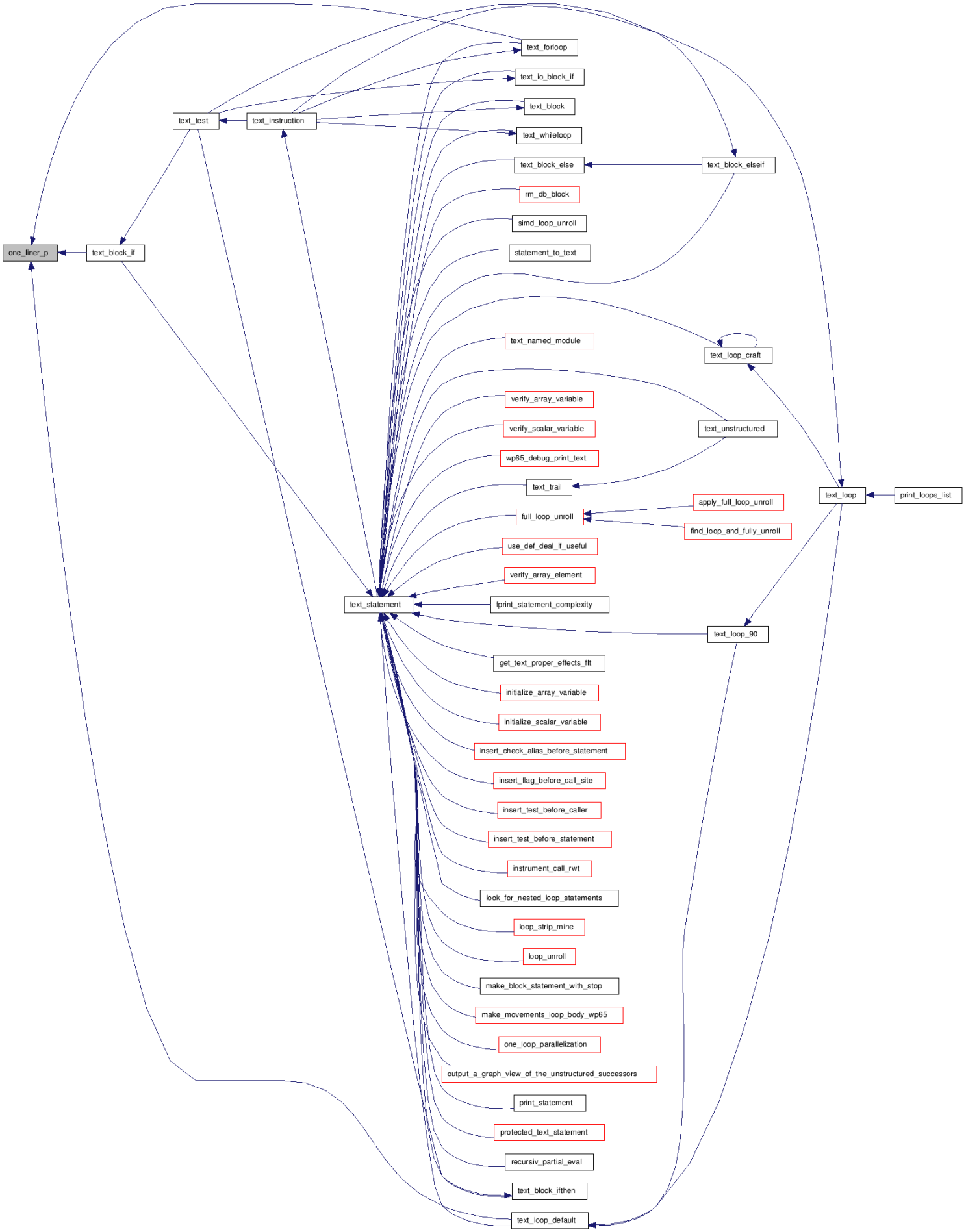


Figure 6 : Graphe d'appels pour one_liner_p()

C’est en tirant avantage des relations entre les fonctions que nous avons pu mettre en place le code de la fonction text_block_elseif() et par la suite mettre en place les tests de non régression pour les fonctions qui traitent des boucles et des blocs d’instructions.

```
static text
text_block_elseif(
   entity module,
   string label,
   int margin,
   test obj,
   int n)
{ … }
```

Extension de la fonction text_named_module

Dans le cadre de l’impression du code Fortran en C, il fallait récupérer les déclarations rattachées à un module (qui peut être une fonction ou une routine ou un programme). Pour pouvoir remonter aux déclarations, nous avons eu recours à la représentation interne de PIPS du code Fortran :

```
tabulated entity = name:string x type x storage x initial:value ;
value = code + symbolic + constant + intrinsic:unit + unknown:unit + expression ;
code = declarations:entity* x decls_text:string x initializations:sequence x externs:entity* ;
```

pour un module (entity) donné, on peut avoir accès à sa valeur initiale par la fonction entity_initial. Comme la valeur initiale que nous voulons récupérer est du type « value » qui peut être du code, il suffit d’appliquer la fonction value_code sur le résultat de la fonction entity_initial. De plus comme le code que nous voulons récupérer est en fait les déclarations du module, il suffit d’appliquer cette fois-ci la fonction code_declarations sur le résultat précédent pour récupérer les déclarations du module passé en paramètre.

```
text
text_named_module(
    entity name, /* the name of the module */
    entity module,
    statement stat)
{  if(ENDP(statement_declarations(stat))) {
                list l = code_declarations(value_code(entity_initial(module)));
}
```

Extension de la fonction text_statement et implémentation de la fonction text_statement_enclosed

Pour pouvoir gérer les blocs d'instructions, nous avons eu à changer la fonction text_statement qui dorénavant fait un appel à la fonction text_statement_enclosed() : une fonction qui lors de l'impression du code Fortran en C rajoute des « ; » lorsqu'il y a un bloc de test ou une boucle sans instructions à l'intérieur, ce qui permettra une compilation réussie du code transformé.

```
text text_statement(
    entity module,
    int margin,
    statement stmt)
{
 return text_statement_enclosed(module, margin, stmt, TRUE);
}
```

La fonction text_statement_enclosed prend en paramètre un booléen braces_p qui indique s'il s'agit ou non d'un bloc avec des instructions à l'intérieur ou non. Lors des appels à text_statement_enclosed() ; braces_p prend la valeur de retour de la fonction one_liner_p() décrite précédemment.

```
text text_statement_enclosed(
    entity module,
    int margin,
    statement stmt,
    bool braces_p)
{ … }
```

Extension de la fonction CParserError

Lors de la validation du code C, nous avons pu isoler un bug au niveau du Parser C qui détecte une double déclaration des variables lors de sa tentative de faire passer le code pour la deuxième fois.

Pour remédier à ce problème il fallait réinitialiser les piles associées au programme chaque fois qu'on exécute le code de la fonction CParserError().

```
void CParserError(char *msg)
{ … }
```

Création de tests de non régression

PIPS est un logiciel en constante évolution grâce au groupe de développeurs qui y travaillent et essayent de l'améliorer. Une façon d'améliorer PIPS est la mise en place des tests de non régression qui sont des fichiers qu'on incorpore dans le répertoire validation. Ces fichiers servent de référence d'exécution. Les tests de non régression servent aussi à la détection de bugs, dans ce cas là, idéalement nous voudrions récupérer des fichiers de quelques lignes de code, si ce n'est des fichiers d'une seule ligne de code afin de pouvoir cerner rapidement le problème.

Dans les deux paragraphes suivants, nous allons mettre en place deux tests de non régression pour vérifier le bon fonctionnement des fonctions implémentées précédemment qui traitent l'impression du C.

Création de « continue »

Dans ce paragraphe, nous allons citer les différentes étapes que nous avons suivis pour mettre en place le test de non régression « continue ».

