Analyse statique et domaines abstraits symboliques

Mémoire d'habilitation à diriger des recherches

Laurent Mauborgne

Habilitation soutenue le 12 février 2007 à l'Université Paris-Dauphine

Jury : Patrick Cousot (rapporteur) Roberto Giacobazzi (rapporteur)
Jean Goubault-Larrecq Vangelis Paschos (coordinateur)
David Schmidt (rapporteur) Reinhard Wilhelm (président)

À Valérie

Table des matières

Chapitre 1

Activités de Recherche

J'ai commencé mon travail de recherche en 1993, lors d'un stage au Laboratoire d'Informatique de l'École Normale Supérieure (LIENS), sous la direction de Patrick Cousot. Le sujet du stage était l'implémentation d'une analyse de nécessité par interprétation en utilisant une structure de données plus efficace pour la représentation des fonctions booléennes. L'idée de Patrick Cousot était d'utiliser l'état de l'art du model checking, à l'époque des arbres de décisions binaires avec partage (BDDs), et d'essayer de voir ce que pouvait donner cette technologie en analyse statique de logiciels. Le model checking étant limité aux ensembles finis, l'application à l'analyse de nécessité de Mycroft (1980), qui utilise un nombre fini de fonctions booléennes, était assez facile d'un point de vue théorique.

L'idée d'essayer d'implémenter une analyse, même élémentaire, mettant en pratique la théorie de l'interprétation abstraite m'a séduit et j'ai beaucoup apprécié la combinaison du travail théorique, et la préoccupation pratique motivant ce stage. J'ai découvert à cette occasion que la théorie de l'interprétation abstraite était aussi très bien adaptée pour prendre en compte les préoccupations d'efficacité des analyses. Le sujet m'a permis de développer d'une part une analyse des langages d'ordre supérieur et d'autre part un nouveau domaine abstrait utilisant les BDDs, publié à SAS en 1996. J'ai donc ensuite cherché un sujet de recherche au sein de la même équipe. Patrick Cousot m'a laissé une liberté totale dans la définition de mon domaine de recherche et mes goûts m'ont porté vers un aspect algorithmes et structures de données de l'analyse statique par interprétation abstraite.

1.1 Structures de données

La caractéristique principale de l'analyse statique par interprétation abstraite est l'utilisation constante d'approximations maîtrisées. Cette liberté supplémentaire permise par ce domaine permet d'explorer l'algorithmique d'une manière totalement nouvelle. J'ai ainsi pu revisiter une partie d'un domaine extrêmement classique en algorithmique, celui des arbres. En informatique comme dans tous les domaines, on a tendance à s'appuyer sur les travaux précédents pour progresser, et en analyse de programme, la tentation est grande d'utiliser des structures de donnée à l'efficacité éprouvée. Mais je me suis aperçu que souvent les structures de données manipulant des ensembles d'arbres sont peu adaptées à un contexte dans lequel l'approximation est non seulement permise mais la plupart du temps obligatoire. Dans cette recherche, il n'a pas été question de faire table rase des structures existantes, mais de trouver quels concepts adapter pour profiter des possibilités de gagner à la fois de la mémoire, du temps mais aussi de l'expressivité en approximant à bon escient.

Le principe de partage des données m'avait semblé un des points clé expliquant l'efficacité des BDDs, autant en model checking qu'en analyse statique. J'ai donc essayé dans le même temps d'appliquer ce principe au maximum, mais dans un premier temps je n'ai pas utilisé les bons concepts et mes structures faisaient trop de repliages pour obtenir une bonne représentation. Au bout d'un an de tentatives infructueuses, je suis reparti de zéro avec un nouveau concept de partage mieux formalisé qui m'a permis de développer les algorithmes génériques présentés dans ma thèse. Le chemin était encore long pour fournir un véritable domaine abstrait d'ensembles d'arbres, et il m'a fallu en particulier développer des représentations efficaces pour les relations possiblement infinies. On observe en effet fréquemment en interprétation abstraite que l'approximation dans un domaine *a priori* fini donne de moins bons résultats que de faire des calculs dans un espace infini en utilisant des approximations qui dépendent de l'histoire des calculs. La possibilité d'approximer certaines opérations

algébriques, comme l'union qui est presque toujours approchée en interprétation abstraite, m'a permis de présenter une structure de donnée plus expressive que les représentations classiques et des algorithmes pouvant être paramétrés par la précision attendue de l'analyse. Cette recherche a été assez longue, puisqu'il m'a fallu encore quatre années pour présenter un résultat cohérent dans ma thèse. Malgré la longueur de cette thèse, je n'avais pas eu le temps d'implémenter les algorithmes qui y étaient présentés. Comme l'utilisation pratique était une de mes motivations premières, et aussi une grande part de la justification de ces recherches, c'est ce à quoi je me suis attelé après la soutenance de ma thèse.

1.2 Analyse statique

L'implémentation des structures de données développées pendant ma thèse m'a permis d'affiner mes algorithmes et de corriger certaines erreurs. J'ai aussi un peu généralisé mes résultats que je présente dans le chapitre 2. Mais le véritable test d'une telle implémentation devait être une analyse statique utilisant ces structures. En effet, une bonne partie de l'intuition qui m'avait conduit à ces structures concernait l'utilisation qui en était faite, qui justifie un emploi assez fréquent (mais pas forcément systématique) du partage des données et de la réutilisation des calculs.

C'est alors que l'occasion de réaliser un véritable analyseur statique s'est présentée avec le projet Astrée (chapitre 4). L'opportunité de mettre véritablement en pratique nos idées et nos implémentations académiques a été immédiatement saisie par toute l'équipe. Notre motivation était à la fois la confrontation aux besoins réels d'un industriel, le défit d'une analyse de codes de grande taille, et le développement d'un analyseur robuste qui servirait de base pour tester nos futures implémentations. En pratique, le démarrage du projet n'a pas vraiment été l'occasion d'appliquer le travail existant, mais il a été le moteur de nouvelles recherches, peut-être plus appliquées que celles que nous avions faites avant ce projet. Seuls les octogones d'Antoine Miné ont été incorporés au bout de quelques mois. Les programmes analysés manipulant principalement des nombres, et les propriétés à prouver étant au départ assez simples, le projet Astrée n'a pas encore eu besoin d'ensembles d'arbres complexes ou de relation infinitaire.

Mais le projet Astrée a sucité le besoin de domaines pour représenter des invariants relationnels parfois assez variés, et nous avons expérimenté une nouvelle contrainte, en plus des contraintes de temps de calcul ou d'occupation mémoire : la contrainte du temps de développement des domaines. Cela m'a amené à implémenter de petits domaines relationnels et à réfléchir à des domaines relationnels assez génériques pour l'instancier facilement à plusieurs formes d'invariants. Le chapitre 3 décrit cette recherche sur les relations symboliques. La partie la plus importante dans Astrée est certainement ce que j'avais d'abord appelé le partitionnement de trace. Au départ, c'était simplement un déroulement de certaines boucles, puis est venue l'idée que ce déroulement pouvait être étendu, enfin il a été généralisé aux partitions de trace, avec la collaboration de Xavier Rival. En écrivant le formalisme de ce domaine, nous nous sommes aperçu que la notion de partition était une restriction inutile et dans ce mémoire j'ai opté pour la terminologie moins restrictive de distinction ou discrimination de traces.

Le projet Astrée a aussi naturellement fourni l'occasion de tester en grandeur nature certaines des idées présentées dans ce mémoire. Les résultats de ces expérimentations sont regroupés dans l'appendice A.

Chapitre 2

Graphes et partage des données

Dans ce chapitre, je revisite certains résultats de Mauborgne (1999b) et Mauborgne (2000a) sur les arbres infinis réguliers, dans un cadre plus général. Je présente aussi quelques nouveaux résultats dont certains permettent une implémentation plus efficace et que je n'ai pas encore publiés.

2.1 Partage des données et interprétation abstraite

2.1.1 Partage maximal incrémental

Le partage maximal des données est un principe vieux comme l'informatique (Michie, 1968; Snyder, 1977; Mehlhorn and Tsakalidis, 1990)[1], qui consiste à ne pas dupliquer en mémoire des données sémantiquement égales. Cela fait bien sûr gagner de la place en mémoire, mais ce n'est pas son intérêt majeur. L'intérêt du partage maximal des données en mémoire apparaît lorsqu'on a besoin de tester fréquemment l'égalité entre deux données. En effet, lorsque les données sont partagées, il suffit de comparer leur adresse physique pour savoir si elles sont égales. Le test d'égalité devient si facile qu'il peut devenir rentable de mémoriser les calculs de manière à les réutiliser si on en a besoin.

En fait, cette apparente facilité du test d'égalité a bien entendu un coût. Car pour réussir à partager les données en mémoire, il faut bien sûr savoir au moment où on crée une donnée si elle n'est pas déjà quelque part en mémoire, de façon à ne pas la dupliquer. Cela signifie qu'à chaque nouvelle donnée manipulée, il a fallu faire un calcul qui compare la nouvelle donnée avec toutes celles qui sont en mémoire. En fait, faire un partage maximal des données peut être vu comme la mémorisation de ce calcul. Bien sûr, le calcul d'égalité ne va pas comparer toutes les données en mémoire avec la nouvelle donnée, mais utiliser des techniques de hashage pour ne comparer cette donnée qu'avec une petite partie de la mémoire. Cette mémorisation pourrait sembler inutilement coûteuse car on peut utiliser les mêmes techniques de hashage pour comparer deux données, au moment où on en a vraiment besoin. Mais si toutes les données à partir desquelles est construite une nouvelle donnée sont partagées au maximum, il est possible de rendre les techniques de hashage beaucoup plus efficaces, puisqu'on n'a plus à comparer que des adresses mémoire et la nouvelle partie créée. Ainsi, le partage maximal montre tout son intérêt si on est capable de l'obtenir de manière *incrémentale*.

Cette technique a été appliquée au cas des listes en LISP (Goto, 1974), sous le nom de hash-consing, puis généralisée au cas des arbres. Les arbres (ou les termes) se prêtent en effet très bien au partage maximal, car pour construire un arbre, on a en général besoin de l'étiquette de la racine et des fils, et si ces fils sont partagés, il est très rapide de trouver si on a déjà mémorisé l'arbre en utilisant un hash de l'étiquette et des adresses mémoires des fils.

2.1.2 Calcul de points fixes

L'essentiel de la partie constructive de l'interprétation abstraite (Cousot, 1978; Cousot and Cousot, 1979a; Cousot, 1996, 2000) consiste en l'approximation de points fixes par des limites de suites d'itérés approchés (Cousot, 1981). Une suite d'itérés $x_0, x_1, \ldots, x_i, \ldots$ pour une fonction F se construit par applications successives de la fonction F, d'abord au point de départ x_0, puis au résultat de F sur ce point (x_1), et en itérant, chaque x_{i+1} étant $F(x_i)$. L'observation qui m'a conduit à essayer d'appliquer les principes du partage maximal à l'interprétation abstraite, c'est qu'en général on approxime

[1] Andrei P. Ershov aurait inventé le concept vers la fin des années 50.

une suite correspondant à un ensemble de comportements possibles d'un programme. Ces ensembles croissent au fur et à mesure des calculs et il apparaît donc naturellement que la suite est construite de manière incrémentale, avec à chaque étape beaucoup d'opportunités de réutiliser des résultats de calculs précédents.

Si les ensembles de comportements qu'on cherche à approximer sont numériques (par exemple les valeurs que peuvent prendre des variables numériques), il est souvent très efficace d'approximer les ensembles par des propriétés numériques (par exemple les bornes pour le domaine des intervalles). Dans ce cas, la réutilisation de calculs est plus rare et généralement trop coûteuse. En revanche, si on s'intéresse à des propriétés purement symboliques, on ne dispose pas de résultats mathématiques puissants permettant de manipuler les propriétés. On peut dans certains cas se ramener aux propriétés numériques, par exemple en comptant certaines structures (Parikh, 1966; Venet, 1996; Rugina, 2004; Feret, 2005a). Mais si on veut être précis, il faut de toutes façons garder au maximum la structure des propriétés et dans ce cas on aura souvent l'occasion de réutiliser des calculs sur des sous-structures.

On peut noter aussi que l'extrapolation des suites d'itérés par élargissement est plus efficace (Bagnara et al., 2005) si la représentation des itérés ne contient aucune redondance, de façon à ce que cette représentation soit aussi proche que possible de sa sémantique. Cela peut être atteint par un partage maximal de toutes les sous-structures d'une structure, dans le cas de propriétés symboliques.

Enfin, le point le plus important qui m'a amené à étudier les possibilité d'avoir du partage dans les structures de données en interprétation abstraite, c'est qu'il est nécessaire de tester l'inclusion à chaque étape de l'itération de point fixe pour savoir si on a atteint (une approximation de) ce point fixe. En l'absence de partage, cela peut obliger à parcourir entièrement les structures, alors qu'il n'est pas utile de regarder ce qui n'a pas changé entre deux itérations.

2.2 Partage maximal de graphes

La structure de graphe permet d'exprimer un grand nombre d'objets utiles en informatique, comme la structure de la mémoire en présence de pointeurs, ou encore les relations de dépendance entre variables, les traces d'exécution d'un programme, ... Il est donc naturel de chercher à s'en servir pour représenter certains comportements de programmes. Le problème réside dans la complexité de la manipulation des graphes, qui évoluent au cours des calculs, dans un contexte où on cherche à partager au maximum les sous-structures communes.

On peut trouver plusieurs définitions des graphes. Ceux qu'il m'a semblé intéressant de manipuler sont ceux qui sont les plus proches de ce qu'on peut trouver en toute généralité dans le tas d'un programme : ils sont constitués d'un ensemble fini de sommets étiquetés et d'un ensemble fini d'arêtes orientées et étiquetées, chaque arête reliant deux sommets. Dans le langage « classique » des graphes, ce sont donc des multigraphes orientés étiquetés. Une propriété intéressante des graphes est que tout ensemble fini de graphes est lui-même (représentable par) un graphe. Les graphes pourront donc nous servir à la fois d'éléments de base mais aussi d'ensembles structurés de ces éléments.

2.2.1 Graphe minimal

Un chemin partant d'un sommet S d'un tel graphe sera une suite d'étiquettes d'arêtes $\ell_1, \ell_2, \ldots, \ell_n$ telle que ℓ_1 est l'étiquette d'une arête partant de S et arrivant en S_1 et pour $1 < i \leq$, ℓ_i est l'étiquette d'une arête partant de S_{i-1} et arrivant en S_i. On dira que ce chemin aboutit en S_n. Si on note p un chemin, on notera $S.p$ l'ensemble des sommets auquels on peut aboutir en suivant p à partir de S.

Le partage maximal dans un graphe va consister à ne pas dupliquer des sommets qui sont indistinguables du point de vue des fonctions qui travaillent sur les graphes. Deux sommets S et T seront dits distinguables si il existe un chemin p (éventuellement vide) tel que l'ensemble des étiquettes de $S.p$ soit différent de l'ensemble des étiquettes de $T.p$. Deux arêtes seront indistinguables si elles ont la même étiquette et des sommets de départ indistinguables et des sommets d'arrivée indistinguables.

Le problème de cette section est donc le suivant : nous avons un ensemble fini de graphes (les graphes qui sont apparus au cours de calcul), à partir desquels nous voulons calculer de nouveaux graphes. Nous cherchons à maintenir un ensemble de sommets $\mathcal{U}$ représentés de manière unique, c'est-à-dire tels que nous maintenons que deux sommets de cet ensembles sont distincts si et seulement si ils sont distinguables. Cela permettra de maintenir un partage maximal pour certains graphes et donc pour ces graphes de

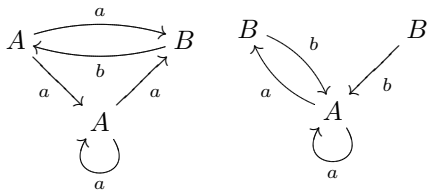

FIG. 2.1 – Deux graphes équivalents

faire de la mémorisation d'applications de fonctions.

On appellera minimal un graphe dont tous les sommets et toutes les arêtes sont distinguables. La première solution au problème de partage maximal est donc de considérer le graphe formé des sommets que l'on cherche à représenter de manière unique et de la transformer en un graphe équivalent minimal. Deux graphes seront dits équivalents dès que tout sommet de l'un est indistinguable d'au moins un sommet de l'autre (voir la figure 2.1). Cette définition implique la même propriété sur les arêtes.

Le problème de trouver un graphe minimal équivalent à un graphe donné est très proche du problème de minimisation d'automates. En effet, un automate de mots sur l'aphabet A n'est rien d'autre qu'un multigraphe dont les sommets sont les états, étiquetés soit par final ou par non-final, et les arêtes les transitions étiquetées par A. Dans le cas d'un automate déterministe, la notion d'équivalence de deux états est la même que celle que nous venons de définir pour deux sommets d'un multigraphe. Dans ce cas, le meilleur algorithme théorique de l'état de l'art est l'algorithme de Hopcroft (Hopcroft, 1971), dont la complexité est en $n \log n$ où n est le nombre d'états de l'automate de départ[2]. Il m'a donc semblé difficile de faire mieux dans le cas général. Il se trouve que l'algorithme de Hopcroft a été généralisé au problème du partitionnement de graphe par Cardon and Crochemore (1982), avec une complexité de $m \log n$ où m est le nombre d'arêtes et n le nombre de sommets. Le problème du partitionnement est de trouver la partition de l'ensemble des sommets la plus grossière parmi toutes celles qui sont plus fines qu'une partition P et qui respectent le graphe de la façon suivante : si deux sommets sont dans la même classe d'équivalence de la partition finale, alors pour toute étiquette d'arête ℓ, il y a le même nombre d'arêtes étiquetées par ℓ qui part des deux sommets, et si une arête partant de l'un des sommets arrive dans une classe d'équivalence, alors il en est de même pour l'autre sommet.

Le principe de l'algorithme est de partir de la partition P, puis de raffiner progressivement cette partition en prenant un ensemble E de la partition courante et une étiquette d'arête ℓ, puis on associe à chaque sommet le nombre d'arêtes d'étiquette ℓ qui arrivent dans E et enfin on partitionne chaque ensemble de la partition en fonction de ces nombres. La clé de la complexité logarithmique est de ne pas réutiliser les plus grosses des classes créées comme critère de raffinement aux étapes suivantes.

Le problème de trouver le graphe minimal peut être résolu en utilisant presque l'algorithme de Cardon et Crochemore. Il suffit de partir d'une partition selon les étiquettes des sommets, et de raffiner en fonction de l'existence d'arêtes et non de leur nombre. On peut montrer (Mauborgne, 2000b) qu'alors deux sommets sont dans la même classe de la partition résultante si et seulement si ils sont indistinguables. L'étape finale consiste donc à ne garder qu'un sommet par classe et à supprimer les arêtes en double, ce qui est linéaire en la taille du graphe.

2.2.2 Partage incrémental

Comme on l'a vu précédemment (cf section 2.1.1, p 7), le partage ne montre tout son intérêt que si il est incrémental. Il faut donc, étant donné notre ensemble de sommets $\mathcal{U}$ représentés de manière unique, être capable de prendre un graphe G dont une partie des sommets est dans $\mathcal{U}$ (ce graphe sera typiquement le résultat d'une fonction sur les graphes précédemment calculés), et d'étendre éventuellement $\mathcal{U}$ de façon à trouver dans $\mathcal{U}$ un sommet indistinguable d'un sommet donné de G.

Si on utilise l'algorithme dérivé de Cardon et Crochemore, la seule solution est de minimiser le graphe constitué des sommets de $\mathcal{U}$, des sommets accessibles depuis $\mathcal{U}$ et de G. La complexité dans un cadre où on va souvent rajouter des sommets est alors beaucoup trop élevée. Il faut de nouveau algorithmes capables d'exploiter le fait que les sommets de $\mathcal{U}$ sont déjà tous distinguables. Il faut bien sûr que ces sommets restent distinguables, et on va donc imposer d'une part que $\mathcal{U}$ soit fermé (tout chemin partant de $\sqcap$ mène dans $\mathcal{U}$) et qu'une fois qu'un sommet est dans $\mathcal{U}$, on n'ajoute plus jamais d'arête partant de ce sommet.

Si le graphe G privé des sommets dans $\mathcal{U}$ (noté $G_{\mathcal{U}}$) n'est pas cyclique, alors on connaît une mé-

[2]Cette complexité dépend aussi linéairement de la taille de l'alphabet A.

thode très efficace pour faire le partage incrémental de $G_{\mathcal{U}}$, le hash-consing (Goto, 1974; Colmerauer, 1982). Il suffit de maintenir une table $\mathcal{D}_{hc}$ qui à une clé associe un sommet. Les clés à considérer dans ce cas sont simplement des couples étiquette de sommet × clé des fils, la clé des fils étant la liste des couples étiquette d'arête × sommet destination de l'arête, ce sommet étant nécessairement dans $\mathcal{U}$. On appellera ces clés des clés de sommet. Le partage incrémental parcourt alors $G_{\mathcal{U}}$. Quand il passe par un sommet dont toutes les arêtes mènent dans $\mathcal{U}$, on peut utiliser la clé de sommet de ce sommet pour savoir rapidement si il est indistinguable d'un sommet de $\mathcal{U}$. Si oui, on le remplace par ce sommet de $\mathcal{U}$, sinon, on le rajoute dans $\mathcal{U}$ et on rajoute sa clé de sommet dans $\mathcal{D}_{hc}$. Un simple parcours, de complexité la taille de $G_{\mathcal{U}}$ suffit donc.

2.2.3 Partage incrémental en présence de cycles

Si le sous-graphe non-partagé contient des cycles, on ne peut pas simplement appliquer directement les techniques de hash-consing. Dans un premier temps, prenons le cas simple d'un graphe fortement connexe ne contenant aucun sommet de $\mathcal{U}$. Dans Mauborgne (1999b), j'ai proposé d'utiliser une table de hashage associant à des clés uniques les graphes fortement connexes associés. Comme les sommets de $\mathcal{U}$ ne peuvent appartenir qu'à des graphes minimaux, ces clés n'ont pas besoin d'être uniques pour tous les graphes, mais juste pour les graphes minimaux. J'ai donc défini les clés de cycles, une clé par sommet S du graphe, chaque clé étant l'arbre couvrant en profondeur (en utilisant un ordre prédéfini sur les étiquettes des arêtes) du graphe en partant de S, avec aux feuilles des chemins. Comme il s'agit de graphes fortement connexes, les feuilles d'un arbre couvrant correspondent toujours à un sommet du graphe qui a déjà été rencontré dans le parcours. Le chemin qu'on met à cette feuille est alors le chemin qu'on a pris dans le parcours pour aller de S à ce sommet (voir figure 2.2). On notera que ces clés ne sont valables que si les chemins désignent un unique sommet. On dira qu'un multigraphe est déterministe si pour tout sommet S, toutes les arêtes qui partent de S ont une étiquette distincte. Dans la suite on se restreindra aux graphes déterministes, sur lesquels les algorithmes semblent plus efficaces. Ainsi, on perd la possibilité de représenter facilement des opérateurs commutatifs, mais cela correspond mieux à ce qu'on peut effectivement programmer directement.

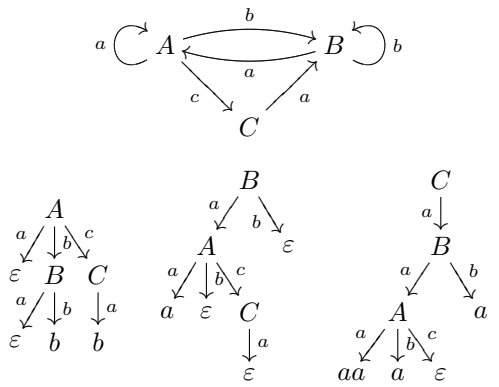

FIG. 2.2 – Un graphe fortement connexe et ses clés de cycles.

Pour retrouver les cycles, on maintient donc une table supplémentaire, $\mathcal{D}_c$ qui associe aux clés d'arbres les sommets correspondant des cycles. Les clés d'arbres elles-mêmes utilisent une autre table afin d'être hash-consées. Pour savoir si un sommet d'un graphe G fortement connexe est indistinguable d'un sommet de $\mathcal{U}$, on peut donc commencer par le minimaliser, puis chercher si la clé d'un de ses sommets est dans $\mathcal{D}_c$. À ce stade, on a le choix, soit on privilégie l'espace mémoire et on ne stocke qu'un seul sommet par graphe fortement connexe de $\mathcal{U}$, et dans ce cas, pour être sûr qu'un sommet de G est indistinguable, il faut construire autant de clés de cycle que de sommets de G et tester pour chacun leur appartenance à $\mathcal{D}_c$, soit on privilégie la vitesse de comparaison et dans ce cas on stocke une clé d'arbre pour chaque sommet de $\mathcal{U}$ dans une composante fortement connexe. Soit m le nombre d'arêtes de G, n son nombre de sommets et m' et n' son nombre d'arêtes et de sommets après minimilisation. La minimilisation a un coût $O(m \log n)$, la construction d'une clé de cycle un coût $O(m')$ et la comparaison de deux clés de cycles se fait en $O(1)$ grâce au hash-consing. Construire toutes les clés d'un graphe coûte $O(m'n')$. Quand on cherche à économiser de l'espace mémoire, ce coût quadratique est payé à chaque test d'instinguabilité et sinon il est payé à chaque fois qu'on rencontre un nouveau graphe distinguable de $\mathcal{U}$.

Alain Frisch, dans l'implémentation de son module `Recursive`[3] pour Ocaml propose une solution moins lourde : il hashe les cycles jusqu'à une profondeur bornée et ensuite compare le sommet avec tous les sommets de même valeur de hash. La comparaison s'effectue en utilisant

[3]Module publié sur la toile

```
globale M

fonction EQU(S1, S2)
 si {S1, S2} ∈ M alors renvoyer vrai
 sinon si étiquette(S1) = étiquette(S2)
  et arêtes(S1) = arêtes(S2)
 alors { M ← M ∪ {S1, S2}
       { pour chaque ℓ ∈ arêtes(S1)
       {   faire { si non EQU(S1.ℓ, S2.ℓ)
       {         { alors renvoyer faux
       { renvoyer vrai
 sinon renvoyer faux

fonction ÉQUIVALENCE(S1, S2)
 M ← ∅
 renvoyer EQU(S1, S2)
```

FIG. 2.3 – Algorithme d'équivalence naïf

l'algoritme d'équivalence naïf (figure 2.3), qui bien qu'étant de complexité quadratique dans le cas le pire se comporte souvent assez bien. Alain Frisch rajoute aussi au test des étiquettes des sommets S_1 et S_2 un test d'égalité de hashage à profondeur bornée, ce qui permet à l'algorithme de conclure plus vite quand les sommets sont distinguables. Sur des exemples aléatoires, il semble que cette technique se comporte moins bien que la technique utilisant la minimisation en $n \log n$ que j'ai développée (cf section A.1), peut-être parce que sur ces exemples, le coût quadratique est souvent atteint, malgré le hashage.

Il reste que la multiplication des clés de cycle était un problème, à la fois en temps et en mémoire. Une seule clé de cycle par graphe fortement connexe devrait suffire si on est capable pour tout cycle de choisir un sommet distingué invariant. Le problème consiste à toujours choisir le même sommet quelles que soient les instances du graphe, et en particulier indépendamment du sommet par lequel on commence à parcourir le graphe. Une solution consistant à trier les sommets a été proposée par Considine (2000), mais le coût était quadratique, donc le gain par rapport à la comparaison de toutes les clés de cycle était au mieux faible. Je me suis aperçu depuis qu'il était possible de faire ce tri en $O(n \log n)$, simplement en utilisant l'algorithme de minimisation. En effet, l'algorithme de minimisation manipule principalement une liste de partitions et il est entièrement déterministe sur ces partitions. Il suffit donc de choisir un ordre de départ de ces partitions indépendant du graphe. Comme la partition se fait en fonction des étiquettes des sommets, il suffit donc

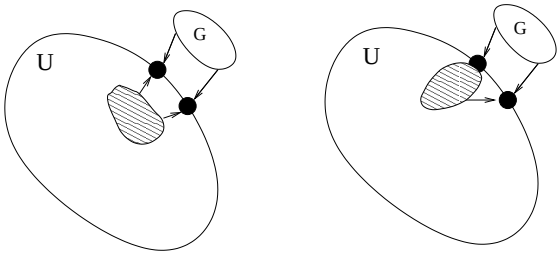

FIG. 2.4 – Les deux cas de partage du graphe G

de trier un ensemble d'étiquettes de sommets, puis de lancer la minimisation sur le graphe en ordonnant la partition de départ en fonction des étiquettes de sommets. On note quelle est la première partition, puis quand on la divise, on choisit de garder en premier cette partition. Ensuite on ne construit que la clé de cycle partant du sommet représentant de la première partition.

2.2.4 Clés partielles pour les graphes quelconques

Finalement, sur un graphe quelconque, on peut diminuer la complexité en parcourant le graphe en partant d'un sommet de manière analogue au parcours de bas en haut d'un arbre. On utilise tout simplement un parcours en profondeur (Tarjan, 1972) qui détermine les composantes fortement connexes. Si le parcours isole une composante fortement connexe sans arête sortante, on peut utiliser simplement l'algorithme de la section 2.2.3. Si la composante fortement G connexe a des arcs sortants, on peut faire le partage incrémental de tous les sommets accessibles depuis G (puisqu'aucun arc ne revient ensuite dans G, par construction des composantes fortement connexes). On a alors que soit chaque sommet de G est indistinguable d'un sommet de $\mathcal{U}$, soit aucun ne l'est. Si chaque sommet de G est indistinguable d'un sommet de $\mathcal{U}$, deux cas peuvent se présenter (figure 2.4) selon qu'au moins un des arcs sortants arrive sur un sommet de $\mathcal{U}$ indistinguable d'un sommet de G ou non.

Lorsqu'on ajoute un nouvelle composante fortement connexe dans $\mathcal{U}$, forcément aucun arc sortant ne mène à un sommet indistinguable d'un sommet de la composante. On choisit donc d'associer à chaque sommet, non pas son étiquette, mais son étiquette enrichie d'un ensemble de couples étiquette d'arête × sommet destination de l'arête, pour les sommets destinations qui sont déjà dans $\mathcal{U}$. Il s'agit en fait d'une clé des fils partielle. En ôtant temporaire-

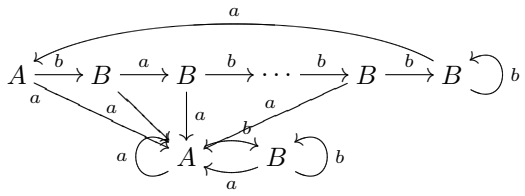

FIG. 2.5 – Un exemple de graphe avec arc sortant menant sur un sommet indistinguable de $\mathcal{U}$

ment les arêtes menant à des sommets de $\mathcal{U}$, on est ramené au cas du graphe fortement connexe sans arête sortante, qu'on peut ensuite minimiser et dont on peut ensuite calculer la clé de cycle (dont les étiquettes sont maintenant des clés partielles). Si on retombe sur un sommet indistinguable de cette composante, dans un graphe fortement connexe G en dehors de $\mathcal{U}$, on obtiendra la même clé de cycle *à condition qu'aucun arc sortant de G ne soit un sommet indistinguable d'un sommet de G* (cf figure 2.5).

Pour isoler les composantes fortement connexes et utiliser le méchanisme de clé de cycle, il a donc fallu que je détecte les cas où un arc sortant de G mène à un sommet de $\mathcal{U}$ indistinguable d'un sommet de G. Cela peut se faire très efficacement (bien que de complexité quadratique dans le cas le pire) en observant quelques propriétés :

1. Si deux arcs sortants de G mènent à deux composantes distinctes de $\mathcal{U}$, seuls des sommets de la composante qui a été ajoutée en dernier à $\mathcal{U}$ peuvent être indistinguables de G.
2. Si S est un sommet de G tel que $S.\ell$ est un sommet de $\mathcal{U}$ de date d'ajout minimale (parmi les sommets accessibles depuis G), alors il suffit de tester si S est indistinguable de tous les sommets S' de $\mathcal{U}$ accessibles depuis $S.\ell$ et tels que $S'.\ell = s.\ell$.
3. La comparaison d'un sommet de $\mathcal{U}$ avec un sommet de G peut se faire en temps $O(n)$ où n est la taille de G.

L'algorithme que j'ai développé consiste donc à associer à chaque sommet une date d'entrée dans $\mathcal{U}$ (la même pour tous les sommets d'une même composante fortement connexe), et pour chaque nouvelle composante fortement connexe G qui se présente, chercher le sommet de $\mathcal{U}$ directement accessible depuis G de plus grande date, puis rechercher dans sa composante les sommets qui pointent vers lui avec la même étiquette d'arête, puis faire le test d'équivalence linéaire.

Les tests (sect A.1) montrent que cet algorithme est efficace, puisqu'en plus de la mémoire il apporte un gain de temps notable par rapport à l'utilisation de graphes sans partage. Mais le partage n'est pas gratuit, et il semble aussi préférable de ne partager que quand cela semble être utile, c'est-à-dire si on pense devoir tester des égalités avec d'autres graphes ou encore si on utilise ces graphes dans une séquence d'itérés pour avoir un élargissement efficace.

2.3 Application aux automates de mots

En analyse statique par interprétation abstraite, on cherche généralement à représenter des ensembles de valeurs. Une des manières les plus communes de représenter un ensemble de valeurs par un graphe est l'utilisation d'un automate de mots.