La première étape consiste à mettre en œuvre un programme dont les instructions correspondent à ceux traités par les fonctions que nous avons implémentées, en l'occurrence les fonctions d'impression du code Fortran en code C. Les instructions dont nous voulons vérifier l'impression sont « read*,x », l'instruction « continue », l'instruction « else if » et les opérateurs préfixés. La gestion de ces instructions a été implémentée au niveau des fonctions suivantes:

- la fonction words_io_inst ;
- la fonction words_nullary_op_c ;
- la fonction text_block_elseif.

```
C     Checking C prettyprint of Fortran code: how about labelless
C     continue?
      subroutine continue_test
      logical l1, l2, l3, l4
      read *,x
      if(x.gt.0.) then
        continue
      elseif(x.lt.-1.) then
        continue
      else
c       x=x+1
      endif
      end
```

la deuxième étape consiste à la création d'un fichier .tpips qui contient les propriétés adéquates de PIPS. Dans notre cas nous avons activé les propriétés d'affichage en C.

```
delete continue
create continue continue.f
setproperty PRETTYPRINT_C_CODE TRUE
setproperty PRETTYPRINT_STATEMENT_NUMBER FALSE
display PRINTED_FILE
close
quit
```

la troisième étape consiste à la création d'un répertoire « result » qui contient le fichier de sortie de l'exécution du .tpips sous forme d'un fichier « out » et une redirection de la sortie vers un fichier « test ».

La redirection de la sortie facilite la comparaison via la commande « diff », le résultat de la redirection est comme suit :

```
void CONTINUE_TEST()
{
int L1;
int L2;
int L3;
int L4;
     float X;
  _f77_intrinsics_read_(X) ;
  if (X>0.)
    ;
  else if (X<-1.)
    ;
  else
    //c       x=x+1
  }
```

Création de « stop_pause »

Le test de non régression qui comporte le fichier « stop_pause » a été mis en place pour vérifier l'implémentation de la fonction d'impression :

- des opérateurs infixés : words_infix_binary_op;
- des instructions stop, pause et continue : words_nullary_op ;
- de la récupération des déclarations : text_named_module.

Le fichier « stop_pause » contient le programme suivant :

```
      subroutine stop_pause
     logical l1, l2, l3, l4
     x = rand()
     if(x.gt.0.) then
       x=x+5.
       stop "finished"
elseif(x.lt.-1.) then
       x=x+7.
       stop 3
     elseif(x.ne.-2.) then
       stop
     elseif(x.ge.-3.) then
       pause 10
     elseif(x.le.-4.) then
       pause
     elseif(x.eq.-5.) then
       continue
     elseif(.NOT. l1) then
       x = rand()
     elseif(l2.or.l3) then
1      continue
     elseif(l3.and.l4) then
       x=x+1.
       endif
     end
```

Quant au fichier .tpips, il contient les mêmes propriétés appliquées au fichier « continue ». Par contre le contenu du fichier de sortie est de la forme

suivante :

```
void STOP_PAUSE()
{
  int L1;
  int L2;
  int L3;
  int L4;
  float X;
  extern float RAND();
  X = RAND();
  if (X>0.) {
    X = X+5.;
    _f77_intrinsics_stop_("finished");
  }
  else if (X<-1.) {
    X = X+7.;
    exit(3);
  }
  else if (X!=-2.)
    exit(0);
  else if (X>=-3.)
    _f77_intrinsics_pause_(10);
  else if (X<=-4.)
    _f77_intrinsics_pause_(0);
  else if (X==-5.)
else if (!L1)
    X = RAND();
  else if (L2||L3)
l1:    ;
  else if (L3&&L4)
    X = X+1.;
}
```

Impression du code C

L'impression du code C a surtout porté sur la gestion d'affichage des « {} » dans les blocs de tests et de boucles. Le test se fait sur l'unicité des instructions à l'intérieur de ces blocs. Nous devons aussi ajouter des « ; » pour faire passer avec succès la compilation des fichiers qui comportent des blocs de tests sans instructions.

Création de « one_liner_01 »

Le test de non régression one_liner_01 a été mis en place pour gérer l'affichage de l'instruction « if ». Le test comporte un « if » avec une instruction unitaire donc l'affichage des « {} » n'est pas nécessaire, un « if » avec deux instructions nécessitant l'affichage des « {} » et finalement un « if » sans instruction nécessitant l'ajout du « ; ».

Comme les tests ont passé avec succès la validation, le code source du test correspond à la sortie, en conséquence nous allons juste afficher le code source des programmes. Par contre les fichiers .tpips comportent d'autres options : le parser C est activé et nous avons ajouté une commande pour compiler le fichier « test » afin de vérifier la validité des transformations que nous avons effectuées.Le code du one_liner_01 se présente comme suit :

```
void one_liner_01()
{
  int x,y,z;

  if(x>0)
    y++;
  if(x>0) {
    y++;
    z++;
  }
  if(x>0)
    ;
}
```

Le fichier .tpips correspondant est le suivant :

```
delete one_liner_01
create one_liner_01 one_liner_01.c
setproperty PRETTYPRINT_C_CODE TRUE
setproperty PRETTYPRINT_STATEMENT_NUMBER FALSE
activate C_PARSER
echo
echo PARSED PRINTED FILE FOR MODULE one_liner_01
echo
display PARSED_PRINTED_FILE[one_liner_01]

echo
echo Compile parsed file one_liner_01
echo

apply UNSPLIT[%PROGRAM]
shell gcc -c one_liner_01.database/Src/one_liner_01.c

close

delete one_liner_01
quit
```

Création de « one_liner_02 »

le test de non régression one_liner_02 a été mis en place pour gérer l'affichage de la boucle while. Il comporte également une boucle avec une seule instruction, plus qu'une instruction et enfin aucune instruction.

```
void one_liner_02()
{
  int x,y,z;
  while(x>0)
```

```
  y++;
 while(x>0) {
  y++;
  z++;
 }
 while(x>0)
  ;
}
```

Création de « one_liner_03 »

Toujours dans le cadre de la gestion de l'impression des boucles, nous avons mis cette fois-ci un test de non régression pour gérer la boucle « for » du langage C.

```
void one_liner_03()
{
  int x,y,z;
  for(;;)
    y++;
for(;;) {
    y++;
    z++;
  }
  for(;;)
    ; }
```

Création de « one_liner_04 »

le test de non régression one_liner_04 a été mis en place pour vérifier à la fois un affichage correct des blocs de « if » et « else » ainsi que le bon placement des commentaires C. Nous avons pu identifier un bug au niveau du parser qui plaçait mal les commentaires lorsqu'il était à l'intérieur d'une branche de test ne contenant pas d'instruction. Un autre

défaut d'affichage se rapportait au dernier commentaire rattaché à la dernière instruction qui se perdait totalement. Le fichier one_liner_04 contient le code suivant :

```
void one_liner_04()
{
  int x,y,z;

  if(x>0)
    y++;
  else
    y++;

  if(x>0) {
    y++;
    z++;
  }
  else {
    y++;
    z++;
}
if(x>0)
    ;
else
    //comment empty branch, misplaced by the parser
    ;

  if(x>0)
    //simple comment, misplaced by the parser
    ;
  //last instruction
  x++;
  //last comment lost by the parser
}
```

Après amélioration du code du parser, nous avons abouti au résultat suivant, mais le code reste en cours de développement et d'amélioration.

```
PARSED PRINTED FILE FOR MODULE one_liner_04

void one_liner_04()
{
   int x;
   int y;
   int z;
if (x>0)
      y++;
   else
      y++;

   if (x>0) {
      y++;
      z++;
   }
   else
    {
      y++;
      z++;
   }
if (x>0)
      ;
   //comment empty branch, misplaced by the parser
   if (x>0)
      ;
   //simple comment, misplaced by the parser
   //last instruction
   x++;
}
Compile parsed file one_liner_04
```

Création de « one_liner_05 »

Le test de non régression one_liner_05 a été mis en place à la suite de one_liner_04 pour vérifier cette fois ci le bon affichage des commentaires rattachés au bloc « else if »

```
void one_liner_05()
{
  int x,y,z;
  if(x>0)
    y++;
  else if(x>0){
    //to please the parser
    y++;
  }
  if(x>0) {
    y++;
    z++;
  }
  else if(x>0) {
    y++;
    z++;
  }
if(x>0)
     ;
  else if(x>0)
    ;
}
```

Cette fois-ci le test de régression a complètement réussi et le commentaire n'a pas été perdu par le parser, comme le montre le fichier « test » ci-dessous.

```
PARSED PRINTED FILE FOR MODULE one_liner_05

void one_liner_05()
{
  int x;
  int y;
  int z;
  if (x>0)
    y++;
  else if (x>0)
    //to please the parser
    y++;
  if (x>0) {
    y++;
    z++;
  }
  else if (x>0) {
    y++;
    z++;
  }

        if (x>0)

            ;

          else if (x>0)

            ;

        }



        Compile parsed file one_liner_05
```

Création de « comment03»