2.3.1 Indistinguabilité et langage reconnu

L'utilisation des techniques de partage maximal de la section 2.2 est immédiate dès que les automates à représenter sont d'une part déterministes et d'autre part soit complets (c'est-à-dire que toute transition à partir d'un état quelconque est valide, elle mène à un nouvel état), soit utile (c'est-à-dire que l'automate ne contient que des états pouvant mener à un état final). On peut noter qu'aucune de ces trois propriétés ne restreint le pouvoir expressif : tout langage reconnaissable par un automate peut être reconnu par un automate déterministe, un automate complet et un automate utile (Hopcroft and Ullman, 1979; Watson, 1993).

Le caractère complet ou utile de l'automate permet d'utiliser la même notion d'indistinguabilité pour le graphe et pour le langage reconnu par un état de l'automate. Le langage reconnu par un état est l'ensemble des chemins partant de cet état et aboutissant à un état étiqueté final. Dans le cas général, on pourrait avoir deux états qui reconnaissent le même langage mais qui soient distinguables si un chemin ne pouvant pas mener ultimement à un état final est présent à partir d'un état et pas de l'autre. Ce cas est impossible pour un automate complet car tous les chemins mènent alors à au moins un état et pour un automate utile car dans ce cas tout chemin doit mener ultimement à un état final. Le caractère déterministe est nécessaire pour pouvoir utiliser les clés de cycles.

2.3.2 Utilisation dans une analyse statique

Sous ma direction, Sébastien Villemot (Villemot, 2002) a implémenté une bibliothèque d'automates déterministes complets. Parmi les très nombreuses applications des automates de mots en analyse statique, il a choisit d'implémenter l'analyse de protocoles de communication par canaux FIFO. Le principe de l'analyse est de calculer l'ensemble des queues de messages possibles sur les canaux d'un ensemble de processus. L'ensemble des canaux ne change pas au cours des calculs. L'intérêt principal de cette analyse était l'existence d'une analyse utilisant des automates (des QDDs (Boigelot and Godefroid, 1996) pour représenter des ensembles relationnels de tuples de mots), implémentée et disponible. La bibliothèque utilisée pour la comparaison est celle de l'outil LASH développé par Bernard Boigelot (Boigelot, 1999). Les résultats expérimentaux (voir section A.2) semblent confirmer que dans le contexte d'une analyse statique, les techniques développées dans ce chapitre peuvent donner de très bons résultats.

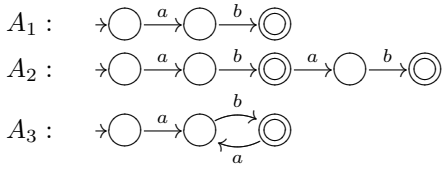

FIG. 2.6 – Repliage : l'automate A_3 est un repliage de A_2 sur A_1

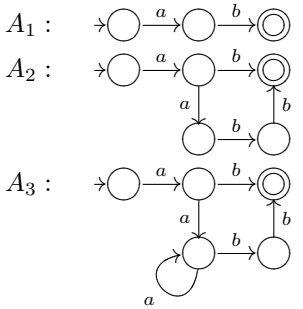

FIG. 2.7 – Extrapolation de chemin (A_3 est l'extrapolé de A_1 et A_2)

2.3.3 Élargissements sur les automates de mots

Les élargissements permettent d'accélérer la convergence des itérés et sont un des points essentiels de l'analyse statique par interprétation abstraite. Sur les automates de mots, comme sur la plupart des représentations utilisant des graphes, on peut utiliser trois grands types d'approximations de l'union (Cousot and Cousot, 1992) utiles pour construire un élargissement (par exemple, le repliage a été utilisé dans Bouajjani and Touili (2002)) :

Le repliage, qui consiste à prendre deux itérés successifs, à les parcourir en parallèle et trouver les points où le deuxième itéré rajoute un sommet puis essayer de remplacer ce sommet par un sommet déjà existant dans le premier itéré. Dans le cas des automates de mot, cela ne peut se faire de manière sûre que si on trouve un état dont le langage est plus grand que le langage de l'état qu'on cherche à remplacer. Si on fait une analogie avec les ensembles d'entiers, cela revient à chercher une congruence.

L'extrapolation de chemin, qui consiste cette fois à regarder les arêtes ajoutées au graphe. On va ajouter les chemins avec cette nouvelle arête répétée indéfiniment si ils ne sont pas déjà dans le nouvel itéré. Pour cela, on regarde le sommet S auquel mène cette arête et on suit les arêtes de même étiquette ℓ, jusqu'à arriver à un éventuel nouveau sommet T duquel aucune arête d'étiquette ℓ ne part. On rajoute une arête d'étiquette ℓ de T vers S, complétant ainsi les chemins se finissant en ℓ^k par ℓ^*. Si on fait une analogie avec les ensembles d'entiers, cela revient à élargir à l'infini.Dans le cas des automates de mots, cela permet d'approcher des séquences qui grossissent à l'intérieur du mot, comme $a^n b^n$.

La limitation de taille, qui consiste à utiliser un sommet $\top$ par lequel on remplace tous les nouveaux sommets si le deuxième itéré dépasse une taille critique donnée. Cet élargissement, déjà présent dans Mauborgne (1994), permet de ne pas tout perdre en cas de comportement d'explosion en mémoire. Ces explosions en mémoire se produisent souvent dans des cas rares, mais il faut tout de même être capable d'y faire face. Cette approximation peut toujours être utilisée et permet de plus d'assurer la terminaison si l'ensemble des étiquettes est fini, ce qui est la cas pour les automates de mots. Dans les automates de mots, l'élément $\top$ est un sommet S avec toutes les arêtes qui partent de S et qui arrivent en S.

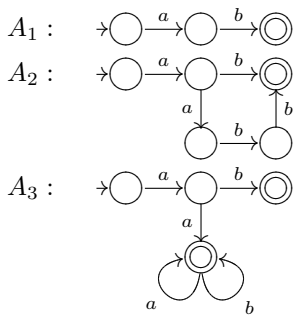

FIG. 2.8 – Limitation de taille (en supposant que l'alphabet soit $\{a, b\}$)

2.3.4 Et les arbres ?

Les résultats sont bons sur les automates de mots, mais ils ne sont hélas pas aussi facilement adaptables pour les automates d'arbres ou les automates de graphes. En effet, les automates d'arbres (Thatcher and Wright, 1968) utilisent des transitions depuis un tuple d'états vers un état (ou depuis un état vers un tuple d'état pour les automates de bas en haut). Les automates d'arbres sont donc des hypergraphes, classe de graphe qui n'est pas contenue dans les multigraphes de ce chapitre. Ils seront un peu abordés au chapitre suivant.

2.4 Application aux ensembles de graphes

Pour les automates de graphes, la situation est bien entendue encore plus complexe qu'avec les automates d'arbres. Malgré quelques tentatives de définition (Brandenburg and Skodinis, 2005), certains pensent qu'il n'est même pas possible de définir des automates de graphes satisfaisants (Courcelle, 1994). Tout dépend bien sûr de ce qu'on attend d'un automate de graphe. Dans le cadre de l'analyse statique par interprétation abstraite, on cherche à avoir une représentation d'ensembles de graphes, aussi expressive que possible, mais en admettant la possibilité de ne pas être algébriquement clos du moment que des opérations approchées sont possibles.

2.4.1 Têtes de graphes

Une des observations qui m'ont poussé à explorer les représentations des graphes, c'est qu'un ensemble fini de graphes est aussi un graphe, d'où l'idée d'utiliser la même structure pour le graphe et son ensemble. Le premier pro-

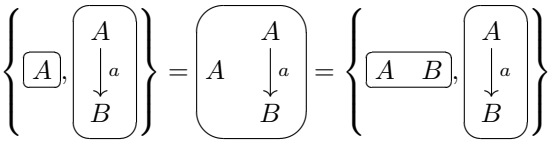

FIG. 2.9 – Deux ensembles de graphes équivalents au même graphe

blème qui se pose avec cette vision simple, c'est que si on représente un ensemble de graphes directement par le graphe composé de l'union des sommets et l'union des arêtes, on perd de l'information, surtout si les graphes de départ sont partagés. Le problème est encore plus évident si l'ensemble est constitué de graphes non connexes (voir figure 2.9). Cette perte d'informations correspond à l'abstraction classique d'un ensemble d'ensembles par l'union de ses éléments. Pour lever l'ambigüité, on peut rajouter aux étiquettes de sommets une étiquette « tête de graphe » (qu'on notera • dans les figures) et pour chaque graphe que l'on souhaite conserver utiliser un nouveau sommet avec cette étiquette et des arêtes partant de cette étiquette vers les sommets du graphe. On appelera ce nouveau sommet la tête du graphe. On ne peut pas choisir de prendre des arêtes des sommets vers la tête de graphe, car sinon cela change les propriétés du sommet en terme de dinstinguabilité à chaque fois qu'on le considère dans un nouveau graphe.

Avec les propriétés d'indistinguabilité, les ensembles de sommets que l'on stocke ont quelques propriétés intéressantes vis-à-vis de leur appartenance à un graphe donné. En effet, si un sommet S appartient à un graphe, alors toutes les arêtes qui en partent font partie de tous les graphes qui contiennent S. Sinon, ce serait un autre sommet, distinct de S mais de même étiquette et avec moins d'arêtes sortantes qui serait distinguable de S qui serait dans un tel graphe. Du coup, on n'est pas obligé d'utiliser une arête par sommet du graphe. On peut à la fois être beaucoup plus économe et garder une représentation unique pour un graphe donné, ce qui permettra un partage maximal. Les sommets d'un graphe vers lesquels on choisit de mettre des arêtes seront appelés les *représentants* du graphe. Un ensemble $\mathcal{S}$ de sommets *représente* le graphe (ou forme un ensemble de représentants du graphe) si l'ensemble des sommets du graphe est exactement l'ensemble des sommets accessibles depuis $\mathcal{S}$. Un sommet S est un représentant canonique de G si et seulement si il

n'existe pas d'arête de G menant à la composante fortement connexe de S et soit S est seul dans sa composante, soit S est le plus petit de sa composante, au sens de l'ordre obtenu par minimisation (cf section 2.2.3).

En général, on a donc plusieurs représentants pour un graphe, ce qui fait que les têtes de graphe peuvent avoir plusieurs arêtes, dont on n'a pas défini l'étiquette. Si on choisit la même étiquette τ pour toutes ces arêtes, alors le graphe constitué de tous les sommets représentés de manière unique, $\mathcal{U}$ ne sera plus déterministe, ce qui pose problème pour la minimisation et les clés de cycles, et même pour la notion de distinguabilité (selon la définition, les sommets $\bullet$ de $\overset{\tau}{\underset{A}{\swarrow}}\bullet\overset{\tau}{\underset{A\xrightarrow{a}B}{\searrow}}$ et $\overset{\bullet}{\underset{A\xrightarrow{a}B}{\downarrow\tau}}$ sont indistinguables). Heureusement, ces sommets n'apparaissent jamais dans un cycle, on ne leur cherche donc jamais de clé de cycle et on ne fait jamais de minimisation avec des têtes de graphes. Il suffit donc de faire une légère entorse à l'indistinguabilité pour ces sommets là et d'utiliser des clés de sommet un peu différentes des clés de sommet de la section 2.2.2 : au lieu d'un couple étiquette $\times$ liste de couples étiquettes d'arêtes $\times$ sommets, on utilise le couple étiquette $\times$ ensemble de sommets représentés de manière unique.

Pour avoir une représentation unique des ensembles finis de graphes quelconques, il suffit donc d'utiliser des listes triées de têtes de graphe, l'ordre utilisé dans les listes étant les dates d'entrée des têtes de graphe dans le graphe $\mathcal{U}$ des sommets représentés de manière unique. On peut aussi utiliser des arbres équilibrés avec hashage commutatif et associatif des sous-arbres.

2.4.2 Sommet de choix

Il peut sembler naturel de n'utiliser que des graphes à ensemble de sommets finis pour le domaine concret, car le programme à analyser ne peut manipuler que des ensembles finis de valeurs. Dans certains cas, il peut être utile de considérer une propriété infinie d'un programme, comme un comportement temporel d'un programme qui ne termine pas, même si cela complique beaucoup les choses. Mais dans tous les cas, il est impératif de pouvoir représenter des ensembles infinis de graphes, car l'analyse statique doit pouvoir représenter toutes les valeurs du programme à la fois, et même si ces valeurs sont finies, les ensembles sont toujours très gros et il est plus efficace de les approximer par des ensembles infinis. On ne peut donc pas simplement utiliser la représentation pour un graphe et

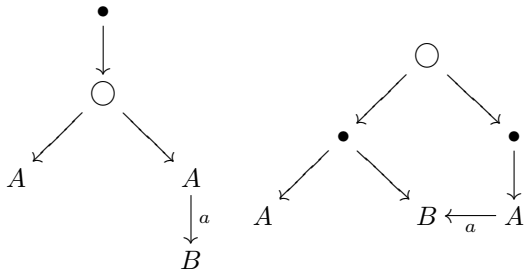

FIG. 2.10 – Représentation des deux ensembles de la figure 2.9 avec des têtes de graphes et sommets de choix

utiliser des représentations d'ensemble classique, comme des arbres équilibrés ou tas de têtes de graphes.

Pour représenter des ensembles infinis de graphes, on va utiliser (comme dans Mauborgne (2000a)) une nouvelle étiquette de sommet, l'étiquette de choix $\bigcirc$, et permettre à ces sommets d'appartenir à des cycles. Pour une plus grande expressivité, il faudra aussi autoriser des ensembles de graphes dont le nombre de composantes n'est pas borné. Pour cela, on va coder une tête de graphe à n représentants par un arbre binaire. Pour pouvoir appliquer les techniques de partage maximal, je n'ai pas trouvé de technique permettant de garder une représentation unique d'un ensemble de représentant. La nouvelle structure représentera des ensembles de listes de représentants, sachant qu'à chaque graphe on peut associer une unique liste de représentant (les représentants canoniques ordonnés selon l'ordre d'entrée dans $\mathcal{U}$ par exemple), mais que plusieurs listes de représentants peuvent représenter le même graphe. On dira qu'un sommet S est un encodage du sommet $\overset{\bullet}{\underset{S_0\cdots S_{n-1}}{\swarrow\ \searrow}}$ si, soit $n = 0$ et S est $\overset{\bullet}{\underset{S_0}{\downarrow 0}}$, soit $n = 1$ et S est $\overset{0}{\underset{S_0}{\swarrow}}\bullet\overset{1}{\underset{S_1}{\searrow}}$ soit $n > 1$ et S est $\overset{0}{\underset{S_0}{\swarrow}}\bullet\overset{1}{\underset{S'}{\searrow}}$ et S' est un encodage de $\overset{\bullet}{\underset{S_0\cdots S_{n-2}}{\swarrow\ \searrow}}$. Les étiquettes 0 et 1 sont supposées être de nouvelles étiquettes d'arêtes.

Pour définir les graphes représentés par un graphe avec des sommets de choix, on définit l'égalité de chemins modulo choix : deux chemins sont égaux modulo choix si ils sont égaux après avoir enlevé toutes les étiquettes associées à des arêtes partant des sommets de choix. On dira qu'un sommet S est un choix valide pour un

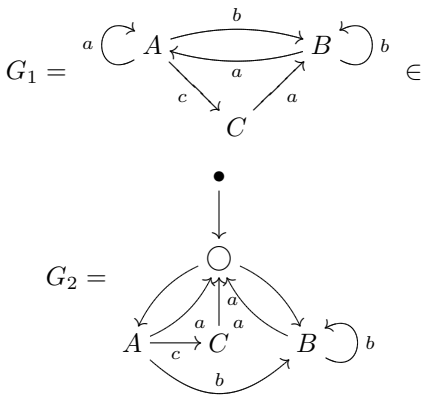

FIG. 2.11 – Le graphe G_1 appartient à l'ensemble des graphes représentés par G_2

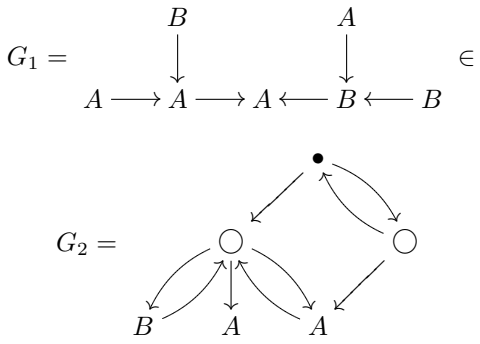

FIG. 2.12 – Le graphe G_1 appartient à l'ensemble des graphes représentés par G_2

sommet S' si pour tout chemin p tel que $S.p$ ne soit pas un choix, il existe un chemin p' égal à p modulo choix tel que $S.p$ et $S'.p'$ aient le même ensemble d'étiquettes. L'ensemble des graphes représentés par un sommet S d'un graphe G étiqueté par $L \cup \{\bullet, \bigcirc\}$ est l'ensemble des graphes G' étiquetés par L tels qu'il existe un sommet T qui soit un choix valide de S et qui soit un encodage de $\overset{\bullet}{\underset{S_0 \cdots S_{n-1}}{\swarrow \; \searrow}}$ et G' est représenté par les S_i. Le cas fini est illustré par la figure 2.10, et le cas infini par les figures 2.11 et 2.12.

2.4.3 Squelettes de graphes

L'ajout de ces sommets de choix pose bien entendu des problèmes, et on ne peut pas les utiliser n'importe où sans perdre très rapidement les propriétés d'unicité qui permettent une implémentation efficace. Le premier problème que j'avais déjà évoqué pour les têtes de graphe est celui du déterminisme si on n'étiquette pas les arêtes partant des choix. Et contrairement au têtes de graphes, le problème ne peut pas être évacué car les choix doivent pouvoir appartenir à des cycles pour représenter des ensembles infinis. D'autre part, pour garder la représentation finie, il faut limiter le nombre d'arêtes sortantes des choix, au moins pour un ensemble donné. J'ai donc choisi d'étiqueter les arêtes sortant d'un choix par le couple étiquette du sommet destination × multi-ensemble des étiquettes d'arêtes partant de ce sommet destination. Cette définition est potentiellement récursive dès qu'une arête relie deux choix. Comme on veut garder les étiquettes d'arêtes représentables et que relier deux choix par un arête n'est pas très intéressant (cela revient à décomposer une union en une union d'union(s)), on va donc interdire que les arêtes relient deux choix. Si de plus on impose le déterminisme, cela revient à imposer que si deux arêtes sortant d'un choix mènent à deux sommets de même étiquette, alors ces étiquettes ne sont pas $\bigcirc$ et les ensembles d'étiquettes d'arêtes sortant de ces sommets sont différents.

Pour assurer un partage maximal des ensembles, il faut que le graphe représentant ensemble donné soit défini de manière unique (modulo l'indistinguabilité de ses sommets). Il reste encore deux causes de non-unicité : un sommet de choix dont ne part qu'une arête est inutile, et un sommet de choix dont ne part aucune arête correspond à un ensemble vide. On en déduit une définition d'automates de graphe que nous appellerons des squelettes de graphe par :

DÃ©finition 2.1 *On appelle squelette de graphe d'étiquettes de sommet dans L tout sommet S d'un graphe dont les sommets sont étiquetés par L plus deux nouvelles étiquettes $\bullet$ et $\bigcirc$ tel que soit $S = \bigcirc$ soit*

- *pour tout chemin p tel que l'étiquette de $S.p$ est $\bullet$, p est égal à un chemin composé uniquement de 1 (l'étiquette de droite de $\bullet$) modulo choix.*
- *et pour tout sommet $S.p$ étiqueté par $\bigcirc$,*
  - *il existe au moins deux arêtes partant de $S.p$,*
  - *et si deux de ces arêtes mènent à des sommets de même étiquette, alors ces sommets n'ont pas le même ensemble d'étiquettes d'arêtes sortantes,*
  - *et aucun de ces sommets n'a pour étiquette $\bigcirc$.*

La preuve de l'unicité de cette représentation est la même que dans Mauborgne (2000a). Il faut simplement noter que la représentation est unique en tant que représentation d'ensembles listes de sommets de graphes, et non en tant que

représentation d'ensembles de graphes. Comme on le verra dans la section 2.5, la difficulté pour représenter des ensembles de graphes est la même que pour représenter des hypergraphes non orientés, ce qui est plutôt difficile...

2.4.4 Opérations algébriques sur les squelettes de graphes

Les squelettes de graphes sont faits de telle façon qu'il est possible de décider l'appartenance d'un graphe à l'ensemble de manière déterministe, car chaque arête à prendre parmi celles qui sortent d'un choix est déterminé de manière unique par l'étiquette du sommet de graphe qu'on est en train de lire et les étiquettes de arêtes sortant de ce sommet. Grâce à cela, on peut faire des algorithmes très efficaces sur les squelettes de graphes. Le schéma de tous les algorithmes pour calculer une fonction f sur les ensembles de graphes consiste à parcourir les arguments de manière synchrone en gardant en mémoire les tuples de sommets déjà rencontrés. Ces algorithmes ont donc une complexité de l'ordre du produit des tailles de leurs arguments.

Un exemple typique est l'intersection de deux ensembles : on associe à chaque couple de sommet rencontré un nouveau sommet, et en cas de sommet de choix, on ne gardera que les choix communs aux deux squelettes (cf figure 2.13. Cet algorithme sera exact. S'il n'est pas trop difficile de voir que le résultat de l'algorithme représentera bien l'intersection des ensembles de graphes représentés par ses arguments, il est peut-être un peu plus difficile de voir que le résultat est un squelette conforme à la définition 2.1. Les points difficiles à obtenir sont ceux concernant les sommets de choix. La procédure PROPAGERVIDE permet d'éliminer tout sommet étiqueté par ◯ sans arête sortante. L'utilisation de CONSTRUIRECHOIX permet d'éviter aussi les choix avec une seule sortie. Par construction des ensembles de choix, tous les sommets sortant d'un choix ont des étiquettes différentes. Enfin, grâce à l'alternance des fonctions mutuellement récursives INTERCHOIX et INTERMÊME, on est assuré qu'aucun des sommets sortant d'un choix n'est un choix.

En revanche, l'union de deux ensembles représentés par des squelettes de graphes n'est pas en général représentable par un squelette. Mais dans le cadre de l'interprétation abstraite, ce n'est pas irrémédiable, et on peut noter que dans la plupart des domaines abstraits, l'union est approchée. Dans le cas des squelettes de graphes, nous avons un algorithme qui donne la meilleure

globale M, P
fonction INTER(S_1, S_2)
 $M \leftarrow \emptyset, \quad P \leftarrow \emptyset$
 renvoyer INTERCHOIX(S_1, S_2)

fonction CHOIX(S)
 si $S =$ ◯ (↙ … ↘ $S_0 \cdots S_n$) **alors renvoyer** $\{S_i \mid i \leq n\}$
 renvoyer $\{S\}$

fonction INTERCHOIX(S_1, S_2)
 si $S_2 = S_2$ **alors renvoyer** S_1
 si $\{S_1, S_2\} \in dom(M)$
 alors renvoyer $M(\{S_1, S_2\})$
 soit S un nouveau sommet
 $M \leftarrow M \cup \{\{S_1, S_2\} \rightarrow S\}$
 soit $C_i =$ CHOIX(S_i)
 soit $R = \emptyset$
 pour chaque $T \in C_1$
  faire { **si** $\exists U \in C_2$ de mêmes étiquettes
  alors { **soit** $V =$ INTERMÊME(T, U)
  si $V \neq$ ◯
  alors { $R \leftarrow R \cup \{V\}$
  $P \leftarrow P \cup \{V \rightarrow S\}$
 CONSTRUIRECHOIX(S, R)
 renvoyer S

fonction INTERMÊME(S_1, S_2)
 S_1 et S_2 ont mêmes étiquettes de sommets et d'arêtes
 soit S un nouveau sommet de même étiquette que S_1
 pour chaque ℓ étiquette d'arête de S_1
  faire { $S.\ell \leftarrow$ INTERCHOIX$(S_1.\ell, S_2.\ell)$
  si $S.\ell =$ ◯ **alors renvoyer** ◯
 renvoyer S

procédure CONSTRUIRECHOIX(S, R)
 enlever les arêtes partant de S
 si $R = \emptyset$ **alors** PROPAGERVIDE(S)
 sinon si $R = \{V\}$
  alors { étiqueter S comme V
  pour chaque arête partant de V
  faire ajouter la même arête partant de S
  sinon { étiqueter S par ◯
  ajouter les arêtes partant de S vers
  chaque sommet de R

procédure PROPAGERVIDE(S)
 étiqueter S par ◯
 pour chaque $T \in P(S)$
  faire { **soit** $C =$ CHOIX(T)
  si $C = \{U\}$ **alors** PROPAGERVIDE(T)
  sinon si $C \neq \emptyset$
  alors CONSTRUIRECHOIX$(T, C \backslash \{S\})$

FIG. 2.13 – Intersection de deux squelettes

approximation possible de l'union : en suivant pratiquement le même algorithme que pour l'intersection, simplement en gardant les sommets sortant des choix qui ne sont que dans un graphe et en prenant l'union de S et $\bigcirc$ comme S. L'approximation inévitable peut arriver lorsqu'on a dans les deux graphes un sommet avec deux arêtes sortantes, car on peut perdre la relation entre les sommets vers lesquels aboutit la première arête et ceux vers lesquels aboutit la seconde arête, selon qu'ils viennent du premier ou du second graphe. Formellement, si E_i est l'ensemble des sommets accessibles depuis la première arête dans le graphe i et F_i l'ensemble des sommets accessibles depuis la seconde arête, la relation pour l'union est $\{(E_1, F_1), (E_2, F_2)\}$; elle sera approximée par $\{(E_1 \cup E_2, F_1 \cup F_2)\}$. On peut noter que cet algorithme sera un peu plus simple que l'intersection car il n'y aura pas de vide à propager.

Il est aussi possible d'approximer la différence de deux squelettes. Ce sera toujours un algorithme semblable à l'intersection, mais les opérations de base seront :

- la différence d'un sommet S avec un sommet T d'étiquette différente sera S,
- la différence d'un sommet S avec un sommet T de même étiquette et ensemble d'étiquettes d'arêtes $\mathcal{L}$ sera
  - $\bigcirc$ si toutes les différences de $(S.\ell, T.\ell)$ sont $\bigcirc$,
  - si toutes les différences sauf un $(S.\ell, T.\ell)$ sont $\bigcirc$, alors la différence de S avec T sera S dans lequel on aura remplacé l'arête d'étiquette ℓ par une arête d'étiquette ℓ menant à la différence de $S.\ell$ et $T.\ell$;
  - sinon la meilleure approximation possible est S
- et pour les choix, on garde tous les choix de S qui ne sont pas dans T, et on applique la différence aux choix qui sont à la fois dans S et T.

2.4.5 Méta-expressions

Pour faire des analyses de programmes, on ne peut pas se limiter aux opérations algébriques classiques. La plupart du temps, les propriétés à inférer sur les programmes vont pouvoir se construire à partir d'opérations de point fixes sur des fonctions croissantes. Cousot and Cousot (1995) proposent de décrire de manière très générique ce processus par l'utilisation de méta-expressions et de transformations de langages formels. Leur formalisme définit les méta-

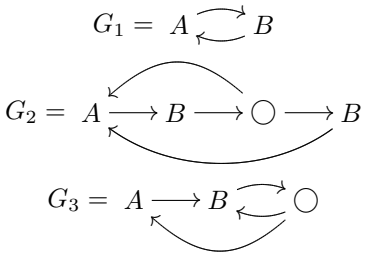

FIG. 2.14 – Repliage : G_3 est le replié de G_2 sur G_1

expressions par la grammaire :

$$e ::= \quad \mathcal{X} \quad | \quad \left\{G' : (G_i \in e_i)_{i<n}\right\} \quad | \quad e_1 \cup e_2$$

où les G sont des termes (arbres avec variables) et les $\mathcal{X}$ sont des variables de langages. Les propriétés de programmes seront alors approchées par des systèmes d'équations de points fixes définissant les valeurs d'un ensemble fini de variables de langages. Si on utilise des ensembles de graphes dans une analyse, on peut facilement étendre ce formalisme en prenant des graphes avec variables au lieu des termes. La sémantique d'un graphe avec variables G sera donnée par la fonction $[\![G]\!] \in v \to \mathcal{G} \to \mathcal{G}$ qui remplace chaque sommet étiqueté par une variable par le représentant d'un graphe associé à la variable. On peut aussi étendre encore le formalisme en utilisant des vecteurs de variables et des graphes à plusieurs représentants, mais cela ne rajoute pas de pouvoir expressif dans un cadre de représentation avec partage des sommets indistinguables. La sémantique des méta-expressions est donnée par

$$\{\!|\mathcal{X}|\!\}\,\rho \quad \stackrel{\text{def}}{=} \quad \rho(\mathcal{X})$$

$$\left\{\!\left|\{G' : (G_i \in e_i)_{i<n}\}\right|\!\right\}\,\rho \stackrel{\text{def}}{=} \left\{ [\![G']\!]\,\kappa \;\middle|\; \kappa \in v \to \mathcal{G} \bigwedge_{1 \le i \le n} [\![G_i]\!]\,\kappa \in \{\!|e_i|\!\}\,\rho \right\}$$

$$\{\!|e_1 \cup e_2|\!\}\,\rho \quad \stackrel{\text{def}}{=} \quad \{\!|e_1|\!\}\,\rho \cup \{\!|e_2|\!\}\,\rho$$

Les méta-expressions permettent par exemple d'exprimer l'intersection $((\{x : x \in e_1, x \in e_2\})$ ou la projection $\left(\left\{x : \begin{smallmatrix} & f & \\ x & & y \end{smallmatrix} \in e\right\}\right)$.

Pour résoudre un système de méta-expressions, on peut calculer les itérés et utiliser des opérateurs d'élargissement pour assurer la convergence (cf les figures 2.14, 2.15 et 2.16 pour des exemples d'extrapolations pouvant servir à définir des élargissements). Mais pour certaines formes de systèmes de

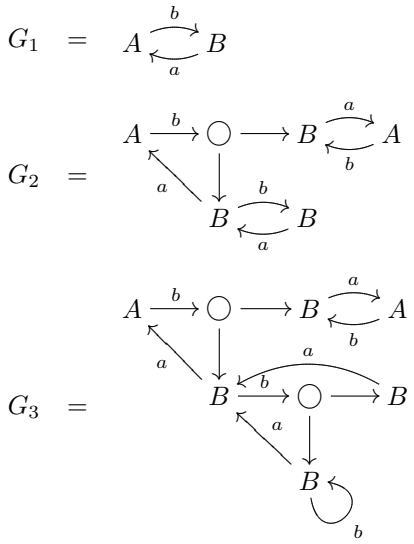

FIG. 2.15 – Extrapolation de chemins

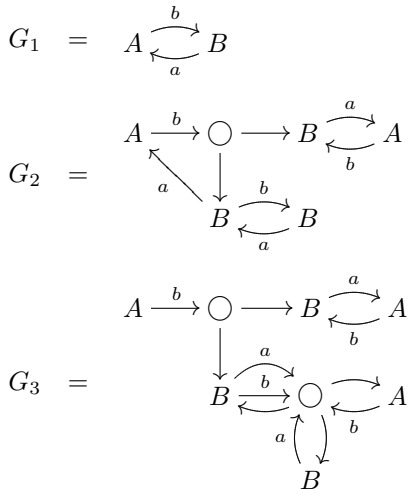

FIG. 2.16 – Limitation de taille

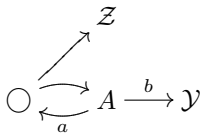

FIG. 2.17 – Un choix direct

méta-expressions, il est possible de calculer directement la meilleure approximation du point fixe. Ce genre de cas, très recherché en model-checking (Fribourg and Olsén, 1997; Abdulla et al., 1999) est aussi extrêment utile en interprétation abstraite (Feret, 2005b) pour avoir les résultats les plus précis permis par l'abstraction.

Un exemple typique qui montre la difficulté et pour lequel on peut calculer directement la meilleure approximation du point fixe est l'équation :

$$\mathcal{X} = \left\{ \begin{array}{c} A \\ {}^{a}\swarrow\ \searrow^{b} \\ x \quad y \end{array} : x \in \mathcal{X}, y \in \mathcal{Y} \right\} \cup \mathcal{Z}$$

Une façon simple d'obtenir un graphe qui représente l'ensemble des graphes solution de l'équation est de prendre un sommet de choix et de faire partir deux arêtes de ce sommet, une vers $\mathcal{Z}$ et l'autre vers $\begin{array}{c} A \\ {}^{a}\swarrow\ \searrow^{b} \\ x \quad \mathcal{Z} \end{array}$ en remplaçant x par le sommet de choix (cf figure 2.17). Cette construction pourrait être appliquée à tout système d'équation de méta-expressions dans lesquels les variables de graphes n'apparaissent qu'une seule fois chacune (sinon, il faudrait en plus un sommet d'intersection). Le problème, c'est que cette construction ne donne pas un squelette valide, dès que la représentation de $\mathcal{Z}$ commence par un choix ou par un sommet étiqueté par A. On pourrait donc avoir deux choix qui se suivent, et peut-être encore un sommet d'étiquette A parmi les choix de $\mathcal{Z}$. Cela signifie qu'on aurait une représentation non déterministe et donc sur laquelle les algorithmes seraient beaucoup moins efficaces, en plus de rendre le partage maximal pratiquement impossible. Par exemple, je ne sais pas faire mieux qu'un algorithme exponentiel pour tester l'inclusion dans le cas des graphes généraux.

Pour revenir au cas simple, on peut trouver directement le squelette. L'algorithme consiste à essayer la construction naïve, et à éliminer les sommets de choix qui se suivent en propageant les ensembles de graphes partagés. On maintient un ensemble sRec de sommets pour la partie qui doit reboucler et un sommet T pour la partie

à laquelle aboutit l'arête qui ne boucle pas (étiquetée par b dans l'exemple). Au départ, Srec est vide et T vaut le sommet de $\mathcal{Y}$. On parcours $\mathcal{Z}$, on appelle le sommet courant de ce parcours Z. On regarde l'ensemble des choix de Z (comme donné par la fonction CHOIX de la figure 2.13). Si cet ensemble ne contient pas de sommet étiqueté par A, alors on peut simplement construire le choix direct entre l'union des choix de Z et les sommets de sRec, et un sommet A avec une arête b vers T et une arête a qui reboucle sur le choix direct. Sinon, il faut continuer le parcours de Z pour partager le sommet d'étiquette A, S. On fait la même construction sur $S.a$ avec pour ensemble sRec l'union des sommets de sRec et des choix de Z autres que S, et pour sommet T l'union de T et de $S.b$. Si on appelle W le résultat de cet appel, on construit ensuite un sommet de choix parmi sRec et un sommet A dont l'arête b pointe sur T et l'arête a sur W. Il faut bien entendu mémoriser les arguments Z, sRec et T de cette fonctions, de manière à reboucler si on tombe à nouveau sur ces arguments, ce qui assure que le processus termine.

2.4.6 Retour aux automates classiques

Arbres

Puisqu'on a réussi à définir une représentation pour les ensembles de graphes, il est facile d'en tirer une représentation pour les ensembles d'arbres, car les arbres, tels qu'utilisés dans les automates d'arbres, peuvent être vus comme des graphes acycliques avec un seul représentant distingué, dont les sommets sont étiquetés dans un alphabet L, et les arêtes étiquetées dans $\mathbb{N}$ (on associe l'étiquette 0 au premier fils, 1 au deuxième, etc.). Comme les arbres n'ont toujours qu'un représentant, on pourra se passer des sommets étiquetés par $\bullet$ dans la représentation des ensembles d'arbres. La seule difficulté pour adapter les squelettes de graphes aux arbres est d'assurer l'unicité de la représentation alors que les choix valides cycliques ne doivent pas être pris en considération. Pour cela, il faut assurer que tout sommet accessible depuis le squelette est un squelette représentant un ensemble non vide d'arbres, car alors cela signifie que ce sommet est nécessaire à la représentation. En effet, il est facile de montrer que si deux squelettes distincts S_1 et S_2 représentent le même ensemble d'arbres, alors il existe un chemin p de S_1 (ou S_2) tel que $S_1.p$ ne représente que des graphes cycliques (prendre le plus petit p tel que $S_1.p$ et $S_2.p$ n'aient pas la même étiquette). Pour assurer de ne garder que les sommets avec cette propriété, il suffit de maintenir en cours de construction une information qui indique si un sommet représente au moins un arbre fini. Pour un sommet de choix, c'est le cas si au moins un des fils représente au moins un arbre, et pour un sommet quelconque, il faut que tous les fils représentent au moins un arbre. Cette information peut être maintenue de manière incrémentale avec un coût linéaire.

Puisqu'il existe des automates d'arbres, la question se pose de savoir comment cette représentation peut se comparer aux automates d'arbres classiques. Concernant le pouvoir expressif, on peut montrer que celui des squelettes restreints aux arbres est exactement le même que celui des automates déterministes de haut en bas (Gécseg and Steinby, 1984). En effet, si on prend un squelette, on peut lui associer l'automate dont les états sont les sommets du squelette et les transitions sont définies pour les arêtes (S, T) telles que T ne soit pas un choix. Soit $T = \overset{a}{\underset{T_0 \cdots T_{n-1}}{\swarrow \searrow}}$, la transition est alors $(S, a) \to (T_i)_{i<n}$. Si la racine du squelette est un choix, c'est l'état initial, sinon on rajoute un état initial et une transition de cet état vers les fils de la racine en lisant l'étiquette de la racine. Les états finaux sont les sommets sans arête sortante.

Réciproquement, à partir d'un automate déterministe de haut en bas, on définit le squelette dont les sommets sont les (q, a) tels qu'il existe un $(q, a) \to \ldots$ dans le système de transition et le langage défini par q n'est pas vide, plus un sommet de choix par état q tel qu'il existe plusieurs (q, a) dans le système de transition de l'automate. On étiquette les sommets (q, a) par a, et on ajoute une étiquette de q vers tous les (q, a) quand le sommet q est défini, et une étiquette de (q, a) vers chaque q_i de la transition associée à (q, a). Le sommet du squelette est soit l'état initial I si il définit un sommet, soit le seul (I, a) qui définit un sommet sinon.

Pour comparer les automates d'arbres en pratique avec les squelettes restreints aux arbres, on pourra se réferer aux expérimentations de la section A.3.

Mots

Toujours selon le même principe, on peut aussi utiliser les squelettes pour représenter les ensembles de mots. L'idée a d'ailleurs déjà été proposée pour les grammaires de graphes Janssens

and Rozenberg (1981); Engelfriet and Heyker (1991). Encore une fois, en utilisant la même construction que pour les arbres, on peut utiliser un squelette pour représenter tout automate de mots déterministe et réciproquement. Mais cette fois, il n'y aurait aucun intérêt à le faire. En effet, par rapport à une utilisation directe des techniques de représentation avec partage (puisque l'indistinguabilité du partage correspond exactement à l'égalité des langages générés), il faudrait utiliser d'une part des sommets de choix qui imposent de nouvelles règles de normalisation et d'autre part éliminer les parties du squelette qui ne produisent que des graphes cycliques. Cela entrainerait un surcoût certain, auquel je ne vois pas d'intérêt.

2.5 Extension aux hypergraphes

Les hypergraphes (Chvatal, 1971) sont une extension naturelle des graphes sur lesquelles on peut généraliser beaucoup de propriété des graphes. En effet, si un graphe (non orienté) est un couple ensemble de sommets, ensemble d'arête où les arêtes sont des ensembles de deux sommets, un hypergraphe étend simplement la notion d'arêtes aux ensembles quelconques de sommets. Cela signifie que les hypergraphes non orientés sont une autre définition pour les ensembles d'ensembles finis. Un ensemble de graphes peut donc être vu comme un hypergraphe non orienté particulier, mais il me semble délicat de les utiliser directement comme représentation des ensembles de graphes, à cause de la non-orientation, qui rend toujours les tests d'équivalence et donc la minimisation, plus difficile.

Un hypergraphe orienté est un ensemble fini de sommets et un ensemble de tuples sur ces sommets. On peut noter que dans le cas des hypergraphes non orientés, les arêtes avaient au plus autant d'éléments que la taille de l'ensemble des sommets, alors que ce n'est plus le cas ici, un tuple pouvant contenir le même élément en plusieurs exemplaires. On peut étendre les multigraphes de cette section en remplaçant simplement la définition d'arête comme un couple par un tuple de longueur quelconque. La notion de déterminisme devient alors : pour tous sommets $S_0, \ldots, S_{n-1}$ et toute étiquette d'arête ℓ, il existe au plus une arête $(S_0, \ldots, S_{n-1}, T)$ d'étiquette ℓ dans l'hypergraphe. Si on restreint la taille des couples d'un hypergraphe et si on n'utilise que deux étiquettes pour les sommets, alors l'hypergraphe est équivalent à un automate d'arbres de bas en haut. Dans le cas des graphes, on avait équivalence avec les automates de mots. On doit donc pouvoir étendre les résultats sur les automates d'arbres, en particulier la minimalisation, de la même façon qu'avec les automates de mots pour pouvoir manipuler efficacement et de manière incrémentale les hypergraphes. C'est l'objet d'une recherche en cours qui me semble prometteuse.

Il est possible d'encoder un hypergraphe orienté par un graphe orienté : on garde l'ensemble des sommets, et on code chaque arête $(S_0, \ldots, S_n)$ d'étiquette ℓ par n arêtes (S_i, S_{i+1}) d'étiquette (ℓ, i). Le problème de ce codage, c'est qu'il ne préserve pas les propriétés fondamentales des hypergraphes utilisées pour le partage : un hypergraphe déterministe ne sera pas forcément codé par un graphe déterministe, et surtout deux sommets indistinguables dans un hypergraphe peuvent être distinguables dans leur codage. En effet, pour que deux sommets T et U soient distinguables dans un hypergraphe, il faut trouver un arbre d'arêtes t tel que $T.t$ et $U.p$ aient une étiquette différente, *avec les mêmes sommets feuilles*. Cette condition n'est pas exigée dans le cas des graphes, puisqu'alors les seules feuilles possibles sont T et U.

Les problèmes qu'il me reste à résoudre concernent principalement la complexité de la minimalisation et l'extension des clé de cycle. Concernant la minimalisation, on trouve facilement des résultats à la Myhill Nerode sur les automates d'arbres (Kozen, 1992), mais je n'ai trouvé qu'une seule implémentation des automates d'arbres qui propose une minimalisation des automates. Il s'agit des automates d'arbres guidés du package MONA (Biehl et al., 1997). Leur minimalisation semble de complexité quadratique et je n'ai trouvé aucun résultat permettant une complexité en $n \log n$ comme pour les automates de mots. En revanche, on sait (Seidl, 1990) que décider l'équivalence de deux automates d'arbres est dans EXPTIME, ce qui donne une borne inférieure pour la minimalisaton des automates d'arbres non déterministes, car il suffit de minimiser l'union de deux automates pour savoir si ils sont équivalents. La complexité du même problème est en $n \log n$ pour les utomates de mots.

Si on enlève la restriction de la taille des arêtes, pour pouvoir représenter des ensembles de graphes de taille non bornée par exemple, une sous-classe existe qui permet de représenter certains hypergraphes, les automates d'arbres sans arité (Thatcher, 1967). Mais dans ce cas, la mi-

nimalisation est NP-complète, même en partant d'automates déterministes (Martens and Niehren, 2005).

Les résultats attendus d'une représentation efficace des hypergraphes permettraient une implémentation plus efficace des automates d'arbres et une extension plus puissante des ensembles de graphes, peut-être en suivant la même démarche que dans Mauborgne (2000c).

Chapitre 3

Relations symboliques

Dans son cadre le plus général, l'analyse statique par interprétation abstraite consiste à trouver un ensemble de comportements, appelé encore propriété, qui contient l'ensemble de tous les comportements possibles d'un programme. Mais bien souvent, les comportements ont une structure. On peut généralement les décomposer en famille de valeurs $(E_i)_{i\in I}$, correspondant par exemple aux points de contrôle du programme (Cousot, 1981), ou aux cellules mémoire (Miné, 2006a). Informellement, une analyse sera relationnelle si elle est capable de trouver des propriétés exprimant des relations entre les familles de valeurs des comportements. Une analyse relationnelle sera souvent nécessaire pour prouver des propriétés précises sur les programmes, mais elle sera en générale bien plus difficile et coûteuse qu'une analyse non relationnelle.

Généralement, il est préférable d'utiliser des propriétés algébriques ou géométriques des relations (par exemple la convexité, comme pour les polyèdres (Cousot and Halbwachs, 1978) ou les octogones (Miné, 2001), ou encore la relation affine (Karr, 1976) ou la relation de congruence (Granger, 1989)). Mais il arrive que les valeurs n'aient pas de structure numérique ou encore qu'on ne puisse savoir à l'avance la structure que devraient prendre les relations en cours de calcul. Dans ce cas, on utilise des relations symboliques, souvent exhaustives. On peut parfois utiliser des représentations qui pourront tirer partie de structures non prévues, ou faire des approximations qui dégagent des structures. Dans tous les cas, il est généralement utile de savoir être relationnel et précis uniquement selon les besoins et d'être capable d'oublier des relations quand elles ne servent plus.

3.1 Relations et graphes

En toute généralité, une relation de support $(E_i)_{i\in I}$ est un sous-ensemble du produit cartésien $\bigotimes_{i\in I} E_i$. En général, quand on manipule des relations symboliques, les E_i seront des ensembles finis. Les relations les plus couramment utilisées sont les relations binaires. Une relation binaire est isomorphe à un graphe, les éléments de la relation étant les arêtes du graphe.

Les techniques du chapitre précédent ne peuvent malheureusement pas être directement utilisées pour les relations binaires. D'une part il est très restrictif dans le cadre des relations d'imposer le détermisme (on n'aurait plus que les relations fonctionnelles), et d'autre part la notion de distinguabilité n'a pas beaucoup de sens ici. Finalement, on est en général beaucoup plus précis si on ne se limite pas aux relations binaires, mais si on utilise des relations entre plus de deux ensembles. Dans ce cas, on se retrouve à manipuler des hypergraphes orientés. Heureusement, ces hypergraphes ont un peu de structure, ce qui fait qu'on peut parfois les manipuler efficacement : toutes les arêtes ont exactement la même taille.

3.1.1 Représentation d'hypergraphes par arbres de décision

Ce qui distingue les hypergraphes représentant des relations des hypergraphes représentant des ensembles de graphes, c'est que dans le cas des relations, on n'a pas de notion de sommet indistinguable. On ne peut donc pas minimiser la représentation en diminuant le nombre de sommets. Tout ce qu'on peut faire, c'est trouver une bonne représentation des arêtes. Si on a une représentation canonique des sommets, une représentation canonique des arêtes nous permettra de faire du partage et de stocker efficacement le résultat d'opérations sur les relations.

L'ensemble des arêtes d'un hypergraphe orienté fini est isomorphe à un ensemble fini de mots. Le problème de représenter efficacement

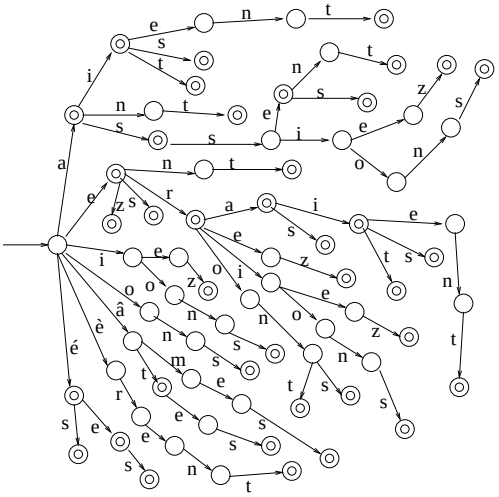

FIG. 3.1 – L'arbre de de la Briandais pour les terminaisons des verbes français

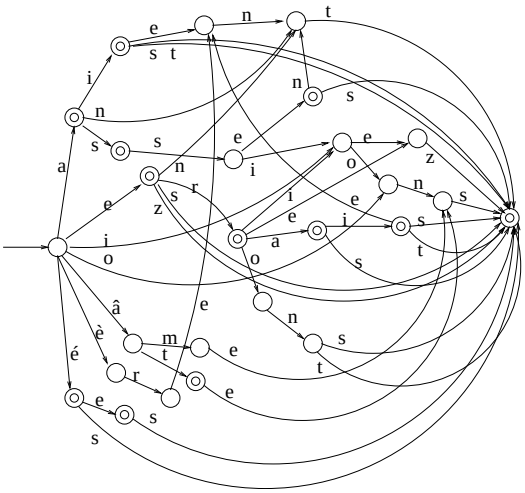

FIG. 3.2 – L'arbre de de la Briandais après hash-consing

un ensemble fini de mots se pose depuis longtemps en informatique, et on connaît un certain nombre de représentations efficaces. Pour représenter efficacement l'ensemble des noms de fichier d'un répertoire, René DE LA BRIANDAIS a inventé une structure d'arbres (de la Briandais, 1959) dans laquelle chaque sommet correspond à une position dans les mots, et dont les arêtes sortantes sont étiquetées par les lettres possibles à cette position. Cette représentation est unique et permet donc la canonicité des représentations d'hypergraphes.

En fait, cette représentation est tout simplement un automate, qu'on peut donc minimiser pour gagner de la place (cf figure 3.2 pour un exemple venant du traitement des langues naturelles (Daciuk et al., 2000)). En fait, cela revient exactement à représenter l'arbre par un DAG en utilisant le hash-consing. Cette représentation est en général très compacte, elle est canonique et n'est pas coûteuse en temps de calcul. Dans le cas des hypergraphes aux arêtes étiquetées, les feuilles du DAG seront les étiquettes des arêtes, ce qui peut sérieusement supprimer des opportunités de partage.

Dans les cas où pratiquement tous les sommets de l'hypergraphe (les lettres de l'alphabet) apparaissent à chaque position de l'arbre, il est plus efficace d'utiliser un tableau qu'une liste dans l'implémentation de l'arbre de de la Briandais. Ces structures sont alors appelées des *tries* (pour Re*trie*val) (Fredkin, 1960). Tous les choix sont alors possibles à chaque étape, et les algorithmes qui testent l'appartenance d'une arête à l'ensemble sont alors plus efficaces. Pour tester l'appartenance, on regarde dans le tableau la position du sommet et si le pointeur à cet endroit est nul on conclut que l'arête n'est pas dans l'ensemble, sinon on itère avec le sommet suivant en suivant le pointeur. C'est donc un processus de décision et il est courant de retrouver les structures de *trie* sous le nom d'arbre de décision. Il est possible d'éliminer certaines décisions inutiles car un seul choix est possible, en étiquetant les arêtes de l'arbre non pas par des sommets du graphe, mais des suites de sommets. On appelle ces arbres les arbres de Patricia (Morrison, 1968).

3.1.2 Relations booléennes

Dans le cas de relations, les tuples ont tous la même taille. On peut alors éliminer d'autres choix, ceux dont tous les pointeurs mènent au même arbre de décision. Il suffit d'étiqueter les sommets de l'arbre par les éléments d'indexation des relations (les éléments du I du support $(E_i)_{i\in I}$), et d'ordonner ces index. Dans l'arbre de décision, la convention sera que toutes les valeurs de E_i sont possible si le nœud suivant $i-1$ est $i+k$ avec $k>0$. Dans le cas où on minimise ces arbres et où les E_i n'ont que deux éléments, cette représentation correspond aux Boolean Decision Diagrams (BDDs) de Randal E. BRYANT (Bryant, 1986). La seule différence est uniquement cosmétique : c'est l'emploi des sommets « vrai » et « faux » au lieu d'un sommet terminal et du pointeur nul.

Théoriquement, le gain par rapport au *trie* minimisé est assez faible : alors que la minimisation

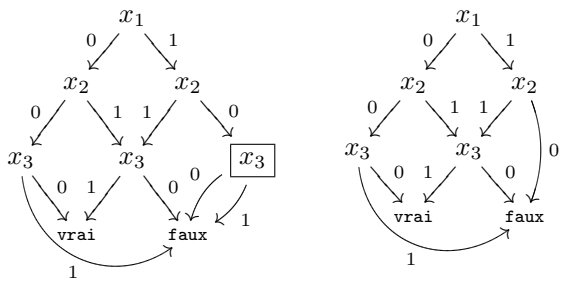

FIG. 3.3 – Arbre de décision et élimination de décision inutile.

fait gagner un facteur linéaire en moyenne, la compression des sous-arbres ne contribue qu'à 1% du gain des BDDs par rapport à un *trie* non minimisé (Liaw and Lin, 1992). Ce qui fait l'intérêt des BDDs en analyse statique, c'est que des relations faisant intervenir des ensembles d'indices différents mais pas forcément disjoints peuvent être représentés de manière uniforme, dans un seul graphe, grâce à la compression de chemins. Alors que le gain théorique est nul puisque la taille moyenne d'un BDD est exponentielle par rapport au nombre d'indices, en pratique les BDDs vont permettre à travers le partage d'exploiter certaines régularités des relations. Ils sont donc largement utilisés pour représenter les relations booléennes en analyse statique (Mauborgne, 1994; Bagnara, 1996; Le Charlier and van Hentenryck, 1993; Stoller and Liu, 1998; Bagnara and Schachte, 1999). L'avantage de l'analyse statique dans le cadre de l'interprétation abstraite est d'ailleurs qu'il est possible d'approcher les relations booléennes par des relations qui auront une structure permettant le partage et donc une représentation plus compacte (Mauborgne, 1994). Cela est d'autant plus intéressant que les opérations sur les BDDs sont proportionnelles à leur taille. On peut donc fixer une limite de taille au-delà de laquelle on applique l'approximation par repliage (section 2.3.3) ou par limitation de taille (on remplace un sommet par le sommet « vrai »).

Il est d'ailleurs possible d'améliorer substentiellement l'élargissement que j'avais proposé dans Mauborgne (1999a) en ne prenant pas un seul itéré, mais deux itérés successifs, représentés par des BDDs B_1 et B_2, tels que $B_1 \Rightarrow B_2$. On peut alors limiter les choix de repliage ou de limitation de taille aux nouveaux sommets de B_2 vis-à-vis de l'implication. Ce sont les sommets tel qu'il existe un chemin p dans B_1 dont l'étiquette est une variable et un choix c mène à « faux », et auquel mène le chemin pc dans B_2.

Les BDDs ont connu un grand succès en model checking de composants matériels dans les années 1990 (Burch et al., 1990). Depuis, cette communauté rapporte plus de succès avec une méthode complètement différente de représentation des relations booléennes, les formules logiques. L'opération clé permettant par exemple de savoir si un modèle vérifie une spécification est la satisfiabilité de la formule (savoir si la relation est vide ou non). La technique repose donc sur des outils de SAT-solving (Biere et al., 1999), qui consiste à utiliser une forme normale conjonctive (faible, car non unique) et à utiliser des méthodes de résolution avec backtracking pour décider de la satisfiabilité de la formule. Ces techniques me semblent peu adaptées à l'analyse statique par interprétation abstraite, car la recherche de points fixes fait intervenir beaucoup de tests de satisfiabilité et les formules peuvent grossir sans limite au fur et à mesure des itérations. Quelques essais ont été menés par David MONNIAUX dans le cadre de l'analyseur statique ASTRÉE développé à l'ENS, et ils semblent montrer très clairement que la technique du SAT-solving n'est pas adaptée.

3.1.3 Relations non booléennes

Décisions multiples

Quand les E_i ne sont pas booléens, on peut utiliser des *tries* et utiliser la même règle de simplification qui compresse les chemins, puisque la seule hypothèse nécessaire pour faire cette simplification est d'avoir des tuples de même taille. On obtient alors des Multiple Decision Diagrams (MDD) (Srinivasan et al., 1990). Mais cette représentation peut souffrir des mêmes défauts que les *tries* si les E_i sont grands, à savoir une trop grosse utilisation de la mémoire si les tableaux à chaque nœud sont creux. Et si on utilise des arbres de de la Briandais, on risque à l'inverse d'avoir des algorithmes d'appartenance moins efficaces à cause de la recherche dans une liste. Il est possible d'avoir une situation avec les mêmes bénéfices que les arbres de de le Briandais, mais sans trop souffrir des cas où tous les sommets sont possibles à une position dans les arêtes, tout simplement en utilisant des arbres binaires de recherche au lieu des listes. Ces structures, appelées Ternary Search Tries (Sedgewick, 1997), sont composées d'une étiquette d'arête, d'un pointeur à gauche vers les éventuelles étiquettes plus petites, un pointeur à droite vers les éventuelles étiquettes plus grandes, et un poin-

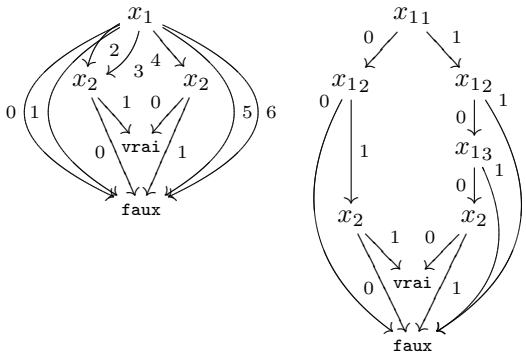

FIG. 3.4 – Encodage booléen

teur au milieu vers la suite de l'arête. Ces représentations sont assez efficaces, mais on peut faire encore mieux par encodage booléen.

Encodage booléen

Une autre façon d'obtenir la même complexité est d'utiliser un encodage booléen. Un encodage booléen d'un ensemble fini E_i est une fonction injective de E_i vers $\{0,1\}^{k_i}$. On peut choisir pour k_i le plus petit entier supérieur au $\log_2$ de $|E_i|$. Une famille d'encodage des $(E_i)_{i\in I}$ permet donc naturellement de représenter des relations de support $(E_i)_{i\in I}$ par des relations de support $\left(\{0,1\}^{k_i}\right)_{i\in I}$. L'effet sur les arbres de décision est alors de remplacer chaque nœud N d'étiquette i par $(2^{k_i}-1)$ nœuds d'étiquette i_1 à i_k, ces nœuds formant un arbre de décision menant aux $N.v$ pour tout v dans E_i, et à « faux » pour les valeurs de $\{0,1\}^{k_i}$ qui ne codent pas de valeur de E_i (voir la figure 3.4). Comme pour les listes ou les arbres de recherche, cette représentation multiplie par une constante la mémoire nécessaire quand toutes les valeurs de E_i sont possibles, mais l'élimination des nœuds inutiles et surtout le partage des sous-arbres permet de regagner de la place, souvent de manière bien plus efficace que les arbres de recherche même quand la relation est creuse.

Représentations mixtes

Quand les ensembles E_i sont infinis, ou tout simplement trop gros, il n'est pas possible d'utiliser l'encodage booléen. Dans ce cas, il existe une littérature abondante proposant différents méchanismes pour utiliser des arbres de décision représentant de manière compacte les relations sur les ensembles infinis. On peut par exemple prendre la décision selon les valeurs d'expressions (Asarin et al., 1997; Møller et al., 1999; Behrmann et al., 1999), faire des choix multiples selon ces expressions (Wang, 2004) ou utiliser des BDDs pour représenter des ensembles de contraintes (une contrainte étant un vecteur de bit) (Clarisó and Cortadella, 2004). En pratique, ces représentations ne semblent pas plus intéressantes que les domaines abstraits numériques déjà utilisés en interprétation abstraite, comme les polyèdres (Cousot and Halbwachs, 1978) ou les octogones (Miné, 2001). Mais dans les cas de relations où certains E_i sont de petite taille et d'autres sont infinis, les domaines classiques ne semble pas assez précis, et il faut utiliser des relations symboliques. La difficulté consiste à relier les valeurs booléennes (représentant les ensembles de petite taille) et les valeurs numériques (représentant les ensembles infinis ou de très grande taille). Il est bien entendu possible d'approximer la relation en ignorant les interactions entre ces deux types de valeurs, mais c'est en général trop imprécis. Bultan et al. (1998) ont proposé d'utiliser des disjonctions de conjonction de BDDs et contraintes de Persburger, mais cela semble trop coûteux car les formules peuvent grossir très vite. Des arbres de décision très expressifs, comme les Hybrid-Restriction Diagrams (Wang, 2004) peuvent être utilisés pour représenter à la fois des contraintes linéaires et booléennes, mais leur expressivité semble les rendre trop complexes. Il faut néanmoins noter que ces structures n'ont jusqu'à présent été testées que dans le cadre du model checking qui impose de faire des calculs exacts sur le modèle. Il serait intéressant d'essayer d'approximer certaines opérations pour voir si il est possible de tirer partie de l'expressivité de cette représentation tout en gardant un coût raisonnable.

Dans le cas où le support de la relation contient peut d'ensembles de petites tailles, on peut utiliser des arbres de décisions étiquetés par un encodage booléen de ces petits ensembles, et dont les feuilles sont des éléments de domaines abstraits classiques représentant les valeurs numériques (Mauborgne, 2004). L'avantage de cette approche est d'être entièrement paramétrée par les domaines abstraits numériques, ce qui permet de choisir les représentations les plus adaptées aux relations mais aussi à la précision qu'on souhaite garder. Elle permet aussi d'utiliser directement les élargissements et fonctions de transfert pour les variables numériques. L'inconvénient c'est en général peu d'opportunité de partage, ce qui limite le nombre de variables booléennes manipulables au sein d'une

relation. Mais il est possible d'approximer une relation par la conjonction de plusieurs relations reliant moins de variables. En permettant à ces relations d'avoir plusieurs variables en commun, on peut choisir la précision de la représentation en fonction du coût. cela permet aussi de ne garder que les parties de la relation qui semblent intéressantes. Ces parties peuvent être calculées par une première analyse, de préférence peu coûteuse (cf section 4.4).

Hypergraphes non orientés

Dans le cas où l'on souhaite représenter des hypergraphes non orientés, il existe une structure adaptée, proche des BDDs, les Zero suppressed Decision Diagrams (ZDD) (Minato, 1993). Ces arbres de décision sont assez efficaces pour représenter des ensembles d'ensembles creux. Pour un hypergraphe, ils utilisent une variable par sommet, la valeur de la variable étant 0 si le sommet est dans l'arête et 1 sinon. Dans les ZDD, la suppression des choix qui mènent à deux sous-arbres identiques est remplacée par la suppression des choix dont la branche 0 mène vers « faux ». Autrement dit, on supprime des arbres les sommets qui sont toujours dans les arêtes. Pour un hypergraphe à n sommets, la taille de la représentation sera de l'ordre de 2^n, alors que pour un hypergraphe orienté d'arêtes de taille au plus k, l'arbre de décision associé sera de taille n^k.

3.2 Relations infinitaires

Les relations peuvent être infinies pour deux raisons non exclusives : un des ensemble du support $(E_i)_{i\in I}$ peut être infini, et le nombre d'ensembles du support (la taille de I) peut être infini. Dans le deuxième cas, que nous n'avons pas encore abordé, nous dirons que la relation est infinitaire (elle est binaire si I est de taille 2, et n-aire si I est de taille n).

3.2.1 Traces et propriétés temporelles

Les relations infinitaires sont donc des ensembles de vecteurs infinis. La principale notion pour laquelle on associe un vecteur infini à une exécution de programme est la notion de trace. Une trace d'exécution est en effet la suite des états du programme au cours du temps, et il existe de nombreux programmes qui sont faits pour ne jamais s'arrêter, comme les systèmes d'exploitation ou plus simplement certains protocoles. Dans certains cas, considérer les traces comme infinies peut faciliter l'expression des problèmes, même si dans la réalité on n'a jamais de trace infinie. Par exemple, pour exprimer le passé d'un avion en vol depuis assez longtemps, ou les propriétés de réactivité d'un logiciel. Les propriété des traces infinies sont généralement appelées des propriétés temporelles. On distingue généralement deux grandes catégories de propriétés, les propriétés de sûreté et les propriétés de vivacité (Lamport, 1977). Intuitivement, les propriétés de sûreté doivent être toujours vraies, et pour le vérifier il suffit de regarder des fenêtres finies sur les traces, alors que les propriétés de vivacité correspondent à des événements qui doivent arriver infiniment souvent, mais sans borne supérieure sur le temps séparant deux événements. Les propriétés de vivacité sont en général plus difficiles à exprimer, mais elles sont indispensables dans beaucoup d'applications, comme les protocoles. La terminaison est un exemple typique de propriété de vivacité.

Pour exprimer ces propriétés, on peut utiliser des logiques spécialement conçues, comme la logique temporelle linéaire (LTL) (Pnueli, 1977) la logique temporelle avec bifurcations (CTL) (Emerson and Halpern, 1985), ou le μ-calcul (Niwiński, 1984). Toutes ces logiques ne sont malheureusement pas adaptées à la manipulation de relations infinitaires car les opérations seraient trop complexes. Les analyses de programmes qui utilisent ces logiques utilisent généralement des automates. Les automates les plus puissants pour ce faire sont les automates d'arbres infinis (Emerson and Jutla, 1991), mais ils ne sont pas utilisés en pratique (ils sont aussi expressifs que le μ-calcul). On fait donc une approximation (ou une restriction) vers les automates de Büchi (Büchi, 1960) qui représentent des ensembles de mots infinis. Un automate de Büchi se définit exactement comme un automate de mots classique, la seule chose qui change, c'est qu'il sert à lire des mots infinis et qu'un mot infini est accepté par l'automate s'il est lu par l'automate et la lecture passe infiniment souvent par des états finaux. Les ensembles de mots représentables par un automate de Büchi sont appelés les langages ω-rationnels. Ces langages sont clos par union, intersection et complémentation, et on peut montrer (Büchi, 1960) que tout langage ω-rationnel est union finie de langages de la forme $U.V^\omega$ où U et V sont des langages rationnels et $X^\omega \stackrel{\text{def}}{=} \{\, u_0u_1\ldots \mid \forall i, u_i \in X\}$.

Pour une utilisation efficace des automates de

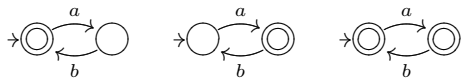

FIG. 3.5 – Trois automates de nombre d'états minimal pour le langage $(ab)^\omega$

Büchi, qui sont une extension simple des automates de mot, on pourrait être tenté de faire la même construction qu'en 2.2 pour obtenir une représentation avec partage maximal. Cela n'est malheureusement pas possible : les automates de Büchi déterministes sont strictement moins expressifs que les automates de Büchi non déterministes, et de plus il n'existe pas en général d'automates de Büchi minimal pour un langage ω-rationnel donné. D'ailleurs les analyses utilisant des automates de Büchi sans restriction sont en général beaucoup trop lentes et ne réussissent pas à analyser des programmes d'importance moyenne.

3.2.2 Langages ouverts, fermés et quasi-ouverts

La raison pour laquelle on ne peut pas utiliser directement les techniques de la section 2.2, c'est que la notion de distinguabilité ne coïncide plus, à cause de l'effet des états finaux, qui peuvent être redondants (figure 3.5). Une façon simple de remédier à cela, c'est de se restreindre aux automates dont tous les états sont finaux. Ces automates définissent alors les langages fermés, au sens de la topologie naturelle des mots infinis. Les langages fermés ont de bonnes propriétés algébriques, puisqu'ils sont clos par union, intersection et intersection infinie, et ils peuvent être manipulés très efficacement (même sans les restrictions d'« entrées équivalentes » que je m'étais imposées dans Mauborgne (1999a)). Mais ils ne permettent pas d'exprimer les propriétés temporelles les plus importantes, à savoir les propriétés de vivacité. La raison en est simple : on ne peut pas exprimer le fait d'être dans un état dont on va nécessairement sortir au bout d'un temps non borné, car si ce temps est non borné, c'est qu'on a une boucle dans l'automate, et comme tous les états sont finaux, rien n'empêche de rester dans cette boucle.