Dans le processus d'amélioration de PIPS, lorsqu'il y a un bug qui se répète on met en place un nouveau test de non régression. Etant donné que le problème de l'affichage des commentaires persistait, il a fallu mettre en œuvre le fichier comment03 suivant :

```
/* Comment of a function declaration */
int filter();
/* Comment of a function body*/
int  main()
{
  /* Declaration comment */
  int i;
  int j;
  j = filter(i);  // Function call comment
  i = j;
}
  int  filter(int x)
{
  int y;
int res;
  y = x;
  /* premier commentaire  */ y=1;
  /* deuxieme commentaire y */
  y=2;
  /* troisieme commentaire y */
  y
  =
  3;
  /* dernier commentaire end y */
  return res;
}
```

Le résultat associé au code ci-dessus est le suivant :

PARSED PRINTED FILE FOR MODULE comment03!

```
  extern int filter(int );
  extern int main();
  extern int filter(int );
```

PARSED PRINTED FILE FOR MODULE main

```
int main()
{
  int i;
  int j;

  j = filter(i);
  // Function call comment
  i = j;
}
```

PARSED PRINTED FILE FOR MODULE filter

```
int filter(int x)
{
  int y;
  int res;
  y = x;
  /* premier commentaire  */
  y = 1;
  /* deuxieme commentaire y */
  y = 2;
  /* troisieme commentaire y */

  y = 3;
  /* dernier commentaire end y */
  return res;
```

La gestion des commentaires C est en cours de développement.

Création de « enum01 »

Au cours de la validation, d'autres bugs ont été détectés nécessitant la mise en œuvre de nouveaux tests de non régression, comme un bug lié à la structure de données « énumération » du langage C que nous allons présenter dans ce paragraphe et un bug de double déclaration de variable qui sera présenté dans le paragraphe suivant.

```
main()
{
  enum fleurs {rose=0x0001, margaritte=0, jasmine};
  enum legumes{carotte=rose+50, haricot};
  enum fleurs ma;
  enum legumes mon;
  ma = rose;
  mon=haricot;
}
```

Le code ci-dessus génère un bug au niveau de PIPS. La structure de données énumération du langage C ne doit pas être assignée à un membre déjà défini.

Création de « error02 »

Le test suivant a été mis en place pour vérifier que PIPS détectait si une variable était déclarée plus qu'une fois et n'acceptait pas des programmes contenant de telles instructions.

```
void error02()
{
  //double declaration of x
  int x;
  int x;
  x=x+1;
}
```

Conclusion

Dans ce chapitre, nous avons mis à jour le prettyprinter C de PIPS. Cela a nécessité l'extension de plusieurs fonctions d'impression d'instructions. La mise à jour a été validé par la mise en place de tests de non régression.

Au cours de la validation, nous avons découverts de nouveaux bugs qui ont aboutit à de nouveaux tests de non régression. Le code source des fonctions que nous avons étendues est disponible en annexe.

Dans le prochain chapitre, nous allons synthétiser les différents problèmes que nous avons rencontrés lors de la phase de validation.

Synthèse des problèmes rencontrés

PIPS se différencie des autres compilateurs source à source par sa capacité à préserver les types de données, la structure du code, le même nombre de blocs d'instructions et essentiellement les commentaires et les lignes blanches apparaissant dans le code original. Le souci de préservation des commentaires et surtout l'évolution du format des commentaires qui, avant la version C99 étaient de la forme « /* … */ », et on savait gérer l'affichage, a évolué vers les commentaires de la forme « //…. ». Ce nouveau format permet de faire des commentaires sur une seule ligne, de la même manière que le langage C++. Ce format n'est pas pris en compte par PIPS, ce qui déclenche une erreur chaque fois qu'on veut analyser un fichier source contenant ce format de commentaire.

Un autre problème a été rencontré lors de la mise à jour de la validation. Nous avons remarqué que toutes les instructions n'ont pas d'étiquettes. Pour avoir l'information sur l'existence ou non d'une étiquette associée à l'instruction, nous avons proposé la solution qui implémente une structure de données comme suit :

```
Struct stmt{
  Bool empty_label ;
  String label ;
        }etiquette;
```

avec le champ booléen empty_label qui aura la valeur TRUE si l'instruction n'a pas d'étiquette et FALSE dans le cas contraire. Etant donné que cette solution est coûteuse du point de vue espace mémoire, nous avons préféré recourir à la définition de macro EMPTY_LABEL_NAME pour désigner les labels vides.

Du point de vue conception, au niveau de PIPS, nous avons pris la décision de regrouper toutes les fonctions d'accès et d'initialisation des entités au niveau du fichier « entity.c » pour offrir à l'utilisateur le moyen de récupérer les entités sans pour autant pouvoir les manipuler directement par soucis de sécurité.

Mais lors de l'implémentation, la conception n'a pas été respectée et des erreurs ont été commises à différents niveaux :

- Au niveau de la définition des macros qui s'est faite dans le fichier « ri-util.h », alors que ces macro ne devraient pas être exportés à l'extérieur du fichier « entity.c »;
- Au niveau des macros, nous avons deux définitions, l'une qui pointe vers l'autre inutilement:

```
#define LABEL_PREFIX « @ »
#define EMPTY_LABEL_NAME LABEL_PREFIX
```

- Au niveau de la fonction empty_label_name où l'utilisateur fait appel à la macro EMPTY_LABEL_NAME alors qu'il ne devrait pas en connaître l'implémentation mais devrait à la place utiliser les fonctions offertes par le module « entity ».

Une autre erreur d'implémentation rencontrée, qui ne respecte pas les meilleures pratiques est la comparaison des chaînes de caractères avec la chaîne vide « ‘ ‘ ». En effet, pour garantir une maintenance facile du logiciel et une meilleure évolutivité, il faut définir des macros pour les constantes. Ainsi nous aurions une gestion plus facile des constantes, si on doit les changer on ne sera pas obligé de parcourir tout le code et les modifier une à une. Il suffirait de changer la macro qui lui est associée.

Conclusion

Dans ce chapitre, nous avons présenté les erreurs d'implémentation de PIPS que nous avons rencontrés lors de la validation.

Dans le prochain chapitre, nous présenterons le calcul des effets propres de la boucle « for ».

Les Effets

Dans ce chapitre, nous calculons les effets propres de la boucle « for ». Nous rappelons que les effets propres nous renseignent sur les opérations de lecture et d'écriture des scalaires. Pour chaque instruction, l'effet correspondant précise la référence actuelle (nom de la variable) et la nature de l'opération mémoire (lecture ou écriture). Quant aux appels de procédures, le calcul est assuré par la propagation interprocédurale des effets mémoire.

Calcul des effets de la boucle « for »

La boucle « pour » en langage C a le modèle suivant :

```
Pour (Index, Condition, Incrément)
        {
                Corps de la boucle
        }
```

Pour calculer les effets propres de la boucle il faut calculer ceux associés à l'index, à la condition puis à l'incrément. Afin de réaliser cela, nous avons développés une fonction proper_effects_of_forloop().

```
static void proper_effects_of_forloop(forloop l)
{
        …
}
```

Cette fonction est implémentée au niveau du fichier proper_effects_engine.c qui regroupe toutes les fonctions génériques de calcul des effets propres.

En premier lieu, nous commençons par calculer les effets propres de l'index, l'index est accessible via la fonction forloop_initialization(). En deuxième lieu, nous calculons les effets propres de la condition, la condition est accessible via la fonction forloop_condition(). En troisième lieu, nous calculons les effets propres de l'incrément que nous récupérons via la fonction forloop_increment().

L’index, la condition et l’incrément sont des expressions dans la représentation interne de PIPS, donc pour récupérer les effets propres qui leur sont associés nous appliquons la fonction generic_proper_effects_of_expression().

La fonction proper_effects_of_forloop() concaténe tous les effets et les renvoie sous forme de liste. Puis au niveau de la fonction proper_effects_of_module_statement() et selon la nature de l’argument, si c’est une boucle for, on fait appel à la fonction proper_effects_of_forloop().

Conclusion

Dans ce chapitre, nous avons présenté le calcul des effets propres relatifs à la boucle « for ». Le calcul des effets mémoires des autres instructions est en cours de développement.

Dans le chapitre suivant, nous présenterons la deuxième partie du stage qui s’articule autour du projet Tera@ps.

Expériences Tera@ps

Dans ce chapitre nous allons présenter la deuxième partie du stage, qui prend en charge l'analyse du code du projet Tera@ps. Le paragraphe suivant introduit le projet Tera@ps en énumérant ses différents objectifs. Par la suite nous allons présenter le résultat des analyses effectuées sur le code source du projet. Les analyses du code ont abouti à la découverte et l'isolation de nouveaux bugs au niveau de PIPS, qui vont conduiront à la mise en place de nouveaux tests de non régression.

Présentation du projet Tera@ps

Description

Le besoin de nombreuses applications de traitement intensif comme le traitement d'images et de signal, l'augmentation rapide des coûts d'accès au silicium, l'évolution rapide des capacités d'intégration, tous ces facteurs ont abouti à la mise en œuvre d'une plate-forme matérielle/logicielle qui repose sur :

- Une architecture de traitement programmable massivement parallèle ;
- Un environnement de développement complet pour la programmation d'applications sur cette « machine multiprocesseurs sur puce ».

La réalisation d'une telle plateforme a nécessité la collaboration de plusieurs partenaires dont : TRT est le laboratoire de recherche du Groupe THALES (société d'électronique), Thomson, EADS (domaine des satellites), MBDA (domaine de la défense), Renault, Valeo, L'INRIA (Institut national de recherche en informatique et en automatique) et le CRI. Tous ces partenaires se sont réunis pour mener à bien le projet Tera@ps.

La réalisation du projet Ter@ops permettra de :

- Réaliser des systèmes dont les densités de performance inégalées en traitement intensif temps réel permettront la mise en œuvre de nouvelles applications embarquées;

- Constituer le pôle français capable de concevoir les plates-formes de traitement numérique embarqué de demain en rivalisant avec les meilleures initiatives au niveau mondial[8] .

Objectifs

Les objectifs majeurs du projet Tera@ps sont :

- Une programmation efficace malgré une architecture complexe;
- Une solution réutilisable et souple;
- Un fonctionnement robuste et sûr;
- Une performance dense : calcul puissant avec une faible consommation.

La réalisation de ces objectifs passe par la conception d'une machine parallèle sur silicium, un modèle de programmation versatile supportant plusieurs types de parallélisme et des outils de programmation assurant une forte productivité dans son exploitation.

Première analyse par PIPS

Dans cette section nous allons mettre sous forme de tableaux les sorties PIPS associées aux fichiers source du projet Tera@ps.

Nom fichier	Sortie PIPS	Modification
stap_param/appli_functions.c	User warning in splitc_error: C syntax error near "float" at preprocessed line 3848 (user line 64)	Déplacement des déclarations
stap_param/gen_radar.c	pips error in general_build_signature: (/home/mensi/MYPIPS/mensi/src/Libs/preprocessor/splitc.y: 191) Unexpected undefined argument 3Aborted	Déplacement des déclarations
stap_param/stap_flc_param.c	pips error in general_build_signature: (/home/mensi/MYPIPS/mensi/src/Libs/preprocessor/splitc.y: 191) Unexpected undefined argument 3Aborted	Elimination des commentaires Déplacement des déclarations
stap_param/gencode.c	No declaration of function: MAT_fft_CF in module: fusion__ApplicationModel_F2_PE0	Déplacement des déclarations

Tableau 1 : Analyse du code stap_param

Nom fichier	Sortie PIPS	Modification
/ter@ps/Thales-TRT/parametre.c	Analyse réussie	
/ter@ps/Thales-TRT$ tpips stap_inlinee_c.c	Analyse réussie	

Tableau 2 : Analyse du code Thales-TRT

Nom fichier	Sortie PIPS	Modification
/aminatosa/TOSA/src/es.c	" is lost at line 141, probably because comments cannot be attached to declarations. Analyse réussie	
/aminatosa/TOSA/src/main_motion_detection.c	" is lost at line 185, probably because comments cannot be attached to declarations. Analyse réussie	

Nom fichier	Sortie PIPS	Modification
/aminatosa/TOSA/src/main_stabilisation.c	pips error in c_parse: (/home/mensi/MYPIPS/mensi/src/Libs/c_syntax/cyacc.y:1185) Scoping not implemented yet, might be the reason	
/aminatosa/TOSA/src/main_motion_detection.c	" is lost at line 185, probably because comments cannot be attached to declarations Analyse réussie	
/aminatosa/TOSA/src/operator.c	" is lost at line 585, probably because comments cannot be attached to declarations.	
/aminatosa/TOSA/src/lib/algomem.c	user warning in splitc_error: C syntax error near "__dest" at preprocessed line 50 (user line 38)	cpp -P -C algomem.c > algomem01.c
/aminatosa/TOSA/src/lib/erreur.c	user warning in splitc_error: C syntax error near "__dest" at preprocessed line 50 (user line 38)	Renommage des variables Élimination des commentaires cpp -P -C erreur.c> erreur01.c

/aminatosa/TOSA/src/lib/file_tools.c	[set_current_module_entity] (/home/mensi/MYPIPS/mensi/src/Libs/ri-util/static.c:31) assertion failed	Renommage des variables Élimination des commentaires
/aminatosa/TOSA/src/lib/fntbmp.c	user warning in CSafeSizeOfArray: Varying size for array "fntbmp!PIPS_STRUCT_14^__val" user warning in CSafeSizeOfArray: Not yet supported properly by PIPSuser warning in discard_C_comment: Comment "/* rq : setjmp renvoie toujours 0 */ " is lost at line 224, probably because comments cannot be attached to declarations. user warning in TK_CHARCON_to_intptr_t: character constant 0x4D42 not recognized user warning in discard_C_comment: Comment "	Renommage des variables

	Segmentation fault	
/aminatosa/TOSA/src/lib/generic_hash_table.c	[hash_put] key redefined: 0xa6eff0.Segmentation fault	Renommage des variables
/aminatosa/TOSA/src/lib/generic_set.c	No declaration of variable: GTL_set_cmp_pointer in module: GTL_set_ctoruser warning in CParserError: Recovery from C parser failure not (fully) implemented yet. user error in CParserError: Illegal Input	
/aminatosa/TOSA/src/lib/handlers.c	[hash_put] key redefined: 0x939820	

/aminatosa/TOSA/src/lib/i_algebre.c	[hash_put] key redefined: 0xc58650 user warning in c_parse: double definition of initial value for variable ALG_add_nb:v pips error in c_parse: (/home/mensi/MYPIPS/mensi/src/Libs/c_syntax/cyacc.y:1185) Scoping not implemented yet, might be the reason.Aborted	
/aminatosa/TOSA/src/lib/i_logic.c	[hash_put] key redefined: 0xa4f020 pips error in c_parse: (/home/mensi/MYPIPS/mensi/src/Libs/c_syntax/cyacc.y:1185) Scoping not implemented yet, might be the reason. Aborted	
/aminatosa/TOSA/src/lib/imagefmt.c	user warning in c_error: C syntax error, unexpected TK_STAR, expecting TK_SEMICOLON near "*" [gen_internal_context_multi_recurse] (/home/mensi/MYPIPS/prod/newgen/src/genC/genClib.c:3179) assertion failed	Élimination des commentaires

	null or undefined object to visit not verified. Aborted	
/aminatosa/TOSA/src/lib/indexcol.c	user warning in FindEntityFromLocalNameAndPrefix: Cannot find entity win with prefix "$" at line 1 user warning in FindEntityFromLocalName: Cannot find entity win Segmentation fault	Renommage des variables
/aminatosa/TOSA/src/lib/i_stdlib.c	[hash_put] key redefined: 0xbb9ee0 user warning in c_parse: double definition of initial value for variable STDI_best_move:thewd1 pips error in c_parse: (/home/mensi/MYPIPS/mensi/src/Libs/c_syntax/cyacc.y:1185) Scoping not implemented yet, might be the reason. Aborted	

/aminatosa/TOSA/src/lib/makimg.c	Analyse réussie	Élimination du "/" du print
/aminatosa/TOSA/src/lib/progress.c	[hash_put] key redefined: 0xa308b0 Segmentation fault	

Tableau 3 : Analyse TOSA

Nom fichier	Sortie PIPS	Modification
/Thomson/.../src/annexb01.c	[general_build_signature] Returns: "int type ; int value1 ; int value2 ; int len ; int inf ; unsigned int bitpattern ; int context ; int k ; void (* mapping) (int len , int info , int * value1 , int * value2) ; void (* reading) (struct syntaxelement * , struct img_par * , DecodingEnvironmentPtr) ; "Program received signal SIGSEGV, Segmentation fault.	cpp -P -C annexb.c>annexb01.c

/Thomson/.../src/biaridecod01.c	[gen_internal_context_multi_recurse] (/export/temp/irigoin/MYPIPS/prod/newgen/src/genC/genClib.c:3179) assertion failed null or undefined object to visit not verified Program received signal SIGABRT, Aborted.	cpp -P -C biaridecod.c>biaridecod01.c

Tableau 4 : Analyse Thomson

Conclusion

Dans ce chapitre nous avons présenté le projet Tera@ps ainsi que ses objectifs. Nous avons aussi mis en évidence les analyses que nous avons réalisées sur le code du projet .