Si on prend le complémentaire d'un langage fermé, on obtient un langage ouvert. Les langages ouverts permettent justement d'exprimer le fait qu'on ne doit pas rester indéfiniment dans un état. On peut aussi les représenter efficacement en observant que le complété d'un automate n'ayant que des états finaux n'a qu'un seul état non final, l'état puits dont toutes les arêtes sortantes rebouclent sur lui-même. Il s'en suit qu'un mot n'est pas dans le langage fermé si et seulement si sa lecture passe (infiniment souvent) par l'état puits. On en déduit que tout langage ouvert est représentable par un automate déterministe complet avec un seul état final dont toutes les arêtes sortantes rebouclent sur lui-même. Cette condition est suffisante pour que l'indistinguabilité de la section 2.2 coïncide avec l'égalité des langages de mots infinis, et donc pour qu'on ait une représentation efficace sans travail supplémentaire. Les langages ouverts sont aussi clos par union et intersection, et ils sont clos par union infinie. Ces langages sont vraisemblablement plus utiles pour l'analyse de propriétés temporelles, mais ils sont encore très limités, puisqu'on ne peut rien dire sur l'état du système après avoir quitté les états dans lesquels il ne fallait pas rester indéfiniment.

La condition qui rend les langages ouverts représentables efficacement est en fait un peu plus générale que la limitation à un seul état où tout est possible. On peut en effet l'étendre au cas où les seuls cycles de l'automate qui contiennent des états finaux sont des cycles de taille 1. On peut caractériser les langages reconnus par ces automates par la propriété suivante : pour tout mot α appartenant au langage, il existe un préfixe u de α tel que si A est l'ensemble des lettres dans α situées après u, $u.A^\omega$ est inclus dans le langage. J'ai appelé ces langages les langages quasi-ouverts. Ils peuvent être représentés aussi efficacement que les langages ouverts et ont les mêmes propriétés algébriques : clôture par union, intersection et union infinie. Les langages quasi-ouverts peuvent exprimer certaines propriétés de vivacité intéressantes, comme la terminaison (représentée de manière classique par un état d'arrêt répété infiniment en fin de trace). Ils ne permettent pas d'exprimer de manière exacte les propriété de vivacité du genre un état reviendra infiniment souvent sans borne finie entre deux apparitions, car cela nécessiterait un cycle contenant au moins un état final et un état non final. L'équité est un exemple typique de propriété qu'on ne peut pas représenter avec des langages quasi-ouverts.

3.2.3 Langages itératifs

À mon avis, une des raisons essentielles qui rendent les langages ω-rationnels si difficiles à manipuler, c'est leur non-déterminisme inhérent.

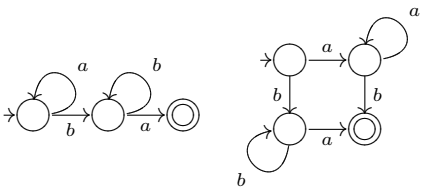

FIG. 3.6 – Deux langages dont l'itération donne le même langage itératif.

En effet, un langage ω-rationnel est une union finie de $U.V^\omega$, et toute représentation devra d'une façon ou d'une autre représenter tous les U et tous les V et les tests d'appartenance ou d'inclusion devront tester différent cas si par exemple deux U ont des mots communs. Ce non-déterminisme peut être réduit si à chaque préfixe possible du langage, on associe un unique V^ω. Toutefois, on ne pourra obtenir une représentation efficace que si on sait déjà représenter efficacement les V^ω. On appellera ces langages les langages itératifs. Pour l'instant, je ne sais construire efficacement l'automate de Büchi que pour les langages itératifs V^ω dont le langage fini V est représenté par un automate déterministe dont les états finaux n'ont aucune arête sortante[1]. On appelle ces langages des langages non-préfixes, car ils ne contiennent pas deux mots préfixes l'un de l'autre. Pour ces langages, il suffit en effet de remplacer les états finaux par une copie de l'état initial, marqué comme final. L'ensemble de ces langages itératifs est exactement celui des langages représentables par un automate de Büchi déterministe dont le seul état final est l'état initial.

Normalisation

Le problème de la représentation unique des langages itératifs n'est pas évident. Il existe en effet plusieurs langages finis dont l'itération donne le même langage infini (voir la figure 3.6), il ne suffit donc pas de représenter un langage rationnel avec partage maximal. Un bon critère pour le choix du langage fini consiste à prendre le plus petit pour le pré-ordre préfixe. Un langage $\mathcal{L}_1$ est plus petit qu'un langage $\mathcal{L}_2$ pour le pré-ordre préfixe si pour tout mot u_2 de $\mathcal{L}_2$, il existe un mot u_1 de $\mathcal{L}_1$ préfixe de u_2. Un tel langage, si il existe, est un bon choix car il minimise la longueur des cycles contenant l'état final dans l'automate du langage itératif. La longueur de ces cycles intervient de manière prépondérante dans

[1] Si un tel automate est minimal, alors il n'aura qu'un seul état final.

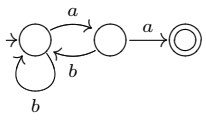

FIG. 3.7 – Le langage des mots contenant une infinité de aa, représentable en forme normale d'itéré.

les opérations sur les automates de Büchi. On peut noter que le pré-ordre préfixe est bien antisymétrique sur les langages non-préfixes (c'est donc un ordre), car si $\mathcal{L}_1$ est plus petit que $\mathcal{L}_2$, $\mathcal{L}_2$ ne peut contenir de mot préfixe d'un mot de $\mathcal{L}_1$ lui-même préfixe d'un mot de $\mathcal{L}_2$.

On peut montrer qu'il existe une borne inférieure à tout ensemble de langages non-préfixes. Il suffit de définir la frontière préfixe d'un langage $\mathcal{L}$ comme l'ensemble des mots de $\mathcal{L}$ qui n'ont pas de préfixe dans $\mathcal{L}$, puis de considérer la frontière préfixe de l'union des langages. Cet ensemble sera par définition non-préfixe, et peut être construit effectivement à partir d'un langage dont on veut représenter l'itéré. On peut montrer que tout langage itératif est inclus dans la borne inférieure préfixe des langages non-préfixes dont il est l'itération. Pour l'inclusion inverse, j'ai montré un cas particulier dans Mauborgne (2003), dans lequel on suppose que le langage est invariant par permutation des lettres. Le cas général est encore ouvert à ma connaissance. Cela dit, l'inclusion inverse n'est pas forcément nécessaire dans le cadre de l'interprétation abstraite, car il suffit d'avoir un représentant unique plus grand que le langage qu'on cherche à représenter pour avoir une bonne approximation. Le fait de lever l'hypothèse d'invariance par permutation permet de représenter beaucoup plus de langages que dans Mauborgne (2003), qui permettait déjà de représenter certaines propriétés de vivacité et d'équité (voir la figure 3.7 pour un exemple très simple).

Automates utilisant des langages itératifs

Il est ensuite possible de construire des automates à représentation unique si à chaque préfixe u du langage on peut associer un plus grand langage itératif L tel que $u.L$ soit dans le langage. Pour cela, on obtient de bons résultats d'expressivité en ajoutant des arêtes avec une étiquette spéciale indiquant l'entrée possible dans une partie itérative. On sort de cette partie itérative soit en atteignant un état puits, soit en empruntant une nouvelle arête d'entrée dans une autre partie itérative correspondant au nouveau préfixe at-

teint. Comme pour un automate de Büchi, pour qu'une lecture de mot soit acceptante, il faut passer infiniment par un état final, mais à la différence des automates de Büchi, on impose de plus de ne passer qu'un nombre fini de fois par des étiquettes d'entrée dans une partie itérative. Il reste encore à implémenter ces automates pour évaluer la perte de précision et l'éventuel gain d'efficacité par rapport aux automates de Büchi dans le cadre de l'analyse de programmes.

3.2.4 Suppression des décisions inutiles

Dans les BDDs, on supprime les états dont tous les arcs sortants mènent au même état car il sont inutiles dans la lecture d'un mot. Mais pour ce faire, il faut savoir le nombre de tels états qui ont été supprimés. En utilisant des variables ordonnées, on peut reconstruire cette information. Dans Mauborgne (1999a), j'ai proposé de le faire aussi pour les relations infinitaires, mais cela impose d'utiliser un nombre fini de noms de variables, et pour garder la cohérence il en découle qu'il faut que le langage soit invariant par les permutations qui préservent les noms de variables.

Cela n'est pas forcément nécessaire en choisissant d'étiqueter les états non pas par des variables mais par des entiers qui représentent le nombre d'états supprimés avant d'arriver à un état donné. Cette représentation a déjà été proprosée dans le cas fini sous le nom de Differential BDD (Anuchitanukul et al., 1995). Elle permet aussi une représentation unique, et les algorithme de la section 2.2 sont parfaitement adaptés à l'ajout d'étiquettes aux états. Il est donc possible d'utiliser cette compression supplémentaire pour toutes les catégories de langages définies ci-dessus.

3.2.5 Ensembles d'arbres et $\widehat{\mu}$-calcul

Dans le cas des logiques de temps avec bifurcation, comme par exemple CTL*, on considère qu'un état peut avoir plusieurs futurs possibles et il est alors nécessaire dans la représentation de ces formules de manipuler des ensembles d'arbres infinis. Pour que ces ensembles d'arbres représentent effectivement des propriétés temporelles, j'ai proposé dans Mauborgne (2000c) d'ajouter des relations aux sommets de choix des squelettes d'arbres. Les relations permettent à la fois de coder les propriétés temporelles le long des chemins des arbres, mais aussi de retrouver une partie de la précision perdue par l'approximation en squelettes. On peut par exemple utiliser ces représentations pour représenter l'ensemble des $f(a^n, b^n, c^n), n \in \mathbb{N}$.

Dans (Cousot and Cousot, 2000), Patrick et Radhia Cousot ont proposé une nouvelle logique temporelle, le $\widehat{\mu}$-calcul, permettant de raisonner de manière symétrique sur le passé et le futur. Ce genre de raisonnement est très utile par exemple pour certifier le comportement d'un avion en vol, puisque l'atterissage comme le décollage sont beaucoup trop loin en terme de nombre d'itérations des opérations automatiques pour qu'on puisse supposer qu'on les connait toutes. Raisonner comme si on était en vol depuis toujours est une bonne approximation. Dans ce cadre, une trace est une suite infinie dans les deux sens (fonction de $\mathbb{Z}$ vers les états), et on raisonne à un instant donné de la trace, le « présent ». Le problème de la représentation de telles traces n'est pas évident. On peut utiliser des automates de Büchi (ou des versions plus simples et plus efficaces, comme dans les paragraphes précédents), un pour représenter le futur avec les états aux instants supérieurs au temps 0, et l'autre pour le passé. L'opération de changement de sens du temps est alors très efficace. Pour stocker l'instant présent, il faut dans ce cadre marquer l'état de cet instant. L'inconvénient de cette représentation est la perte des informations relationnelles entre les passé et l'avenir, ce qui diminue grandement l'intérêt de cette logique. En effet, si on a deux traces, une avec des états toujours à a et une autre avec des états toujours à b, on se retrouve avec un état passé qui est soit a^ω, soit b^ω et un état futur qui est le même, mais on ne sait plus qu'on aura du a seulement si on avait du a par le passé.

Il est possible de retrouver une partie de ces informations en utilisant non plus un couple d'ensemble de traces infinies dans un seul sens, mais un ensemble de couples de traces infinies dans un seul sens. En effet, un couple n'est rien d'autre qu'un arbre très simple et il est donc possible d'utiliser des schémas d'arbres (Mauborgne, 2000c). Encore une fois, on utilisera un marqueur spécifique pour représenter le temps présent. Le retournement du temps est alors immédiat et l'avancée du temps présent consiste simplement à décaler ce marqueur d'un cran. Avant élargissement, cela peut donner de gros schémas, mais il est possible d'en réduire la taille en utilisant des compteurs (Mauborgne, 1999b).

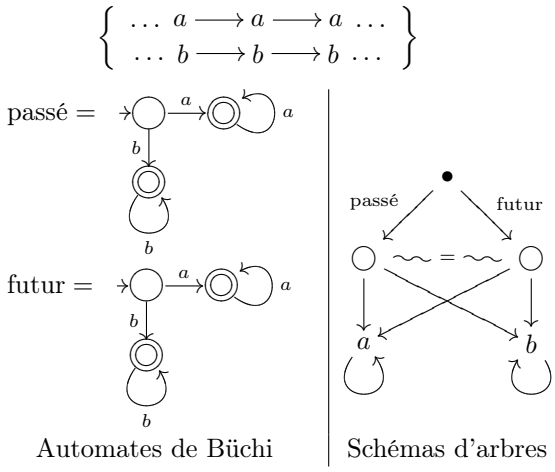

FIG. 3.8 – Représentations pour des traces infinies dans les deux sens.

3.3 Discrimination de traces

Les relations étant des ensembles de vecteurs, la façon la plus précise de les représenter consiste à représenter chacun des vecteurs de la relation. Les relations symboliques que nous avons vues jusqu'à présent essaient ensuite de mettre en commun ce qu'il est possible de mettre en commun, de façon à extraire une structure qui donnera une représentation compacte. Les opération approchées permettent d'imposer plus de structure, au prix de quelques vecteurs supplémentaires ajoutés à l'ensemble. Une approche intéressante dans le cadre de l'analyse statique est de regarder comment ces vecteurs sont ajoutés et de garder une structure qui respecte au mieux cette histoire.

En général, l'analyse statique par interprétation abstraite simule, avec des approximations, l'ensemble des exécutions d'un programme sur toutes ses entrées possibles. Si l'analyse est sensible au flot de contrôle, elle suit ensuite le flot de contrôle du programme, et à chaque point de branchement (tests conditionnels, boucles, ...), elle calcule une sur-approximation des ensembles d'états qui doivent suivrent tel ou tel branchement, simule les calculs dans chaque branche puis approxime l'union des résultats de chaque branche. Il y a donc principalement deux événements qui ajoutent des vecteurs dans les relations, la lecture d'une entrée et l'union des branches d'un point de contrôle. Comme chaque échantillon d'entrées définit une trace d'exécution du programme, et qu'au sein de chaque trace, on ne suit qu'une seule branche de chaque point de contrôle, il semble donc pertinent de guider la structure des relations par les traces d'exécution des programmes. Une grande partie du travail présenté dans cette section, restreint à l'analyse d'accessibilité, a été réalisé en collaboration avec Xavier RIVAL (Mauborgne and Rival, 2005).

3.3.1 Abstraction d'ensembles de traces et recouvrements

Traces

On peut décrire de manière très générale un programme concret comme une relation de transition (pas nécessairement finie) $\rightarrow \subset Q \times Q$, où Q est l'ensemble de tous les états possibles du programme, plus un ensemble d'états initiaux, $I \subset Q$. Dans ce contexte, la lecture d'entrées du programme peut correspondre soit au choix d'un état initial, soit au choix entre plusieurs transitions à partir d'un état q donné où on lit ces entrées. Une *trace* d'un programme $(Q, \rightarrow, I)$ est une suite finie ou infinie d'états, telle que le premier élément de la suite soit dans I, et deux états consécutifs q_i et q_{i+1} de la suite soient reliés par la relation de transition (on écrira $q_i \rightarrow q_{i+1}$). L'ensemble des traces d'un programme est naturellement clos par préfixe. Un ensemble de traces finies T, clos par préfixe, définit de manière unique un programme : celui dont les états sont les états apparaissant dans les traces, dont l'ensemble d'états initiaux est l'ensemble des premiers états des traces, et dont la relation de transition est définie par l'ensemble des couples (q_1, q_2) tels que q_1 et q_2 soient les derniers états des traces t_1 et t_2, et t_2 est la trace composée de t_1 suivie de l'état q_2. On en déduit que si on était capable de représenter exactement l'ensemble des traces d'un programme, on pourrait exprimer toutes les propriétés de ce programme. Un tel ensemble n'est bien entendu pas calculable en général, et toute analyse statique va donc approximer d'une façon ou d'une autre l'ensemble des traces des programmes.

En model checking abstrait par exemple, on utilise généralement des suites d'ensembles d'états (Clarke et al., 1994; Cleaveland et al., 1995). Si on ne s'intéresse qu'aux états accessibles, on approxime les suites par leur état final. En général, cette première approximation est insuffisante, et elle est suivie d'autres approximations, qui permettent de représenter effectivement les relations. On constate que pour guider cette approximation, on utilise souvent une disjonction des états fondés par exemple sur des étiquettes de points de contrôles. Cela revient à baser cette disjonction sur un critère de

distinction des traces en fonction du point de contrôle de leur dernier état.

Pour fixer les choses, nous appellerons D_T le domaine abstrait utilisé pour représenter les ensembles de traces (par exemple le domaine des intervalles (Cousot and Cousot, 1977a), qui représentera l'ensemble des traces dont les variables de l'état final ont des valeurs décrites par des intervalles), et γ_T la concrétisation de ce domaine vers les ensembles de traces clos par préfixe. Nous noterons $\mathcal{P}_{\preceq}(Q^\star)$ l'ensemble des ensembles de suites de Q clos par préfixe.

Discrimination de traces

Suivant l'idée selon laquelle on utilisera une disjonction indexée par un ensemble fini (comme dans le cas des étiquettes de points de contrôle), nous allons définir des discriminations de traces basées sur des indices.

DÃ©finition 3.1 (Recouvrement) *Une fonction $\delta : E \to \wp(F)$ est appelée un* recouvrement *de F si et seulement si $\bigcup_{x\in E}(\delta(x)) = F$.*

L'idée est d'utiliser un recouvrement pour déterminer à chaque indice quel ensemble de traces on approxime. Cet ensemble sera donné par la *projection sur l'indice* x : $p^\delta_x(T) \stackrel{\text{def}}{=} T \cap \delta(x)$. Le domaine induit par un recouvrement δ de $Q^\star$ sera l'ensemble des fonctions de $dom(\delta)$ dans D_T. On notera ce domaine D^δ_T. L'effet de cette approximation est donné par la concrétisation :

$$\gamma^\delta_T(f) \stackrel{\text{def}}{=} \bigcup_{x\in dom(\delta)} (\gamma_T(f(x)) \cap \delta(x))$$

Si l'approximation par D_T admet une fonction d'abstraction α_T, on peut définir l'abstraction sur D^δ_T par :

$$\alpha^\delta_T(T) = \lambda x.\alpha_T\left(p^\delta_x(T)\right)$$

Dans les cas où p^δ_x n'est pas calculable, on pourra choisir une approximation conservative de cette fonction. Si (α_T, γ_T) forme une connection de Galois, $(\alpha^\delta_T, \gamma^\delta_T)$ aussi.

Cette nouvelle abstraction sera toujours au moins aussi précise que l'abstraction sans discrimination de trace. Suivant Mauborgne and Rival (2005), on peut montrer qu'elle sera strictement plus précise dès qu'elle permet de distinguer deux ensembles de traces T_1 et T_2 tels que $\alpha_T(T_1) \cap \alpha_T(T_2)$ soit strictement supérieur à l'infimum de D_T. La définition formelle de la notion de distinction est donnée par :

DÃ©finition 3.2 (Distinction) *On dit qu'une fonction δ de $E \to \wp(F)$* distingue *les sous-ensembles X_i de F $(i \in I)$ si pour tout x dans E et tout i dans I :*

$$X_i \cap \delta(x) \neq \emptyset \Rightarrow \forall j \neq i, X_j \cap \delta(x) = \emptyset$$

Si il n'existe pas deux ensembles de traces T_1 et T_2 tels que $T_1 \cap T_2 = \emptyset$ et $\alpha_T(T_1) \cap \alpha_T(T_2)$ soit strictement supérieur à l'infimum de D_T, cela signifie que le domaine D_T abstrait chaque trace sur un élément abstrait distinct, et donc qu'en fait il ne fait aucune approximation. Lorsqu'on cherche à gagner de la précision sur les approximations de D_T, on peut donc identifier des ensembles de traces T_1 et T_2 disjoints qui satisfont cette condition et sur lesquels l'imprécision semble poser un problème pour l'analyse. Ensuite, on raffine δ en fonction de ces propriétés d'intérêt en construisant $\delta' : E \times \{1, 2\} \to \wp(Q^\star)$ définie par :

$$\delta'(x, i) = \delta(x) \backslash T_i$$

Ainsi, on δ' distingue les mêmes ensembles que δ' plus T_1 et T_2. L'inconvénient, bien sûr, c'est que chacune de ces opérations multiplie le nombre d'indice, et donc le coût de l'analyse par deux.

Dans certains cas, on peut rendre l'analyse plus précise en augmentant moins le nombre d'indices (et donc avec un coût moins grand). En effet, si le recouvrement utilise deux indices a et b tels que $\delta(a) \cap \delta(b) \neq \emptyset$, on peut remplacer δ par $\delta' : E \cup \{c\} \to \wp(Q^\star)$, avec c un nouvel indice qui n'est pas dans E, définie par :

$$\begin{aligned} \delta'(a) &= \delta(a) \backslash \delta(b) \\ \delta'(b) &= \delta(b) \backslash \delta(a) \\ \delta'(c) &= \delta(a) \cap \delta(b) \\ \delta'(x) &= \delta(x) \quad \text{sinon} \end{aligned}$$

Pour comparer la précision de D^δ_T et $D^{\delta'}_T$, on utilise les opérateurs de clôture des domaines abstraits (Cousot and Cousot, 1979b). Suivant la définition,

$$\gamma^\delta_T \circ \alpha^\delta_T(S) = \bigcup_{x\in E} \gamma_T \circ \alpha_T\left(S \cap \delta(x)\right) \cap \delta(x)$$

La clôture $\gamma_T \circ \alpha_T$ étant croissante, il est facile de conclure que $\gamma^{\delta'}_T \circ \alpha^{\delta'}_T$ est plus petit que $\gamma^\delta_T \circ \alpha^\delta_T$. Si on itère cette opération jusqu'à ce qu'elle ne soit plus possible, et qu'on enlève les indices x tels que $\delta(x) = \emptyset$, on obtient alors une partition. On dira qu'on a un partitionnement. Il n'est pas toujours souhaitable d'utiliser le partitionnement construit à partir du recouvrement,

car au final le nombre d'indices peut augmenter de façon quadratique sans forcément que le gain de précision n'ait été nécessaire dans l'analyse. Le découpage en fonction d'une propriété d'intérêt me semble en général plus approprié.

3.3.2 Exemple de l'accessibilité

L'instanciation immédiate de cette approche est celle qui l'a inspirée : calculer l'ensemble des états accessibles en partitionnant par les points de contrôle des programmes. Dans ce cas, le domaine D_T est un domaine représentant un ensemble de tuples de valeurs, parmi lesquelle le point de contrôle du programme dont on suppose qu'il fait partie de l'état du programme au même titre que les valeurs des variables ou de la pile d'exécution. Pour fixer les choses, nous écrirons $\mathcal{L}$ l'ensemble des points de contrôle et $\mathcal{M}$ l'ensemble des états possibles de la mémoire hors des points de contrôle. L'ensemble des états Q est donc égal à $\mathcal{L} \times \mathcal{M}$.

Partition selon le point de contrôle final

On définit $\delta_{\mathcal{L}} : \mathcal{L} \to \wp(Q^\star)$ comme le recouvrement qui à chaque point de contrôle associe les traces se terminant en ce point de contrôle. Si on définit $\tau_\dashv$ comme le dernier élément de la suite τ, la définition formelle de $\delta_{\mathcal{L}}$ est alors $\lambda l.\, \{\tau \in Q^\star \mid \exists \rho \in \mathcal{M},\ \tau_\dashv = (l, \rho)\}$. Ce recouvrement se trouve être une partition, donc tout raffinement par propriété d'intérêt donnera une partition. Cette partition, présentée dans (Cousot, 1981), est très commune et souvent faite sans le dire quand on décrit des sémantiques abstraites.

Dans le cas où le programme est composé de procédures, on peut encore de la même façon ajouter un partitionnement par les valeurs de la pile d'appels. Si on partitionne complètement en fonction de la pile, cela revient à faire de l'*inlining* des procédures, ce qui ne peut fonctionner que si les procédures ne sont pas récursives.

Partition selon le flot de contrôle

Le partitionnement des traces selon le flot de contrôle a été introduit par Handjieva and Tzolovski (1998). Leur idée était d'étendre les points de contrôle avec des listes de tags représentant l'histoire des calculs. Pour pouvoir faire un partitionnement selon ces nouveaux « points de contrôle », encore faut-il qu'il n'y en ait qu'un nombre fini. Pour ce faire, ils associent à chaque

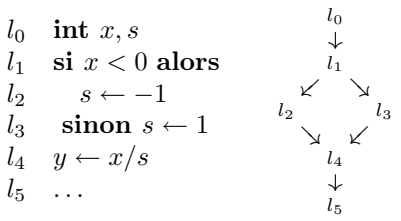

FIG. 3.9 – Programme du calcul du signe et partitionnement selon ses points de contrôle.

boucle un entier qui borne le nombre de tags générés par le boucle. Cela signifie qu'on ne distingue que les k premiers tours de boucle et qu'on garde l'union des tours de boucle suivants. Cela permet de résoudre simplement le problème posé par le programme de la figure 3.9. Pour montrer que ce programme ne fait pas de division par zéro en ligne 4, on ne peut pas simplement utiliser le domaine des intervalles, car celui-ci donne l'intervalle $[-1, 1]$ pour s. Si on sait à l'avance que s ne vaudra normalement que des nombres impairs, on peut utiliser des congruences (Granger, 1989), ou si on sait que le calcul de s est lié à x, on peut utiliser une relation comme en section 3.1.3. Il n'est pas toujours aussi évident de connaître à l'avance la forme de l'invariant ou la forme des relations qui permettront d'éviter les états d'erreur. On peut en général simplement partitionner les traces en fonction de l'histoire du flot de contrôle, et en ligne 4 dire que s vaut -1 si on avait pris la branche l_2 et 1 si on avait pris la branche l_3. Cette simple disjonction permet de résoudre le problème.

Formellement, si $\mathcal{B} \subseteq \mathcal{L}$ est l'ensemble des points de contrôle qui introduisent un branchement (comme les conditionelles, les boucles, ...), on définit l'ensemble de toutes les branches du programme par :

$$\mathbb{C} \stackrel{\text{def}}{=} \{\, (b, l) \in \mathcal{B} \times \mathcal{L} \mid \exists \rho, \rho' \in \mathcal{M},\ (b, \rho) \to (l, \rho') \}$$

Tout recouvrement définit un ensemble de propriétés ou encore une abstraction des traces. Ici, l'abstraction est l'abstraction de flot de contrôle qui consiste à ne garder de la trace que les branches. On appelle $\mathrm{cf}(\tau) \subseteq \mathbb{C}^\star$ la suite construite itérativement par :

$$\begin{array}{rcl} \mathrm{cf}(\epsilon) & = & \epsilon \\ \mathrm{cf}(b.l.u) & = & (b, l).\mathrm{cf}(u) \quad \text{si } (b, l) \in \mathcal{B} \\ \mathrm{cf}(l.u) & = & \mathrm{cf}(u) \quad \text{sinon} \end{array}$$

On peut alors définir le partitionnement selon le flot de contrôle comme étant la discrimination

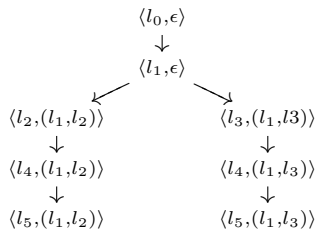

FIG. 3.10 – Effet du partitionnement par flot de contrôle du programme 3.9

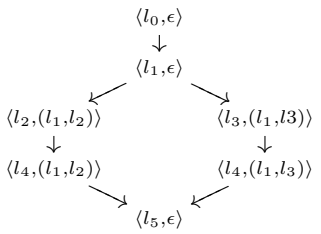

FIG. 3.11 – Partitionnements des traces du programme 3.9, avec fusion des traces en l_4

des traces selon $\delta_{\text{cf}} : \mathcal{L} \times \mathbb{C}^\star \to \wp Q^\star$ avec

$$\delta_{\text{cf}}(l, \beta) \stackrel{\text{def}}{=} \{\sigma \in \delta_{\mathcal{L}}(l) \mid \text{cf}(\sigma) = \beta\}$$

Cette discrimination n'est pas effective car $\mathcal{L} \times \mathbb{C}^\star$ n'est pas fini. Handjieva and Tzolovski (1998) utilisent donc un paramètre $\kappa : \mathcal{B} \to \mathbb{N}$ qui limite *a priori* le nombre d'indices effectivement utilisés pour l'analyse d'un programme. On ne garde plus que les suites β de $\mathbb{C}$ telles que la taille de chaque ensemble de (b, l) de β avec $l \in \mathcal{L}$ soit inférieur à $\kappa(b)$. Le partitionnement limité par κ devient donc :

$$\delta^\kappa_{\text{cf}}(l, \beta) \stackrel{\text{def}}{=} \{\sigma \in \delta_{\mathcal{L}}(l) \mid \lambda_\kappa(\text{cf}(\sigma)) = \beta\}$$

Où λ_κ est la fonction de limitation des suites définie par κ. Si une suite contient trop de (b, l), cette fonction supprime suffisamment de (b, l) à la fin de la suite pour que leur nombre devienne égal à $\kappa(b)$. Cela permet aussi de ne pas partitionner selon certains tests, car chaque branchement selon lequel on partitionne multiplie au moins par deux le nombre d'indices. Cette discrimination, bien que très précise est donc trop chère pour être utilisée en pratique sur de gros codes.

Des recouvrements plus souples

De fait, il n'est généralement pas utile de partitionner les traces en fonction d'événements très anciens, et c'est de toutes façons souvent trop coûteux. Le cadre que nous avons développé ici permet une grande flexibilité. Il est possible de décider à l'avance de fusionner en un point de contrôle donné les traces qui avaient été distinguée en fonction d'un branchement donné (cf figure 3.11), ou alors de décider pendant l'analyse, si le nombre d'indices du recouvrement qu'il faut gérer en certains points de programmes est trop grand d'en fusionner certains, en fonction de leur ancienneté ou de leur utilité constatée.

Une autre source d'ajout de vecteurs dans les relations à représenter est la lecture des données. Il est possible de partitionner selon certaines de ces valeurs, s'il n'y en a pas trop, ou encore de distinguer selon des propriétés de départ de ces valeurs (on n'aura pas forcément une partition). On peut retarder ce genre de discrimination à l'entrée de certaines procédures seulement, et décider seulement au moment de l'analyse si le nombre de valeurs selon lesquelles distinguer les traces est raisonnable ou non. C'est ce qui est fait par exemple dans ASTRÉE pour résoudre rapidement certains problèmes de précision (Cousot et al., 2005).

3.3.3 Domaine abstrait des recouvrements de traces

Les recouvrements d'un ensemble F sont naturellement ordonnés selon leur « finesse ».

DÃ©finition 3.3 *Un recouvrement $\delta_1 : E_1 \to \wp(F)$ est dit* plus fin *qu'un recouvrement $\delta_2 : E_2 \to \wp(F)$ si pour tout $x \in E_2$, il existe $y \in E_1$ tel que $\delta_1(y) \subseteq \delta_2(x)$. On écrira $\delta_1 \precsim \delta_2$.*

En raisonnant modulo l'équivalence induite par $\precsim$, les recouvrements d'un ensemble forment un treillis complet de supremum la classe du recouvrement avec un seul indice et d'infimum la classe du recouvrement de $\wp(F) \to \wp(F)$ qui à chaque ensemble associe lui-même. On note $\mathfrak{R}$ ce treillis. Ce treillis peut servir de base à un domaine abstrait, en utilisant des techniques inspirées des domaines cofibrés (Venet, 1996). L'intérêt d'un tel domaine est double : il permet d'une part de changer *dynamiquement* le recouvrement en cours de calcul, et d'autre part d'utiliser des recouvrements infinis. Le changement dynamique de recouvrement permet de s'adapter en cours de calcul aux propriétés qu'on analyse, mais cela requiert de garder des surapproximations sûres. L'utilisation de partitionnement infini (ou très gros) permet de retarder les approximations, ce qui est toujours bon

pour la précision, mais cela requiert la définition d'opérateurs d'élargissement pour faire l'approximation en cours de calcul.

le domaine abstrait des recouvrements de traces est défini comme l'ensemble $\mathbb{D}^\sharp$ des couples (δ, f) avec $\delta \in \mathfrak{R}$ et $f \in D_T^\delta$. La concrétisation de ce domaine est :

$$\gamma_{\mathfrak{R}}(\delta, f) = \bigcup_{x \in dom(\delta)} \gamma_T(f(x)) \cap \delta(x)$$

L'ordre abstrait est induit par l'ordre d'inclusion des concrétisés : (δ, f) est plus précis que (δ', f') si et seulement si $\gamma_{\mathfrak{R}}(\delta, f) \subset \gamma_{\mathfrak{R}}(\delta', f')$. Si on veut changer de recouvrement en cours de calcul il est utile de savoir calculer le f' le plus précis tel que (δ', f') soit plus grand que (δ, f). Si on dispose d'une fonction d'abstraction α_T, ce sera tout simplement $\alpha_T^{\delta'}(\gamma_{\mathfrak{R}}(\delta, f))$, soit

$$\lambda x.\alpha_T \left(\left(\bigcup_{y \in dom(\delta)} \gamma_T\left(f(y)\right) \cap \delta(y) \right) \cap \delta'(x) \right)$$

pour chaque x de $dom(\delta')$, il faut donc prendre tous les y de $dom(\delta)$ tels que l'intersection de $\delta'(x)$ et $\delta(y)$ soit non vide. Les choses sont assez faciles si δ' est un partitionnement de δ (on rajoute des partitions à l'aide de gardes) ou dans le cas inverse (on fusionne des partitions).