Dans le prochain chapitre, nous presenterons les résultats des experiences effectuées sur le code du projet Tera@ps.

Résultats des expériences

Les expériences sur le code du projet Tera@ps ont servis de benchmark pour le logiciel PIPS, qui était conçu pour traiter et analyser des programmes scientifiques. Par contre le projet Tera@ps est un projet industriel, dont une partie du code a été généré automatiquement, ce qui a permis d'effectuer de nouvelles analyses sur PIPS et la détection de nouveaux bugs.

Comme le code du projet Tera@ps est généré en partie automatiquement, lors des analyses des fichiers, il fallait effectuer des traitements au préalable pour faire passer le code sous PIPS. Les majeurs modifications qui ont été effectuées sont dues au fait que PIPS supporte la norme antérieure à la norme C99[8]. Les extensions apportées par la norme C99 comme le format des commentaires « //… » et les déclarations des variables peuvent se faire dans le corps du code et non pas seulement au début du programme. Ces extensions ne sont pas encore traitées par PIPS et le parser les considère comme étant des erreurs.

Pour pouvoir analyser ces fichiers, nous avons dû les formater, en supprimant les commentaires au format incompatible et en mettant les déclarations au début du programme comme l'ancienne norme.

Dans la suite nous allons introduire quelques bugs identifiés au niveau de PIPS grâce au code Tera@ps. Face à la découverte d'un bug, nous avons le choix entre deux stratégies : soit simplement déclarer le bug et le contourner, soit le corriger, cela dépend surtout de l'impact du bug sur le fonctionnement de PIPS.

Découverte et isolation de bugs

Signature « __extension__ »

Lors du prétraitement du code Tera@ps, nous avons dû compiler les fichiers avant de les passer sous PIPS ce qui ajoutait le mot clé « __extension__ » de gcc devant quelques déclarations. Le mot ne change rien au comportement du code, mais il est utilisé dans le but de supprimer les avertissements dus aux différents standards C.

Face à ce problème deux solutions ont été présentées :

- Ignorer le mot clé au niveau de l'analyseur lexical;
- Prendre en compte le mot en rajoutant des règles de grammaire.

Comme c'est une extension gcc, la solution choisie était de le supprimer dés le début lors de l'analyse lexicale.

Support des extensions de la norme C99

Le code du projet Tera@ps respecte la norme C99, dont plusieurs extensions ne sont pas gérées par PIPS. Dans le cas de la définition de la taille tableaux avec des types dynamiques et parce que le code contient un nombre important de déclarations sous cette forme, nous avons été amené à modifier le traitement de l'allocation de l'espace des tableaux.

Pour les déclarations mal placées dans le code, nous avons été obligé de les déplacer manuellement pour ne pas déclencher un échec lors des analyses.

Par la suite, et dans l'objectif d'étendre PIPS à la norme C99, nous prendrons le choix entre être strict avec leur emplacements ou non.

Conclusion

Dans ce chapitre, nous avons présenté les résultats des expériences Tera@ps. Les expériences ont abouti détection de nouveaux bugs au niveau de PIPS dont le non support des extensions apportées par la norme C99. Après avoir mis en place de nouveaux tests de non régression et des changements au niveau de l'analyseur lexical, il reste encore le choix entre étendre PIPS à la norme C99 ou non.

Conclusion : contributions et perspectives

La parallélisation de programmes n'est pas un domaine innovant, mais qui grâce à l'évolution matérielle devient d'actualité. L'architecture multi-cœurs domine le marché et il devient nécessaire que les programmes exploitent au mieux cette architecture.

Cette exploitation couvre aussi les programmes séquentiels déjà écrits, qui peuvent être parallélisés. Mais la parallélisation des programmes s'avère être une tâche délicate parce qu'on vise à réécrire les programmes tout en conservant leurs sens.

Le paralléliseur source à source PIPS parallélise les programmes séquentiels écrits en langage Fortran et est en cours d'extension pour paralléliser aussi les programmes écrits en langage C.

Afin de paralléliser un programme, PIPS effectue plusieurs analyses et transformations sur les programmes. Ces analyses nécessitent une phase de « parsing » qui permet la traduction du code source dans la représentation interne de PIPS définie avec l'outil Newgen. Cet outil permet la définition des type de données utilisateurs, perçus comme des types abstraits.

Parmi les principales transformations, nous pouvons peut citer :

- la propagation de constante ;
- l'élimination du code mort;
- le déroulage de boucles;
- la distribution de boucles ;
- la transformation des boucles.

et parmi les analyse effectuées, nous pouvons citer :

- le calcul des effets mémoires;
- le calcul des use-def chains,
- l'analyse interprocédurale;
- les préconditions;
- les transformeurs ;
- les régions.

Une partie de notre stage a été consacrée au calcul des effets propres de la boucle « for » du langage C. On appelle effet propre l'action de lire ou d'écrire une variable au sein d'une instruction. Ils sont utilisés par la suite pour la définition des use-def chains.

Dans le but d'étendre PIPS, nous avons dû mettre à jour le prettyprinter C afin d'imprimer du code Fortran en code C.

La phase de mise à jour du prettyprinter a nécessité l'extension de plusieurs fonctions afin de traiter les structures de test comme le « if » et le « else if », les structures de boucles comme la boucle « for » et le « while », les opérateurs préfixés comme le « AND » et le « OR », l'opérateur infixé « NOT » ainsi que les commentaires.

Après la mise à jour du prettyprinter, nous avons mis en place les tests de non régression afin de vérifier la cohérence des résultats produits par les fonctions citées ci dessus et par conséquent cela permet de maintenir la validation de PIPS à jour.

Une partie du stage a été également consacrée à l'analyse du code source du projet Tera@ps. Le code source du projet Tera@ps a servi de benchmark pour PIPS, le code est généré automatiquement par des outils, contrairement aux programmes scientifiques que nous avons l'habitude d'analyser.

La partie consacrée au projet Tera@ps a permis la détection de nouveaux bugs au niveau de PIPS et donc la mise en place de nouveaux tests de non régression.

Dans nos futurs travaux, nous allons continuer à étendre le calcul des effets propres que nous avons entamé avec la boucle « for », nous allons viser de nouvelles instructions, telle que la boucle « while ».

Toujours dans la phase d'extension de PIPS au langage C, nous allons étendre la grammaire de l'analyseur syntaxique afin de l'étendre à la norme C99 et supporter nouveaux formats d'instructions.

Et en s'inspirant du projet Tera@ps, qui implique des applications multimédia temps réel, nous allons effectuer une bibliographie spécifique à la parallélisation du langage C. Par la suite nous allons modéliser automatiquement des applications écrites en C dans le format du projet Tera@ps en partant d'une implantation existante pour Fortran.

Bibligraphie

[1] AHO, SETHI, ULLMAN, COMPILERS. PRINCIPLES, TECHNIQUES, AND TOOLS, ADDISON-WESLEY, (1986)

[2]HTTP://ARSTECHNICA.COM/NEWS.ARS/POST/20080501-INDUSTRY-STANFORD-HOPE-TO-FIX-WHAT-AILS-PARALLEL-PROCESSING.HTML

[3] RANDY, ALLEN, KEN KENNEDY. OPTIMIZING COMPILERS FOR MODERN ARCHITECTURES, MORGAN KAUFMANN, (2002)

[4] HTTP://WWW.BULLETINS-ELECTRONIQUES.COM/ACTUALITES/53646.HTM

[5] BASSEM BEN MESSAOUD, BRUNO VIDALENC. EVALUATION D'UN ENVIRONNEMENT DE PARALLELISATION AUTOMATIQUE PIPS

[6] BEATRICE CREUSILLET. REGIONS EXACTES ET PRIVATISATION DE TABLEAUX(1994)

[7] HTTP://SVNBOOK.RED-BEAN.COM

[8] HTTP://TERAOPS-EMB.IEF.U-PSUD.FR

[9] HTTP://FR.WIKIPEDIA.ORG/WIKI/C_%28LANGAGE%29#COMMENTAIRE

Annexe

La représentation intermédiaire de PIPS du code Fortran

action = read:unit + write:unit ;
application = function:expression x arguments:expression* ;

approximation = may:unit + must:unit + exact:unit ;

area = size:int x layout:entity* ;

basic = int:int + float:int + logical:int + overloaded:unit + complex:int + string:value + bit:int + poin
ter:type + derived:entity + typedef:entity ;

callees = callees:string* ;

call = function:entity x arguments:expression* ;

cast = type x expression ;

cell = reference + preference ;

code = declarations:entity* x decls_text:string x initializations:sequence x externs:entity* ;

constant = int + litteral:unit + call:entity + unknown:unit ;

controlmap = persistant statement-control ;

control = statement x predecessors:control* x successors:control* ;

descriptor = convexunion:Psysteme* + convex:Psysteme + none:unit ;

dimension = lower:expression x upper:expression ;

effect = cell x action x approximation x descriptor ;

effects_classes = classes:effects* ;

effects = effects:effect* ;

entity_effects = entity->effects ;

entity_int = entity->int ;

evaluation = before:unit + after:unit ;

execution = sequential:unit + parallel:unit ;

expression = syntax x normalized ;

forloop = initialization:expression x condition:expression x increment:expression x body:statement ;

formal = function:entity x offset:int ;

functional = parameters:parameter* x result:type ;

instruction = sequence + test + loop + whileloop + goto:statement + call + unstructured + multitest
+ forloop + return:expression + expression ;

loop = index:entity x range x body:statement x label:entity x execution x locals:entity* ;

mode = value:unit + reference:unit ;

multitest = controller:expression x body:statement ;

normalized = linear:Pvecteur + complex:unit ;

parameter = type x mode ;

persistant_expression_to_effects = persistant expression -> effects ;

persistant_statement_to_control = persistant statement -> persistant control ;

persistant_statement_to_int = persistant statement -> int ;

persistant_statement_to_statement = persistant statement -> persistant statement ;

predicate = system:Psysteme ;

preference = persistant reference ;

qualifier = const:unit + restrict:unit + volatile:unit + register:unit + auto:unit ;

ram = function:entity x section:entity x offset:int x shared:entity* ;

range = lower:expression x upper:expression x increment:expression ;

reference = variable:entity x indices:expression* ;

sequence = statements:statement* ;

sizeofexpression = type + expression ;

statement_effects = persistent statement->effects ;

statement = label:entity x number:int x ordering:int x comments:string x instruction x declarations:e
ntity* x decls_text:string ;

storage = return:entity + ram + formal + rom:unit ;

subscript = array:expression x indices:expression* ;

symbolic = expression x constant ;

syntax = reference + range + call + cast + sizeofexpression + subscript + application ;

tabulated entity = name:string x type x storage x initial:value ;

test = condition:expression x true:statement x false:statement ;

transformer = arguments:entity* x relation:predicate ;

type = statement:unit + area + variable + functional + varargs:type + unknown:unit + void:unit + str
uct:entity* + union:entity* + enum:entity* ;

unstructured = entry:control x exit:control ;

value = code + symbolic + constant + intrinsic:unit + unknown:unit + expression ;

variable = basic x dimensions:dimension* x qualifiers:qualifier* ;

whileloop = condition:expression x body:statement x label:entity x evaluation ;

L’extension de la représentation intermédiaire de PIPS au langage C

basic = int:int + float:int + logical:int + overloaded:unit + complex:int
+ string:value + bit:int + pointer:type + derived:entity + typedef:entity;

instruction = sequence + test + loop + whileloop + goto:statement + call +
unstructured + forloop + expression;

forloop = initialization:expression x condition:expression x
incrementation:expression x body:statement ;
statement = label:entity x number:int x ordering:int x comments:string x
instruction x declarations:entity* x decls_text:string ;
syntax = reference + range + call + cast + sizeofexpression + subscript + application;
cast = type x expression ;
sizeofexpression = type + expression ;
subscript = array:expression x indices:expression* ;
application = function:expression x arguments:expression* ;
type = statement:unit + area + variable + functional + varargs:type +
unknown:unit + void:unit + struct:entity* + union:entity* + enum:entity*;
variable = basic x dimensions:dimension* x qualifiers:qualifier* ;
qualifier = const:unit + restrict:unit + volatile:unit + register:unit + auto:unit;
whileloop = condition:expression x body:statement x label:entity x
evaluation ;
evaluation = before:unit + after:unit ;
value = code + symbolic + constant + intrinsic:unit + unknown:unit + expression;

La fonction words_nullary_op_fortran

```
    // function added for fortran  by A. Mensi
static list
words_nullary_op_fortran(call obj,
            int precedence,
            bool __attribute__ ((unused)) leftmost)
```

```
{
 list pc = NIL;
 list args = call_arguments(obj);
 entity func = call_function(obj);
 string fname = entity_local_name(func);
 if(same_string_p(fname,RETURN_FUNCTION_NAME))
   pc = CHAIN_SWORD(pc, "return");
 else
   pc = CHAIN_SWORD(pc, fname);
 // STOP and PAUSE and RETURN in fortran may have 0 or 1 argument.A Mensi
 if(gen_length(args)==1) {
   if(same_string_p(fname,STOP_FUNCTION_NAME)
      || same_string_p(fname,PAUSE_FUNCTION_NAME)
      || same_string_p(fname,RETURN_FUNCTION_NAME)) {
     expression e = EXPRESSION(CAR(args));
     pc = CHAIN_SWORD(pc, " ");
     pc = gen_nconc(pc, words_subexpression(e, precedence, TRUE));
   }
   else {
     pips_internal_error("unexpected arguments");
   }
 }
 else if(gen_length(args)>1) {
   pips_internal_error("unexpected arguments");
 }
 return(pc);
}
```

La fonction words_nullary_op_c

```
// Function written by C.A. Mensi to prettyprint C or Fortran code as C code
static list
words_nullary_op_c(call obj,
                   int precedence __attribute__ ((unused)),
                   bool leftmost __attribute__ ((unused)))
{
 list pc = NIL;
 list args = call_arguments(obj);
```

```
entity func = call_function(obj);
string fname = entity_local_name(func);
int nargs = gen_length(args);
bool parentheses_p=TRUE;

/* STOP and PAUSE and RETURN in Fortran may have 0 or 1 argument.
   STOP and PAUSE are prettyprinted in C using PIPS specific C functions. */

if(nargs==0){
  if(same_string_p(fname,STOP_FUNCTION_NAME))
    pc = CHAIN_SWORD(pc, "exit(0)");
  else if(same_string_p(fname,RETURN_FUNCTION_NAME))
    pc = CHAIN_SWORD(pc, "return");
  else if(same_string_p(fname,PAUSE_FUNCTION_NAME))
    pc = CHAIN_SWORD(pc, "_f77_intrinsics_pause_(0)");
  else if(same_string_p(fname,CONTINUE_FUNCTION_NAME))
    pc = CHAIN_SWORD(pc, "");
}
else if(nargs==1){
  expression e = EXPRESSION(CAR(args));
  basic b=expression_basic(e);
  if(same_string_p(fname,STOP_FUNCTION_NAME)){
    if(basic_int_p(b)){
      // Missing: declaration of exit() if Fortran code handled
      pc = CHAIN_SWORD(pc, "exit");
    }
    else if(basic_string_p(b)){
      pc = CHAIN_SWORD(pc, "_f77_intrinsics_stop_");
    }
  }
  else if(same_string_p(fname,RETURN_FUNCTION_NAME)){
    pc = CHAIN_SWORD(pc, "return");
    parentheses_p = FALSE;
    //pips_user_error("alternate returns are not supported in C\n");
  }
  else if(same_string_p(fname,PAUSE_FUNCTION_NAME)){
    pc = CHAIN_SWORD(pc, "_f77_intrinsics_pause_");
```

```
  }
  else {
    pips_internal_error("unexpected arguments");
  }
  pc = CHAIN_SWORD(pc, parentheses_p?"(":" ");
  pc = gen_nconc(pc, words_subexpression(e, precedence, TRUE));
  pc = CHAIN_SWORD(pc, parentheses_p?")":"");
 }
 else {
  pips_internal_error("unexpected arguments");
 }
 return(pc);
}
```

La fonction words_infix_binary_op

```
static list
words_infix_binary_op(call obj, int precedence, bool leftmost)
{
 list pc = NIL;
 list args = call_arguments(obj);
 int prec = words_intrinsic_precedence(obj);
 list we1 = words_subexpression(EXPRESSION(CAR(args)), prec,
                                prec>=MINIMAL_ARITHMETIC_PRECEDENCE? leftmost: TRUE);
 list we2;
 string fun = entity_local_name(call_function(obj));
 if ( strcmp(fun, "/") == 0 )
   we2 = words_subexpression(EXPRESSION(CAR(CDR(args))),
MAXIMAL_PRECEDENCE, FALSE);
 else if ( strcmp(fun, "-") == 0 ) {
   expression exp = EXPRESSION(CAR(CDR(args)));
   if ( expression_call_p(exp) &&
        words_intrinsic_precedence(syntax_call(expression_syntax(exp))) >=
        intrinsic_precedence("*") )
     /* precedence is greater than * or / */
     we2 = words_subexpression(exp, prec, FALSE);
   else
     we2 = words_subexpression(exp, MAXIMAL_PRECEDENCE, FALSE);
```

```
}
else if ( strcmp(fun, "*") == 0 ) {
  expression exp = EXPRESSION(CAR(CDR(args)));
  if ( expression_call_p(exp) &&
       ENTITY_DIVIDE_P(call_function(syntax_call(expression_syntax(exp))))) {
    basic bexp = basic_of_expression(exp);
    if(basic_int_p(bexp)) {
      we2 = words_subexpression(exp, MAXIMAL_PRECEDENCE, FALSE);
    }
    else
      we2 = words_subexpression(exp, prec, FALSE);
    free_basic(bexp);
  }
  else
    we2 = words_subexpression(exp, prec, FALSE);
}
else {
  we2 = words_subexpression(EXPRESSION(CAR(CDR(args))), prec,
                            prec<MINIMAL_ARITHMETIC_PRECEDENCE);
}
if ( strcmp(fun,PLUS_C_OPERATOR_NAME) == 0 )
  fun = "+";
else  if ( strcmp(fun, MINUS_C_OPERATOR_NAME) == 0 )
  fun = "-";
else  if ( strcmp(fun,BITWISE_AND_OPERATOR_NAME) == 0 )
  fun = "&";
else if (!is_fortran){
  if(strcasecmp(fun, GREATER_THAN_OPERATOR_NAME)==0)
    fun=C_GREATER_THAN_OPERATOR_NAME;
  else if(strcasecmp(fun, LESS_THAN_OPERATOR_NAME)==0)
    fun=C_LESS_THAN_OPERATOR_NAME;
  else if(strcasecmp(fun,GREATER_OR_EQUAL_OPERATOR_NAME)==0)
    fun=C_GREATER_OR_EQUAL_OPERATOR_NAME;
  else if(strcasecmp(fun,LESS_OR_EQUAL_OPERATOR_NAME)==0)
    fun=C_LESS_OR_EQUAL_OPERATOR_NAME;
  else if(strcasecmp(fun, EQUAL_OPERATOR_NAME) ==0)
    fun=C_EQUAL_OPERATOR_NAME;
```

```
  else if(strcasecmp(fun,NON_EQUAL_OPERATOR_NAME)==0)
    fun=C_NON_EQUAL_OPERATOR_NAME;
  else if(strcasecmp(fun,AND_OPERATOR_NAME)==0)
    fun=C_AND_OPERATOR_NAME;
  else if(strcasecmp(fun, OR_OPERATOR_NAME)==0)
    fun=C_OR_OPERATOR_NAME;
 }
 if ( prec < precedence )
  pc = CHAIN_SWORD(pc, "(");
 pc = gen_nconc(pc, we1);
 pc = CHAIN_SWORD(pc, strdup(fun));
 pc = gen_nconc(pc, we2);
 if( prec < precedence )
  pc = CHAIN_SWORD(pc, ")");
 return(pc);
}
```

La fonction words_prefix_unary_op

```
static list
words_prefix_unary_op(call obj,
                      int __attribute__ ((unused)) precedence,
                      bool __attribute__ ((unused)) leftmost)
{
  list pc = NIL;
  expression e = EXPRESSION(CAR(call_arguments(obj)));
  int prec = words_intrinsic_precedence(obj);
  string fun = entity_local_name(call_function(obj));
  if (strcmp(fun,"++pre") == 0)
    fun = "++";
  else
    if (strcmp(fun,"--pre") == 0)
      fun = "--";
    else
      if (strcmp(fun,"*indirection") == 0)
        /* Since we put no spaces around an operator (to not change Fortran),
the blank
           before '*' is used to avoid the confusion in the case of divide
```

```
operator, i.e

          d1 = 1.0 / *det  in function inv_j, SPEC2000 quake benchmark. */

        fun = " *";

      else if (strcmp(fun,"+unary") == 0)

         fun = "+";

       else if(!is_fortran){

        if(strcasecmp(fun, NOT_OPERATOR_NAME)==0)

    fun=C_NOT_OPERATOR_NAME;

       }

  pc = CHAIN_SWORD(pc,strdup(fun));

  pc = gen_nconc(pc, words_subexpression(e, prec, FALSE));

  return(pc);

}
```

La fonction text_block_elseif

```
static text

text_block_elseif(

  entity module,

  string label,

  int margin,

  test obj,

  int n)

{

 text r = make_text(NIL);

 list pc = NIL;

 statement tb = test_true(obj);

 statement fb = test_false(obj);


 pc = CHAIN_SWORD(pc, strdup(is_fortran?"ELSEIF (":"else if ("));

 pc = gen_nconc(pc, words_expression(test_condition(obj)));

 pc = CHAIN_SWORD(pc, strdup(is_fortran?")

THEN":(one_liner_p(tb)?")":") {")));

 ADD_SENTENCE_TO_TEXT(r,

                      make_sentence(is_sentence_unformatted,

                                    make_unformatted(strdup(label), n,

                                                     margin, pc)));
```

```
 MERGE_TEXTS(r, text_statement_enclosed(module, margin+INDENTATION,
tb,!one_liner_p(tb)));

 if (!is_fortran && !one_liner_p(tb)) {
   ADD_SENTENCE_TO_TEXT(r, MAKE_ONE_WORD_SENTENCE(margin,
strdup("}")));
 }

 if(statement_test_p(fb)
    && empty_comments_p(statement_comments(fb))
    && entity_empty_label_p(statement_label(fb))) {
  MERGE_TEXTS(r, text_block_elseif(module,
                                   label_local_name(statement_label(fb)),
                                   margin,
                                   statement_test(fb), n));

 } else {
   MERGE_TEXTS(r, text_block_else(module, label, margin, fb, n));
 }
 ifdebug(8){
   fprintf(stderr,"elseif================================\n");
   print_text(stderr,r);
   fprintf(stderr,"==============================\n");
 }
 return(r);
}
```

La fonction text_named_module

```
text
text_named_module(
    entity name, /* the name of the module */
    entity module,
    statement stat)
{
  text r = make_text(NIL);
  code c = entity_code(module);
  string s = code_decls_text(c);
```

```
text ral = text_undefined;

debug_on("PRETTYPRINT_DEBUG_LEVEL");
is_fortran = !get_bool_property("PRETTYPRINT_C_CODE");

/* This guard is correct but could be removed if find_last_statement()
 * were robust and/or if the internal representations were always "correct".
 * See also the guard for reset_last_statement()
 */
if(!get_bool_property("PRETTYPRINT_FINAL_RETURN"))
  set_last_statement(stat);

precedence_p = !get_bool_property("PRETTYPRINT_ALL_PARENTHESES");

if (is_fortran)
  {
    if ( strcmp(s,"") == 0
       || get_bool_property("PRETTYPRINT_ALL_DECLARATIONS") )
    {
      if (get_bool_property("PRETTYPRINT_HEADER_COMMENTS"))
        /* Add the original header comments if any: */
        ADD_SENTENCE_TO_TEXT(r, get_header_comments(module));

      ADD_SENTENCE_TO_TEXT(r,
                           attach_head_to_sentence(sentence_head(name), module));

      if (head_hook)
        ADD_SENTENCE_TO_TEXT(r, make_sentence(is_sentence_formatted,
                                              head_hook(module)));

      if (get_bool_property("PRETTYPRINT_HEADER_COMMENTS"))
        /* Add the original header comments if any: */
        ADD_SENTENCE_TO_TEXT(r, get_declaration_comments(module));

      MERGE_TEXTS(r, text_declaration(module));
      MERGE_TEXTS(r, text_initializations(module));
    }
```

```
    else
    {
      ADD_SENTENCE_TO_TEXT(r,
                           attach_head_to_sentence(make_sentence(is_sentence_formatted,
                                                                 strdup(s)),
                                                   module));
    }
  }
else
  {
    /* C prettyprinter */
    pips_debug(3,"Prettyprint function %s\n",entity_name(name));
    if (!compilation_unit_p(entity_name(name)))
    {
      /* Print function header if the current module is not a compilation unit*/
      ADD_SENTENCE_TO_TEXT(r,attach_head_to_sentence(sentence_head(name), module));
      ADD_SENTENCE_TO_TEXT(r,MAKE_ONE_WORD_SENTENCE(0,"{"));
      /* get the declarations when they are not located in the module statement. A.Mensi */
      if(ENDP(statement_declarations(stat))) {
        list l = code_declarations(value_code(entity_initial(module)));

        MERGE_TEXTS(r,c_text_entities(module, l, INDENTATION));
      }
    }
  }

set_alternate_return_set();

if (stat != statement_undefined) {
  MERGE_TEXTS(r, text_statement(module, is_fortran?0:INDENTATION, stat));
}

ral = generate_alternate_return_targets();
reset_alternate_return_set();
MERGE_TEXTS(r, ral);

if (!compilation_unit_p(entity_name(name)) || is_fortran)
```

```
    {
      /* No need to print TAIL (}) if the current module is a C compilation unit*/
      ADD_SENTENCE_TO_TEXT(r, sentence_tail());
    }

  if(!get_bool_property("PRETTYPRINT_FINAL_RETURN"))
    reset_last_statement();

  debug_off();
  return(r);
}
```

La fonction text_statement_enclosed

```
text text_statement_enclosed(
    entity module,
    int margin,
    statement stmt,
    bool braces_p)
{
  instruction i = statement_instruction(stmt);
  text r= make_text(NIL);
  text temp;
  string label =
    entity_local_name(statement_label(stmt)) + strlen(LABEL_PREFIX);
  string comments = statement_comments(stmt);

  pips_assert("Blocks have no comments", !instruction_block_p(i)||empty_comments_p(comments));

  /* 31/07/2003 Nga Nguyen : This code is added for C, because a statement can have its own declarations */
  list l = statement_declarations(stmt);

  if (!ENDP(l) && !is_fortran)
    {
      /* printf("Statement declarations : ");
        print_entities(l); */
      MERGE_TEXTS(r,c_text_entities(module,l,margin));
    }
```

```
pips_debug(2, "Begin for statement %s with braces_p=%d\n", statement_identification(stmt),braces_p);
pips_debug(9, "statement_comments: --%s--\n",
        string_undefined_p(comments)? "<undef>": comments);
 if(statement_number(stmt)!=STATEMENT_NUMBER_UNDEFINED &&
   statement_ordering(stmt)==STATEMENT_ORDERING_UNDEFINED) {
  /* we are in trouble with some kind of dead (?) code...
     but we might as well be dealing with some parsed_code */
  pips_debug(1, "I unexpectedly bumped into dead code?\n");
}

if (same_string_p(label, RETURN_LABEL_NAME))
  {
    pips_assert("Statement with return label must be a return statement",
                        return_statement_p(stmt));

    /* do not add a redundant RETURN before an END, unless requested */
    if(get_bool_property("PRETTYPRINT_FINAL_RETURN")
     || !last_statement_p(stmt))
    {
      sentence s = MAKE_ONE_WORD_SENTENCE(margin, RETURN_FUNCTION_NAME);
      temp = make_text(CONS(SENTENCE, s ,NIL));
    }
    else {
    temp = make_text(NIL);
    }
  }
else
  {
    temp = text_instruction(module, label, margin, i,
                                        statement_number(stmt));
  }

/* note about comments: they are duplicated here, but I'm pretty
 * sure that the free is NEVER performed as it should. FC.
 */
if(!ENDP(text_sentences(temp))) {
  MERGE_TEXTS(r, init_text_statement(module, margin, stmt));
```

```
  if (! string_undefined_p(comments)) {
    if(is_fortran) {
     ADD_SENTENCE_TO_TEXT(r, make_sentence(is_sentence_formatted,
                                           strdup(comments)));
    }
    else {
    text ct = C_comment_to_text(margin, comments);
    MERGE_TEXTS(r, ct);
    }
  }
  MERGE_TEXTS(r, temp);
}
else {
  /* Preserve comments and empty C instruction */
  if (! string_undefined_p(comments)) {
    if(is_fortran) {
     ADD_SENTENCE_TO_TEXT(r, make_sentence(is_sentence_formatted,
                                           strdup(comments)));
    }
    else {
    text ct = C_comment_to_text(margin, comments);
    MERGE_TEXTS(r, ct);
    }
  }
  else if(!is_fortran && !braces_p) {
    // Because C braces can be eliminated and hence semi-colon
    // may be mandatory in a test branch or in a loop body.
    // A. Mensi
    sentence s = MAKE_ONE_WORD_SENTENCE(margin, strdup(";"));
    ADD_SENTENCE_TO_TEXT(r, s);
  }
  free_text(temp);
}
attach_statement_information_to_text(r, stmt);

ifdebug(1) {
  if (instruction_sequence_p(i)) {
```

```
    if(!(statement_with_empty_comment_p(stmt)
       && statement_number(stmt) == STATEMENT_NUMBER_UNDEFINED
       && unlabelled_statement_p(stmt))) {
     user_log("Block statement %s\n"
                  "Block number=%d, Block label=\"%s\", block comment=\"%s\"\n",
                  statement_identification(stmt),
                  statement_number(stmt), label_local_name(statement_label(stmt)),
                  statement_comments(stmt));
     pips_error("text_statement", "This block statement should be labelless, numberless"

                  " and commentless.\n");
     }
    }
  }
  ifdebug(8){
   fprintf(stderr,"text_statement_enclosed==================================\n");
   print_text(stderr,r);
   fprintf(stderr,"============================\n");
  }

   pips_debug(2, "End for statement %s\n", statement_identification(stmt));

   return(r);
}
```

La fonction CParserError

```
void CParserError(char *msg)
{
  entity mod = get_current_module_entity();
  string mod_name = entity_undefined_p(mod)? "entity_undefined":entity_module_name(mod);

  /* Reset the parser global variables ?*/

  pips_debug(4,"Reset current module entity %s\n", mod_name);
```

```
/* The error may occur before the current module entity is defined */
error_reset_current_module_entity();

// Get rid of partly declared variables
if(mod!=entity_undefined) {
  value v = entity_initial(mod);
  code c = value_code(v);

  code_declarations(c) = NIL;
  code_decls_text(c) = string_undefined;
  CleanLocalEntities(mod);
}

// Could not rebuild filename (A. Mensi)
// c_in = safe_fopen(file_name, "r");
// safe_fclose(c_in, file_name);

/* Stacks are not allocated yet when dealing with external
   declarations. I assume that all stacks are declared
   simultaneously, hence a single test before freeing. */
if(!entity_undefined_p(mod)) {
  reset_entity_type_stack_table();
  if(!stack_undefined_p(SwitchGotoStack)) {
    stack_free(&SwitchGotoStack);
    stack_free(&SwitchControllerStack);
    stack_free(&LoopStack);
    stack_free(&BlockStack);
    /* Reset them to stack_undefined_p instead of STACK_NULL */
    SwitchGotoStack = stack_undefined;
    SwitchControllerStack = stack_undefined;
    LoopStack = stack_undefined;
    BlockStack = stack_undefined;

    stack_free(&ContextStack);
    stack_free(&FunctionStack);
    stack_free(&FormalStack);
    stack_free(&OffsetStack);
```

```
    stack_free(&StructNameStack);
    ContextStack = stack_undefined;
    FunctionStack = stack_undefined;
    FormalStack = stack_undefined;
    OffsetStack = stack_undefined;
    StructNameStack = stack_undefined;
   }
  }

  reset_current_C_line_number();
  /* get rid of all collected comments */
  reset_C_comment(TRUE);
  pips_user_warning("Recovery from C parser failure not (fully) implemented yet.\n"
                    "C parser is likely to fail later if re-used.\n");
  pips_user_error(msg);
  debug_off();
}
```