Pour définir un élargissement sur ce domaine, on peut utiliser un élargissement déjà existant et un élargissement sur la base $\mathfrak{R}$. Dans ce cas, on commence par calculer le recouvrement donné par l'élargissement de la base, puis les meilleures approximations des fonctions sur ce recouvrement, finalement, on élargit ces fonctions point à point avec l'élargissement de D_T. Ce domaine est un exemple typique où on peut vouloir utiliser un ordre de construction (suivi par les itérés et l'élargissement) différent de l'ordre d'approximation. On peut par exemple vouloir partir d'un recouvrement à un indice (transparent, donc) et au fur et à mesure des calculs vouloir raffiner les recouvrement en fonction des besoins. L'ordre des calculs sera alors l'opposé de $\precsim$. On peut aussi partir d'un système entièrement partitionner et fusionner au fur et à mesure quand le coût devient trop grand. L'ordre des calculs sera alors $\precsim$. L'inconvénient de la première méthode c'est que les calculs deviennent irrémédiablement de plus en plus coûteux et la deuxième méthode peut perdre prématurément de la précision. C'est pourquoi il me semble préférable d'intégrer les fonctions dans l'élargissement de la base, de façon à avoir des élargissements qui ne se contentent pas forcément d'aller toujours dans un sens ou dans l'autre.

3.4 Applications

Toutes les structures de données exposées dans ce chapitre représentent des relations symboliques en exploitant certaines structures des données. Elles ont été choisies ou développées de façon à permettre des analyses effectives, rapides et précises. Mais leur mise en pratique pose parfois d'autres problèmes qu'il m'a été donné d'aborder en participant à l'aventure d'Astrée. J'en discute certains dans le chapitre suivant.

Chapitre 4

Le projet ASTRÉE

Entre 2000 et 2002, j'ai participé avec l'équipe Sémantique et Interprétation Abstraite au projet DAEDALUS. Il s'agissait de réunir des industriels utilisant ou développant des logiciels critiques, des industriels concevant des outils d'analyse statique et des équipes de la recherche académique pour développer la certification automatique de logiciels par analyse statique. Le rôle des équipes académiques, dont je faisais partie, était de développer les outils théoriques ou les prototypes académiques permettant aux industriels de l'analyse statique de résoudre les problèmes des utilisateurs. Il se trouve qu'AIRBUS faisait partie des utilisateurs et qu'il n'a pas été satisfait des progrès du participant chargé de prouver l'absence d'erreur à l'exécution de la commande de vol électrique des avions.

C'est donc au cours d'une visite de Patrick COUSOT chez AIRBUS, à Toulouse, que Famantanantsoa RANDIMBIVOLOLONA lui a demandé si son équipe pouvait développer un prototype montrant que l'analyse statique par interprétation abstraite pouvait résoudre son problème. Ainsi, en novembre 2001 a commencé le projet d'Analyseur Statique de logiciels Temps RÉel Embarqué (ASTRÉE [1]), dans lequel se sont embarqués avec enthousiasme Patrick COUSOT, Radhia COUSOT, Bruno BLANCHET, Jérôme FERET, Antoine MINÉ, David MONNIAUX, Xavier RIVAL et moi-même. Ce chapitre emprunte énormément aux contributions de ces collègues et amis sans lesquels le projet ASTRÉE n'aurait même jamais décollé.

Cette aventure nous aura permis de tester nos idées sur l'analyse de programme en grandeur nature. Elle aura aussi soulevé nombre de problèmes intéressants qui n'apparaissent généralement pas quand on se contente de rester dans le domaine académique.

[1] Projet soutenu en grande partie par le « Projet exploratoire ASTRÉE » du Réseau National de recherche et d'innovation en Technologies Logicielles (RNTL).

4.1 Problèmatique

4.1.1 Logiciels critiques

Le problème auquel devait faire face AIRBUS était la certification de la commande de vol électrique des avions. Ce logiciel est un exemple tout à fait typique de logiciels critiques, c'est-à-dire un logiciel dont on a identifié qu'il ne devait pas faire certaines erreurs, car les erreurs en question seraient trop coûteuses, d'un point de vue économique ou même d'un point de vue simplement humain car ils mettent des vies en jeu. Dans le cas des avions modernes[2] d'AIRBUS, la commande de vol électrique est l'interface exclusive entre les pilotes et l'avion. Par exemple, il n'y a pas de cable direct sur lequel un pilote pourrait tirer pour actionner les volets, tout passe par un processeur et des servo-moteurs. On imagine bien dans ces conditions qu'une panne totale du logiciel (qui se bloquerait dans un état d'erreur à l'exécution) aurait des conséquences catastrophiques. En revanche on peut se permettre d'attacher moins d'importance à certaines erreurs fonctionnelles qui provoqueraient simplement une plus grande consommation de carburant. La vérification de l'absence d'erreur étant très coûteuse, il est important de savoir identifier des erreurs facilement descriptibles et effectivement critiques.

Comme chacun en fait l'expérience régulièrement, il est très difficile de produire des logiciels sans erreur. Malgré cela, le gain apporté par les logiciels est tellement grand que les logiciels prennent une place de plus en plus importante. Les industriels qui conçoivent des logiciels critiques suivent des normes de développement très strictes et très coûteuses. Ces normes drastiques permettent de produire des logiciels pratiquement sans faute, mais l'expérience a montré, parfois de façon spectaculaire comme dans

[2] Il n'y a plus de lien mécanique à partir de l'A380 et l'A400M (Traverse et al., 2004)

le cas du crash d'Ariane 5.01 en 1996, qu'elles ne suffisent pas toujours. Bien sûr, le monde de l'aéronautique en a tiré les leçons sur les normes de développement, mais on en a surtout déduit que l'effort devait aussi porter sur la vérification du logiciel et non seulement sur sa méthode de production. Les méthodes classiques de vérification, par inspection manuelle ou par test, étant à la fois coûteuses et non exhaustives, AIRBUS a commencé à se tourner vers d'autres méthodes.

4.1.2 Les méthodes de vérification de logiciels

Les logiciels critiques ayant généralement pas ou peu de bugs, les besoins ne sont pas les mêmes que pour les logiciels moins critiques. Alors que pour ces logiciels une méthode permettant de trouver des bugs probables peut être précieuse, elle ne sera d'aucun intérêt pour un logiciel critique. D'une part la plupart des heuristiques utilisées par ces méthodes se fondent sur des bugs connus, identifiés et donc vraisemblablement évités par construction en suivant les normes de développement des logiciels critiques, et d'autre part ces bugs « probables » ont toutes les chances de ne pas être des bugs, mais il faudra une grande dépense de moyens pour le prouver. Finalement, ces méthodes ne couvrant pas tous les bugs possibles, même s'ils ne signalent aucun bug ils améliorent peu la confiance que l'on peut apporter aux logiciels critiques.

Méthodes industrielles classiques

L'inspection manuelle d'un programme consiste à vérifier qu'on a bien programmé ce qu'on pensait. C'est en général très insuffisant pour montrer qu'un petit programme est correct et de toute façon impossible à réaliser sur de très gros programmes.

La principale méthode utilisée dans l'industrie est donc le test. Un test consiste à exécuter le programme dans un environnement simulé sur un jeu de données d'entrée fixé. Les tests ont deux avantages principaux pour la vérification de logiciels critiques. Tout d'abord, ils sont inévitables en l'état actuel des connaissances et le resteront sans doute encore longtemps. Seuls les tests permettent de s'assurer que le programme correspond à l'ensemble de ses spécifications, au moins sur certaines données. De plus, un test ne fait jamais de fausse alerte : si un test indique que le programme est faux sur une donnée, il n'y a pas à chercher si le test s'est trompé. Il suffit de tracer le test pour voir où le comportement du programme n'a pas été correct.

Les tests n'ont hélas pas que des qualités, sinon les industriels s'en seraient contentés. Le problème principal du test est sa non-exhaustivité intrinsèque. Il n'est pas possible de tester le programme sur toutes ses entrées possibles, car ces entrées sont trop nombreuses, surtout pour des programmes temps réel qui acquièrent très régulièrement de nouvelles données. Les industriels pratiquent donc le test unitaire qui consiste à tester indépendamment de toutes petites unités de programme (typiquement moins de 100 instructions), en essayant d'atteindre tous les points de contrôle de ces unités. Ils essaient aussi de multiplier les tests pour couvrir une partie estimée suffisamment significative des entrées possibles, de façon à pouvoir s'estimer confiant dans le logiciel critique. Mais au fur et à mesure de l'évolution des technologies, les logiciels critiques croissent, et le temps de calcul nécessaire à des tests jugés satisfaisants croît beaucoup plus vite que la puissance de calcul des ordinateurs, à tel point que la méthode du test ne permet plus à elle seule d'avoir une confiance suffisante dans les logiciels critiques.

Preuves formelles

L'idée pour lutter contre cette augmentation de la taille et de la complexité des logiciels critiques est de s'aider d'ordinateurs pour prouver automatiquement leur correction. Cette approche peut être très générale, et elle peut permettre de prouver des propriétés d'une grande diversité. Elle a été testée par AIRBUS pour prouver des propriétés unitaires, qui concernent donc des petits morceaux du logiciel pris indépendamment (Souyris and Favre-Felix, 2004). Les preuves formelles consistent à vérifier automatiquement que des propriétés fournies par l'utilisateur sont vérifiées par un modèle du programme. Cette méthode souffre d'un certain nombre de difficultés, qui ne sont pas forcément insurmontables, mais qui retardent leur utilisation à une échelle industrielle.

La première difficulté rencontrée est l'expression des propriétés par l'utilisateur. Ces propriétés sont en effet exprimées dans une logique, à la syntaxe parfois particulière et il semble que la formation des ingénieurs à cette logique puisse poser des problèmes. C'est le cas en particulier, d'après nos interlocuteurs d'AIRBUS, des logiques modales temporelles, difficiles à maîtriser. Si l'expression des propriétés est complexe, alors il est possible de faire des erreurs dans leur

conception et c'est tout le processus de vérification qui est remis en question.

La deuxième difficulté consiste en l'élaboration d'un modèle pour le logiciel. Cela requiert souvent l'intervention d'un spécialiste, et il reste encore à être sûr que le modèle est bien conforme au programme. C'était par exemple un défaut du système de preuves formelles qu'AIRBUS a testé, puisqu'il prenait les flottants pour des réels, négligeant les erreurs d'arrondis qui peuvent pourtant conduire à de véritables aberrations (voir la figure 4.1, page 41). Même si le modèle est correct, il reste le problème du coût de ce modèle, qui n'est pas négligeable. Il doit être payé à chaque modification du logiciel, et maintenu avec le logiciel tout au long de sa vie. Chaque version du logiciel doit avoir son modèle. Il est parfois possible de générer automatiquement ce modèle grâce à des techniques d'analyse statique. C'est sûrement la voie de l'avenir pour cette approche.

Finalement, la plupart du temps les outils de vérifications de propriété qui doivent fournir la preuve de la propriété ne sont pas entièrement automatiques. Leur utilisation impose de recourir à un expert qui saura guider l'outil, et cet expert doit être un ingénieur spécialisé.

Analyse statique généraliste

Le but des outils d'analyse statique est de déterminer automatiquement avant l'exécution d'un programme ses propriétés. Un outil généraliste va utiliser un modèle d'un ou plusieurs langages, pour être ensuite capable d'analyser n'importe quel programme écrit dans ces langages. Ils sont généralement capables de donner des informations variées sur le programme, comme les valeurs des variables et leurs relations, les parties du programme inutilisées... L'intérêt d'un analyseur généraliste est d'une part la possibilité d'analyser n'importe quel programme du langage, le même outil peut servir pour une très grande classe d'applications, et d'autre part en général plus il saura inférer de propriétés différentes, plus il sera précis car il sera capable d'utiliser chacune des propriétés pour raffiner les autres. Un exemple typique est l'analyse de la forme des structures de données qui peut être bien plus précisément connues si on connait aussi la valeur des variables numériques qui peuvent intervenir dans le flot de contrôle.

Dans le cadre de DAEDALUS, AIRBUS a donc testé l'utilisation d'un analyseur statique généraliste, *PolySpace Verifier*. Aux yeux d'AIRBUS, il avait deux défauts principaux :

- un temps de calcul prohibitif (quelques semaines pour 100 000 lignes de codes),
- et un nombre de fausses alarmes beaucoup trop important (5% du programme).

Ces performances s'expliquent aisément par la difficulté à être capable d'analyser un langage moderne (ici du `C`) extrêmement complexe. Le temps de calcul aurait pu convenir dans une optique de certification, mais un temps de calcul plus court, de l'ordre d'une nuit, aurait permis d'utiliser l'outil en cours de développement et donc de réduire les coûts de production par une détection précoce des erreurs.

Le point le plus gênant est le nombre d'alertes. Si ce nombre est assez faible dans un cadre de logiciel normal, ici il aurait fallu montrer que les 5 000 erreurs possibles soulignées par l'analyseur n'étaient que des fausses alarmes, ce qui aurait constitué une charge beaucoup trop grosse.

Analyseurs statiques spécialisés

Selon la théorie de l'interprétation abstraite, il est possible d'obtenir des résultats plus précis en restreignant la classe des programmes qu'un outil est capable d'analyser. L'idée est relativement simple : la plupart des résultats d'indécidabilité sont prouvés par un argument diagonal. Ainsi si il est impossible de construire un programme P qui décide si un programme quelconque termine sur toutes ses entrées, c'est que sinon on arrive à une contradiction en appliquant ce programme au programme qui ne termine que si P répond non sur ce programme. On voit que cet argument n'est pas applicable si P n'est pas dans la même catégorie que les programmes qu'il analyse. Dans le cadre qui nous intéresse, les logiciels critiques suivent une norme très stricte (la norme DO178B) qui les place dans une catégorie suffisamment restreinte pour qu'il semble possible de les analyser précisément en un temps raisonnable. La société *AbsInt* avait d'ailleur montré qu'il était possible en restreignant les propriétés à démontrer aux propriété de plus grand temps d'exécution, et les architectures de compilation à un seul processeur, d'obtenir d'excellents résultats améliorant ceux obtenus par le test unitaire (Alt et al., 1996).

Ce que nous a demandé Famantanantsoa RANDIMBIVOLOLONA était donc de fournir un prototype capable d'analyser la famille des logiciels synchrones utilisés dans les commandes de vol électrique de l'A340, avec moins de 100 alertes, de façon à ce que le coût de la réduction de ces alertes en fausses alertes ne soit pas trop important pour AIRBUS. Le but de l'ana-

lyse était de montrer l'absence d'erreur à l'exécution, car ces erreurs sont critiques et elles ne nécessitent pas de spécification sophistiquée ou d'annotation du code source.

4.1.3 Quelques principes du développement d'ASTRÉE

L'analyseur ASTRÉE a donc été développé en premier lieu pour la commande de vol de l'A340, avec des contraintes de temps assez fortes, puisqu'il s'agissait de montrer la faisabilité de la technique en vue de l'exploiter pour le développement de l'A380 ou de l'A400M.

Nous avons donc choisi de développer l'analyseur par raffinements successifs. Nous sommes partis d'un analyseur très simple, utilisant un vecteur d'intervalles entiers pour approximer les valeurs de variables de tous types, un seul intervalle servant à stocker l'ensemble des valeurs possibles de toutes les cases d'un tableau donné. Puis nous avons raffiné l'analyseur en fonction des problèmes rencontrés. Parfois le raffinement était tel qu'il a été nécessaire de réécrire entièrement l'analyseur, mais cela a servi à obtenir un programme plus clair et plus efficace.

Les raffinements étaient motivés principalement par deux types de problème : les problèmes d'efficacité et les problèmes de précision. Il faut toutefois noter que l'efficacité a souvent été améliorée *en augmentant la précision* de l'analyse, car des domaines plus précis ont parfois permis d'arriver à un résultat avec moins d'itérations, et donc même si les itérations étaient plus longues, le gain final pouvait être appréciable. L'autre façon d'améliorer l'efficacité a consisté à profiler les analyses et à faire porter nos efforts d'implémentation ou d'algorithmique sur les fonctions les plus gourmandes.

Pour améliorer la précision, nous avons utilisé quand c'était possible des *traces abstraites d'exécution*. Une trace abstraite consiste en la suite des valeurs abstraites calculées par l'analyseur en cours d'itérations. L'examen de cette suite nous a toujours permis de faire remonter l'imprécision, soit à une opération abstraite, soit à un élargissement. Dans le cas des opérations abstraites imprécises, il fallait trouver la cause de l'imprécision qui menait à une alarme. Nous avons rencontré des cas où l'opération concrète pouvait être mieux approchée dans le domaine abstrait existant, mais certaines fois, il a fallu inventer de nouveaux domaines abstraits pour être capable d'exprimer un invariant nécessaire à la preuve d'absence d'alerte. Des exemples seront fournis dans la section 4.3. Il est aussi arrivé que nous améliorions la précision des opérations abstraites en faisant de meilleurs réductions entre les domaines. Cela nous a amené à développer un mécanisme sophistiqué de communication entre les domaines abstraits.

Dans les cas où l'imprécision était introduite par un élargissement, il fallait revoir l'heuristique de l'élargissement ou éventuellement retarder l'emploi de l'élargissement. Nous avons expérimenté plusieurs stratégies d'élargissements, de l'élargissement systématique à l'élargissement seulement quand aucune variable ne se stabilise en passant par différentes heuristiques.

Ce processus nous a permis d'analyser une petite partie de la commande de vol de 10 000 lignes au-delà des espoirs que nous avions au début, puisque nous avons réussi à prouver l'absence d'erreur à l'exécution en juin 2002, avec une analyse de l'ordre de la minute. Puis nous avons analysé le code complet de l'A340, avec l'ajout de certains domaines relationnels qui ont permis encore d'arriver à 0 alerte en novembre 2003. Depuis, nous analysons les différentes versions de l'A380 en cours de développement. Les programmes sont de plus en plus gros (jusqu'à 700 000 lignes de code, ce qui nous prend 26 heures pour 0 fausse alerte), et nos analyses ont permis de trouver quelques erreurs en cours de développement. Ce processus a été extrêmement enthousiasmant, en nous fournissant en permanence de nouveaux défis et la satisfaction de les résoudre.

En l'état actuel de l'analyseur, avec toutes les adaptations que nous avons dû apporter pour améliorer l'efficacité et la précision de l'analyse, il me semble qu'ASTRÉE, avec ses 90 000 lignes de Caml, est à nouveau mûr pour une refonte totale qui devrait nous faire encore gagner de la clarté et de l'efficacité.

4.1.4 Description des codes analysés par ASTRÉE

Les logiciels sur lesquels AIRBUS (en la personne de Jean SOUYRIS) fait passer ASTRÉE sont suffisamment spécifiques pour permettre des optimisations conséquentes, mais ils ont aussi des caractéristiques qui font de l'analyse précise et rapide un vrai défi. Ces logiciels sont générés automatiquement depuis un langage graphique, SCADE ou SAO. Le source produit est du `C` synchrone avec les restrictions suivantes :

Restrictions

Pour l'instant, le logiciel critique ne contient pas les points suivants :
- allocation de mémoire dynamique,
- types `union`,
- fonctions récursives,
- branchements en arrière,
- effets de bord conflictuels,
- appels aux librairies `C`.

Cela signifie que nous n'avons pas cherché à être capables d'analyser ces points à priori. Cela pouvait faire gagner du temps de calcul ou parfois simplement du temps de programmation, car notre équipe est assez petite et ne programme pas Astrée à plein temps. C'est le cas par exemple des fonctions récursives. En l'état, Astrée boucle si on lui présente une fonction récursive[3], pourtant la gestion des fonctions récursives ne nous poserait aucun problème particulier. Les branchements en arrière (avec des `goto`) demanderaient une modification de notre itérateur (mais pas des domaines abstraits). Les appels aux librairies demanderaient d'écrire un descriptif formel de l'effet de ces librairies.

Enfin, Astrée nous sert aussi pour tester certains de nos travaux théoriques, ce qui nous permet d'avoir des résultats plus rapides et plus valables qu'avec un prototype académique. De nouveaux utilisateurs demandent aussi à utiliser Astrée sur leur code qui peut comporter de nouvelles caractéristiques. C'est ainsi qu'Antoine Miné a récemment rajouté un modèle mémoire plus précis, capable de gérer précisément l'arithmétique de pointeurs et les structures avec des `union` (Miné, 2006a).

Ce qu'il reste de difficile à gérer

Les restrictions laissent encore le champ libre à beaucoup de points difficiles du langage `C`, certains utilisés de manière intensive dans le logiciel critique. Les points non triviaux sur lesquels Astrée est assez précis :
- les tests et les boucles,
- les branchements en avant,
- les pointeurs (y compris sur des fonctions),
- les structures et les tableaux,
- les calculs flottants.

Le problème des boucles est réglé depuis longtemps dans le cadre de l'interprétation abstraite par l'utilisation d'élargissements, mais il a fallu trouver les stratégies les plus efficaces pour arriver à un résultat précis.

[3]Seulement si le nombre potentiel d'appels récursifs n'est pas borné.

```
int main () {
  double x; float y, z, r;
  x = 1125899973951488.0;
  y = x + 1;
  z = x - 1;
  r = y - z;
  printf("%f\n", r);
}
```

Fig. 4.1 – Exemples d'erreurs d'arrondi : le programme affiche `134217728.000000` et non `2`.

Les calculs flottants sont un vrai problème souvent mal traité en analyse statique. Le problème est qu'on ne peut pas les considérer comme des réels, car il faut tenir compte des erreurs d'arrondi qui sont encadrées par la norme IEEE 754 (IEEE Computer Society, 1985) mais dont les résultats peuvent être surprenants, même en l'absence de boucle de rétroaction. Si on considère le programme de la figure 4.1, on voit que dans les flottants on n'a pas toujours $(x + 1) + (x - 1) = 2$. Sur l'exemple, on trouve 134217728, mais on pourrait aussi bien trouver 0 avec x plus grand.

Finalement, la structure même des logiciels critiques que nous avons eu à analyser peut poser problème, en raison d'une part de la taille de ces programmes et d'autre part de leur génération automatique :
- gros code (de 70 000 à 700 000 lignes de `C`),
- beaucoup de variables globales (jusqu'à 35 000, hors tableaux et structures),
- une grosse boucle avec beaucoup d'interdépendances,
- flot de contrôle non local utilisant des booléens.

Ce dernier point signifie qu'on ne peut souvent pas se contenter d'analyses locales, et que des relations doivent être maintenues entre des variables calculées en des points très éloignés.

Le nombre de variables globales oblige à manipuler de très gros invariants. Nous avons essayé des analyses de vivacité ou de dépendance pour essayer de limiter le nombre de variables à considérer en chaque point, mais la forme du programme en grosse boucle rétroactive ne permet pas d'utiliser ces idées.

Les erreurs détectées par Astrée

Astrée devait au départ détecter les erreurs à l'exécution, c'est à dire les opérations qui conduisent à la levée d'exception dans le mode d'exécution choisi par Airbus. Cela comprend :

- les opérations flottantes invalides ($\sqrt{-1}$, $0/0$),
- les générations de flottants infinis,
- les divisions par 0,
- les accès mémoires incorrects (en dehors des bornes d'un tableau par exemple).

Pour détecter ces erreurs, il faut bien entendu définir très précisément la sémantique des programmes que nous analysons. Nous avons réalisé qu'ASTRÉE devait tenir compte de plusieurs aspects de la sémantique. La première, impérative est la norme du `C`. Mais cette norme permet souvent des libertés d'implémentation. Par exemple la norme ne précise pas dans quel ordre les arguments d'une fonction doivent être évalués. Elle ne précise pas non plus ce qu'il se passe en cas de dépassement de capacité des entiers, ou quelle doit être la valeur du plus grand entier représentable. Si l'analyse devait tenir compte de toutes les implémentations possibles, elle devrait considérer des choix non-déterministes, ce qui risquerait à la fois d'être coûteux et surtout imprécis. L'information du compilateur utilisé permet de fixer ces points, et dans certains cas ASTRÉE est prévu pour pouvoir être modifié en fonction de ces choix.

Finalement, certains choix du compilateur, comme par exemple le fait d'implémenter une arithmétique modulo en cas de dépassement de capacité entière, ne correspondent pas forcément aux intuitions qu'avaient les personnes qui ont programmé en utilisant les entiers. C'est pourquoi, suivant les désirs de l'utilisateur, nous avons ajouté la possibilité de détecter les erreurs suivantes :

- dépassement de capacité pour certains entiers (par défaut les entiers signés, mais pas les non-signés),
- violation d'assertions.

Les assertions peuvent porter sur n'importe quelle expression `C` valide en ce point de programme, ce qui donne une grande liberté de spécification pour des erreurs spécifiques à certains utilisateurs. L'environnement est aussi connu par l'utilisateur et il est souvent indispensable qu'ASTRÉE possède ces renseignements pour pouvoir conclure à l'absence d'erreur. L'idéal serait de pouvoir disposer d'un modèle de l'environnement physique avec lequel le système embarqué interagit (Cousot, 2005), mais en l'état actuel, ASTRÉE n'utilise que des bornes sur les entrées analogiques discrétisées lues de manière asynchrone, et cela semble suffire pour l'instant. Ces données, ainsi que le temps d'exécution maximum (correspondant au plus long temps de vol possible de l'avion) sont précisés dans un fichier de configuration ou au choix dans le code source.

4.2 La structure générale d'ASTRÉE

4.2.1 Options et paramètres

La première étape lorsqu'on exécute ASTRÉE consiste à lire les arguments qui lui sont passés, et ce n'est déjà pas une mince affaire. À l'heure où ces lignes sont écrites, ASTRÉE admet 202 options possibles, en plus de la liste des fichiers `C` à analyser. La plupart de ces options sont directement issues du processus de développement d'ASTRÉE par raffinement. L'idée sous-jacente est que chaque modification de domaine ou chaque nouveau domaine nécessaire pour une famille de programmes ne le sera pas forcément pour d'autres familles. En laissant le choix au lancement d'utiliser tel ou tel domaine, on peut obtenir des analyses efficaces pour chaque famille. Au sein d'un même domaine, le choix de paramètres peut être fait par un utilisateur averti pour adapter la précision de l'analyse pour une famille de programmes. Les options permettent aussi l'affichage de différents résultats de l'analyse, comme l'invariant en tête de boucle, ou un sur-ensemble des valeurs d'une variable, ce qui peut aider à déterminer si une alarme provient d'une erreur dans le code ou d'une imprécision de l'analyse. Pour s'y retrouver dans ce foisonnement d'options, Antoine MINÉ a développé une interface graphique qui regroupe les options et en donne une vue synthétique.

Les paramètres *sémantiques* de l'analyse, tels que le comportement de l'environnement (pour l'instant seulement des intervalles de valeurs pour les acquisitions de données), ou le temps maximal d'exécution, peuvent être regroupés dans un fichier de configuration. Cela permet d'analyser un code source sans le modifier. Mais il est aussi possible de mettre toutes les informations dans le même fichier, en utilisant des directives spécifiques.

Ce mécanisme de directives permet de donner des indications à ASTRÉE bien plus précises que les options puisqu'elles peuvent être liées à un point de programme donné. Il permet de faire référence facilement à des variables locales. Il a été utilisé pour tester des domaines abstraits (voir par exemple les packs de variables 4.4.3). Ces directives sont restées, car elles peuvent permettre de tester de nouvelles stratégies pour l'analyse, puis pour un utilisateur d'automatiser ces stra-

tégies, par la génération automatique des directives qui peut s'effectuer avec n'importe quel langage de manipulation de texte.

4.2.2 Les différentes étapes

La première étape d'une analyse par ASTRÉE est la même que pour un compilateur : les fichiers comportant le programme (plus éventuellement un fichier de configuration sémantique) sont parsés, et un arbre abstrait est construit pour chaque fichier. Ensuite, les arbres sont fusionnés en un seul arbre qui correspondrait à un seul fichier. Cet arbre est finalement décoré par des informations de type, de taille des données (selon l'architecture sur laquelle le programme doit être compilé) et de table des symboles. La fusion des arbres abstraits est un point faible d'ASTRÉE, elle n'est pas aussi générale qu'une véritable édition de liens. Cette faiblesse n'est pas un handicap pour analyser les programmes analysés pour AIRBUS. La méthode correspond à la voie la plus rapide pour arriver au résultat en terme de développement d'ASTRÉE. À terme, lorsque nous aurons plus de temps, il faudra réécrire cette phase.

L'arbre décoré permet déjà de faire des analyses peu coûteuses qui simplifient les analyses suivantes ou qui les guident. ASTRÉE propage les constantes (section 4.4.1), détermine des groupes de variables pour lesquelles un invariant relationnel peut être intéressant (section 4.4.3), les zones du programme sur lesquelles il peut être intéressant de partitionner l'analyse (sections 3.3.2 et 4.4.2). ASTRÉE calcule un graphe de dépendance des variables et vérifie aussi l'absence d'effets de bords qui pourraient fausser l'analyse.

La dernière transformation consiste à mettre à plat les expressions de façon à bien extraire le flot de contrôle et permettre des analyses qui suivent le flot de contrôle ou qui le remontent. Sur ce graphe, ASTRÉE va ensuite calculer les invariants grâce à un itérateur abstrait (section 4.2.3). Puis ces invariants sont utilisés pour vérifier l'absence d'erreur à l'exécution du programme. Il est aussi possible d'explorer ces invariants à l'aide d'une interface graphique développée par Antoine MINÉ. Dans un futur proche, il devrait être aussi possible d'utiliser ces invariants pour analyser en arrière et trouver les causes des alarmes automatiquement (Rival, 2005).

4.2.3 L'itérateur

Le cœur de l'analyseur est l'itérateur qui permet de découvrir les invariants précis permettant de trouver les erreurs ou de prouver leur absence. Suivant les techniques de l'analyse statique par interprétation abstraite (Cousot and Cousot, 1977a), l'itérateur calcule une approximation de la suite des ensembles d'états possibles en chaque point de programme. Cette suite des états possibles se contruit comme l'ensemble vide en tout point de programme, sauf au départ de la fonction principale où toutes les variables sont initialisées à l'ensemble de leurs valeurs possibles. Le passage d'un point de la suite à un autre se fait en propageant tous les états des points de programme vers les suivants en suivant les fonctions de transition du programme.

En général, une bonne manière d'approximer cette itération consiste à stocker pour chaque point de contrôle un état abstrait qui représente un ensemble d'état, puis à suivre des itération chaotiques avec élargissement (Cousot and Cousot, 1977b; Cousot, 1999). Mais cette approche n'est pas réaliste dans l'analyse de programme avec plusieurs dizaine de milliers de variables globales et plusieurs centaines de milliers de points de contrôle, car pour l'instant les ordinateurs n'auraient pas assez de mémoire, même pour une simple analyse d'intervalles. Il est possible de ne stocker qu'un faible nombre d'environnements abstraits en utilisant une stratégie d'itération récursive, qui consiste à suivre un ordre topologique faible (Bourdoncle, 1993) du graphe de flot de contrôle et à calculer des points fixes locaux sur les boucles internes du graphe puis sur les boucles de plus en plus externes. L'itération revient donc à suivre le flot de contrôle du programme en ne stockant des environnements abstraits que pour le point de contrôle courant c et les point de contrôle où l'on fait les élargissements sur les boucles contenant c. Au maximum, le nombre d'invariants abstraits stockés à un instant donné de l'itération est donc un plus le nombre maximum d'imbrication de boucles dans le graphe de flot de contrôle.

Avec les limitations actuelles d'ASTRÉE (absence de récursion et de `goto` en arrière), les seules boucles sont les boucles `while`, et on choisit naturellement comme point d'élargissement les têtes de boucle `while`. On pourrait faire moins d'itérations en gardant les environnements abstraits en chaque tête de boucle (car ils sont *a priori* plus proches du point fixe), mais cela n'a pas été testé. Le coût risque en effet d'être élevé, même s'il n'est pas ingérable : pour

le plus gros programme sur lequel ASTRÉE a été utilisé avec succès (700 000 lignes), on trouve 1 256 boucles whiles et un niveau d'imbrication maximal de 2 boucles.

En suivant cette stratégie d'itération, on atteint tout de suite un invariant tant qu'on n'entre pas dans une boucle, et ASTRÉE vérifie alors l'absence d'erreur à l'exécution en même temps que le calcul de l'invariant. Pour les boucles, ASTRÉE calcule d'abord l'invariant, puis cet invariant est propagé dans le corps de la boucle en testant l'absence d'erreur à l'exécution. En cas d'erreur à l'exécution, la norme ne définit souvent pas ce qui se passe, et pour l'utilisateur ces cas ne doivent de toutes façons jamais arriver, donc si on en détecte un, il importe peu de savoir si il y en a d'autres qui découlent de cette erreur. ASTRÉE considère donc une erreur à l'exécution comme la fin du programme, *pour les valeurs abstraites considérées comme erronées*. Cela signifie par exemple qu'en cas de dépassement de valeur entière, ASTRÉE va signaler l'erreur et poursuivre l'itération asbtraite avec seulement les entiers inférieurs ou égaux au plus grand entier représentable en machine. Comme ces états d'erreur sont considérés comme indésirables, ou inutiles, ils sont en fait supprimés *pendant* la phase de recherche d'invariant. Cela signifie que l'invariant est recherché en faisant l'hypothèse que tout se passe bien, ce qui limite les cas possibles, puis pendant la phase de test d'erreur, on utilise cet invariant pour déterminer si cette hypothèse était correcte. Cette approche est correcte car elle va toujours au moins signaler la première erreur possible, puisque pour cet erreur, l'état en tête de boucle avant qu'elle se produise est un état sans erreur. On peut ainsi guider l'analyse avec des assertions, qui vont servir à calculer des invariants précis et qui peuvent être de plus vérifiés par l'analyse. Il faut toutefois être conscient qu'on ne peut considérer qu'une assertion a été prouvée par ASTRÉE que si l'analyse produit zéro alerte.

Pour gérer les appels de procédures, ASTRÉE utilise une stratégie d'appel par copie, qui consiste à distinguer les procédures selon l'endroit d'où on les appelle. Cette stratégie conduirait ASTRÉE à boucler sur des procédures récursives. Elle permet en général d'être plus précis mais au prix d'une disjonction sur les états d'un point de contrôle des fonctions semblable à ce qu'on obtient par un partitionnement selon le point d'appel des fonctions (voir section 3.3.2). Cette stratégie est très efficace sur les codes qu'ASTRÉE analyse actuellement, car les seules fonctions appelées très souvent sont de petites fonctions. On pourrait adapter cette stratégie au cas récursif en définissant des recouvrements finis des ensembles de points d'appels et en associant à chaque point de discrimination un point d'élargissement.

Dans les programmes de contrôle fournis par AIRBUS, les premiers tours de boucle sont consacrés à l'initialisation du système, puis le système rentre en régime permanent. ASTRÉE est très précis sur ce code en distinguant un certain nombre de premiers tours de boucle des suivants. L'itération consiste alors à faire ces premiers tours de boucle les uns à la suite des autres, sans union avec les états précédents, puis à calculer un invariant qui ne sera valable que sur les tours de boucles supérieurs. Cet invariant sera ensuite utilisé pour tester l'absence d'erreur dans la boucle pour les tours de boucles finaux. L'état abstrait en sortie de boucle sera l'union des tests de sortie de chaque tour de boucle et du test de sortie appliqué à l'invariant de fin de boucle. Beaucoup de boucles internes petites (moins d'une centaine de tours) peuvent ainsi être dépliées et analysées très précisément. Cette technique est à nouveau un cas particulier de discrimination de traces : c'est une discrimination selon le tour de boucle, limité au paramètre de dépliage, avec fusion en fin de boucle.

4.2.4 Convergence

Pour assurer la convergence de l'itération en un temps raisonnable, il est nécessaire d'avoir recours à des élargissements (Cousot, 1978). On peut commencer par noter qu'ASTRÉE doit principalement faire face à deux types de comportement « infini » : d'une part l'infini de la taille des nombres (pour avoir une analyse qui termine en temps raisonnable, on raisonne comme si les entiers pouvaient prendre toutes les valeurs de $\mathbb{N}$ et on cherche à ce que l'analyse termine toujours), et d'autre part l'infiniment petit de la précision des calculs (cette fois, on considère les flottants comme des réels). Ce deuxième cas, assez rarement pris en compte en analyse statique, correspond à l'analyse d'une variable qui tend vers une valeur réelle, et dont la convergence peut être extrêmement longue. En pratique, ce problème de convergence est plus facile à résoudre que le cas de l'infiniment grand, car les équations physiques modélisées par le programme analysé sont faites pour converger. En pratique, en effet, dans tous les cas de ce style que nous avons observé, la limite est un attracteur. Pour résoudre le problème, il suffit donc d'augmenter légèrement la valeur de la variable pour atteindre un

post point fixe. Cette technique de perturbation des calculs flottants est aussi utilisée pour résoudre les problèmes d'erreur de calcul du post point fixe dans les domaines abstraits utilisant des flottants (Miné, 2004a).

Pour le problème de l'infiniment grand, les élargissements relèvent souvent de l'heuristique, et chaque domaine abstrait peut avoir sa stratégie, indépendamment des autres domaines. Toutefois, un certain nombre de critères sont appliqués sur (presque) tous les domaines abstraits. Ces critères sont appliqués en envoyant soit un ordre d'union, soit un ordre d'élargissement aux domaines abstraits. On voit facilement que si après un certain temps, tous les domaines abstraits font des élargissements pour tous les points d'élargissement, l'analyse terminera (en temps raisonnable). Dans un premier temps, il apparaît qu'il est préférable d'attendre que l'itérateur ait effectué plusieurs tours de boucle, collectant un nombre suffisant de comportements, pour commencer à approximer en utilisant des élargissements. Ensuite, il arrive que certaines variables interdépendantes ne se stabilisent pas en même temps. Dans ce cas, un élargissement pourrait obtenir de très grosses valeurs là où de simples unions donneraient de bon résultats. Astrée garde donc des anciennes valeurs de variables, et si une variable s'est stabilisée, on impose une union. Ce critère est naturellement ignoré à partir d'un certain nombre d'itérations.

De la même façon, il est généralement préférable de ne pas élargir si l'analyse découvre une nouvelle branche du flot de contrôle (un test qui était par exemple toujours vrai devient vrai ou faux en rajoutant des états par exemple) (Cousot, 1999). Ce critère n'est pas encore implémenté dans Astrée mais il a l'avantage de ne pas mettre en cause la terminaison de l'analyse, et il semble qu'il puisse améliorer notablement la précision.

4.3 Les domaines abstraits

En interprétation abstraite, le domaine abstrait est à la fois l'ensemble des propriétés représentables par l'abstraction et les fonctions de transfert entre ces propriétés. Quand on dispose d'un domaine abstrait effectif, on peut calculer les itérés approchés : il suffit de traduire chaque expression de l'ensemble d'équations à résoudre en une composition de fonctions de transfert abstraits pour trouver un itéré abstrait à partir du précédent. Dans le cas d'Astrée, il s'agit de traduire les fonctions de transfert concrètes du graphe de flot de contrôle en directives qui sont ensuite interprétées par le domaine abstrait. Mais pour être précis, Astrée doit être capable de représenter des catégories variées de propriétés. Il est plus simple d'un point de vue du développement de programmer plusieurs domaines abstraits simples (en tout cas pas trop compliqués) et d'utiliser un état abstrait qui est la conjonction des propriétés représentées par chaque domaine abstrait. C'est pourquoi Astrée utilise en fait plusieurs domaines abstraits.

4.3.1 Le domaine des intervalles

Le premier de ces domaines est aussi le plus simple. C'est le domaine des intervalles (Cousot and Cousot, 1977a), qui consiste à associer à chaque variable de l'environnement abstrait une borne inférieure et une borne supérieure (ou alors, si l'environnement abstrait est vide, on n'associe rien aux variables). Ce domaine très simple nous a déjà permis de mettre au point des techniques utiles pour tous les domaines abstraits qui ont suivi. Ce domaine est toujours utilisé dans Astrée, car c'est le domaine des propriétés qui nous intéressent pour les dépassements de capacité entière ou flottante. Certaines opérations abstraites qui ne sont correctes qu'à condition de ne pas faire de dépassement utilisent les calculs sur les intervalles pour éviter les cas interdits.

Calculs flottants

Au départ, un élément de ce domaine était implémenté comme un tableau de couples d'entiers, chaque indice du tableau correspondant à une valeur de l'environnement abstrait. Cela posait bien entendu des problèmes de précision pour les valeurs flottantes. Il était par exemple impossible de montrer qu'une division par un flottant compris entre 0.5 et 1 ne produisait pas de valeur interdite (division par zéro). Pour obtenir une précision suffisante sans compromettre les performances de l'analyse, il a fallu utiliser des flottants (des rationnels auraient été trop coûteux en temps et en mémoire). Cela a posé le problème de l'arrondi des calculs, sachant que la norme autorise différents modes d'arrondi. Pour couvrir tous les cas, nous avons choisi l'arrondi systématique vers l'infini (au plus grand pour les nombres positifs et au plus petit pour les négatifs). Ce mode d'arrondi a été gardé pour tous les autres domaines abstraits pour assurer la correction de l'analyse par rapport à tous les modes d'arrondi.

Partage et structures de donnée fonctionnelles

La structure de tableau s'est montrée trop inefficace en pratique : en effet, avec un invariant de dizaines de milliers d'intervalles propagé sur des centaines de milliers de points de contrôle, on ne peut pas se permettre des fonctions de transfert linéaire par rapport à la taille de l'invariant. Avec un tableau, il faut pourtant recopier tout l'environnement abstrait à chaque test et parcourir tous les éléments des deux tableaux à chaque union, au moment où les tests se rejoignent. En fait, en général, seules de petites parties de l'environnement diffèrent. Nous avons donc développé une structure d'arbre binaire de recherche avec des opérateurs binaires dont le comportement par défaut est de renvoyer le même résultat que les arguments si les arguments sont égaux. Ainsi, si deux arbres binaires de recherche partagent les sous-arbres communs, le coût d'une opération binaire sera au plus le logarithme de la taille de l'invariant fois le nombre de variables différentes dans les deux arbres. Cela a permis à la fois un gain de temps considérable, mais aussi un gain de mémoire grâce au partage des données qui ne changent pas. De plus, cette représentation est purement fonctionnelle, c'est-à-dire que l'arbre n'est jamais modifié, ce qui facilite grandement le développement de la parallélisation d'ASTRÉE (Monniaux, 2005). Cette structure de donnée a ensuite été reprise pour tous les domaines abstraits qui manipulent des ensembles de valeurs abstraites. David MONNIAUX a aussi essayé d'utiliser des arbres de Patricia (Morrison, 1968), mais les résultats n'ont pas été concluants.

Élargissement

L'élargissement classique pour les intervalles consiste à comparer les bornes des intervalles de deux itérés successifs, et si une des bornes grossit, on la remplace par l'infini (en fait, on supprime la contrainte). En général, si cet élargissement est grossier, il est facilement rattrapé par l'application d'un opérateur de rétrécissement (Cousot and Cousot, 1977a). Dans ASTRÉE, cette opération consiste simplement à propager le post point fixe et à intersecter le post point fixe propagé avec le précédent. Cette opération est itérée un nombre fini de fois, décidé au lancement de l'analyse. Cela permet en pratique de borner une variable par une autre (ou en tout cas par la borne supérieure de l'autre), alors même que le domaine des intervalles n'est pas relationnel. Si on prend par exemple le premier programme de la figure 4.2, la suite des itérés pour `x` en tête de boucle donne $[0,0]$, $[0,1]$, $[0,2]$, etc. Si la borne supérieure de `y` est assez grande, l'élargissement donne $[0,\infty]$ pour `x`. En propageant ce post point fixe d'un tour de boucle, on obtient un intervalle compris entre 0 et la borne supérieure de `y`. Mais dans le second programme, la propagation donne soit cet intervalle si `b` est vrai, soit $[0,\infty]$ si b est faux. L'invariant est l'union de ces deux intervalles si l'analyse a déterminé que `b` pouvait être vrai ou faux, ce qui signifie que le rétrécissement est inutile ici. Il est possible d'être plus précis si on pense connaître *a priori* un majorant pour `y`. Il suffit d'*étager* l'élargissement (Blanchet et al., 2002) en rajoutant des étapes, par exemple ici le majorant supposé de `y`. La première étape d'élargissement qui amène la borne supérieure de `x` à ce majorant est alors un post point fixe, et l'analyse trouve donc que `x` est plus petit que ce majorant (à condition que `y` soit bien plus petit que le majorant supposé). ASTRÉE peut ainsi utiliser un ensemble de valeurs spécifiées par l'utilisateur pour ses élargissements étagés. Une autre solution pour avoir un étage spécifique à une variable est aussi d'utiliser une assertion qui permettra au rétrécissement d'arriver à un bon invariant. Dans l'exemple de la figure 4.2, on peut ajouter `assert(x<k);` où `k` est un majorant de `y` juste après le **while**(1){. À noter que cette possibilité n'est pas utilisée à l'heure actuel, car les utilisateurs souhaitent n'apporter aucune modification au code analysé. Toute modification entrainerait en effet d'importants coût de recertification du processus de production du code.

```
x = 0;
while(1) {

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

}
```

```
x = 0;
while(1) {
  if(b) {
    x++;
    if(x>y) x=0;
  }
}
```

FIG. 4.2 – Deux programmes au comportement différent vis-à-vis du rétrécissement

4.3.2 Autres domaines pour les ensembles d'états

Bien évidemment le domaine des intervalles n'a pas suffit à lui tout seul pour prouver l'absence d'erreur à l'exécution des programmes

d'Airbus. Chaque source d'imprécision qui ne pouvait se résoudre avec les domaines précédents a motivé le développement d'un nouveau domaine, qu'Astrée peut utiliser ou non en fonction des souhaits de l'utilisateur. Pour faciliter l'insertion de nouveaux domaines, nous avons défini une interface commune à tous ces domaines. Cette interface contient bien sûr les signatures des fonctions abstraites nécessaires pour analyser du `C`, mais aussi les fonctions nécessaires pour réaliser le produit réduit entre les domaines utilisés par l'analyse. Le produit réduit complet (Cousot and Cousot, 1979b) entre tous les domaines abstraits utilisés par une analyse consisterait à garder dans chaque domaine l'information la plus précise compte-tenu des informations calculées par les autres domaines. Une telle précision serait difficile à atteindre, et Astrée implémente une version approchée de cette réduction à l'aide de fonctions de communication entre les domaines. Les informations communiquées peuvent être à la fois des contraintes (intervalles, bornes de valeurs absolues, égalité de variables, ...) mais aussi des demandes aux autres domaines (demandes si deux variables sont sûrement égales ou inégales, demandes de calculs précis sur une variable, de regroupements de variables, ...). La composition de deux domaines abstraits se fait aisément à l'aide d'un foncteur qui transmet aux domaines les ordres de fonctions de transfert (émanant de l'itérateur) et les contraintes, et qui collecte les demandes de chaque domaine.

Les domaines actuellement disponibles dans Astrée sont les suivants :

Linéarisation et propagation symbolique

La plupart des domaines abstraits sont bien plus précis sur les expressions linéaires. Un domaine est donc chargé d'associer aux expressions une version linéaire en transformant certaines variables (en général celles de plus petite amplitude de variation) en intervalles. Un autre domaine associe à chaque variable la dernière expression à laquelle elle a été affectée et propose le remplacement des variables par ces expressions. Cela peut souvent faire gagner de la précision. Pour ces deux domaines, la principale difficulté est la gestion correcte des erreurs d'arrondi (Miné, 2004a).

Progressions arithmético-géométriques

Les calculs menés dans des programme de contrôle/commande, même si ils sont stables dans les réels peuvent accumuler des erreurs d'arrondi. L'influence de ces erreurs d'arrondi peut être maîtrisée si on impose que le programme ne tourne pas en continu et qu'il soit régulièrement réinitialisé. C'est naturellement le cas des avions. Dans ce cas, on peut utiliser un invariant lié au nombre de tours de boucle[4] qui se trouvera naturellement être une borne de la valeur absolue par une progression arithmético-géométrique (i.e. l'itération d'une fonction affine). Ce domaine, développé par Jérôme Feret (Feret, 2005b), permet aussi de remplacer un premier domaine que nous avions développé et qui associait à chaque variable l'intervalle de sa différence avec le nombre de tours de boucle et de sa somme avec le nombre de tours de boucle. Ce domaine permettait de borner des variables qui comptent le nombre d'occurences d'un évènement extérieur. Le domaine des progressions arithmético-géométriques est bien plus général et pas plus coûteux.

Valeurs absolues, égalités et inégalités

Pour appliquer le domaine des progressions arithmético-géométriques à des cas de rétroaction non triviaux, plusieurs domaines simples ont été développés. L'un stocke les classes d'égalité des variables, un autre les inégalités, et encore un les classes d'égalité au signe près. Ces domaines permettent ainsi d'alléger la tâche des autres domaines qui peuvent faire des demandes concernant telles variables d'intérêt et ainsi utiliser des informations relationnelles spécifiques.

Ellipses et filtres expansés

Les programmes de contrôle-commande pour lesquels Airbus voulait prouver l'absence d'erreur à l'exécution contiennent un certain nombre de filtres numériques dont les invariants sont mal approximés par les domaines classiques. Par exemple, pour un filtre du second ordre, qui calcule sa sortie comme une fonction linéaire de l'entrée courante et de la sortie du filtre aux deux étapes précédentes, la forme de l'invariant sera naturellement contenue dans une ellipse, et aucun domaine composé de relations linéaires entre les variables ne pourra fournir d'invariant stable. Un domaine simple des ellipses est utilisé dans

[4] En fait, dans les programmes synchrones, il se produit des interruptions par une horloge, qui se produisent nécessairement pendant des phases d'attente de l'interruption. Astrée compte donc le nombre de telles phases, qui sont directement liées au temps pendant lequel le programme tourne.

ASTRÉE (Feret, 2004), et un domaine en dimension quelconque est en cours de développement par Radhia COUSOT. Ce dernier constituerait un domaine relationnel générique très intéressant pour les variables flottantes. Pour revenir aux filtres, il est possible d'être très précis en expansant les affectations des filtres de façon à tenir compte plus précisément des dernières sorties de filtre. Cette expansion doit bien entendu tenir compte des erreurs d'arrondi. Les erreurs d'arrondis et le reste de l'expansion du filtre sont approximées par des ellipses. Un tel domaine nécessite de connaître le filtre et de fournir à l'analyse la forme des invariants qu'il produit, sous forme symbolique. Il est ainsi possible d'analyser très précisément tout filtre linéaire (Feret, 2005c).

Octogones

Les octogones sont un domaine abstrait développé par Antoine MINÉ permettant de représenter efficacement des invariants sur un ensemble de variables $(x_i)_{i<n}$. Ces invariants décrivent un intervalle pour chaque expression $x_i \pm x_j$. Les octogones permettent ainsi de manipuler des informations relationnelles entre variables sans avoir à écrire un domaine spécifique à telle ou telle partie de programme. Bien que peu coûteux comparés aux autres domaines abstraits relationnels, comme les polyèdres (Cousot and Halbwachs, 1978), ils occupent une place de taille quadratique par rapport au nombre de variables liées, ce qui est bien trop pour le style de programme analysé par ASTRÉE. Les variables sont donc décomposées en petits groupes (voir section 4.4.3) et le domaine manipule un ensemble de petits octogones (reliant en moyenne 3 à 4 variables). Cette décomposition en petits groupes serait aussi profitable aux polyèdres, les rendant peut-être utilisables par ASTRÉE, mais les octogones ont un avantage décisif sur les représentations de polyèdres, à savoir la possibilité d'utiliser des flottants dans les contraintes tout en restant correct vis-à-vis des erreurs d'arrondi (Miné, 2006b). Au final, les octogones ont permis d'améliorer considérablement la précision des analyses pour un coût raisonnable (Blanchet et al., 2003).

Arbres de décision

Le programme utilisant des booléens pour stocker des conditions du flot de contrôle, il a été nécessaire d'utiliser un domaine d'arbres de décisions tel que décrit dans la section 3.1.3 (page 26). Ce domaine utilise l'interface commune aux autres domaines abstraits pour utiliser une combinaison quelconque de domaines abstraits aux feuilles des arbres de décisions. Comme ce domaine a un coût exponentiel en fonction du nombre de variables booléennes utilisées, il utilise aussi des groupes de variables (section 4.4.3) en limitant le nombre de variables booléennes par groupe. Deux utilisations ont été testées, une avec très peu d'arbres de décision et tous les autres domaines abstraits aux feuilles et l'autre avec beaucoup d'arbres et des domaines simples aux feuilles (intervalles avec éventuellement des propagations symboliques). L'effet d'un arbre de décision avec tous les autres domaines aux feuilles revient à partitionner l'analyse selon les valeurs des booléens (Cousot and Cousot, 1979b). On notera que cette approche est différente des discriminations de traces selon la valeur de ces booléens à un instant donné. Cette approche semble trop coûteuse pour les plus gros programmes industriels. Les résultats pratiques de l'utilisation des petits arbres sont décrits dans la section A.4.

Discrimination de traces

La discrimination de traces telle que décrite dans la section 3.3.2 (page 34), avec possibilité d'utiliser comme critère certains branchements du programme ou dynamiquement les valeurs de variables. Ce domaine a permis de résoudre plusieurs problèmes de précision à moindre frais de développement et avec une complexité tout à fait acceptable (section A.5). Les exemples ayant justifié l'utilisation de ce domaine ont été utilisés pour insérer automatiquement des directives de discrimination de traces (section 4.4.2).

4.3.3 Les modèles mémoire

Tous les domaines de la section précédente manipulent un environnement abstrait composé de variables scalaires supposées indépendantes. Les codes analysés par ASTRÉE contiennent aussi des tableaux, des structures et même des pointeurs. La première approche a consisté à associer à chaque variable du programme une unique variable abstraite dont les valeurs représentent l'ensemble des valeurs possibles des scalaires atteignables par la variable du programme. Pour une analyse correcte, il faut bien sûr traîter les affectations à ces variables différemment des variables abstraites qui ne correspondent qu'à une seule variable concrète, mais l'analyse était très rapide. Malheureusement la précision n'était

pas suffisante et nous avons ensuite introduit un mécanisme d'éclatement qui permet d'associer aux tableaux ou structures pas trop gros (la limite de taille est un paramètre de l'analyse) autant de variables abstraites que de scalaires, permettant ainsi des affectations plus précises.

Ce modèle mémoire très simple faisait l'hypothèse d'absence d'alias entre les valeurs abstraites et ne permettait pas d'analyser des codes contenant des types `union`. Antoine Miné a développé un nouveau modèle mémoire avec une représentation au niveau des octets et la générations de variables abstraites scalaires selon les besoins, avec gestion des recouvrements de ces pseudo-variables (Miné, 2006a). Cela permet une représentation fine de la mémoire et des analyses bien plus précise, sans changer la structure des autres domaines abstraits ni la complexité des analyses.

4.4 Pré-analyses

L'itérateur d'Astrée ne peut être efficace que si des analyses préliminaires lui facilitent le travail : il n'est pas possible d'obtenir des informations certaines des domaines abstraits tant qu'on n'a pas atteint un post point fixe global, à cause des possibilités de réduction entre domaines. Astrée utilise donc des analyses préliminaires avec des domaines très simples de façon à ce que ces premières informations, parfois grossières, puissent être utilisées dans l'analyse plus fine du programme.

4.4.1 Simplification du code source

L'idée de simplifier le code source avant de faire une analyse poussée est venue de l'utilisation dans le programme analysé de gros tableaux de constantes, avec de plus plusieurs niveaux d'indirection. L'éclatement de ces gros tableaux multipliait par trois le nombre de variables abstraites et la représentation par une seule « variable » n'était pas assez précise. Plutôt que de développer un domaine abstrait de représentation de constantes, nous avons décidé de propager ces constantes (suivant Kildall, 1973). Nous avons constaté une nette amélioration des performances de l'analyseur. Cette amélioration est sensible pour toute transformation du code source permettant de réduire la taille et la complexité de ce code de façon significative.

Propagation des constantes

Dans un premier temps, notre propagation des constantes se contentait de recopier les valeurs des variables marquées comme constantes dans le code source, puis de supprimer ces variables si possible. Devant l'amélioration produite par cette simplification, nous avons décidé d'aller plus loin en détectant les variables effectivement constantes et en propageant autant que possible les expressions constantes. La détection des variables effectivement constantes se fait par une analyse par interprétation abstraite classique, mais le domaine utilisé n'est pas celui des constantes. En effet, l'évaluation d'une expression qui ne contient que des constantes ne doit pas être une constante au moins pour deux raisons : d'une part certaines sous-expressions pourraient produire un dépassement de capacité (ce qui peut se gérer en générant une alerte et en arrêtant l'analyse), d'autre part la correction d'Astrée vis-à-vis de toutes les erreurs d'arrondi donne un ensemble de valeurs et non une constante en cas d'opérations flottantes. Notre propagation utilise donc des expressions constantes. Le domaine abstrait reste malgré tout de hauteur finie (hauteur 3), ce qui permet de se passer d'élargissement. Une fois les invariants calculés, on peut les remplacer dans le code source par des constantes entières ou des intervalles flottants. La dernière étape consiste à supprimer toutes les affectations de variables qui ne sont plus utilisées par une analyse en arrière, puis la suppression des variables qui ne sont du coup plus affectées ni lues dans le programme résultant. Cette propagation des constantes fait gagner entre 6 et 14% sur la taille du code source.

Propagation des variables locales

Les programmes générés automatiquement contiennent souvent des instructions et des variables inutiles. Typiquement, un paramètre est rangé dans une variable temporaire, qui sert ensuite pour le calcul d'une autre variable locale qui sera ensuite sauvegardée dans la variable sensée contenir le résultat. Une propagation des variables locales en deux étapes est en cours d'implémentation pour Astrée. La première phase propage en avant les variables locales : après une affectation x=y; la variable `x` est remplacée par `y` jusqu'à la prochaine écriture de `x` ou `y`. Cette propagation n'est effectuée que dans les cas où cela permet de supprimer l'affectation x=y;. La deuxième phase propage en arrière les affectations x=y; qui restent jusqu'à une affectation

y=e; qu'on remplace par x=e; (à condition toujours que cela permette de supprimer l'affectation de départ). Les premières expérimentations montrent que cela devrait permettre de diminuer encore la taille du code source de 13 à 16% sur les exemples analysés par ASTRÉE.

4.4.2 Stratégies de discrimination

Le domaine des discriminations de traces présenté dans ce mémoire (et Mauborgne and Rival, 2005) permet de déterminer au cours des calculs quels ensembles de traces discriminer, mais en général il est difficile d'implémenter une stratégie efficace, sauf peut-être en réponse aux demandes des autres domaines abstraits. En pratique pour ASTRÉE, à chaque fois que nous avons identifié une imprécision qui pouvait se résoudre par discrimination des traces, nous avons testé par introduction manuelle de quelques directives de discrimination dans le code analysé, puis nous avons élaboré une stratégie générale permettant de placer automatiquement les directives partout où un problème semblable pouvait se présenter.

Accès de tableau

Un mécanisme très courant dans les applications de contrôle/commande est d'utiliser des fonctions définies par interpolation linéaire. Ces fonctions prennent une entrée `x`, cherchent entre quelles bornes du tableau on peut placer cette entrée et calculent l'interpolation de `x` entre ces deux bornes. Un exemple est donné en figure 4.3. Sur cet exemple, les valeurs effectives de `y` sont dans l'intervalle $[-1, 2]$. Mais une analyse d'intervalles classique ne pourrait pas le découvrir en général. Supposons que l'analyse ait trouvé que `x` est dans $[-100, 0]$, alors au moment de l'affectation de `y`, `i` vaut 0 ou 1. Même si on utilise cet ensemble de valeurs pour `i` au lieu d'un intervalle, on obtient que `y` est dans $[50.5, -0.5]$, à cause de l'absence d'information relationnelle entre `i` et `x`. Mais si on distingue les traces en fonction du nombre de tours dans la boucle **while**, alors on obtient le résultat exact. Si on revient à l'exemple $\texttt{x} \in [-100, 0]$, la boucle peut faire aucun ou un tour. Le cas aucun tour donne $\texttt{i} = 0$, $\texttt{x} \in [-100, -1[$ et $\texttt{y} = -1$, alors que le cas avec un tour de boucle donne $\texttt{i} = 1$, $\texttt{x} \in]-1, 0]$ et $\texttt{y} \in]-1, -0.5]$. Au final, si on fusionne les traces après cette affectation, on trouve l'intervalle $[-1, -0.5]$ pour `y`.

En analysant les raisons des imprécisions sur cet exemple, on trouve deux causes principales

```
tc = { 0;   0.5;   1; 0};
tc = { 0;   0.5;   1; 0};
tx = { 0;    -1;   1; 3};
ty = {-1; -0.5; -1; 2};
int i = 0;
while(i<4 && x>tx[i+1]) i++;
y = tc[i] * (x - tx[i]) + ty[i];
```

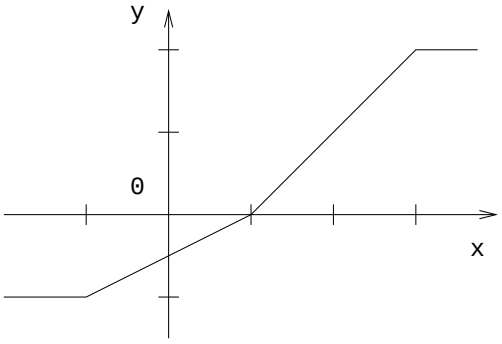

FIG. 4.3 – Interpolation linéaire

dans le calcul de `y`. D'une part l'expression contient plusieurs sous-termes qui ne sont pas indépendants (par exemple `x` et `tx[i]`) et d'autre part elle utilise des accès de tableau indexés par une variable. Le premier point indique une perte de précision probable pour des domaines non relationnels, et le deuxième point revient à une sorte de disjonction d'affectations indexée par l'indice de tableau. Dans ASTRÉE, nous repérons donc automatiquement les expressions avec au moins deux sous-termes de la forme accès de tableau avec la même variable entière (`i` sur l'exemple), puis nous cherchons la dernière affectation de cette variable avant son utilisation dans cette expression. Si cette affectation se fait dans un test ou une boucle, nous insérons une directive de discrimination selon ce flot de contrôle, sinon nous insérons une directive de discrimination selon les valeurs de la variable. Pour éviter que cette discrimination coûte trop cher, nous insérons aussi une directive de fusion après l'expression contenant les accès de tableau. Ce mécanisme est assez général pour traiter précisément les accès de tableau sans avoir à développer de nouveaux domaines abstraits relationnels spécifiques.

Divisions entières

Un autre exemple d'imprécision était assez générique pour justifier l'implémentation d'une autre stratégie dans ASTRÉE. Il est illustré dans la figure 4.4 par le calcul du barycentre d'une va-

```
int r = 0;
double x = 0.0, input = 0.0;
while(1) {
  r = random(0,50);
  input = random(-100,100);
  x = (x*r+input)/(r+1);
}
```

FIG. 4.4 – Lissage par calcul de barycentre (random(i,j) retourne un entier aléatoire entre i et j).

leur courante avec une nouvelle valeur (servant à lisser des variations d'une variable externe). Dans cette exemple, le calcul du poids du barycentre, `r`, est simulé par un tirage aléatoire pour simplifier la présentation. Mais même si ce poids est effectivement aléatoire, il est aisé de voir que `x` ne peut pas sortir des bornes de `input`. Dans le domaine des intervalles, si on part de l'invariant $[-100, 100]$, on trouve après un tour de boucle que `x` est dans $[-5100, 5100]$. Cette divergence peut être bornée par le domaine des contraintes arithmético-géométriques (Feret, 2005b), mais l'imprécision est tout de même trop grande si on compose plusieurs de ces calculs.

En fait, si on prend n'importe quelle valeur individuelle de `r`, on trouve bien que `x` reste dans $[-100, 100]$. ASTRÉE identifie donc les expressions contenant une division avec la même variable entière utilisée dans le dividende et le diviseur, puis comme dans le cas des accès de tableaux on insère une directive de discrimination sur la plus proche affectation de cette variable. On peut noter que si l'intervalle de variation de cette variable est trop grand, alors le domaine des discriminations de traces peut décider *dynamiquement* de ne pas tenir compte de cette directive, ce qui évite d'éventuelles explosions combinatoires.

4.4.3 Regroupement de variables

Très vite, il nous est apparu que les domaines abstraits ayant un coût quadratique ou plus seraient trop gourmands pour analyser les gros codes avec beaucoup de variables globales. De tels domaines, comme celui des octogones ou des décisions d'arbres sont pourtant nécessaires à la précision de l'analyse. La solution utilisée dans ASTRÉE consiste à ne pas mettre toutes les variables dans le même groupe, mais à utiliser plusieurs petits groupes de variables. Cela signifie qu'on ne garde plus que les relations des variables au sein d'un même groupe. Pour que cette solution ait un intérêt, il faut donc regrouper intelligemment les variables afin d'une part de ne pas perdre les relations nécessaires et d'autre part de ne pas s'encombrer de relations inutiles. Pour tester des stratégies de regroupement de variables, ou améliorer localement la précision de l'analyse, ASTRÉE permet aussi l'utilisation de directives décrivant des groupes de variables. Comme pour les discriminations de traces, nous avons implémenté des stratégies automatisées construisant ces groupes sans l'aide des directives.

Approche locale

Les codes générés par AIRBUS sont composés de calculs élémentaires successifs, chaque calcul étant isolé au sein d'un bloc `C`. C'est bien souvent au sein de ces calculs qu'il est utile de garder les relations entre variables, car chaque calcul élémentaire a été développé indépendemment des autres. Une première stratégie consiste donc à regrouper les variables utilisées au sein d'un même bloc. Les variables de conditionnelles ou de test de boucle apparaissant à la fois dans le bloc contenant le test et dans les blocs de la boucle ou des conditionnelles. Une fois les groupes déterminés, il sont utilisés partout dans l'analyse, de manière à ne pas perdre d'invariants relationnels globals. C'est la stratégie qui est implémentée dans ASTRÉE pour les octogones, en limitant les variables à celles qui apparaissent dans les parties potentiellement linéaires des expressions utilisant plusieurs variables, car le domaine des octogones n'est précis que pour ces expressions. Les expériences montrent que d'une part les octognes ne sont pas trop gros (moins d'une dizaine de variables par groupe en moyenne), et d'autre part que plus de la moitié des octogones ainsi représentés sont effectivement utiles au cours des calculs (Miné, 2004b), c'est-à-dire qu'au cours des calculs, ils trouvent des intervalles de valeur plus petits que tous les autres domaines pour certaines variables. En l'absence de blocs explicites, des parties locales du codes pourraient être extraites par des analyses de dépendance, permettant d'utiliser la même stratégie sur des codes différents.

Approche globale

Dans les codes générés automatiquement, les relations utiles ne sont pas toujours regroupées au même endroit dans le programme. C'est le cas

de certains informations de contrôle, des tests étant stockés dans des booléens qui sont ensuite utilisés très loin de leur affectation, parfois dans des fonctions différentes. Dans ce cas, on ne peut se retreindre à une approche locale, mais l'approche globale pose le problème de trouver une autre structure aux groupes. Nous avons implémenté un regroupement qui essaie d'identifier à l'avance les groupes de variables pour lesquels le domaine pourra faire gagner de la précision. Pour cela, on construit des groupes de variables qui seront en relation d'une façon représentable par le domaine, puis on valide les groupes pour lesquels on trouve des utilisations dont on sait que le domaine est plus précis que les intervalles. Pour les arbres de décisions, les relations sont créées soit par l'affectation d'un booléen en fonction de variables numériques (du genre b=(x>0);), soit par des affectations de booléens en fonction de résultats de tests numériques ou de variables numériques en fonction de booléens. Ces relations ne seront effectivement utilisées que si on lit à la fois une variable booléenne et une variable numérique du même groupe. Cela correspond aux lectures de variables numériques dans des branches de tests portant sur une variable booléenne du groupe.

Il faut aussi tenir compte des dépendances entre variables, en particulier booléennes, car si un booléen b_1 est utilisé dans le calcul d'un autre b_2, tester b_2 revient à tester certaines valeurs de b_1. La stratégie complète doit donc aussi regrouper autour des groupes de variables candidats les variables qui dépendent de ces candidats. Le problème, c'est qu'il n'y a pas de limite raisonnable à ce processus dans un contexte où toutes les variables semblent interdépendantes. L'idée, c'est que la correction du code ne dépend que d'une petite chaîne de telles dépendances pour être gérable par les concepteurs du code. On fixe donc une limite à la taille de cette chaîne de dépendance. À l'heure actuelle, une limite de 3 booléens donne de bons résultats pour le domaine des arbres de décision.

La même stratégie pourrait être utilisée pour les octogones, les groupes candidats étant générés par les affectations suffisamment linéaires pour que les octogones en gardent quelque chose (c'est déjà un critère dans l'approche locale actuelle), un peu d'aggrégation, puis une validation par l'emploi de deux variables d'un même groupe dans un test. Cette stratégie n'a pas encore été expérimentée, principalement parce que l'approche locale donne déjà de bons résultats sur les programmes d'AIRBUS.

4.4.4 Adaptation dynamique

La plupart des analyses présentées dans cette section pourraient être bien plus fines en utilisant les résultats de l'analyse globale d'ASTRÉE. Par exemple, il est possible d'utiliser le résultat de l'analyse pour savoir quels octogones ont été utiles et de n'utiliser que ces octogones dans les analyses suivantes. Mais cela ne peut se faire que dans une phase de mise au point de l'analyse et pas dans l'optique d'une analyse unique d'un code en vue de sa certification. Une approche plus intéressante consiste à utiliser les informations disponibles en cours d'analyse, mais on ne peut le faire que dans des domaines où des informations partielles suffisent. C'est le cas des actions qui ne modifient pas la correction de l'analyse mais peuvent éventuellement en améliorer la précision. Dans la section 3.3.3, on mentionne la possibilité de rajouter des critères de discrimination en cours d'analyse. La stratégie actuelle s'appuie beaucoup sur l'identification de sous-termes qui partagent une variable. On pourrait facilement l'étendre au cas de sous-termes utilisant deux variables dont l'analyse a montré qu'elles étaient égales (en utilisant le domaine des égalités de la section 4.3.2) et créer ainsi dynamiquement de nouvelles discriminations de traces utiles. La terminaison du processus est assurée car cet ajout ne peut être fait qu'une fois au plus par point de programme.

Une autre application qui ne remet pas en cause la correction de l'analyse est celui des groupes de variables. Tous les domaines de filtres utilisent des groupes de variables, chaque groupe correspondant à un filtre. La détermination de ces groupes est faite dynamiquement car elle peut nécessiter le calcul des intervalles de variation de certaines variables et une linéarisation des expressions pour qu'elles rentrent dans le cadre des filtres linéaires. Encore une fois, le nombre total de groupes est borné par la taille du programme.

On peut utiliser une stratégie similaire avec les groupes déterminés par une approche globale. Dans cette approche on détermine statiquement des groupes candidats qui auront une chance de contenir des informations relationnelles pour le domaine, puis on identifie de manière presque syntaxique des endroits où ces groupes font gagner de la précision. Dans certains cas, de nouvelles opportunités de précision peuvent être identifiées par d'autres domaines abstraits. Dans ASTRÉE, il est donc possible pour un domaine de demander aux autres domaines d'être aussi précis que possible sur un

ensemble de variables. Cette demande provoque l'ajout des groupes candidats contenant les variables de cet ensemble. Cela permet de ne pas surcharger l'analyse par l'emploi d'un trop gros nombre de groupes tout en étant précis quand on en a besoin.

4.5 Les défis

Le but initial du projet Astrée était de montrer la faisabilité d'un analyseur statique utilisant l'interprétation abstraite arrivant à produire moins d'une centaine d'alarmes sur une portion significative d'un code de taille industrielle, et ce en utilisant des ressources raisonnables. Le projet a largement dépassé cet objectif en analysant complètement et sans fausse alerte la commande de vol électrique de l'A340, puis récemment celle de l'A380, pourtant sept fois plus gros. Ce faisant, l'équipe d'Astrée a rencontré un grand nombre de défis passionnants qui ont motivé une grande partie de sa recherche au cours de ces dernières années. De nouveaux défis sont apparus au fur et à mesure de l'aventure et un grand nombre attendent encore d'être relevés.

Agrandir la bibliothèque des domaines abstraits

Le principe d'Astrée d'utiliser un analyseur spécialisé par propriété et par famille de programme ne limite pas l'utilisation d'Astrée aux programmes de la commande de vol électrique des A300. La conception d'Astrée permet d'employer pour chaque famille un ensemble de domaines abstraits différents. Mais pour cela, il faudrait disposer d'une bibliothèque de domaines abstraits beaucoup plus importante. Le succès d'Astrée a motivé un certain nombre d'autres industriels à s'y intéresser, et leurs familles de programmes fourniront sûrement l'opportunité d'imaginer et de développer de nouveau domaines qui rendront Astrée plus adaptable.

Analyser les propriétés fonctionnelles

Pour augmenter la confiance que l'on peut accorder aux logiciels, il faut être capable de montrer plus que l'absence d'erreur critique à l'exécution. Il faut pouvoir aussi montrer la préservation d'invariants globaux complexes, utiliser un modèle de l'environnement pour prouver des propriétés relationnelles entre les entrées et les variables du programme. Il faut aussi être capable de raisonner efficacement sur le comportement temporel du programme pour s'assurer que tout se passe comme il faut. Un certain nombre d'éléments de ce mémoire peuvent servir de point de départ pour relever ces défis qui motiveront certainement une part importante de mes futures recherches.

Et beaucoup d'autres encore

Les utilisateurs souhaitent aussi pouvoir certifier ou trouver les erreurs d'autres styles de programmes, normalement moins critiques, comme les périphériques de communication qui utilisent des structures de données symboliques, ou les programmes asynchrones qui posent de nouveaux défis pour une analyse efficace. Il n'est pas facile d'énumérer toutes les applications et les problèmes théoriques qu'ils soulèvent, mais une chose est sûre, le projet Astrée m'aura fourni une opportunité extraordinaire de mettre en pratique le domaine qui m'intéressait et je pense qu'il servira encore longtemps de moteur et de banc d'essai pour les recherches à venir de l'équipe d'interprétation abstraite du LIENS.

Chapitre 5

Enseignement

J'ai commencé l'enseignement assez tôt, dès l'obtention de mon DEA, et je n'ai pas cessé d'enseigner depuis. Mes domaines d'enseignement ont d'abord porté sur les outils que j'employais tous les jours dans mon environnemnt de travail, le système UNIX, puis on m'a demandé d'enseigner l'algorithmique et la programmation dans différents cadres très intéressants. Quand j'ai été recruté à l'ENS, on m'a assez naturellement associé aux travaux dirigés en compilation et en sémantique. À partir de ma soutenance de thèse, j'ai enseigné les parties les plus abouties de mon travail de recherche, et depuis peu, j'essaie à travers les cours d'amener de nouveaux élèves à faire de l'analyse statique par interprétation abstraite.

5.1 Systèmes d'exploitation

La première fois qu'on m'a proposé d'enseigner, j'étais scientifique du contingent à la DGA, et la société /FORM/UNIX cherchait des personnes capables de former des industriels à l'utilisation de ses serveurs de calcul qui tournaient sous un UNIX propriétaire. Je l'ai fait en été 1994. La matière enseignée était finalement assez simple car elle s'adressait à des personnes qui n'avaient jamais vu le monde UNIX.

Plus tard, Patrick Cousot m'a proposé d'assurer des travaux pratiques d'un cours de système d'exploitation (encore UNIX) à l'École Polytechnique. Le cours s'intitulait « Systèmes d'exploitation UNIX et réseaux d'ordinateurs » et couvrait la théorie des systèmes d'exploitation, les systèmes de fichiers, les librairies, les signaux, les processus, la mémoire et une part importante de description des réseaux sous UNIX. J'ai assuré les TPs en 1994/95 sous la responsabilité de Robert Ehrlich qui m'a beaucoup appris.

Alors que j'étais moniteur, on m'a à nouveau proposé les travaux pratiques de ce cours en 1996/97. Il était à l'époque repris par François Bourdoncle, Jean-Jacques Lévy et Christian Queinnec. Le responsable des travaux pratiques était Frédéric Ruget.

J'ai par la suite continué à enseigner les bases de l'environnement UNIX en 1999 et en 2000 au cours de séances d'initiations pour les normaliens.

5.2 Algorithmique et programmation

Les cours de programmation et d'algorithmique m'ont toujours plu et ils ont occupé une part importante de mon enseignement.

5.2.1 Travaux dirigés

Mes premiers travaux dirigés en programmation ont eu lieu à l'École Nationale Supérieure des Techniques Avancées, grâce à Philippe Granger qui assurait un cours d'initiation à la programmation en C. Il s'agissait principalement de TDs en salle de cours, sans machine.

Ensuite, en tant que moniteur, j'ai participé pendant 2 ans aux travaux pratiques du tronc commun de l'École Polytechnique assuré par Robert Cori et Jean-Jacques Lévy. Le cours s'intitulait « Algorithmes et programmation », et couvrait une vaste portion de la programmation en C, en abordant les structures de données des listes aux graphes en passant par les arbres et les files, touchait à la récursivité, au backtracking et à la programmation dynamique, et abordait la validation de programme, le parallélisme asynchrone, le *garbage collection*, les exceptions, les modules, le polymorphisme, la programmation symbolique, la surcharge.... Heureusement, j'étais dans un groupe de polytechniciens forts, les petits cours étaient assurés par Patrice Bertin et la conception des travaux pratiques revenait à Bruno Salvy.

Puis en 2000, avec la suppression du service militaire obligatoire, l'enseignement de l'École Polytechnique a été réformé, l'initiation à l'informatique n'a plus été divisée qu'en deux niveaux, et j'ai assuré en tant que chargé de travaux pratiques la conception et l'animation des TPs du cours « Les bases de la programmation et de l'algorithmique » sous la responsabilité de Robert Cori. Bien que ce cours s'adressait aux « initiés », il se contentait d'aborder les listes et les arbres et les concepts de base du langage Java. J'ai fait ces TPs jusqu'en 2004, puis en 2005 sous la responsabilité de Gilles Dowek qui a ajouté un peu de sémantique au cours. Ce point de vue m'a semblé intéressant car il rajoute un aspect scientifique et formel qui manque aux élèves qui connaissent déjà un peu la programmation.

Dans le même temps, j'ai aussi assuré les travaux pratique du deuxième cours d'informatique du cursus polytechnicien « Programmation et algorithmique » qui vise à mettre les débutant qui ont suivi le premier cours au même niveau que les « initiés ». Ce cours a été assuré de 2001 à 2003 par Jean-Éric Pin et Jean Berstel, et en 2004, mon responsable était Luc Maranget.

J'ai aussi participé avec Louis Granboulan aux travaux pratiques et travaux dirigés du cours de Jacques Stern à l'École Normale Supérieure. Le cours s'intitulait « Algorithmique et programmation » et couvrait un champ aussi large que le premier cours d'algorithmique auquel j'avais été confronté à l'École Polytechnique. Le faible nombre d'étudiants et leur niveau élevé nous permettait de leur faire suivre un rythme soutenu et de faire des travaux pratiques qui étaient presque des mini-projets.

5.2.2 Examens

Les examens sont un exercice important dans l'enseignement et j'ai eu l'occasion d'en rédiger quelques uns pour les cours d'algorithmique à l'École Polytechnique. Le plus important, et le plus formateur pour moi, a été la conception des sujets de l'oral du concours des Écoles Normales Supérieures pour l'« Épreuve Pratique de Programmation et d'Algorithmique ». J'ai ensuite fait passer cet oral avec Jean Goubault-Larrecq et Nicolas Schabanel. L'oral consistait en un exercice de programmation de 4 heures, dont seuls les résultats de programmes étaient notés, suivi d'une interrogation individuelle permettant au candidat d'expliquer ses algorithmes et ses choix d'implémentation. Les contraintes de cette épreuve sont très grande pour le jury, car il impose à la fois un style très précis pour l'exercice de programmation dont on doit pouvoir exploiter les résultats facilement tout en donnant l'occasion aux candidats de montrer différents aspects de leurs qualités d'informaticiens.

5.2.3 Cours

En septembre 2003, on m'a proposé à l'École Polytechnique d'assurer un mini-cours d'initiation au langage Ocaml. Ce cours abordait les points suivants :

1. L'historique et la motivation.
2. La syntaxe, les types et opérations élémentaires, les enregistrements.
3. Les fonctions, le filtrage, le polymorphisme.
4. Les entrées-sorties, les exceptions, la mise au point.
5. Les modules et la compilation séparée.

J'ai aussi à cette occasion rédigé les travaux pratiques associés.

Ayant été recruté comme professeur chargé de cours à l'École Polytechnique, on m'a confié des petits cours d'algorithmique avancée, au sein du cours de François Morain et Jean-Marc Steyaert intitulé « Informatique fondamentale ». Ce cours, dispensé en 2006, abordait principalement les points suivants :

1. Graphes (représentation, accessibilité, parcours, point d'articulation, planarité, coloriage).
2. Langages (automates finis, analyse lexicale, analyse syntaxique, grammaires).
3. Programmation distribuée (calcul distribué, protocoles de routage, deadlocks, broadcast, élections, réseau ad-hoc).
4. Logique (calcul propositionnel, logique des prédicats, complexité).

5.3 Compilation et sémantique

Lorsque j'ai été nommé attaché temporaire de recherche et d'enseignement à l'École Normale Supérieure, les cours de première année les plus proches de mon domaine de recherche étaient ceux assurés par Patrick Cousot, c'est-à-dire « Langages de programmation et compilation » et « Sémantique des langages de programmation ». J'ai assuré les travaux pratiques

de ces cours depuis 1999, et parfois aussi remplacé Patrick Cousot ponctuellement pour certains cours de compilation. Le cours de compilation aborde l'analyse lexicale, les grammaires, le typage, le polymorphisme, les grammaires attribuées, les portées des identificateurs, la récursivité, les modes de passage des paramètres, la pile d'exécution, le code intermédiaire et un peu d'analyse statique et de preuve de correction du compilateur. Le cours de Sémantique présentait les différentes sémantiques de la littérature et leurs connections grâce à la théorie de l'interprétation abstraite.

5.4 Domaines abstraits symboliques

J'ai donné des cours de troisième cycle à partir de ma soutenance de thèse. Au début, ces cours parlaient principalement des ensembles d'arbres infinis et de propriétés temporelles, pour petit à petit parler un peu plus d'analyse statique et de domaines abstraits symboliques.

En février 2000, Neil Jones m'a invité à faire un cours pour thésards, afin de présenter l'essentiel de mon travail de thèse en détail, à l'Université de Copenhague. Le plan du cours, en 6 séances de 2 heures, était :

1. Grammaires, automates d'arbre, μ-calcul, automates de Büchi, automates avec contraintes.
2. Hash-consing, extension au cas des cycles.
3. Applications aux squelettes d'arbres, exemples d'utilisations.
4. BDDs, relations ouvertes et fermées, relations ω-régulières.
5. Relations ω-déterministes, unicité, algorithmes.
6. Schémas d'arbres, application à la preuve de terminaison sous hypothèse d'équité.
7. Type principal, sémantique petits pas, compteurs.

J'ai ensuite donné des cours au sein du DEA « Programmation : sémantique, preuves et langages ». En 2000, j'ai participé au cours d'Éric Goubault sur le parallélisme et les propriétés temporelles (Éric assurant la responsabilité du cours et enseignant la partie parallélisme et modèles géométriques (Goubault, 2000)). Puis, en 2002 et 2003, j'ai participé au cours d'« Analyse statique par interprétation abstaite » de Radhia Cousot. D'autres intervenants participaient aussi à ce cours : en 2002, Arnaud Venet parlait aussi d'analyse d'alias et en 2003, Bruno Blanchet enseignait aussi l'analyse d'échappement. En 2004, j'ai été responsable du cours d'« Analyse statique de propriétés temporelles et probabilistes », avec David Monniaux et Damien Massé. Le contenu du cours était :

1. Représentations avec partage (BDDs, partage incrémental de cycles)
2. Relations infinies et propriétés temporelles
3. Ensembles d'arbres et propriétés de traces
4. Logiques temporelles
5. Analyse avant/arrière (D. Massé)
6. Analyse de composants matériels (C. Hymans)
7. Sémantique probabiliste dénotationnelle (D. Monniaux)
8. Processus Décisionnels de Markov (D. Monniaux)
9. Méthodes de Monte-Carlo (D. Monniaux)

J'ai ensuite enseigné des domaines abstraits symboliques, finis et infinis au sein du cours d'« Analyse statique par interprétation abstraite » du Master Parisien de Recherche en Informatique depuis 2005. Ce cours est sous la responsabilité de Patrick et Radhia Cousot et fait aussi intervenir Bruno Blanchet, Mathieu Martel, David Monniaux, Jérôme Feret, Antoine Miné et Xavier Rival.

5.5 Interprétation abstraite

L'interprétation abstraite entre dans une phase d'applications et de développements qui nécessite beaucoup de spécialistes pour suivre les demandes du monde industriel. D'autre part, l'utilisation d'outils basés sur l'interprétation abstraite est bien plus efficace si on connaît quelques éléments de la théorie pour comprendre le comportement de l'outil, ses messages et aussi pour être capable de l'ajuster à des situations particulières. Je trouve donc particulièrement important de transmettre cette théorie et de pousser de jeunes chercheurs à explorer cette voie, développer de nouveaux outils ou aider les équipes existantes à étoffer les outils existants.

5.5.1 Cours

J'ai eu la possibilité en 2006 de faire un cours d'une journée à l'École Jeunes Chercheurs en Programmation. Ce cours abordait les points suivants :

1. Présentation informelle et motivations.

2. Éléments de théorie de l'interprétation abstraite : familles de Moore, abstraction, opérateur de clôture, différentes notions de complétion, connection de Galois, transfert de points fixes et application à l'accessibilité.
3. Exemples d'application : model checking, abstraction par prédicats, typage, analyse d'intervalles, élargissements.

J'aurais aimé avoir plus de temps pour aborder d'autres applications importantes, comme la mise en pratique de la théorie dans ASTRÉE.

Il est aussi prévu que je donne un cours d'« Analyse statique » à l'École Polytechnique, pour les élèves en troisième année, au début de l'année 2007. L'objectif de ce cours est de former de futurs utilisateurs d'outils d'analyse statique et de donner les bases qui permettront de suivre le cours du Master Parisien de Recherche en Informatique de Patrick et Radhia COUSOT d'analyse statique par interprétation abstraite. Le plan de ce cours n'est pas encore définitivement arrêté, mais il devrait aborder différents paradigmes d'analyses statique, les comparer grâce à la théorie unificatrice de l'interprétation abstraite et surtout fournir de nombreux exemples. Ce cours s'appuiera aussi sur des travaux pratiques dont l'objet sera le développement d'un petit analyseur de C.

5.5.2 Encadrements

Les encadrements sont les meilleurs moyens d'intéresser un élève à une discipline en lui faisant découvrir les problèmes passionnants qu'on y rencontre.

À l'École Polytechnique, les enseignants proposent chaque année au moins un sujet de stage. En 2004, j'ai proposé pour un élève assez fort et motivé, Guillaume KIRSH, de faire quelques analyses simples sur un sous-ensemble de Java dont je fournissais le back-end. Il s'en est tiré honorablement et cela a été l'occasion pour moi de découvrir les difficultés pratiques ou conceptuelles que pouvait rencontrer un étudiant qui découvre l'analyse statique. Je pense que ces enseignements me seront très utiles pour le cours d'analyse statique que je dois donner en 2007.

J'ai aussi encadré quelques stages du DEA Programmation : sémantique et preuves.

Automates de mots

En 2002, j'ai proposé à Sébastien VILLEMOT d'évaluer les techniques de partage maximal d'automates de mots dans le cadre d'une analyse statique. Il a implémenté une analyse de protocoles de communications qui a permis de comparer cette technique aux techniques classiques dans le même contexte. Son mémoire s'intitule *Automates finis et interprétation abstraite, applications à l'analyse statique de protocoles de communication* (Villemot, 2002).

Détection d'intrusion

En 2004, Élie BURSZTEIN est venu demander à faire un stage en interprétation abstraite sur un sujet qu'il avait déjà en tête. Il voulait explorer ce que l'interprétation abstraite pouvait faire pour exprimer et repérer des comportements anormaux sur un réseau. Il fallait que ce soit très rapide pour traiter un grand débit de données et lever l'alarme le plus tôt possible. Son stage a abouti à la conception d'une analyse fondée sur une approximation des protocoles de communication utilisés dans les réseaux. Son mémoire s'intitule *Conception d'un détecteur d'intrusion réseau par comportement anormal à l'aide de l'interprétation abstraite* (Bursztein, 2004). Élie est actuellement en thèse avec Jean GOUBAULT-LARRECQ

Vérification de spécifications temporelles

Les logiciels réactifs critiques sont spécifiés à l'aide d'opérations élémentaires qui sont ensuite reliées entre elles pour former le programme effectivement exécuté (Randimbivololona et al., 1999). Ces opérations élémentaires (parfois assez complexes) doivent vérifier une spécification, généralement informelle, qui précise en particulier le comportement temporel de l'opération. Le but du stage de Guillaume CAPRON était de formaliser ces spécifications, puis de proposer une analyse pour prouver la correction des opérations. Il a réalisé les premières étapes d'une analyse utilisant les techniques d'analyse en avant et en arrière et en combinant les domaines abstraits des polyèdres (Cousot and Halbwachs, 1978) et des schémas d'arbres avec compteurs (Mauborgne, 1999b). Son mémoire s'intitule *Analyse statique de réactivité par interprétation abstraite* (Capron, 2004). Il poursuit son sujet de stage dans le cadre d'une thèse que je suis de près, sous la direction de Radhia COUSOT.

Annexe A

Résultats pratiques

A.1 Partage de graphes

On compare plusieurs implémentations raisonnables du partage de graphes : celui qui semble le moins coûteux *a priori*, le partage partiel par hash-consing classique (pour les parties sans boucle, donc), le partage systématique par hashage à profondeur bornée (selon l'implémentation d'Alain Frisch), le partage systématique par clés de cycle, et le partage selon les besoins (quand on sait qu'on aura à tester des égalités) avec clés de cycle.

A.1.1 Les données

Les données sur lesquelles ces algorithmes sont testés correspondent à une simulation aléatoire d'opérations basiques susceptibles d'être utilisées fréquement dans le cadre d'une analyse statique. Au départ on suppose déjà calculé un sommet sans arête, et on effectue au hasard un nombre déterminé d'opérations. Pour chaque opération, on calcule un nouveau sommet. La méthode de calcul est choisie aléatoirement entre :

1. prendre un nouveau sommet d'étiquette aléatoire, duquel partent deux arêtes vers des sommets calculés précédemment,
2. dupliquer un sommet existant et toutes les arêtes qui partent de ce sommet sauf une qu'on fait pointer vers un autre sommet déjà calculé,
3. dupliquer un sommet S et tous les sommets et arêtes accessibles depuis S, sauf les arêtes menant à un sommet T choisi au hasard, et qu'on remplace par des arêtes menant à S,
4. prendre deux sommets au hasard, choisir le premier si ils sont indistinguables et le deuxième sinon.

En théorie, les deux premières opérations sont effectuées rapidement, que ce soit avec ou sans partage, grâce au hash-consing. La troisième opération est potentiellement très coûteuse quand on cherche une représentation unique et peu coûteuse sinon, et la quatrième est très peu coûteuse si les deux sommets sont représentés de manière unique et très coûteuses sinon. Les opérations 1 et 2 correspondent grosso-modo à des calculs d'itérés normaux, l'opération 3 correspondrait plus à un opérateur d'élargissement, avec repliage, et l'opération 4 peut être utilisée pour mémoiser ou tester si on arrive au point fixe.

Les tests de distinguabilité utilisés quand on n'a pas de partage et le hashage à profondeur bornée sont théoriquement plus efficaces si on a plus d'étiquettes de sommets. En effet, on pourra conclure à l'indistinguabilité beaucoup plus tôt dans le parcours du graphe. On a donc testé les cas avec une seule étiquette (comme si les graphes n'étaient pas étiquetés), avec deux étiquettes et avec 1000 étiquettes.

A.1.2 Les Graphiques

Dans tous les graphiques, on montre les courbes du temps d'exécution en fonction du nombre d'opérations, sur une même machine avec des implémentations en Ocaml 3.08. Comme certains algorithmes peuvent être sensibles à certaines combinaisons d'opérations rares (par exemple le coût quadratique très rarement observé pour la méthode avec clés de cycle), chaque courbe correspond à la moyenne des temps pour les opérations tirées avec 5 graines aléatoires différentes.

Sur tous les grahiques, les courbes nommées "Partage par hash" se rapportent à un partage systématique par hashage à profondeur bornée, tel qu'implémenté par Alain Frisch. Si un nombre est indiqué entre parenthèses, c'est la profondeur du hashage, par défaut 4. Les courbes nommées "Sans partage des cycles" se rapportent à un partage limité au hash-consing

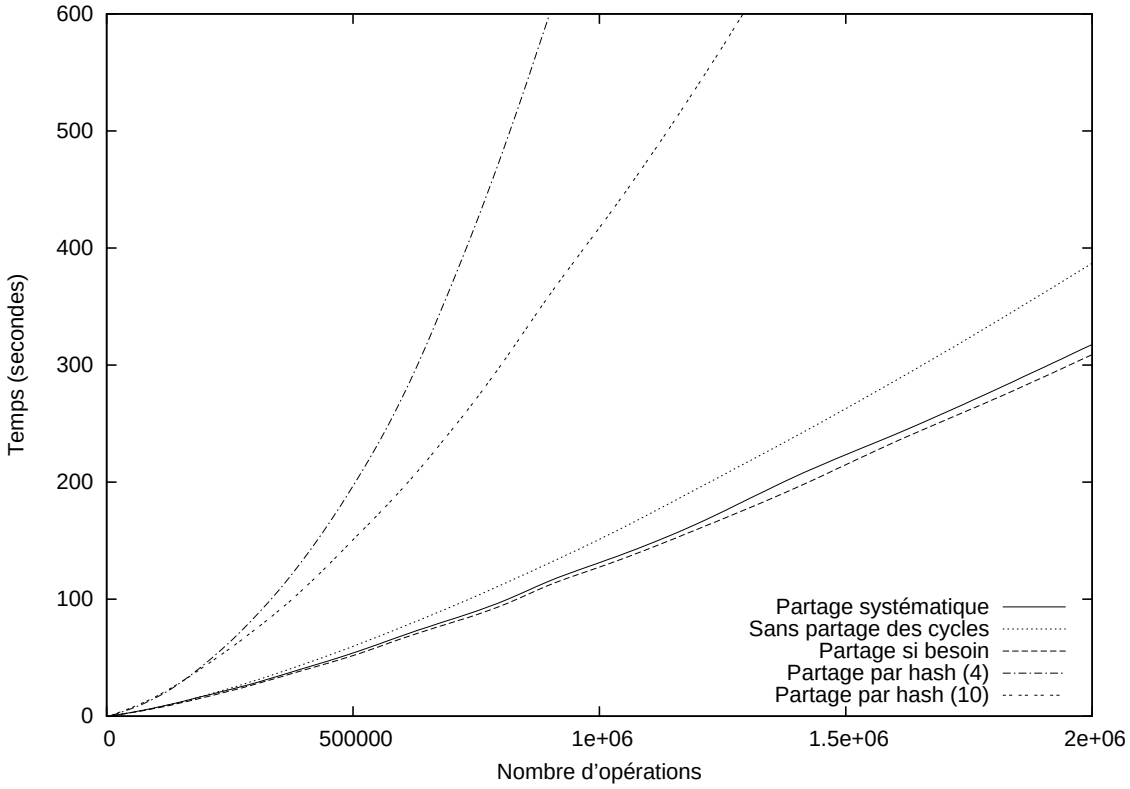

FIG. A.1 – Manipulation aléatoire de graphes avec 1000 étiquettes

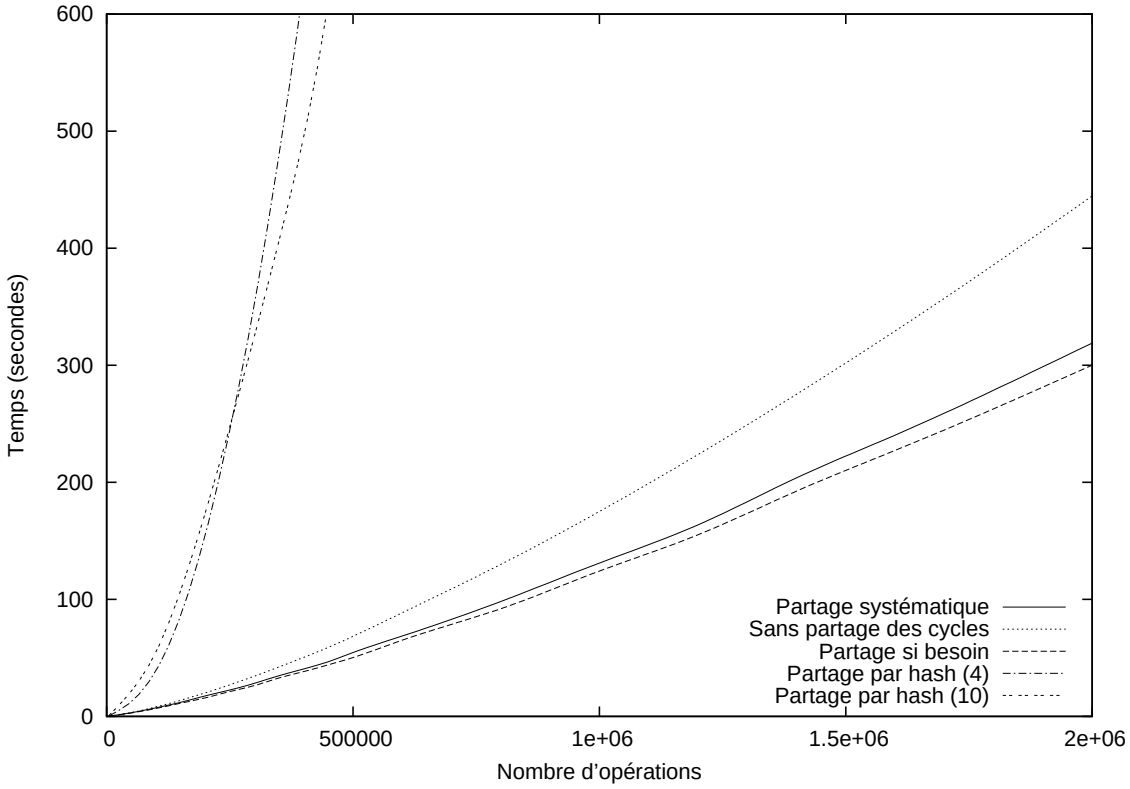

FIG. A.2 – Manipulation aléatoire de graphes avec 2 étiquettes

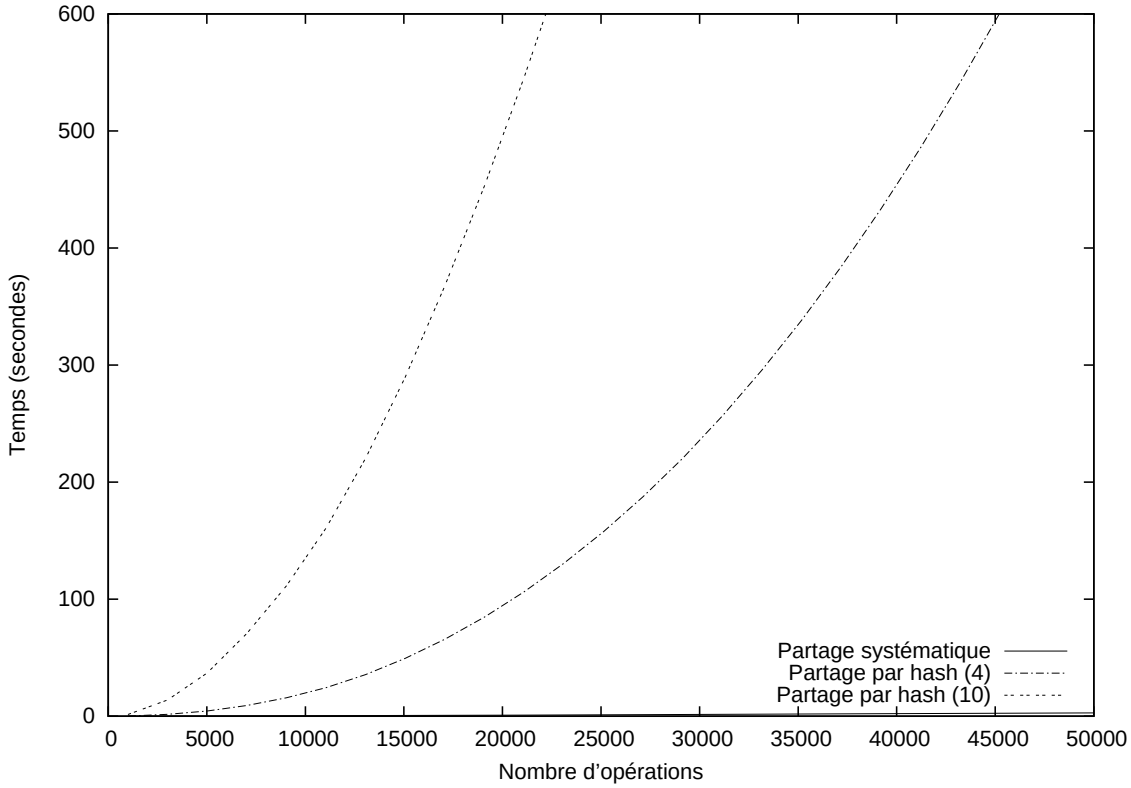

FIG. A.3 – Manipulation aléatoire de graphes non étiquetés

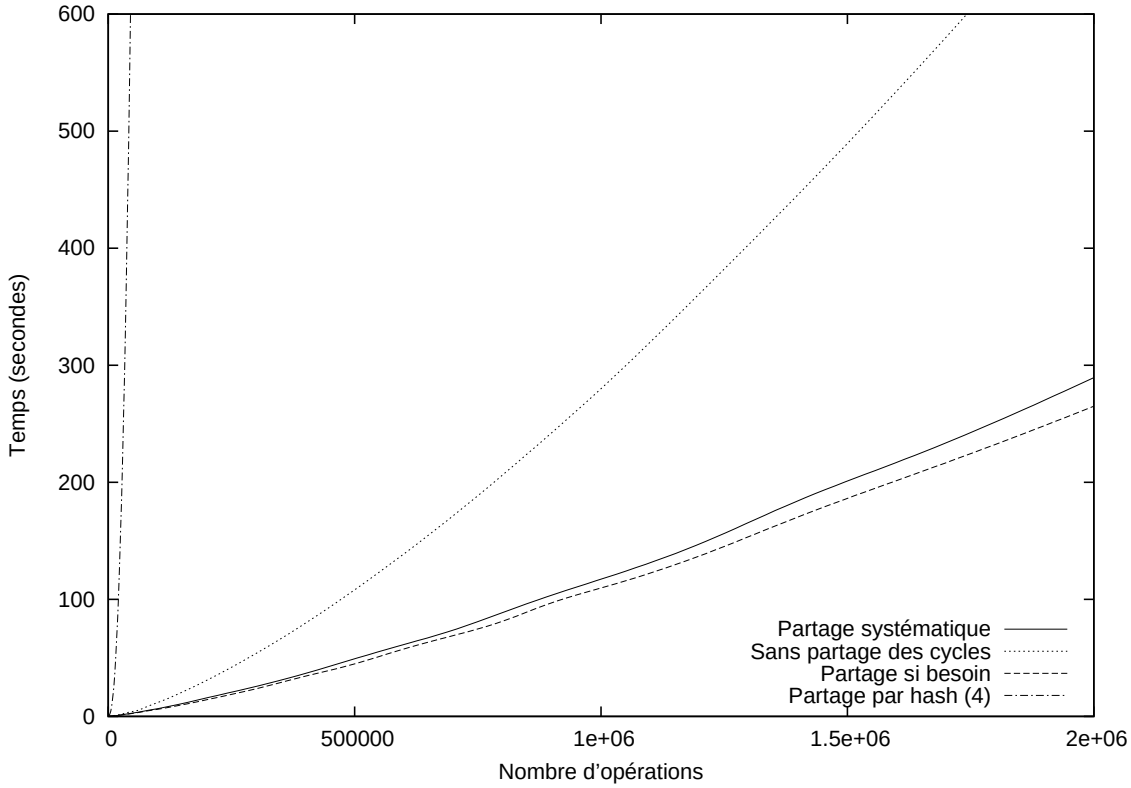

FIG. A.4 – Manipulation aléatoire de graphes non étiquetés

classique, ce qui fait que la majorité des sommets ne sont pas partagés. Les courbes nommées "Partage systématique" se rapportent à un partage systématique avec clé de cycle, selon la méthode présentée dans la section 2.2. Enfin les courbes nommées "Partage si besoin" n'utilisent ce partage que sur les arguments des opérations 2 et 3.

A.1.3 Les résultats

Tous les tests montrent qu'à la limite, il est plus efficace de partager avec les clés de cycle que de ne pas partager, et étonnamment il est plus efficace de ne pas partager (du points de vue du temps, pas de la mémoire) que de faire du partage par hashage à profondeur bornée. C'est vrai à la fois en moyenne et pour chaque graine aléatoire testée. C'est vraisemblablement dû à l'opération 3, potentiellement très coûteuse sur des gros cycles, et peut-être aussi au coût du hashage lui-même, puisqu'il croît très vite avec la profondeur du hashage. Il est intéressant de noter que la profondeur de hashage semble améliorer notablement les performances si les graphes ont beaucoup d'étiquettes distinctes (figure A.1). En revanche, en présence de gros graphes non-étiquetés, comme on peut s'y attendre, augmenter la profondeur de hashage entraine un gros surcoût (figure A.3). Pour le cas intermédiaire à 2 étiquettes (figure A.2), le hashage profond semble en général trop coûteux, mais il est rentable sur certaines graines aléatoires, ce qui semble suggérer que pour certains problèmes avec une structure particulière, on peut essayer de hasher plus profond.

On note aussi dans tous les cas que le fait de ne partager les sommets que quand cela semble utile, même si cela peut occasionner parfois des calculs en double, fait un peu gagner de temps par rapport au partage systématique.

Si on regarde l'effet de l'augmentation du nombre d'étiquettes, on constate l'effet attendu quand il n'y a pas de partage (l'augmentation du nombre d'étiquettes fait échouer les tests d'égalité plus tôt), mais l'effet sur la représentation partagée avec clé de cycle est mineur. Pour 2 millions d'opérations avec partage systématique, on passe d'environ 5 minutes sans étiquettes à 5 minutes et demi avec 2 ou 1000 étiquettes.

A.2 Automates de mots

Les automates sont implémentés dans une analyse de queues de communication. Les implémentations testées sont d'une part une implémentation des automates classiques en Java, une implémentation des automates avec partage maximal en Java et l'implémentation fournie dans la bibliothèque LASH écrite en C (Boigelot, 1999). Les deux premières implémentations disposent d'opérateurs d'élargissements et la bibliothèque LASH ne dispose que de méta-transitions qui ne permettent qu'une extrapolation exacte.

Ces implémentations sont d'abord testées sur le protocole du bit alterné qui permet à un émetteur et un récepteur de communiquer sur un canal avec pertes des données aléatoires (Bartlett et al., 1969). Le protocole peut être modélisé avec deux processus à 10 et 8 états, soit au total 80 points de contrôle pour lesquels il faut trouver l'état des queues possibles. L'ensemble des messages qui attendent dans les queues de communications est alors régulier, et on peut le trouver assez rapidement de manière exacte en utilisant soit les élargissements (repliage) soit les méta-transitions du LASH. Pour permettre une comparaison du comportement des automates, on montre ce qu'il se passe sans élargissement. Le graphique (figure A.5) montre que les gains en temps sont très importants quand on utilise des automates avec partage maximal dans un cadre d'analyse statique.

Le deuxième protocole analysé est plus complexe. Il s'agit du protocole à retransmission bornée qui permet de prendre en charge intégralement la transmission d'un fichier sur un canal avec pertes (Tanenbaum, 1981; Helmink et al., 1994). Dans l'analyse de ce protocole, nous avons utilisé 390 points de contrôle. Cette fois, l'ensemble des messages des queues n'est pas régulier pour certains points de contrôle, et il n'a donc pas été possible de faire l'analyse avec la bibliothèque LASH. Le graphique (figure A.6) montre le temps de calcul en mettant la limitation de taille de plus en plus tard (on obtient alors des résultats de plus en plus précis). Cette fois encore, l'avantage du partage maximal est très net. Il faut noter toutefois que les automates construits dans cette analyse n'ont que de très petits cycles (rarement plus de deux états), ce qui est assez favorable au partage puisque les minimisations seront très peu coûteuses et les clés de cycle petites.

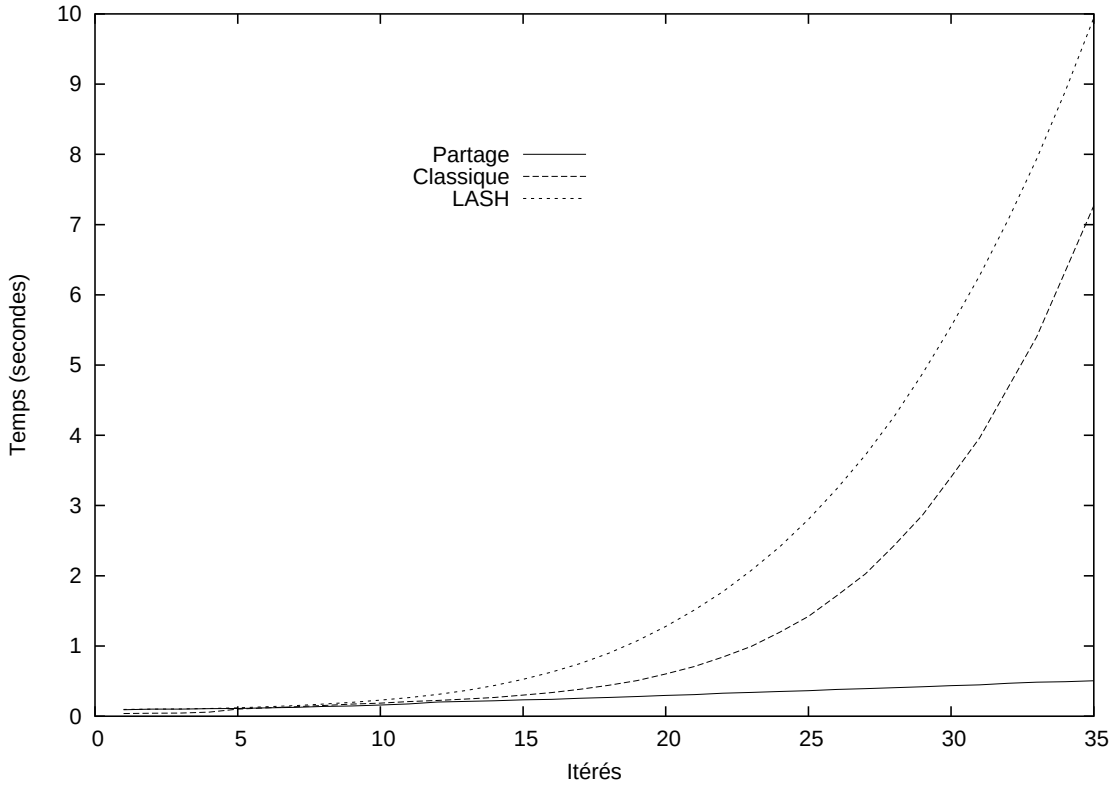

FIG. A.5 – Manipulation d'automates au cours d'une analyse

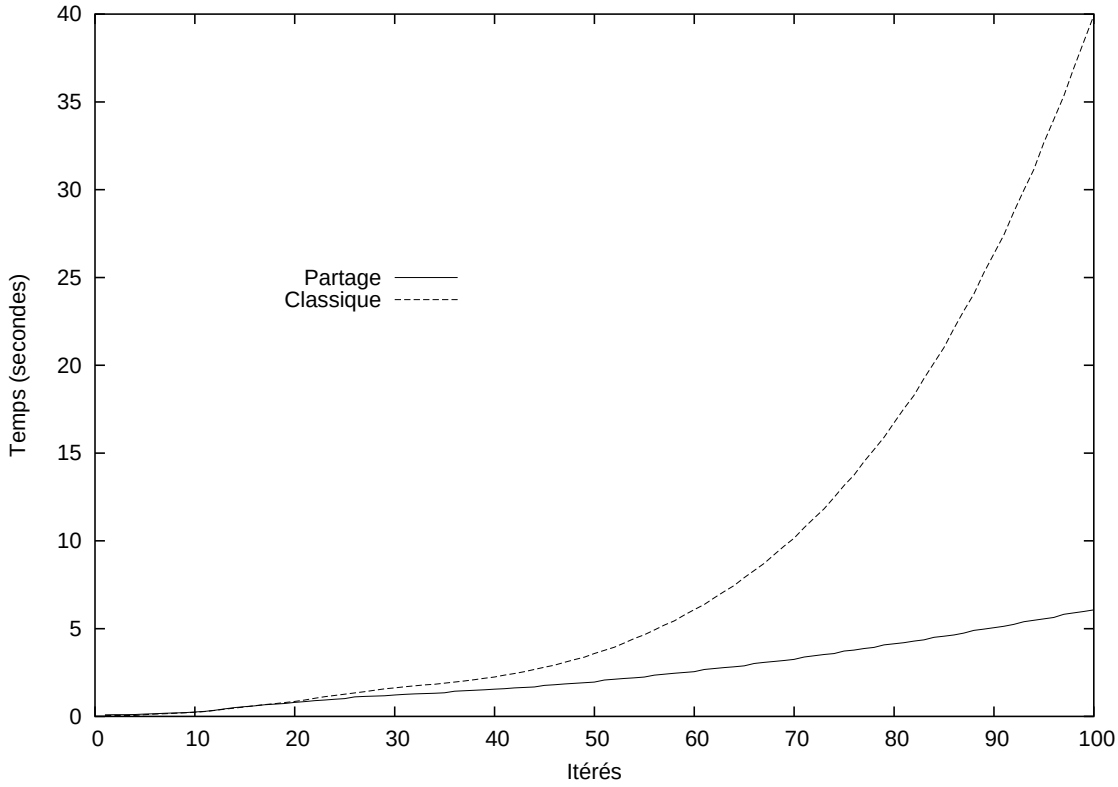

FIG. A.6 – Manipulation d'automates au cours d'une analyse avec élargissement

Bibliographie

Abdulla, P. A., Bouajjani, A., Jonsson, B., and Nilsson, M. (1999). Handling global conditions in parameterized system verification. In Halbwachs, N. and Peled, D., editors, *Conference on Computer Aided Verification (CAV'99)*, pages 134–145.

Alt, M., Ferdinand, C., Martin, F., and Wilhelm, R. (1996). Cache behavior prediction by abstract interpretation. In Cousot, R. and Schmidt, D., editors, *Static Analysis, Third International Symposium (SAS'96)*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag.

Anuchitanukul, A., Manna, Z., and Uribe, T. E. (1995). Differential BDDs. In van Leeuwen, J., editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 218–233. Springer-Verlag.

Asarin, E., Bozga, M., Kerbrat, A., Maler, O., Pnueli, A., and Rasse, A. (1997). Data-structures for the verification of timed automata. In Maler, O., editor, *Hybrid and Real Time Systems (HART'97)*, Lecture Notes in Computer Science, pages 346–360. Springer-Verlag.

Bagnara, R. (1996). A reactive implementation of pos using ROBDDs. In Kuchen, H. and Swierstra, S. D., editors, *8th International Symposium on Programming Languages, Implementation, Logic and Programs*, volume 1140 of *Lecture Notes in Computer Science*, pages 107–121. Springer-Verlag.

Bagnara, R., Hill, P. M., Mazzi, E., and Zaffanella, E. (2005). Widening operators for weakly-relational numeric abstractions. In Hankin, C., editor, *Static Analysis : Proceedings of the 12th International Symposium*, Lecture Notes in Computer Science, London, UK. Springer.

Bagnara, R. and Schachte, P. (1999). Factorizing equivalent variable pairs in ROBDD-based implementations of *Pos*. In Haeberer, A. M., editor, *7th International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485. Springer.

Bartlett, K. A., Scantlebury, R. A., and Wilkinson, P. T. (1969). A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5) :260–261.

Behrmann, Larsen, Pearson, Weise, and Yi (1999). Efficient timed reachability analysis using clock difference diagrams. In Halbwachs, N. and Peled, D., editors, *Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*. Springer.

Biehl, M., Klarlund, N., and Rauhe, T. (1997). Algorithms for guided tree automata. In Raymond, D. R., Wood, D., and Yu, S., editors, *First International Workshop on Implementing Automata (WIA'96)*, volume 1260 of *Lecture Notes in Computer Science*, pages 6–25. Springer-Verlag.

Biere, A., Cimatti, A., Clarke, E. M., Fujita, M., and Zhu, Y. (1999). Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC'99)*, pages 317–320. ACM Press.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2002). *The Essence of Computation : Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, chapter Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, pages 85–108. Lecture Notes in Computer Science 2566. Springer.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2003). A static analyzer for large safety-critical software. In *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages

196–207, San Diego, California, USA. ACM Press.

Boigelot, B. (1999). *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Faculté des Sciences Appliquées de l'Université de Liège.

Boigelot, B. and Godefroid, P. (1996). Symbolic verification of communication protocols with infinite state spaces using QDDs. In Alur, R. and Henzinger, T. A., editors, *8th International Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag.

Bouajjani, A. and Touili, T. (2002). Extrapolating tree transformations. In Brinksma, E. and Larsen, K. G., editors, *Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 539–554.

Bourdoncle, F. (1992). Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4) :407–423.

Bourdoncle, F. (1993). Efficient chaotic iteration strategies with widenings. In *International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer-Verlag.

Brandenburg, F. J. and Skodinis, K. (2005). Finite graph automata for linear and boundary graph languages. *Theoretical Computer Science*, 332 :199–232.

Bryant, R. E. (1986). Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35 :677–691.

Büchi, J. R. (1960). On a decision method in restricted second order arithmetics. In Nagel, E. et al., editors, *International Congress on Logic, Methodology and Philosophy of Science*. Stanford University Press.

Bultan, T., Gerber, R., and League, C. (1998). Verifying systems with integer constraints and boolean predicates : a composite approach. In *ISSTA '98 : Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123, New York, NY, USA. ACM Press.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, J. (1990). Symbolic modek checking : 10^{20} states and beyond. In *Fifth Annual Symposium on Logic in Computer Science (LICS'90)*, pages 428–439. IEEE Computer Society.

Bursztein, É. (2004). Conception d'un détecteur d'intrusion réseau par comportement anormal à l'aide de l'interprétation abstraite. Mémoire de stage de DEA Programmation : sémantique, preuves et langages, ENS.

Capron, G. (2004). Analyse statique de réactivité par interprétation abstraite. Mémoire de stage de DEA Programmation : sémantique, preuves et langages, ENS.

Cardon, A. and Crochemore, M. (1982). Partitioning a graph in $O(|A| \log_2 |V|)$. *Theoretical Computer Science*, 19 :85–98.

Chvatal, V. (1971). Hypergraphs and Ramseyian theorems. *Proceedings of the American Mathematical Society*, 27(3) :434–440.

Clarisó, R. and Cortadella, J. (2004). The octahedron abstract domain. In Giacobazzi, R., editor, *Static Analysis Symposium (SAS'04)*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer.

Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5) :1512–1542.

Clarke, E. M., Kroening, D., and Lerda, F. (2004). A tool for checking ANSI-C programs. In Jensen, K. and Podelski, A., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer.

Cleaveland, R., Iyer, S. P., and Yankelevich, D. (1995). Optimality in abstractions of model checking. In Mycroft, A., editor, *Static Analysis, Second International Symposium (SAS'95)*, volume 983 of *Lecture Notes in Computer Science*, pages 51–63. Springer-Verlag.

Colmerauer, A. (1982). PROLOG and infinite trees. In Clark, K. L. and Tärnlund, S.-A., editors, *Logic Programming*, volume 16 of *APIC Studies in Data Processing*, pages 231–251. Academic Press.

Considine, J. (2000). Efficient hash-consing of recursive types. Technical Report BUCS-TR-2000-006, CS Department, Boston University.

Courcelle, B. (1994). Monadic second-order definable graph transductions : A survey. *Theoretical Computer Science*, 126 :53–75.

Cousot, P. (1978). *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. PhD thesis, Université de Grenoble.

Cousot, P. (1981). Semantic foundations of program analysis. In Muchnick, S. and Jones, N., editors, *Program Flow Analysis : Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Cousot, P. (1996). Abstract interpretation. *ACM Computing Surveys*, 28(2) :324–328.

Cousot, P. (1999). The calculational design of a generic abstract interpreter. In Broy, M. and Steinbrüggen, R., editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam.

Cousot, P. (2000). Interprétation abstraite. *Technique et science informatique*, 19(1-2-3) :155–164.

Cousot, P. (2005). Integrating physical systems in the static analysis of embedded control software. In Yi, K., editor, *Programming Languages and Systems, Third Asian Symposium (APLAS'05)*, volume 3780 of *Lecture Notes in Computer Science*, pages 135–138. Springer.

Cousot, P. and Cousot, R. (1977a). Abstract interpretation : a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, New York, NY.

Cousot, P. and Cousot, R. (1977b). Automatic synthesis of optimal invariant assertions : mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, volume 12(8) of *ACM SIGPLAN Notices*, pages 1–12.

Cousot, P. and Cousot, R. (1979a). Constructive version of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1) :43–57.

Cousot, P. and Cousot, R. (1979b). Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'79)*, pages 269–282, San Antonio, Texas. ACM Press, New York, NY.

Cousot, P. and Cousot, R. (1992). Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4) :511–547.

Cousot, P. and Cousot, R. (1995). Formal languages, grammar and set-constraint-based program analysis by abstract interpretation. In *Conference on Functional Programming and Computer Architecture (FPCA'95)*, pages 170–181. ACM Press.

Cousot, P. and Cousot, R. (2000). Temporal abstract interpretation. In *Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 12–25, Boston, Mass. ACM Press, New York, NY.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2005). The Astrée analyzer. In Sagiv, M., editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer.

Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear constraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, New York, NY.

Daciuk, J., Mihov, S., Watson, B. W., and Watson, R. (2000). Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1) :3–16.

de la Briandais, R. (1959). File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, pages 295–298.

Emerson, E. A. and Halpern, J. Y. (1985). Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer System Science*, 30(1) :1–24.

Emerson, E. A. and Jutla, C. S. (1991). Tree automata, mu-calculus and determinacy. In *32nd Annual Symposium on Foundations of Computer Science (FOCS'91)*, pages 368–377. IEEE Computer Society.

Engelfriet, J. and Heyker, L. (1991). The string generating power of context-free hypergraph grammars. *Journal of Computer and System Sciences*, 43(2) :328–360.

Feret, J. (2004). Static analysis of digital filters. In Schmidt, D. A., editor, *European Symposium on Programming (ESOP'04)*, number 2986 in Lecture Notes in Computer Science, pages 33–48. Springer.

Feret, J. (2005a). Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming*, 63(1) :59–130.

Feret, J. (2005b). The arithmetic-geometric progression abstract domain. In Cousot, R., editor, *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, number 3385 in Lecture Notes in Computer Science, pages 42–58. Springer.

Feret, J. (2005c). Numerical abstract domains for digital filters. In *International workshop on Numerical and Symbolic Abstract Domains (NSAD'05)*, Electronic Notes in Theoretical Computer Science. Elsevier.

Fredkin, E. (1960). Trie memory. *Communications of the ACM*, 3(9) :490–499.

Fribourg, L. and Olsén, H. (1997). Reachability sets of parameterized rings as regular languages. *Electronic Notes on Theoretical Computer Science*, 9.

Gécseg, F. and Steinby, M. (1984). *Tree Automata*. Akadémia Kiadó, Budapest.

Goto, E. (1974). Monocopy and associative algorithms in extended Lisp. Technical Report TR 74-03, University of Tokyo.

Goubault, E. (2000). Geometry and concurrency : a user's guide. *Mathematical Structures in Computer Science*, 10(4) :411–425.

Goubault, J. (1993). Implementing functional languages with fast equality, sets and maps : an exercise in hash consing. In *Journées Francophones des Langages Applicatifs (JFLA'93)*, pages 222–238.

Goubault, J. (1994). HimML : Standard ML with fast sets and maps. In *5th ACM SIGPLAN Workshop on ML and it's Applications (ML'94)*.

Granger, P. (1989). Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30 :165–190.

Handjieva, M. and Tzolovski, S. (1998). Refining static analyses by trace-based partitioning using control flow. In Levi, G., editor, *Static Analysis Symposium (SAS'98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 200–214. Springer.

Helmink, L., Sellink, M. P. A., and Vaandrager, F. W. (1994). Proof-checking a data link protocol. In Barendregt, H. and Nipkow, T., editors, *Types for Proofs and Programs (TYPES'93)*, volume 806 of *Lecture Notes in Computer Science*, pages 127–165. Springer-Verlag.

Hopcroft, J. E. (1971). An $n \log n$ algorithm for minimizing states in a finite automaton. In Kohavi, Z. and Paz, A., editors, *Theory of machines and computations*, pages 189–196. Academic Press.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Adison-Wesley Publishing Company, Reading, USA.

IEEE Computer Society (1985). IEEE standard for binary floating-point arithmetic.

Janssens, D. and Rozenberg, G. (1981). A characterization of context-free string languages by directed node-label controlled graph grammars. *Acta Informatica*, 16 :63–85.

Jeannet, B. (2003). Dynamic partitioning in linear relation analysis : Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1) :5–37.

Karr, M. (1976). Affine relationships among variables of a program. *Acta Informatica*, 6 :133–151.

Kildall, G. A. (1973). A unified approach to global program optimization. In *1st ACM Symposium on Principles of Programming Languages (POPL'73)*, pages 194–206. ACM Press, New York, NY.

Kozen, D. (1992). On the Myhill-Nerode theorem for trees. *Bulletin of the EATCS*, 47 :170–173.

Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2) :125–143.

Le Charlier, B. and van Hentenryck, P. (1993). Groundness analysis for Prolog : Implementation and evaluation of the domain `prop`. In *PEPM'93*, pages 99–110.

Liaw, H.-T. and Lin, C.-S. (1992). On the OBDD-representation of general boolean functions. *IEEE Transactions on Computers*, 41(6) :661–664.

Martens, W. and Niehren, J. (2005). Minimizing tree automata for unranked trees. In Bierman, G. M. and Koch, C., editors, *International Symposium on Database Programming Languages 2005 (DBPL'05)*, volume 3774 of *Lecture Notes in Computer Science*, pages 232–246. Springer.

Mauborgne, L. (1994). Abstract interpretation using TDGs. In Le Charlier, B., editor, *Static Analysis Symposium (SAS'94)*, volume 864 of *Lecture Notes in Computer Science*, pages 363–379. Springer-Verlag.

Mauborgne, L. (1998). Abstract interpretation using typed decision graphs. *Science of Computer Programming*, 31(1) :91–112.

Mauborgne, L. (1999a). Binary decision graphs. In Cortesi, A. and Filé, G., editors, *Static Analyis Symposium (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, pages 101–116. Springer.

Mauborgne, L. (1999b). *Representation of Sets of Trees for Abstract Interpretation*. PhD thesis, École Polytechnique.

Mauborgne, L. (2000a). Improving the representation of infinite trees to deal with sets of trees. In Smolka, G., editor, *European Symposium on Programming (ESOP 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 275–289. Springer.

Mauborgne, L. (2000b). An incremental unique representation for regular trees. *Nordic Journal of Computing*, 7(4) :290–311.

Mauborgne, L. (2000c). Tree schemata and fair termination. In Palsberg, J., editor, *Static Analyis Symposium (SAS'00)*, volume 1824 of *Lecture Notes in Computer Science*, pages 302–320. Springer.

Mauborgne, L. (2003). Infinitary relations and their representation. *Science of Computer Programming*, 47 :121–144.

Mauborgne, L. (2004). ASTRÉE : Verification of absence of run-time error. In Jacquart, R., editor, *Building the information Society (18th IFIP World Computer Congress)*, pages 384–392. The International Federation for Information Processing, Kluwer Academic Publishers.

Mauborgne, L. and Rival, X. (2005). Trace partitioning in abstract interpretation based static analyzers. In Sagiv, M., editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer.

Mehlhorn, K. and Tsakalidis, A. K. (1990). Data structures. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume A : Algorithms and Complexity, pages 301–342. Elsevier Science Publishers.

Michie, D. (1968). "memo" functions and machine learning. *Nature*, 218 :19–22.

Minato, S. (1993). Zero-suppressed BDDs for set manipulation in combinatorial problems. In *30th ACM/IEEE Design Automaton Conference (DAC'93)*, pages 272–277.

Miné, A. (2001). The octagon abstract domain. In *Analysis, Slicing and Transformation (AST'01) (part of the Working Conference on Reverse Engineering)*, IEEE, pages 310–319. IEEE CS Press.

Miné, A. (2004a). Relational abstract domains for the detection of floating-point run-time errors. In Schmidt, D. A., editor, *European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer.

Miné, A. (2004b). *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique.

Miné, A. (2006a). Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In Irwin, M. J. and Bosschere, K. D., editors, *ACM SIGPLAN/-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63. ACM Press.

Miné, A. (2006b). The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19 :31–100.

Miné, A. (2006c). Symbolic methods to enhance the precision of numerical abstract domains. In Emerson, E. A. and Namjoshi, K. S., editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *Lecture Notes in Computer Science*, pages 348–363. Springer.

Møller, J. B., Lichtenberg, J., Andersen, H. R., and Hulgaard, H. (1999). Difference decision diagrams. In *Computer Science Logic*, pages 111–125. The IT University of Copenhagen.

Monniaux, D. (2005). The parallel implementation of the Astrée static analyzer. In Yi, K., editor, *Programming Languages and Systems, Third Asian Symposium (APLAS'05)*, volume 3780 of *Lecture Notes in Computer Science*, pages 86–96. Springer.

Morrison, D. R. (1968). PATRICIA — practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4) :514–534.

Mycroft, A. (1980). The theory and practice of transforming call-by-need into call-by-value. In Robinet, B., editor, *Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer-Verlag.

Niwiński, D. (1984). Equational μ-calculus. In Skowron, A., editor, *Computation Theory*, volume 208 of *Lecture Notes in Computer Science*, pages 169–176. Springer-Verlag.

Parikh, R. J. (1966). On context-free languages. *Journal of the ACM*, 13(4) :570–581.

Pnueli, A. (1977). The temporal logic of programs. In *Foundation of Computer Science (FOCS'77)*, pages 46–57. IEEE.

Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., and Schoen, D. (1999). Applying formal proof techniques to avionics software : A pragmatic approach. In Wing, J. M., Woodcock, J., and Davies, J., editors, *World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1798–1815. Springer.

Rival, X. (2005). Understanding the origin of alarms in Astrée. In Hankin, C. and Siveroni, I., editors, *Static Analysis Symposium (SAS'05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 303–319. Springer.

Rugina, R. (2004). Quantitative shape analysis. In Giacobazzi, R., editor, *Static Analysis Symposium (SAS'04)*, pages 228–245. Springer.

Sedgewick, R. (1997). *Algorithms in C, Parts 1-4 : Fundamentals, Data Structures, Sorting, Searching*. Addison Wesley Professional.

Seidl, H. (1990). Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3) :424–437.

Snyder, L. (1977). On uniquely representable data structures. In *18th annual IEEE Symposium on Foundations of Computer Science (FOCS'77)*, pages 142–146, Washingtown, DC. IEEE Computer Society.

Souyris, J. and Favre-Felix, D. (2004). Proof of properties in avionics. In Jacquart, R., editor, *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions*, pages 527–536. Kluwer.

Srinivasan, A., Kam, T., Malik, S., and Brayton, R. K. (1990). Algorithms for discrete function manipulation. In *IEEE International Conference on Computer-Aided Design (ICCAD'90)*, pages 92–95.

Stoller, S. D. and Liu, Y. A. (1998). Efficient symbolic detection of global properties in distributed systems. In Hu, A. J. and Vardi, M. Y., editors, *10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 357–368. Springer.

Tanenbaum, A. S. (1981). *Computer Networks*. Prentice-Hall.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160.

Thatcher, J. W. (1967). Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Computer System Sciences*, 1(4) :317–322.

Thatcher, J. W. and Wright, J. B. (1968). Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2 :57–82.

Traverse, P., Lacaze, I., and Souyris, J. (2004). Airbus fly-by-wire - a total approach to dependability. In Jacquart, R., editor, *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions*, pages 191–212. Kluwer.

Venet, A. (1996). Abstract cofibered domains : Application to the alias analysis of untyped programs. In Cousot, R. and Schmidt, D. A.,

editors, *Static Analysis Symposium (SAS'96)*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382, Aachen, Germany. Springer-Verlag.

Venet, A. and Brat, G. P. (2004). Precise and efficient static array bound checking for large embedded C programs. In Pugh, W. and Chambers, C., editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*, pages 231–242. ACM.

Villemot, S. (2002). Automates finis et interprétation abstraite, applications à l'analyse statique de protocoles de communication. Mémoire de stage de DEA Programmation : sémantique, preuves et langages, ENS.

Wang, F. (2004). Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. In Alur, R. and Peled, D., editors, *Computer Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 295–307. Springer.

Watson, B. W. (1993). A taxonomy of finite automata construction algorithms. Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands.