Analyse statique : de la théorie à la pratique ; analyse statique de code embarqué de grande taille, génération de domaines abstraits

David Monniaux

To cite this version:

David Monniaux. Analyse statique : de la théorie à la pratique ; analyse statique de code embarqué de grande taille, génération de domaines abstraits. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2009. tel-00397108

HAL Id: tel-00397108
https://theses.hal.science/tel-00397108v1

Submitted on 19 Jun 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MÉMOIRE D'HABILITATION À DIRIGER LES RECHERCHES

Spécialité : informatique

présenté par

David Monniaux

à l'Université Joseph-Fourier, Grenoble

Sujet :

Analyse statique : de la théorie à la pratique,
Analyse statique de code embarqué de grande taille, génération de domaines abstraits

Static analysis: from theory to practice,
Static analysis of large-scale embedded code, generation of abstract domains

Remerciements

Merci aux membres de l'équipe « sémantique et interprétation abstraite » du LIENS, et à ceux de l'équipe « Synchrone » de VERIMAG.

Merci à ceux qui ont accepté de relire ce manuscrit, et aux membres du jury.

Merci à mon épouse.

Table des matières

Remerciements **iii**

1 Introduction **1**

1.1 Une fusée qui explose, et tout est dépeuplé 1

1.1.1 Quelques accidents de systèmes informatisés critiques . 2

1.1.2 Quelles solutions ? 3

1.2 Analyse statique 6

1.3 Interprétation abstraite 12

1.4 Remarques sur le présent mémoire 17

2 Le projet Astrée **19**

2.1 L'outil Astrée 19

2.1.1 Contexte industriel 19

2.1.2 Fonctionnement 22

2.2 Études et développements annexes 24

2.2.1 Efficacité 24

2.2.2 Preuve formelle de structures de données 25

2.2.3 Implémentation parallèle 26

2.2.4 Preuve formelle et définition de l'élargissement 28

2.2.5 SAT-solving 29

2.2.6 Filtres récurrents linéaires 30

2.2.7 Analyse de programmes asynchrones, cas d'un pilote de périphérique 33

3 Analyse de programmes et calculs en virgule flottante **39**

3.1 Sémantique des opérations flottantes 41

3.1.1 Généralités 41

3.1.2 Sémantique des expressions 42

3.1.3 Réécritures optimisantes 43

3.1.4 Double arrondi 44

3.1.5 Des fonctions mal spécifiées et peu sûres 47

3.1.6 Une situation difficile 48

3.2 Abstraction 48

3.2.1 Abstractions non sûres 48
3.2.2 Raisonnements corrects 49

4 Élimination des quantificateurs et abstractions 51
4.1 Algorithmes 52
4.1.1 Élimination de quantificateurs existentiels sur les conjonctions 52
4.1.2 Méthodes par substitution 55
4.1.3 Méthode géométrique 57
4.2 Application à l'analyse modulaire de programmes 61
4.2.1 Domaines de modèles de contraintes linéaires 61
4.2.2 Formulation de la recherche de postconditions et d'invariants 63
4.2.3 Généralisation au cas non-linéaire 65
4.2.4 Correction 65
4.2.5 Extensions et perspectives 66

5 Témoins d'insatisfiabilité 69
5.1 Insatisfiabilité de formules 69
5.1.1 Procédures de décision 69
5.1.2 Exemples de théories avec témoins d'insatisfiabilité . . 71
5.2 Cas des inégalités polynomiales réelles 72
5.2.1 Élimination de quantificateurs et décision 73
5.2.2 Incompatibilités algébriques 74
5.2.3 Sommes de carrés 75
5.2.4 Programmation semidéfinie 76

6 Conclusion et perspectives 79

Bibliographie 85

Index 105

Chapitre 1

Introduction

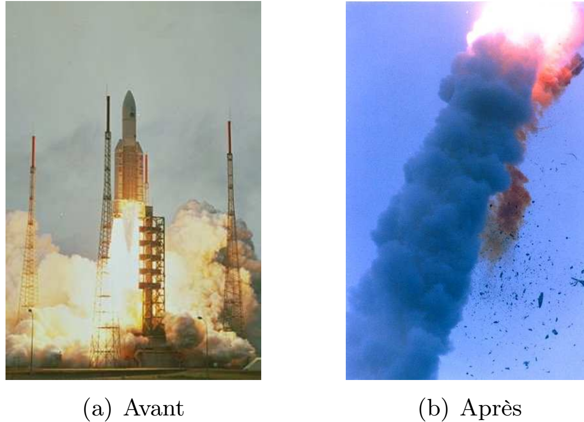
> *Personne n'ignore qu'il y a deux entrées par où les opinions sont reçues dans l'âme, qui sont ses deux principales puissances, l'entendement et la volonté. La plus naturelle est celle de l'entendement, car on ne devrait jamais consentir qu'aux vérités démontrées ; mais la plus ordinaire, quoique contre la nature, est celle de la volonté ; car tout ce qu'il y a d'hommes sont presque toujours emportés à croire non pas par la preuve, mais par l'agrément. Cette voie est basse, indigne et étrangère : aussi tout le monde la désavoue.*
>
> Pascal, *De l'art de persuader*

Mes travaux sont plutôt de nature mathématique et algorithmique ; toutefois ils ont pour visée plus ou moins lointaine une application concrète : la réalisation de systèmes informatiques fiables, notamment dans le cas où des vies humaines sont en jeu. Il me semble donc important de replacer ces travaux non seulement dans leur contexte scientifique, mais aussi dans leur contexte sociétal.

1.1 Une fusée qui explose, et tout est dépeuplé

Dans la vie courante, nous constatons des dysfonctionnements informatiques : caisses enregistreuses de supermarché « bloquées », affichages électroniques en panne, ordinateur personnel qui se fige en plein travail, etc. Dans la plupart des cas, ces dysfonctionnements n'ont pas grande importance, du moins pris isolément : quelques minutes, quelques heures, de travail seront perdus, les clients attendront un peu, mais cela n'est finalement pas grave.[1]

[1] Il est cependant possible que ces dysfonctionnements, pris globalement, aient une importance non négligeable pour l'économie. Une étude de 2002 évaluait à 59 milliards de dollars, soit 0,6% du produit intérieur brut, le coût annuel pour les États-Unis d'Amérique



(a) Avant (b) Après

Fig. 1.1 – L'accident du vol inaugural de la fusée Ariane 5. Crédit photo : ESA.

On distinguera le cas des *systèmes critiques*, dont les dysfonctionnements peuvent entraîner morts, blessures, ou du moins des coûts élevés.

1.1.1 Quelques accidents de systèmes informatisés critiques

L'accident causé par un logiciel défectueux le plus connu du grand public est sans doute l'explosion du vol inaugural de la fusée Ariane 5 le 4 juin 1996 : à environ 37 s après son décollage, la fusée a dévié de sa trajectoire avant de s'autodétruire. Une commission d'enquête établira par la suite que cet accident était dû à la réutilisation d'un logiciel prévu pour la fusée Ariane 4 et dont on n'avait jamais vérifié qu'il fonctionnait bien sur la nouvelle fusée [120]. D'autres missions spatiales, américaines celles-ci, ont échoué en raison de problèmes informatiques : citons notamment *Mars Polar Lander* et *Deep Space 2* [3], *Mars Climate Orbiter* [161, 193] et le satellite de communications militaires MILSTAR 3 [165]. La sonde *Deep Space One* a eu des problèmes de bloquages entre processus [166], l'engin *Mars Pathfinder* a subi des pannes du fait d'un problème d'inversion de priorité entre processus concurrents [169]. Ces problèmes ont d'ailleurs poussé la NASA à essayer diverses techniques de test et de vérification de logiciels [26].

L'accident d'Ariane 5, pour spectaculaire et coûteux qu'il ait été, n'a cependant pas provoqué de dommages corporels. Cela n'a pas toujours été le cas. Entre 1985 et 1987, un logiciel de pilotage défectueux dans l'équipement de radiothérapie *Therac-25* a provoqué la mort d'au moins 3 patients et grièvement blessé au moins 3 autres [119]. En 1991, lors de l'opération *Desert Storm*, 28 soldats américains ont été tués par un missile Scud. Un missile

de leur infrastructure insuffisante en matière de tests de logiciels, dont 38 à la charge des utilisateurs de logiciels et 21 pour les développeurs [178, p. 23].

anti-missile *Patriot* aurait dû intercepter le Scud, mais une erreur dans le logiciel de contrôle du Patriot a empêché cela [19, 188]. En 2000, au moins 27 patients traités par radiothérapie dans un hôpital panaméen ont reçu des doses excessives en raison d'un mauvais calcul par un logiciel de préparation ; 21 sont morts dans les années suivantes [23, 79, 100].

Les avions commerciaux actuels font un ample usage de techniques informatisées. Ainsi, Airbus, à partir de l'A320 (premier vol commercial en 1988) et Boeing, à partir du B777 (1995), utilisent des *commandes de vol électriques* (*fly-by-wire*), c'est-à-dire que les liaisons hydromécaniques entre le cockpit et les gouvernes sont supprimées, ou du moins réservées à des cas de détresse, et remplacées par une signalisation électrique sous contrôle informatique [15, 27, 28]. Selon les normes en vigueur dans l'aéronautique commerciale, ces commandes sont classées au niveau A de sécurité, c'est-à-dire qu'un fonctionnement anormal pourrait empêcher ou contribuer à empêcher la continuation sûre du vol et l'atterrissage [177]. Au regard des risques tant humains que financiers, on ne s'étonnera pas que le développement des logiciels utilisés au niveau A soit soumis à des règles très strictes, quelque peu relâchées pour les niveaux B et C. Cela n'a cependant pas empêché un Airbus A340 sur la route de Hong-Kong à Londres de devoir se poser en urgence à Amsterdam en 2005, en raison d'un problème d'alimentation de ses moteurs dû à une panne des calculateurs contrôlant les flux de carburant [5, 12]. L'informatique embarquée dans les avions est maintenant complexe, avec de véritables ordinateurs et des réseaux internes [78].

La mauvaise conception de matériels vendus à un grand nombre d'exemplaires peut avoir de graves conséquences pour le constructeur, à la fois pour ses finances et son image de marque, sans parler d'éventuelles victimes. En 1994, un professeur de mathématiques qui faisait des expériences de théorie des nombres s'est aperçu que le microprocesseur Pentium calculait incorrectement certaines divisions [37, 157]. Le fabricant, Intel, le plus important fabricant de microprocesseurs pour ordinateurs personnels, a dû accepter de remplacer gratuitement les microprocesseurs défectueux, après que sa proposition de ne remplacer que ceux des utilisateurs qui pouvaient démontrer un préjudice eut suscité l'indignation. Toyota a dû rappeler des dizaines de milliers d'automobiles de son modèle *Prius* ; des dysfonctionnements logiciels provoquaient des allumages intempestifs des feux d'avertissement, ainsi que des arrêts du moteur, y compris aux allures routières. Or, la tendance actuelle est à la complication de l'informatique embarquée dans les automobiles.

1.1.2 Quelles solutions ?

Certaines faiblesses en matière de sûreté informatique proviennent de l'utilisation d'approches, de méthodes, provenant d'autres branches de l'ingénierie mais inadaptées dans le cas de réalisation de logiciels. Ainsi, dans le cas des pannes matérielles « aléatoires » (usure etc.), on estime souvent que

les pannes de plusieurs pièces sont des évènements indépendants (ou presque indépendants).[2] Dans ce cas, on peut augmenter la sécurité du système en utilisant du matériel *redondant* : au lieu de mettre un dispositif, on en mettra plusieurs. Dans le cas de commandes informatisées, cette approche souffre de plusieurs défauts. Tout d'abord, si la redondance est facile à organiser dans le cas de systèmes où il suffit qu'un seul des dispositifs redondants fonctionne, par exemple des points de verrouillage dont il suffit qu'un seul soit enclenché, elle est plus difficile quand il s'agit de commander un dispositif : que faire si l'on a deux commandes et que l'une dit « oui » et l'autre dit « non » ? Les dispositifs redondants de commande doivent donc souvent être compliqués de dispositifs de « vote » (si deux sur les trois calculateurs disent « oui » et le troisième « non » alors je choisis « oui »). Une mauvaise conception d'ensemble de la redondance est responsable de l'incident de carburant de l'A340 cité plus haut. Un tel dysfonctionnement est d'ailleurs indétectable si l'on se concentre sur la fiabilité de chaque calculateur pris isolément

Plus grave, la redondance entre dispositifs identiques est inutile s'il s'agit de dysfonctionnements logiciels : si un premier boîtier signale une erreur suite à un problème de conception de son logiciel, un second boîtier fonctionnant avec le même logiciel sur les mêmes données va aboutir au même résultat, et donc tomber également en panne.[3] C'est justement ce qui est arrivé avec le vol Ariane 501 [120]. Pour pallier ce type de problèmes, on s'assure d'avoir une redondance non seulement au niveau du matériel lui-même, mais au niveau de l'architecture matérielle (utilisation de différents types de microprocesseurs), des logiciels (développement des logiciels par plusieurs équipes séparées) et même des spécifications, ce afin de limiter les modes communs de pannes [27, §4.2]. L'idée est que les fautes de conceptions d'équipes de développement indépendantes sont supposées être indépendantes (ou presque) en termes de probabilités. Cette hypothèse est cependant contestée, notamment parce que face au même problème, différents programmeurs travaillant indépendamment peuvent adopter la même analyse erronée, trébucher sur la même difficulté [30, 114].[4] Il ne suffit donc pas d'utiliser des systèmes redondants conçus indépendamment les uns des autres, il faut aussi que la qualité intrinsèque de chaque système soit élevée.

J'adhère à la distinction que Leveson [118] fait entre les causes techniques d'un accident ou incident, et ses causes profondes ou systémiques, découlant de l'organisation ou des pratiques de l'organisation ayant réalisé le produit

[2] Au sens des probabilités : si la probabilité de panne d'une pièce est p, la probabilité de panne simultanée de k pièces est p^k.

[3] Certains suggèrent donc que l'on fasse travailler les différents dispositifs sans synchronisation stricte : en cas de problème transitoire qui provoquerait un dysfonctionnement des calculateurs, si les calculateurs ne sont pas synchronisés on peut espérer qu'une partie d'entre eux ne soient pas affectés par le transitoire.

[4] C'est en fait un constat courant chez les enseignants : la même erreur peut se retrouver dans différentes copies alors que les étudiants n'avaient pas la possibilité d'échanger des informations pendant l'examen.

défectueux. Comme elle l'explique bien, pour prendre un exemple hors informatique, l'accident de la navette spatiale *Challenger* en 1986 a été provoqué par un joint défectueux (cause technique), mais le rapport de la commission d'enquête [175], notamment l'annexe rédigée par Richard Feynman, explique que cet accident n'a été rendu possible que parce que les procédures de sécurité n'étaient pas bien appliquées, les problèmes relevés ignorés, etc. (causes systémiques). On retrouve des phénomènes similaires dans d'autres accidents ou incidents cités plus haut.

La résolution des problèmes systémiques relève du génie logiciel (organisation des équipes de développeurs et de testeurs, bonnes pratiques de programmation...) voire des études sur la gestion d'entreprise (mise en place de mécanismes effectifs permettant de rapporter les dysfonctionnements et de traiter ces rapports...), domaines dans lesquels je n'interviens pas.[5] Les buts pratiques de mes recherches concernent plutôt la détection automatique des problèmes techniques possibles.

Certains des exemples mentionnés montrent les limites d'une approche de sécurité qui ne considérerait que les causes techniques. Quelle serait l'utilité d'une méthode d'analyse automatique si les responsables ignorent délibérément les informations obtenues sur la sécurité de leur système,[6] si les ingénieurs n'utilisent pas la méthode d'analyse, ou l'utilisent sur un logiciel différent de celui qui est effectivement utilisé, ou en prenant des paramétrages pour un modèle de machine différent de celui utilisé ?[7] Quelle est la validité d'une analyse, dépendant de certaines hypothèses, si le système est finalement utilisé hors du cadre de celles-ci, dans des conditions différentes de ce pour quoi il a été prévu ? Enfin, quel est l'impact d'une analyse de bonne qualité si, de toute façon, ses résultats sont ignorés pour prendre des décisions effectives ? Toutefois, même si les solutions purement techniques ne

[5] Jackson [104] fait un panorama des méthodes actuellement employées dans le but d'améliorer la fiabilité des logiciels : normes de développement, méthodes formelles, utilisation de meilleurs langages de programmation...

[6] Richard Feynman, membre de la commission d'enquête sur l'accident de la navette *Challenger*, a notamment accusé les gestionnaires de la NASA d'avoir continué à faire voler les navettes à un rythme ne permettant pas de maintenir les engins en bon état, et d'avoir donné au public et aux décideurs politiques des estimations de probabilité de panne différant de trois ordres de grandeur par rapport à ce que l'on pouvait déduire des données des ingénieurs (1/100 000 au lieu de 1/100 vols) afin d'obtenir le maintien du financement de leurs activités [175, annexe F]. Si les responsables ignorent délibérément les données factuelles concernant la fiabilité de leur système, il ne sert à rien de leur fournir des moyens supplémentaires d'investigation technique. Dans le cas du Therac-25, on peine à comprendre comment le fabricant a pu ne pas réagir, ou réagir si faiblement, au vu des rapports d'incidents qui s'accumulaient [119].

[7] Les enquêtes postérieures aux accidents cités dénoncent l'insuffisance des procédures de test. Dans le cas de l'accident d'Ariane, on avait réutilisé un système prévu pour une fusée précédente sans le re-tester dans le nouvel environnement [120] ; pour le lancement de Milstar, les données de paramétrage validées n'étaient pas celles effectivement utilisées dans l'engin [118, 165]. Quant au Therac-25, l'analyse *a posteriori* de Leveson and Turner [119] laisse apparaître un manque quasi-total de procédures de contrôle.

suffisent pas, nous pouvons chercher à améliorer l'état de l'art en la matière ; c'est la visée pratique des travaux exposés dans ce mémoire.

Gilles Kahn, membre de la commission d'enquête sur l'explosion d'Ariane 5, aurait voulu écrire dans le rapport que cet accident provenait d'un problème de déficit de recherche[8] — ce qui lui a été refusé [54, p. 237]. Peut-être que si des méthodes performantes d'analyse automatique avaient été disponibles à l'époque, elles auraient été utilisées et l'accident évité. Pour être adoptées par l'industrie, les méthodes proposées devront s'intégrer dans le flot de travail, ne pas harceler les développeurs par des alertes superflues, et consommer des ressources de calcul modérées. Ces trois axes — meilleure insertion dans la chaîne de production du logiciel, précision de l'analyse, modération du coût en temps et en mémoire — ont été les principaux objectifs pratiques de mes travaux.

1.2 Analyse statique

Ma recherche s'est principalement inscrite dans le champ de l'*analyse statique de programmes par interprétation abstraite*. Il me semble important de replacer ce domaine à la fois dans le champ scientifique et dans le champ des technologies d'ingénierie logicielle.

L'*analyse de programmes* consiste à dériver automatiquement des informations à propos de ceux-ci. Elle peut servir à fournir un résultat au programmeur (par exemple, « vous avez probablement fait une erreur à la ligne 51 »), à l'utilisateur final (« le logiciel que vous avez acheté ne pourra jamais provoquer d'accès mémoire incorrect ») mais aussi à d'autres outils informatiques, comme des optimiseurs de compilateurs (« la variable `x` n'est pas au même endroit en mémoire que la variable `y` »).

L'*analyse statique* procède par examen du programme sans l'exécuter, à l'inverse de l'*analyse dynamique* qui observe l'exécution des programmes et désigne donc des formes avancées de *test*. Le terme d'analyse statique, suivant le locuteur, peut désigner des techniques très différentes tant dans leurs moyens que dans leurs finalités.[9] Ainsi, certains rangent dans cette catégorie des outils qui fournissent une information essentiellement *syntaxique*, concernant des règles *stylistiques* d'écriture de programmes, découlant du désir d'une organisation industrielle d'avoir une présentation uniforme des programmes qu'elle développe, une quantité minimale de documentation par ligne de code, etc. Certains outils avertissent de l'usage de constructions considérées comme dangereuses ou du moins suspectes,[10] voire interdites

[8] Certains ont d'ailleurs comparé le coût très important de l'explosion avec les sommes finalement assez modestes consacrées en France à la recherche en informatique, et notamment en sûreté de fonctionnement.

[9] On trouvera sur `http://spinroot.com/static/` une liste d'outils très divers se réclamant de l'analyse statique.

[10] Certains parlent de *pattern-based static analysis*.

par des règles visant à une programmation plus « sûre ». Ainsi, les messages d'avertissement des compilateurs (« vous avez écrit `if(x=y){`, vous voulez peut-être écrire `if(x==y){` ») ou d'outils comme le `lint` traditionnel d'Unix peuvent être considérés comme des analyses statiques. Certaines industries imposent des « règles de codage » dont la majorité peuvent être testées statiquement voire syntaxiquement [96], comme par exemple MISRA-C pour l'automobile [135].

Même si nous admettons l'utilité pratique de ces analyses,[11] celles-ci ne répondent pas à des problèmes aussi importants que :

- « Mon programme peut-il « planter » ? » ;
- plus généralement, « Mon programme répond-il à la spécification » ? », la quasi-totalité des spécifications excluant les « plantages ».

Pour répondre fiablement à ce genre de questions, il faut procéder à des analyses qui considèrent la *sémantique* des programmes, c'est-à-dire l'ensemble de leurs exécutions possibles.

Il est bien connu que l'analyse de programmes est un problème indécidable, équivalent au problème de l'arrêt des machines de Turing. Plus précisément, il est impossible qu'une méthode d'analyse de programme soit à la fois[12] :

1. automatique (sans intervention humaine, annotations, etc.) ;
2. correcte (*sound* — elle ne donne pas de résultats faux) ;
3. complète (*complete* — elle est capable de prouver tous les résultats vrais) ;
4. non limitée à des exécutions ou des mémoires bornées.

Le dernier point mérite une petite explication. Tout système informatique concret actuel (j'exclus ici les éventuels systèmes quantiques, etc.) a un nombre fini n de bits d'information, et donc une mémoire bornée. Tous les problèmes sont donc théoriquement décidables sur ces systèmes, ne serait-ce qu'en énumérant les 2^n états possibles. Il est cependant clair qu'on tombe alors dans des problèmes non plus de *calculabilité*, mais de *complexité*, c'est-à-dire qu'on peut obtenir le résultat désiré avec un nombre fini d'opérations, mais que ce nombre peut être prohibitif.

[11] Et même plus : nous pensons qu'il *faut* utiliser l'option `-Wall` de `gcc`, l'option `-w` de Perl...

[12] Ce résultat, dans sa version formelle dans le cadre de la théorie de la calculabilité sur les fonctions récursives partielles, est connu sous le nom de théorème de Rice ou de Rice-Myhill-Shapiro [174, p. 34][171, corollaire B]. Le problème est profond, et est lié aux théorèmes d'incomplétude de Gödel. Cook [44] a démontré la complétude de la logique de Hoare appliquée à des programmes à variables entières non bornées relativement à l'arithmétique de Péano [202, ch. 7], et le problème de l'arrêt sur de tels programme peut donc se formuler comme une formule de l'arithmétique. Par ailleurs, le problème de tester si une formule de l'arithmétique de Péano est démontrable s'exprime comme le problème de l'arrêt d'un tel programme.

Chacune des quatre conditions ci-dessus est nécessaire au théorème, et la relaxation de chaque condition ouvre la porte à des méthodes :

1. Les *assistants de preuve* opèrent sur une version formalisée du raisonnement mathématique, dans laquelle sont exprimés les programmes et les propriétés à vérifier. Ils aident l'utilisateur à construire des preuves, et vérifient que les preuves construites sont correctes. Tandis que la vérification des preuves est automatisée, leur construction ne l'est que partiellement, et requiert en général une intervention humaine. Citons notamment les outils PVS[13], COQ[14], ACL2[15] et ISABELLE[16].
2. Certains outils ne visent pas à prouver l'absence d'erreurs dans un programme, mais à rechercher les erreurs existantes. Par souci d'efficacité, ces outils ne garantissent en général pas qu'ils trouveront toutes les erreurs possibles (ils sont *incorrects* ou *unsound*). Il existe un certain nombre d'outils industriels, dont les bases et le fonctionnement sont malheureusement rarement décrits autrement que dans des brochures commerciales. Citons par exemple les logiciels de COVERITY[17].
3. Certains outils ne « mentent » jamais (du moins, en l'absence de « bugs » au sein de l'outil) mais peuvent échouer à prouver des propriétés vraies (ils sont *incomplets*). En termes d'analyses de programmes, ces outils afficheront des messages d'avertissement pour des erreurs éventuelles qui ne peuvent pas se produire dans la réalité (*fausses alarmes*) parce qu'ils ne sont pas arrivés à prouver que ces erreurs ne peuvent pas se produire. C'est le cas des prouveurs automatiques de théorèmes, ou des analyses par *interprétation abstraite*. La plupart des outils industriels basés sur l'abstraction prennent quelques libertés avec la correction sémantique ; il semble que POLYSPACE VERIFIER[18] soit l'un des rares à effectivement garantir la correction de ses résultats.
4. Enfin, certains outils supposent un état du système, une mémoire, bornés et « pas trop grand ». Nous avons déjà évoqué la méthode d'énumération des états accessibles, dite *model-checking* explicite. Il existe des méthodes algorithmiques astucieuses pour représenter des ensembles d'états, comme les BDD [2]. On parle alors de *model-checking* symbolique ; citons notamment NuSMV[19]. Clarke et al. [39] font un panorama des différentes techniques.

   Lorsque les traces d'exécution sont bornées, il est souvent possible de transformer le problème de la recherche d'une trace vérifiant une cer-

[13]Voir `http://pvs.csl.sri.com/` et [163].

[14]Voir `http://coq.inria.fr/` et l'ouvrage de Bertot and Castéran [18].

[15]Voir `http://www.cs.utexas.edu/users/moore/acl2/` et l'ouvrage de Kaufmann et al. [113]

[16]Voir `http://isabelle.in.tum.de/` et le tutorial de Nipkow et al. [160].

[17]`http://www.coverity.com/`

[18]`http://www.polyspace.com/`

[19]`http://nusmv.fbk.eu/`

taine propriété (par exemple, aboutissant sur un état indésirable) en un problème de *satisfiabilité* d'une formule logique, que l'on résout à l'aide d'algorithmes de *SAT-solving*, notamment ceux de la famille DPLL (Davis-Putnam-Logemann-Loveland) [57, 58].

Si le système a une mémoire non bornée ou trop grande, il est éventuellement possible de se ramener à un système borné raisonnable par application de techniques relevant des points précédents. On trouvera donc des mélanges de *model-checking* avec des méthodes de preuve assistée (interventions humaines), des réductions « incorrectes » de problèmes, des réductions incomplètes...

Insistons sur le fait qu'il est possible de combiner plusieurs des techniques ci-dessus. Ainsi, des techniques SAT propositionnelles combinées avec des procédures de décision (des prouveurs automatiques à la fois complets et corrects, mais opérant sur une logique restreinte) donnent du SAT modulo théorie ou SMT [80]. La technique de raffinement par contre-examples (*counterexample-based refinement* ou CEGAR) calcule une abstraction qui, si elle est insuffisamment précise, produit un contre-exemple, une « fausse alarme » dont la fausseté est automatiquement détectée ; l'abstraction est raffinée en fonction de ce contre-exemple. [38] Ces techniques CEGAR peuvent elles-mêmes se baser sur du SMT, et sacrifier une certaine partie de la correction dans l'optique de trouver des erreurs plutôt que de tenter de prouver leur absence, comme dans l'outil SLAM[20]. Ce domaine est en perpétuelle évolution et nous ne saurions dans cette introduction prétendre à une quelconque exhaustivité...

Pour dire qu'une méthode est correcte, encore faut-il savoir par rapport à quoi. On opère à partir d'une sémantique du langage, c'est-à-dire d'une définition mathématique du sens des constructions du langage, permettant de distinguer pour un programme donné les exécutions possibles des exécutions impossibles. Idéalement, cette sémantique devrait être formalisée le plus précisément possible sous forme mathématique ; en pratique, on formalise bien plus facilement des petits langages jouets [202], ou un sous-langage du langage considéré, qu'un langage industriel comme C au complet.[21] Les méthodes d'analyse ou de preuve de programmes (ou de matériels, de spécifications...) basées sur une sémantique et des raisonnements mathématiques sont appelées *méthodes formelles*.

[20] `http://research.microsoft.com/slam/`, voir par exemple [11].

[21] Notons également que les sémantiques de langages s'appuient souvent sur des modèles d'exécution simplifiés et parfois incorrects. Par exemple, nous verrons ainsi au Ch. 3 que la sémantique effective des calculs en virgule flottante peut ne pas correspondre aux attentes habituelles des programmeurs. Autre exemple, Sarkar et al. [183] ont réussi, au prix de grandes difficultés, à proposer une sémantique des accès mémoire sur les systèmes multiprocesseurs ou multicœurs de l'architecture Intel, et cette sémantique n'est pas une sémantique par simples entrelacements alors que c'est ce modèle qu'ont généralement en tête les programmeurs et les auteurs d'algorithmes opérant en mémoire partagée sans verrouillages.

Une erreur consiste en une violation de la *spécification* du programme, c'est-à-dire d'une description de ce qu'il était censé faire, de propriétés qu'il était censé vérifier. Le plus souvent, les spécifications sont imprécises, écrites en langue naturelle (français, anglais, etc.) ; avant de pouvoir appliquer des méthodes formelles, il faudra les traduire dans un langage mathématique, ce qui est souvent malaisé. La rédaction de spécifications précises est donc un exercice délicat, pourtant indispensable si l'on se donne l'objectif ambitieux de montrer que le programme fait ce que demande sa spécification. Il ne faut d'ailleurs pas oublier que même si l'on arrive à cette preuve que le programme met en œuvre sa spécification, il est possible que celle-ci soit défectueuse, en d'autre terme, que l'ordinateur fera une chose stupide parce que c'est cela qui était attendu de lui !

Plus modestement, on pourra en revanche se reposer sur des spécifications implicites, découlant automatiquement du programme : par exemple, que le programme ne doit pas soulever d'exception système (dépassement de capacité, division par zéro, accès à une mémoire inexistante...), qu'il ne doit pas produire de comportement indéfini selon la norme du langage de programmation utilisé, évènements que nous appellerons « erreurs à l'exécution » (*runtime errors*).

Les méthodes purement automatiques et sémantiquement correctes ont parfois la réputation de produire un nombre trop élevé de « fausses alarmes ». Prenons le cas d'un logiciel de 200 000 lignes de code, pour lequel une analyse ne produit que 3% d'alarmes, un résultat apparemment satisfaisant. Ceci représente un total de 6 000 alarmes qu'il faudra individuellement étudier, au prix d'un grand travail humain, sans parler de la lassitude des équipes découvrant que la très grande majorité des alarmes sont « fausses ». Un tel outil est donc très peu utile en pratique.[22] Un industriel pragmatique pourra préférer un outil qui ne garantit pas de trouver tous les problèmes possibles mais dont les alarmes se révèlent souvent justifiées. C'est ainsi que les problèmes d'une part de la *preuve d'absence* d'erreurs, et d'autre part de leur *recherche*, apparemment identiques, appellent en pratique des méthodes différentes.

Il existe en revanche des applications pour lesquelles il n'est pas grave que quelques pourcents des propriétés vraies ne puissent être prouvées. Prenons par exemple un compilateur pour un langage qui vérifie que l'on ne tente pas d'accéder à un tableau en sortant de ses bornes. Il est coûteux de tester les indices à chaque accès, aussi un compilateur optimisant pourra supprimer ces tests s'il détermine qu'ils sont inutiles. Pour reprendre l'exemple des 3% ci-dessus, une analyse capable de supprimer 97% des tests fournira en général une excellente optimisation. La suppression des tests de borne de tableau motivait d'ailleurs les premières recherches en inférence d'invariants et de

[22] Xie and Aiken [204] rapportent ainsi que leur outil de détection de fuites de mémoire, pourtant sémantiquement incorrect, permet en pratique de trouver plus d'erreurs que des outils sémantiquement corrects parce qu'il ne noie pas l'utilisateur sous un flot d'alarmes dont il est difficile de savoir si elles sont vraies ou fausses.

préconditions [194] et en interprétation abstraite [46, 49].

Les impératifs de précision des analyses varient considérablement d'une application à l'autre, mais il en est de même des contraintes de temps. Un optimiseur de compilateur destiné à produire la version finale d'un programme de calcul destiné à un supercalculateur coûteux pourra se permettre des analyses un peu coûteuses : qu'importe si une compilation optimisante dure 20 heures sur une station de travail bon marché ! La situation est bien différente pour un compilateur *just in time* qui doit intervenir en une fraction de seconde au cours de l'exécution d'un programme.

Nous nous sommes principalement intéressés à la preuve automatique de propriétés de programmes destinés à des systèmes critiques, au développement long. On pourrait croire que pour de telles applications, il n'est pas rédhibitoire qu'une analyse prenne une semaine, l'analyse arrivant à la fin du processus de développement. En pratique, lorsqu'une analyse affiche des « fausses alarmes », l'utilisateur va tenter d'éliminer celles-ci (par ajout de directives guidant l'analyseur, etc.) et relancer le processus, dont il appréciera qu'il soit bref. Par ailleurs, les logiciels sont développés selon un processus incrémental : on programme, on teste les parties déjà prêtes, on ajoute des modules, etc. ; il importe donc qu'on puisse en pratique utiliser l'analyse tout au long du processus de développement. Dans ce cadre, on voudra par exemple qu'elle prenne au maximum 16 heures et puisse s'exécuter chaque nuit, afin que les développeurs puissent obtenir lecture des résultats le matin.[23]

S'il existe un assez grand nombre d'outils commerciaux se réclamant de l'analyse statique, les données techniques sur la plus grande partie d'entre eux sont assez floues. Notamment, les présentations qui en sont faites expliquent rarement clairement si l'analyse est correcte dans tous les cas, c'est-à-dire qu'elle ne peut ignorer d'erreurs parmi la classe d'erreurs qu'elle recherche. C'est pourtant une propriété très importante, et on peut conjecturer que si elle n'est pas mentionnée, c'est parce que ces outils ne sont pas sémantiquement corrects. Par exemple, la société COVERITY[24] ne décrit pas le fonctionnement précis de ses outils, mais les publications sur les travaux dont elle s'est inspirée décrivent des analyses sans correction sémantique [67]. À notre connaissance, le seul outil d'analyse industriel généraliste et sémantiquement correct pour la détection des erreurs à l'exécution est l'outil de

[23] Brooks [31, ch. 19] mentionne que Microsoft compile et teste l'entièreté de ses grands logiciels chaque nuit. Des chercheurs de Microsoft m'ont affirmé que des outils d'analyse statique sont appliqués automatiquement lorsque les développeurs veulent modifier la copie centralisée des codes sources, et que si ces techniques produisent des avertissements, les développeurs doivent soit modifier le code pour les faire disparaître, soit fournir des raisons d'accepter de conserver ce code malgré les avertissements.

[24] `http://www.coverity.com/`

POLYSPACE ;[25] mais cet outil a de l'avis de divers utilisateurs tendance à produire un trop grand nombre de fausses alarmes. Zitser [205], Zitser et al. [206] ont comparé différents outils pour la recherche de dépassements de tampons (*buffer overrun*), une des principales causes de vulnérabilité aux virus et aux piratages des ordinateurs connectés à Internet, en l'appliquant à quelques programmes d'utilisation courante sur le réseau ; les deux seuls outils capables de détecter une forte proportion des erreurs à l'exécution, SPLINT et POLYSPACE, avaient un fort taux de fausses alarmes (respectivement 2% et 10%).

1.3 Interprétation abstraite

La sémantique de tout programme informatique séquentiel peut s'exprimer sous la forme d'un système de transition : on se donne un état initial $\sigma_0 \in \Sigma$, décrivant les valeurs initiales des variables du programme, et une relation de transition $\tau \subseteq \Sigma \times \Sigma$. Une exécution du programme est une suite (éventuellement infinie) d'états $\sigma_0 \rightarrow_\tau \sigma_1 \rightarrow_\tau \sigma_2 \rightarrow_\tau \dots$. L'état du programme est, dans un modèle simple, donné par un couple $(p, v_1, \dots, v_n)$ où p est le point de programme atteint et $v_1, \dots, v_n$ sont les valeurs des variables.[26]

Si pour tout état σ il existe au plus un état σ' tel que $\sigma \rightarrow_\tau \sigma'$, on dit que le système est *déterministe* : étant donné un état initial σ_0 il n'y a qu'une exécution possible. Ceci est normalement le cas des programmes informatiques non parallèles, du moins tant qu'ils n'interagissent pas avec le monde extérieur : un calcul donne un résultat donné, et si on le relance sur la même machine on doit obtenir le même résultat. Des programmes avec des entrées extérieures auront des points de non-déterminisme : quand on est dans l'état σ, suivant le choix extérieur, il existera plusieurs états σ' tels que $\sigma \rightarrow_\tau \sigma'$.[27]

[25] POLYSPACE, depuis rachetée par MathWorks, a industrialisé des outils basés sur le travail d'Alain Deutsch à l'INRIA sur l'analyse du code d'Ariane 5. (`http://www.mathworks.com/products/polyspace/index.html`).

[26] On doit bien évidemment utiliser des modèles plus compliqués si on a une pile d'appels, une pile de variables locales, de l'allocation dynamique etc. mais nous ne compliquerons pas inutilement l'exposé. De même, on peut également représenter des programmes parallèles dans ce cadre, en prenant un espace d'état $\Sigma_1 \times \cdots \times \Sigma_m$ s'il y a m programmes.

[27] On m'objectera le cas des programmes utilisant des générateurs aléatoires. En fait, ceux-ci sont le plus souvent pseudo-aléatoires et utilisent quelques paramètres externes, comme l'heure de lancement du programme, pour initialiser une suite de calculs déterministes fournissant les données « aléatoires ». L'algorithme, déterministe, doit utiliser un phénomène extérieur, qu'on modélise soit comme une composante de l'état initial, soit comme une entrée. Comme l'a dit von Neumann [198], « Toute personne qui envisage d'utiliser des méthodes arithmétiques pour produire des chiffres aléatoires est, bien sûr, en état de péché. ». On pourra donc modéliser le comportement de ces programmes par un système déterministe ou non-déterministe. Il existe également, beaucoup plus rarement il est vrai, des générateurs matériels d'aléa.

Il peut être également intéressant de considérer que les générateurs « aléatoires » le sont

Il en est de même d'un ensemble de programmes déterministes lancés en parallèle, si l'on ne connaît pas précisément la politique d'ordonnancement — il s'agit là d'une *abstraction* correcte quelle que soit l'ordonnanceur.

L'ensemble des *états accessibles* du système est l'ensemble des σ tels qu'il existe une suite $\sigma_0 \to_\tau \sigma_1 \to_\tau \cdots \to_\tau \sigma_n = \sigma$, avec éventuellement $n = 0$. On note alors $\sigma_0 \to_\tau^* \sigma$. Une *propriété de sûreté* est un ensemble d'états S qu'on ne doit pas quitter — souvent spécifiée par son complémentaire, un ensemble d'états qu'il ne faut pas atteindre, représentant des situations indésirables (« plantage »...). Autrement dit, on vérifie que $\sigma_0 \to_\tau^* \sigma \implies \sigma \in S$. Un preuve de sûreté par *model checking*, dans sa version la plus simple, consiste à calculer l'ensemble des états accessibles et à vérifier cette inclusion.[28]

Si l'on étudie un programme à p points de programme et n variables de 32 bits, la taille de l'ensemble des états est *a priori* $p.2^{32n}$. On constate que très rapidement, avec un nombre modeste de variables, le nombre d'états dépasse celui (estimé) des atomes dans l'Univers. Clairement, pour la plupart des programmes, certaines simplifications sont nécessaires pour les analyser. On parle alors d'*abstraction* : d'une façon ou d'une autre, certains détails de l'exécution du programme sont ignorés afin de rendre le problème plus simple.

Une méthode d'abstraction très utilisée est la définition de classes d'équivalences. Par exemple, en *model checking*, quand on analyse des programmes contenant des variables entières ou virgule flottante, considérer toutes les combinaisons de valeurs de ces variables serait en général trop coûteux. On commence donc souvent par tout simplement ignorer la valeur de ces variables pour ne conserver que les variables booléennes ou énumérées, ce qui revient à considérer comme équivalents les états qui ne diffèrent que par les valeurs de ces variables. Un état σ est remplacé par sa classe d'équivalence $\alpha(\sigma)$. La relation de transition τ est remplacée par une relation $\tau^\sharp$ telle que :

$$\forall x, y \in \Sigma \; x \to_\tau y \implies \alpha(x) \to_{\tau^\sharp} \alpha(y) \tag{1.1}$$

L'abstraction *optimale* est la plus petite relation $\to_{\tau^\sharp}$ qui vérifie cette relation.

Un exemple d'abstraction par classes d'équivalences est l'*abstraction par prédicats* [87], qui considère un ensemble fini P de prédicats et abstrait chaque état σ par un vecteur $\sigma^\sharp$ de $|P|$ booléens indiquant si cet état vérifie ou non chacun des prédicats.

vraiment, notamment pour des calculs de complexités d'algorithmes ou des preuves de correction d'algorithmes stochastiques. Mon travail de thèse avait consisté à développer des sémantiques et des techniques d'analyses pour ces programmes stochastiques [139, 142, 144, 145, 146, 150, 152, 153].

[28] En pratique, on peut aussi partir des états indésirables S^{C} et on calcule les états *coaccessibles*, c'est-à-dire les états σ tels que $\sigma_\tau^* e$ avec $e \in S^{\mathsf{C}}$, et on montre que σ_0 n'y est pas.

On peut étendre ce cadre d'abstraction état par état à des abstractions d'ensembles d'états : $\alpha : \mathcal{P}(\Sigma) \to \Sigma^\sharp$. Par exemple, on peut abstraire les valeurs possibles d'une variable numérique x par un intervalle $[m_x, M_x]$, ce qui nous donne pour n variables $x, y, z, \ldots$ un produit d'intervalles $[m_x, M_x] \times [m_y, M_y] \times [m_z, M_z] \times \ldots$. Plus précisément, on associe à $\sigma^\sharp = ((m_x, M_x), (m_y, M_y), (m_z, M_z), \ldots)$ l'ensemble d'états

$$\begin{aligned} \gamma(\sigma^\sharp) &= \{x, y, z, \cdots \mid m_x \leq x \leq M_x \wedge m_y \leq y \leq M_y \wedge m_z \leq z \leq M_z \wedge \ldots\} \\ &= [m_x, M_x] \times [m_y, M_y] \times [m_z, M_z] \times \ldots \end{aligned} \quad (1.2)$$

Ainsi, sur les intervalles, l'abstraction optimale de la sémantique de $y := x + y + 1$ (soit $(x, y) \to_\tau (x, x + y + 1)$) par rapport aux intervalles est $((m_x, M_x), (m_y, M_y)) \mapsto ((m_x, M_x), (m_x + m_y + 1, M_x + M_y + 1))$.

Pour un ensemble d'états $E \subseteq \Sigma$, $\alpha(E)$ est abstrait par n couples $[m, M]$ correspondant aux valeurs minimales et maximales des variables $x, y, z, \ldots$ sur E. (Il s'agit donc de l'abstraction optimale.) On surcharge la notation τ : pour $E \subseteq \Sigma$, nous notons $\tau(E) = \{\sigma' \mid \exists \sigma \in E\ \sigma \to_\tau \sigma'\}$. $\tau' : \Sigma^\sharp \to \Sigma^\sharp$ est une abstraction correcte de τ si

$$\forall E \subseteq \Sigma,\ \tau(E) \subseteq \gamma \circ \tau^\sharp \circ \alpha(E) \quad (1.3)$$

On peut facilement munir $\Sigma^\sharp$ d'une relation d'ordre "raisonnable" : $\sigma_1^\sharp \sqsubseteq \sigma_2^\sharp \iff \gamma(\sigma_1^\sharp) \subseteq \gamma(\sigma_2^\sharp)$. On dit que $\Sigma^\sharp$ est un *domaine abstrait.*

Plus généralement, on peut définir des abstractions entre deux ensembles ordonnés par une *correspondance de Galois* (α, γ) :

$$\forall a \forall b\ \alpha(a) \sqsubseteq b \iff a \subseteq \gamma(b) \quad (1.4)$$

La meilleure abstraction de τ est $\alpha \circ \tau \circ \gamma$. Nous ne rentrerons pas plus dans la théorie des correspondances de Galois et renvoyons le lecteur intéressé à Cousot and Cousot [48].

Intuitivement, dans une abstraction par correspondance de Galois, α envoie un ensemble d'états sur sa meilleure abstraction : pour chaque variable, on prend le plus petit intervalle, etc. Imaginons maintenant que nous choisissons d'abstraire des couples (x, y) de grandeurs numériques par des polyèdres convexes. Si E est un ensemble fini de points, alors $\alpha(E)$ est son enveloppe convexe. Cependant, si E est un ensemble infini de points, par exemple un disque, $\alpha(E)$ n'est pas forcément un polyèdre. Sur l'exemple du disque, on comprend bien le problème : pour tout $n \geq 3$ on peut prendre un polyèdre convexe à n côtés tangents au disque, et on a une approximation de plus en plus fine (en aire) quand $n \to \infty$ (voir Fig. 1.2), mais l'approximation optimale (le disque lui-même) n'est pas un polyèdre. Le problème est que l'ensemble des polyèdres convexes n'est pas clos par intersections infinies.

Nous travaillerons souvent dans un cadre plus lâche que les correspondances de Galois : nous nous donnerons une unique fonction $\gamma : \Sigma^\sharp \to \mathcal{P}(\Sigma)$.

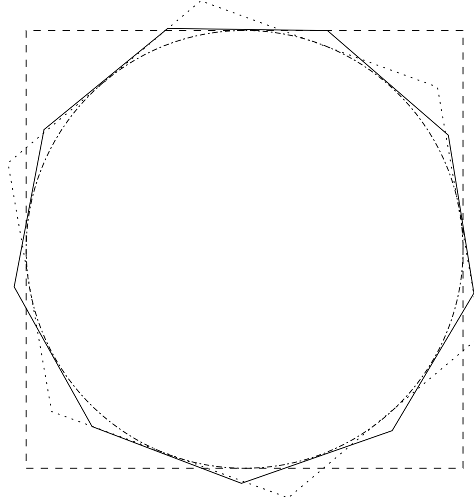


FIG. 1.2 – Un disque peut être approximé avec une précision arbitraire par un polyèdre convexe, mais il n'existe pas de meilleure approximation sous cette forme.

$\tau^\sharp : \Sigma^\sharp \to \Sigma^\sharp$ est une abstraction correcte de τ si :

$$\forall \sigma^\sharp \in \Sigma^\sharp \ \gamma \circ \tau^\sharp(\sigma^\sharp) \subseteq \tau \circ \gamma(\sigma^\sharp) \tag{1.5}$$

Dans le cas des polyèdres convexes, γ envoie une représentation algorithmique du polyèdre vers l'ensemble des points qu'elle représente.

La distinction entre la représentation en machine d'un ensemble d'états et cet ensemble peut paraître académique, mais il est parfois utile de distinguer les deux. Ainsi, si l'on prend les *difference-bound matrices* [133], représentant des contraintes de la forme $v_1 - v_2 \leq C$, les deux ensembles de contraintes suivants représentent le même ensemble d'états :

$$x - y \leq 1 \wedge y - z \leq 2 \wedge x - z \leq 4 \tag{1.6}$$

$$x - y \leq 1 \wedge y - z \leq 2 \wedge x - z \leq 3 \tag{1.7}$$

Remarquons que les contraintes $x - z \leq 4$ et $x - z \leq 3$ sont superflues (on peut les ôter sans changer l'ensemble des états représentés), et que la deuxième est optimale (c'est la meilleure contrainte possible pour $x - z$). Dans le cas des *difference-bound matrices*, il y a une unique *forme normale* donnant toutes les contraintes $v_1 - v_2 \leq C$ avec C optimal. Les problèmes de mise en forme normale, d'éliminations de contraintes superflues interviennent pour de nombreux domaines abstraits.[29]

[29] Idéalement, on aimerait que γ définisse une congruence pour les opérations abstraites, c'est-à-dire que quelle que soit l'opération algorithmique $f^\sharp$ à effectuer on aimerait que

Jusqu'à présent, nous n'avons évoqué que la *correction* de $\tau^\sharp$: cette fonction doit représenter tous les comportements possibles du système sans en omettre aucun. Pour que l'analyse soit effective, elle doit également être calculable, et ce dans des contraintes raisonnables de temps et d'espace mémoire (ces contraintes dépendant elles-mêmes de l'application). Par exemple, les coûts d'opérations sur les *difference-bound matrices* sont dans le pire cas en $\Theta(n^3)$ où n est le nombre de variables, mais on passe en $\Theta(2^n)$ pour les polyèdres convexes.

Pour le moment, nous avons considéré le problème de savoir ce qui advient du programme au bout d'un nombre donné p de pas de calcul (il suffit de considérer $\tau^{\sharp p}$). Dans le cas de programmes avec des boucles, on voudrait plutôt connaître ce qui arrive au bout d'un nombre *quelconque* de pas de calcul. Pour cela, il faut étudier la suite croissante $u_0 = \emptyset$, $u_{k+1} = S \cup \tau(u_k)$ où S est l'ensemble des états initiaux, et notamment sa limite u_∞, qui est exactement l'ensemble des états accessibles du système (et d'ailleurs le plus petit point fixe de $\phi : x \mapsto S \cup \tau(x)$). On remplace cette suite non calculable en général par $u_0^\sharp = \bot$, $u_{k+1}^\sharp = S^\sharp \sqcup \tau^\sharp(u_k^\sharp)$, avec $a \cup b \subseteq \gamma(a \sqcup b)$. Si cette suite est stationnaire, restant sur une valeur $u_\infty^\sharp$, alors $\phi \circ \gamma(u_\infty^\sharp) = S \cup \tau \circ \gamma(u_\infty^\sharp) \subseteq \gamma(u_\infty^\sharp)$. On démontre qu'un tel $u_\infty^\sharp$ vérifie $u_\infty \subseteq \gamma(u_\infty^\sharp)$, c'est-à-dire que c'est une abstraction correcte de l'ensemble des états accessibles.[30]

En général, cette suite $u_k^\sharp$ n'est pas stationnaire, ce qui permettrait d'arrêter le calcul en un nombre fini de pas : par exemple, sur des intervalles, on pourra avoir $u_1 = (1, 1)$, $u_2 = (1, 2)$, $u_n = (1, n)$ etc. Dans ce cas, on introduira un *opérateur d'élargissement* $\triangledown$ (*widening*) qui accélérera la convergence : $u_{k+1}^\sharp = u_k^\sharp \triangledown \tau^\sharp(u_k^\sharp)$ garantie ultimement stationnaire, avec $a \cup b \sqsubseteq \gamma(a \triangledown b)$ [48, p. 517]. Par exemple, on pourra prendre $(1, 2) \triangledown (1, 3) = (1, \infty)$.

La construction des opérateurs d'élargissement dépend beaucoup du domaine abstrait considéré et est plus ou moins malaisée. L'élargissement introduit une surapproximation supplémentaire, qu'on peut tenter de contrôler par différents artifices, tels que les élargissements étagés [20, 93, 94, 130], les déroulements de boucles et délais d'élargissement [21, §7.1]...

En bref, les techniques d'interprétation abstraite souffrent de deux sources d'imprécision :

- Le domaine abstrait choisi peut être inadéquat : soit il ne représente pas les propriétés recherchées — auquel cas l'insuffisance est évidente ; soit il ne représente pas des propriétés indispensables pour établir les propriétés recherchées — auquel cas la recherche de la source de l'im-

$\gamma(x^\sharp) = \gamma(y^\sharp) \implies \gamma \circ f^\sharp(x^\sharp) = \gamma \circ f^\sharp(y^\sharp)$, c'est-à-dire que les ensembles d'états calculés ne dépendent pas des détails des représentations utilisées. Ce n'est pas toujours le cas, notamment en cas d'utilisation d'élargissements [129, 131, 133], et cela peut parfois donner des comportements peu intuitifs.

[30] Par le théorème de Tarski, $u_\infty = \lim_{k\to\infty} u_k = \bigcap\{x \mid \phi(x) \subseteq x\}$ et donc $u_\infty \subseteq x$ pour tout x tel que $\phi(x) \subseteq x$. Comme $\phi \circ \gamma(u_\infty^\sharp) \subseteq \gamma(u_\infty^\sharp)$, le résultat s'ensuit.

précision peut être plus ou moins pénible. On a proposé différentes méthodes à cet effet, dont la plus originale est peut-être celle de Jung et al. [108], à savoir l'usage de filtres statistiques bayésiens censés distinguer les vraies et les fausses alarmes, fonctionnant sur le modèle... de la détection automatisée des courriels publicitaires par .
La recherche en interprétation abstraite va donc proposer des domaines abstraits de précision variable, sachant qu'une précision accrue tend à augmenter les coûts de calcul.[31] Dans le présent mémoire, nous verrons notamment au Ch. 2 comment, dans le logiciel ASTRÉE, notre équipe a pu réconcilier les impératifs de vitesse, de taille mémoire occupée, et de précision.
– Les opérateurs d'élargissement peuvent avoir trop élargi. Ceci motive toute une recherche actuelle visant à se passer d'élargissements, par différentes techniques : par exemple, Costan et al. [45], Gaubert et al. [81] itérations de politiques (technique venant de la théorie des jeux). Au 2.2.6 et 4.2, nous verrons comment des techniques provenant d'autres domaines scientifiques (automatique, logique assistée par ordinateur) permettent, sur certaines classes de programmes, d'obtenir des analyses très précises, voire optimales dans un certain sens, et ce sans utiliser d'élargissements.

1.4 Remarques sur le présent mémoire

Le présent mémoire est un document de synthèse sur mes recherches après ma thèse, condensant en quelques dizaines de page des articles de recherche bien plus longs [20, 21, 50, 51, 52, 136, 137, 138, 141, 143, 147, 151]. Je n'y ai pas repris les articles [150, 152] que j'ai publié après ma thèse sur des travaux déjà présentés dans ma thèse [153] ou réalisés avant ma thèse [140, 149].

[31] Dans certains cas une précision accrue peut simplifier les calculs. L ;imprécision peut notamment rendre abstraitement accessibles des ensembles d'états ou de traces d'exécution qui ne sont pas accessibles dans le système concret, et dont l'analyse peut être coûteuse.

Chapitre 2

Analyse statique appliquée à des problèmes industriels : le projet ASTRÉE

Il semble qu'Astrée, qu'on dit qui est retirée dans le ciel, est encore ici-bas cachée parmi ces hommes.

Fénelon, *Les aventures de Télémaque*, livre VII (1699)

De décembre 2001 à août 2007, j'ai été impliqué dans le développement de l'analyseur statique ASTRÉE, aux côtés de la plupart des membres de l'équipe de Patrick Cousot.

Notre équipe a écrit un certain nombre d'articles sur cet outil [21, 50, 51, 52, 128, 136, 173, ...], aussi je me bornerai à en résumer les principaux traits et les aspects que j'ai plus particulièrement développés personnellement.

Astrée a été réalisé dans le cadre d'une coopération d'équipe entre, outre moi-même et par ordre alphabétique, Bruno Blanchet, Patrick et Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné et Xavier Rival. Dans ce chapitre, j'ai tenté de rendre à chacun sa contribution, et, sinon, j'ai utilisé un « nous » collectif.

Astrée, initialement un développement « preuve de concept » sans ambition d'utilisation à long terme, est maintenant en phase d'intégration dans un des ateliers logiciels de la division « avionique » d'Airbus, après évaluation au sein de l'équipe dirigée par Famantanantsoa Randimbivololona [62, 189].

2.1 L'outil Astrée

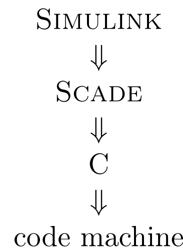
2.1.1 Contexte industriel

Traditionnellement, en automatique, les calculateurs analogiques étaient conçus à l'aide de schémas consistant en blocs, représentant des opérateurs

(intégrateur, limiteur...) correspondant souvent directement à des composants électroniques, et en flèches, représentant les liaisons électriques. Il n'est donc pas surprenant que la même représentation ait été ensuite utilisée par les automaticiens pour spécifier des calculateurs numériques, avec des blocs représentant les opérateurs à temps discret.

Ces diagrammes permettent de *simuler* les systèmes de contrôles : les parties à temps discret sont directement simulables en machine, les parties à temps continu, représentant l'environnement matériel du système (mécanique, électricité...) sont approchées par des méthodes de résolution d'équations différentielles ordinaires. Les automaticiens peuvent donc très efficacement utiliser ces *modèles* pour prototyper les systèmes, notamment à l'aide d'outils tels que SIMULINK.[1] La phase suivante, dans un processus de production « en cascade » du logiciel, est le codage sous la forme exécutable par le système embarqué. La méthode traditionnelle était de fournir les schémas, constituant une spécification détaillée, à un équipementier, un fournisseur chargé de produire le matériel et le logiciel implémentant cette spécification. Bien entendu, celui-ci peut employer des programmeurs qui implémentent la spécification à la main (en langage machine, en langages C, Ada, Fortran...). On peut cependant faire mieux : puisqu'on a des modèles simulables, on peut aussi bien les compiler en code exécutable. C'est ainsi qu'avec les méthodes d'*ingéniérie basée sur les modèles*, les programmes d'automatiques, spécifiés en détail dans des langages comme SIMULINK+STATEFLOW, SAO[2], ESTEREL,[3], LUSTRE[4] ou SCADE,[5] sont automatiquement compilés vers des langages de plus bas niveau. On supprime ainsi une étape de programmation manuelle, coûteuse et éventuellement source de *bugs*.

On utilise ainsi, typiquement, une chaîne de compilation de la forme :



On peut penser que les résultats de preuves, d'analyses faites à haut niveau, sur les modèles, seront également valables sur le code généré et,

[1] SIMULINK est un logiciel de la société *MathWorks*, créateurs du célèbre logiciel de calcul numérique *Matlab*.

[2] *Spécification assistée par ordinateur*, un langage de programmation graphique développé en interne par Airbus [176].

[3] ESTEREL est un langage synchrone développé par la société Esterel Technologies.

[4] LUSTRE est un langage de flot de données synchrone développé au laboratoire VERIMAG [34].

[5] Un langage graphique industriel basé sur LUSTRE.

partant, sur le code effectivement embarqué. Divers problèmes doivent cependant tempérer cet espoir. Tout d'abord, la sémantique des langages de spécification peut être peu claire, ou recéler des particularités malvenues. Ainsi, Scaife et al. [185] ont identifié différents problèmes sémantiques dans STATEFLOW, et proposé un sous-ensemble « sûr » de ce langage. Les problèmes que nous avons relevés dans la sémantique des flottants dans des langages comme C ou Java (voir chapitre 3) se retrouvent certainement dans les langages de spécification. Ensuite, il n'est pas exclu que les compilateurs utilisés contiennent des erreurs. Malgré divers efforts pour réaliser des compilateurs certifiés corrects à la construction [117], ou pour vérifier *a posteriori* que la compilation a bien préservé la sémantique [172], force est de constater que les compilateurs effectivement utilisés dans l'industrie n'ont pas été formellement prouvés corrects.

Il est ainsi doublement intéressant de valider certaines propriétés sur le code machine, ou du moins sur le langage bas niveau à l'avant-dernière position de la chaîne de compilation :

- Une telle vérification peut être effectuée chez l'équipementier, lequel n'a par contre pas forcément autorité pour vérifier ou discuter les schémas de spécifications.
- Cette analyse n'est pas affectée par les éventuels dysfonctionnements de la chaîne de compilation, et peut prendre en compte les caractéristiques qui ne sont pas visibles sur la spécification en langage de haut niveau.

Bien sûr, ce positionnement a également des inconvénients :

- Une analyse directement sur le code machine permettrait de prendre en compte la résolution des ambiguïtés sémantiques du langage de bas niveau, ainsi que les éventuels bugs du compilateur.
- En revanche, il est probablement plus simple d'analyser la spécification en langage de haut niveau.

Airbus, un des deux grands fabricants mondiaux d'avions de ligne, a une division « avionique et produits de simulation », qui joue le rôle d'équipementier interne (en sus de commandes à des fournisseurs comme Thales, Honeywell, Rockwell-Collins...). Cette division réalise notamment les commandes de vol électriques des A330, A340, A380 et suivants, tant sur le plan matériel que logiciel. Désireuse d'utiliser des méthodes formelles, elle s'est d'abord dirigée vers la preuve assistée en logique de Floyd-Hoare [98, 202] avec l'outil CAVEAT,[6] puis a voulu utiliser des outils purement automatiques d'analyse statique, notamment l'outil de POLYSPACE.

Airbus a demandé à notre équipe au Laboratoire d'informatique de l'École normale supérieure d'étudier la possibilité de prouver automatiquement l'absence d'erreur à l'exécution dans les codes C des commandes de vol électriques. Parmi les contraintes :

[6] CAVEAT est un outil développé par le Commissariat à l'énergie atomique. `http://www-list.cea.fr/labos/fr/LSL/caveat/index.html`

- Les programmes à étudier ne doivent pas être modifiés, ou du moins minimalement. Il n'est notamment pas question de construire à la main un « modèle » ou une maquette du programme.
- Ils sont écrits dans un langage et un style de relativement bas niveau.
- Ils sont générés partiellement automatiquement, mélangeant des portions écrites à la main avec du code généré [28, §12.6.3.2], donc deux styles différents.
- Ils contiennent un très grand nombre de variables flottantes, y compris « `static` » (variables locales dont la valeur est conservée d'une itération à l'autre).
- Ils sont de grande taille (en centaines de milliers de lignes de code).

L'analyse devait détecter les erreurs à l'exécution, ainsi que les violations éventuelles des règles du langage C (comportements non spécifiés). Ces notions ont d'ailleurs dû être précisées au cours de l'étude, mais, en résumé, il s'agit de :

- Débordements de tableaux.
- Déréférencements de pointeurs nuls ou incorrects.
- Dépassements arithmétiques, notamment lors des conversions flottants vers entiers (on retrouve ici la préoccupation de ne pas avoir un accident semblable à celui d'Ariane 5).

La plupart des logiciels d'analyse statique existants souffrent de différents défauts pour l'application citée :

- Les outils universitaires prennent souvent en entrée des langages jouets, ou des sous-ensembles restreints de langages « sémantiquement commodes ».
- Le plus souvent, les outils ne passent pas à l'échelle sur de gros programmes.
- Quand ils passent à l'échelle, il sont souvent sémantiquement incorrects — il n'y a aucune garantie que l'outil considère l'ensemble des comportements possibles du programme, comme par exemple dans les outils d'Engler and Musuvathi [67].
- Parfois, ils changent la sémantique des programmes afin de la rendre plus commode : entiers machine (sur un nombre fini de bits) assimilés à des entiers mathématiques ($\mathbb{Z}$), flottants assimilés à des réels ($\mathbb{R}$).

2.1.2 Fonctionnement

ASTRÉE est constitué de trois grandes parties :

- Un module d'entrée, très semblable à celui d'un compilateur : analyse lexicale et syntaxique, typage, fusion de codes sources (analyse multifichiers). Ce module, réalisé spécialement pour ASTRÉE, est antérieur aux outils CIL[7] [156] et FRAMA-C.[8]

[7] `http://manju.cs.berkeley.edu/cil/`
[8] `http://frama-c.cea.fr/`

– Une pré-analyse visant à simplifier le code à analyser (propagation de constantes, suppression de variables inutiles).
– L'analyseur proprement dit.

L'outil prend en entrée, outre le code source du logiciel à analyser, différents paramètres l'adaptant au système cible analysé, notamment des contraintes sur les valeurs se trouvant dans les registres d'entrée-sortie.

L'analyseur est constitué de cinq couches :
– L'itérateur, qui parcourt l'arbre syntaxique.[9]
– Les domaines de partitionnement — traces [128, 173], disjonction sur valeur de variable.
– Le domaine représentant la mémoire du programme analysé, les relations « peut pointer vers » pour les pointeurs, et les relations avec les cases mémoires abstraites représentant les quantités numériques. Ce domaine a fait l'objet de multiples modifications destinées à augmenter sa capacité à gérer des programmes « sales » (arithmétique de pointeur, types union,[10] transtypages sauvages,[11] conversions entre entiers et pointeurs...).
– La couche de communication entre domaines numériques [52].
– Les domaines de contraintes numériques.

Parmi les domaines de contraintes arithmétiques, citons notamment :
– Les intervalles entiers.
– Les intervalles flottants.
– La propagation symbolique [134].
– Différents domaines de filtrage [70].

[9]Cet itérateur a été modifié par Xavier Rival pour permettre l'analyse en arrière, par parcours d'une structure de graphe.

[10]En langage C, `union` permet d'introduire des types de données faisant résider au même endroit en mémoire alternativement différents types de données $T_1, \dots, T_n$. L'usage conforme à la norme est de s'en servir comme un type disjonctif, c'est-à-dire que si l'on a rangé dans une « union » une donnée de type T_i, on ne la relit que selon le type T_i. La possibilité de relire une donnée écrite avec un type à l'aide d'un autre type (*type-punning*) permet toutes sortes d'usages « créatifs » et au comportement indéfini selon la norme C.

[11]Exemple : pour recopier un flottant de 32 bits dans un autre, recopier sa représentation bit-à-bit sous forme d'entiers de 32 bits à l'aide de conversions de types de pointeurs.

```
float a, b ;
*((int*) &a) = *((int*) &b) ;
```

Le comportement de ce programme est indéfini selon la norme du langage C, pour deux raisons : non seulement, comme dans le cas du *type-punning* via les `union`s, il n'y a pas en général de garantie sur ce qui arrive quand on lit une donnée d'un type à l'aide d'un autre type, mais en plus la norme du langage C permet maintenant au compilateur de supposer, à des fins d'optimisation, que deux données de type différent ne peuvent être au même endroit en mémoire (« aliasées »), de sorte que l'optimiseur pourra, par exemple, supposer que la valeur du flottant `a` n'a pas été changée par l'opération ci-dessus. Voir notamment l'option `-fstrict-aliasing` de `gcc`.

Ce genre d'« astuces » est cependant assez courant. Lors du portage d'applications de plate-formes 32 bits vers des plate-formes 64 bits, on se rend souvent compte que le programmeur a supposé que les pointeurs et les `int` ont la même taille...

2.2 Études et développements annexes

Le développement d'ASTRÉE a suggéré plusieurs études et développements annexes.

2.2.1 Efficacité

Les publications sur l'interprétation abstraite mesurent souvent la complexité des opérations (abstraction d'une opération $x := e$, union abstraite $\sqcup$...) en fonction du nombre v de variables. Cette mesure est toutefois quelque peu trompeuse. Par exemple, une union abstraite en complexité linéaire $O(v)$ peut être *a priori* considérée comme efficace, mais en fait, ce n'est pas le cas pour toutes les applications, comme nous allons le voir.

À la sortie de chaque test, l'analyseur doit calculer une surapproximation de l'union : $a^\sharp \sqcup b^\sharp$ telle que $\gamma(a^\sharp \sqcup b^\sharp) \supseteq \gamma(a^\sharp) \cup \gamma(b^\sharp)$. On peut raisonnablement supposer que pour le type de programmes réactifs considérés, la proportion p_t de tests parmi lignes de code est grossièrement constante : pour un programme de n lignes on aura environ $p_t n$ tests, donc leur analyse nécessitera $p_t n$ opérations $\sqcup$. De même, la proportion de déclarations de variables rémanentes (correspondant à la classe `static` du langage C) est grossièrement constante, disons que pour n lignes de code on a $p_d n$ déclarations de rémanents. Supposons que la complexité de $\sqcup$ est proportionnelle au nombre total de variables actives au point de programme, alors la complexité globale de l'analyse des tests du programme est proportionnelle à $p_n p_d n^2$. La complexité de l'analyse est donc *quadratique* en la taille du code, ce qui est prohibitif.

Nous nous sommes très vite rendu compte de cette inefficacité, sur des programmes de taille modérée (environ 10 000 lignes). Notre équipe a donc introduit des domaines et des implémentations astucieuses, pour lesquels la complexité de $f^\sharp(x^\sharp) \sqcup g^\sharp(x^\sharp)$ dépend principalement du nombre de variables qui ont été altérées par f et g ; autrement dit, pour le $\sqcup$ en fin de test, par le nombre de variables dont la valeur est affectée par les deux branches du test.

Plus précisément :

- Bruno Blanchet a suggéré la représentation de fonctions $V \to X^\sharp$, où V est l'ensemble des variables et $X^\sharp$ peut par exemple être un domaine d'intervalles, à l'aide d'arbres binaires équilibrés. Pour les opérations idempotentes ($f(a, a) = a$), comme $\sqcup$, l'implémentation court-circuite le cas où les deux opérandes sont physiquement égaux en mémoire.
- J'ai également expérimenté le remplacement des arbres binaires équilibrés par des arbres de Patricia, c'est-à-dire de représentations de fonctions partielles $\{0, 1\}^k \to X^\sharp$ par des diagrammes de décision binaire. Les performances étaient légèrement inférieures et la solution n'a donc pas été retenue.

– Antoine Miné a introduit la notion de « pack » de variables pour les domaines relationnels : au lieu de prendre les relations sur toutes les variables, on se donne une famille de petits ensembles de variables, et on ne calcule relationnellement que sur chacun de ces petits ensembles.

Sur le strict plan de l'ingéniérie logicielle, les grands volumes de données manipulées par ASTREE (plusieurs gigaoctets) et les temps de calcul importants (10, 20 heures...) ont posé des problèmes spécifiques. Il a fallu passer sur une architecture 64-bits, d'où des difficultés de portabilité de certaines bibliothèques. J'ai *profilé* plusieurs fois l'analyseur, et ai notamment dû trouver des paramètres appropriés pour le *garbage collector* de CAML (les réglages par défaut sont adaptés à de moindres volumes de données), et ai même dû examiner le code assembleur généré pour les calculs arithmétiques d'une bibliothèque anormalement lente sur certains modèles de processeurs. Nous pouvons donc confirmer ce que disait Weise [199] : en matière d'analyse de programme de grande taille, les inefficacités d'implémentation, d'algorithme, de structure de données coûtent très vite cher !

2.2.2 Preuve formelle de structures de données

Comme expliqué ci-dessus, ASTRÉE fait appel à des arbres binaires équilibrés pour représenter des fonctions de la forme $V \to A$, où A est un domaine abstrait et V l'ensemble des variables (ou, plus généralement, un ensemble d'ensembles de variables liées entre elles). Par exemple, le domaine des intervalles associe à chaque variable numérique un intervalle. V est assimilé à un intervalle d'entiers $[0, n-1[$.

La plupart des opérations $f^\sharp \sqcup g^\sharp$ sont liées à des tests entre deux branches de programme relativement courtes, qui chacune n'altèrent qu'une petite portion de l'état abstrait. Il semble donc exagérément coûteux de passer en revue chaque variable v et calculer $f^\sharp(v) \sqcup g^\sharp(v)$, alors que seuls les cas où $f^\sharp(v) \neq g^\sharp(v)$ sont intéressants, puisque si $f^\sharp(v) = g^\sharp(v)$ alors $f^\sharp(v) \sqcup g^\sharp(v) = f^\sharp(v)$. Nous préfèrerions un système qui détecterait directement que des portions entières de $f^\sharp$ et $g^\sharp$ sont identiques.

Nous avons choisi de représenter les fonctions $V \to A$ par des arbres binaires « presque équilibrés » — quand un rééquilibrage local est possible, nous le faisons, mais nous ne cherchons pas à rééquilibrer à grande échelle après certaines opérations. Les opérations unaires ou binaires sur ces fonctions procèdent par récurrence sur la structure des arbres. Pour une opération ϕ telle que $\phi(x, x) = x$, quand on détecte deux sous arbres physiquement identiques (à la même adresse en mémoire), on rend directement le sous-arbre en question.

À l'idéal, on aimerait qu'il y ait équivalence entre « les sous-arbres a et b sont identiques » et « a et b désignent la même adresse en mémoire ». Ceci est possible en utilisant des techniques de *hash consing* : une grande table de hachage contient des références sur tous les sous-arbres déjà créés et non

encore détruits,[12] et avant de créer un nœud nouveau on vérifie s'il n'est pas déjà présent dans la table, auquel cas on réutilise l'occurence déjà présente. Bien sûr, cette technique ne fonctionne que pour des structures de données purement fonctionnelles (non modifiables après création).

Nous n'avons cependant pas recouru au *hash consing* et nous sommes restreints à un système plus faible où les fonctions de manipulation des arbres essayent, dans la mesure du possible, de repérer quand le résultat d'une opération est isomorphe à un des arguments (ou à un sous-arbre d'un des arguments).

Nous avons ainsi défini diverses fonctions opérant sur les arbres binaires et dont la spécification est donnée par rapport aux fonctions associées aux arbres. Ainsi, on a une fonction `map2iz` qui prend en argument deux arbres a et b représentant des fonctions partielles $\tilde{a}$ et $\tilde{b}$ de support S identique et une fonction f telle que $f(x, x) = x$ pour tout x, et qui rend un arbre c représentant une fonction $\tilde{c}$ tel que

$$\forall x \in S \; \tilde{c}(x) = f(\tilde{a}(x), \tilde{b}(x)) \tag{2.1}$$

Par la suite, j'ai réimplementé une partie de ces fonctions dans l'assistant de preuve Coq et les ai prouvées correctes par rapport à leur spécification, à savoir non seulement des relations semblables à 2.1, mais aussi la conservation de l'invariant de bon ordonnancement des arbres. Le code Caml extrait automatiquement de cette implémentation a remplacé le code écrit manuellement pour les fonctions concernées.

2.2.3 Implémentation parallèle

Le lecteur se reportera à [136] pour plus de détails.

Les codes de contrôle-commande obtenus par compilation de langages synchrones ont, dans leur forme la plus simple, la forme suivante :

```
while (true) {
  attente_horloge();
  actions_a_realiser();
}
```

Dans le cas de programmes multi-horloges, de périodes $\tau_1, \ldots, \tau_k$ multiples d'une période principale τ, la solution la plus simple consiste à faire un ordonnancement statique et à ranger les tâches dans des séquenceurs numérotés de 0 à $n-1$, où $n = \text{ppcm}\left(\frac{\tau_1}{\tau}, \ldots, \frac{\tau_k}{\tau}\right)$. Comme le temps d'exécution

[12] Dans un système où la mémoire est gérée par un *garbage collector*, on doit donc utiliser des tables de hachage à références faibles (*weak hash tables*), c'est-à-dire que le fait que la table de hachage pointe sur l'objet n'est pas un obstacle à sa destruction, et qu'en cas de destruction de l'objet l'entrée correspondante de la table est invalidée.

du corps de la boucle, dans le pire cas, est le maximum des temps d'exécution des séquenceurs, on aura intérêt à répartir les tâches dans les séquenceurs de façon à ce qu'ils aient tous environ le même temps d'exécution. On obtient alors des programmes de la forme :

```
int sequence = 0;
while (true) {
  attente_horloge();
  executer_sequenceurs(sequence);
  sequence = sequence + 1;
  if (sequence >= n) sequence = 0;
}
```

Souvent, la correction de tels programmes par rapport aux erreurs à l'exécution ne dépend pas de l'ordonnancement précis des différents séquenceurs, si l'on excepte le fait qu'il faut que les procédures d'initialisation de toutes les tâches appelées par les séquenceurs aient été appelées préalablement à l'utilisation des fonctionnalités de la tâche. On peut donc espérer que le programme suivant est également correct (il est évident que s'il l'est, *a fortiori* le programme précédent l'est) :

```
int sequence;
for(sequence = 0; sequence < n; sequence++) {
  executer_sequenceurs(sequence);
}
while (true) {
  sequence = choix_arbitraire(0, n-1);
  executer_sequenceurs(sequence);
}
```

En général, bien sûr, cette transformation peut être une abstraction trop brutale, qui empêche de démontrer les propriétés désirées. Intuitivement, sur les programmes que nous considérons, la correction vis-à-vis des erreurs à l'exécution repose sur la stabilité de boucles de rétroaction, qui ne dépend pas de leur fréquence de cadencement.

La première boucle peut être déroulée, le problème principal est l'obtention d'un invariant pour la deuxième boucle qui soit suffisamment petit pour permettre de prouver les propriétés que l'on souhaite. L'analyse statique en avant de cette boucle revient à trouver un post-point fixe de $X \mapsto f_0(X) \cup \cdots \cup f_{n-1}(X)$; on itère donc $X^\sharp \mapsto f_0^\sharp(X^\sharp) \sqcup^\sharp \cdots \sqcup^\sharp f_{n-1}^\sharp(X^\sharp)$.

Remarquons que les calculs $f_i^\sharp(X^\sharp)$ peuvent être réalisés indépendamment les uns des autres. On peut même espérer que si les séquenceurs sont suffisamment bien équilibrés, leur complexité d'analyse est du même ordre. Une implémentation parallèle divisera donc les $f_0^\sharp(X^\sharp)$ entre plusieurs

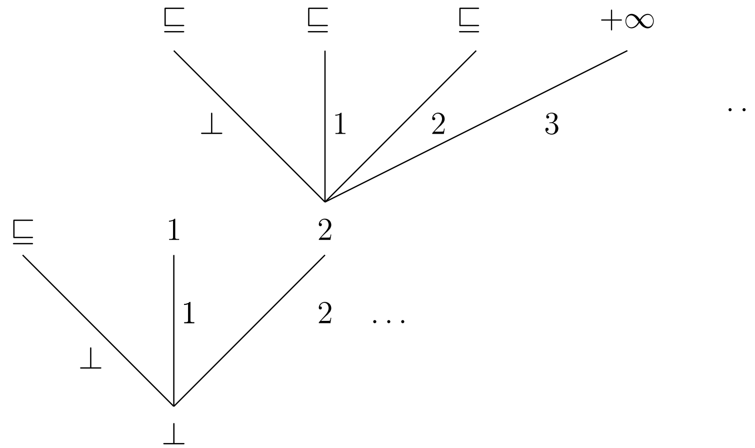


Fig. 2.1 – Arbre d'élargissement : la racine représente le point de départ des itérations, en général $\bot$, une feuille $\sqsubseteq$ représente la terminaison de l'élargissement, une arête $x \xrightarrow{n} y$ indique que l'élargissement de x par n donne y.

k fils d'exécution.[13] On peut également utiliser une implémentation distribuée ; si l'on désire minimiser les transferts de données, on pourra calculer $X^\sharp \mapsto \bigsqcup_{j=1}^{k} \left(\bigsqcup_{i \in F_j} f_i^\sharp(X^\sharp) \right)$ avec une partition F_j de $\{0, \dots, n-1\}$.Le résultat de ce calcul sera identique à celui d'un calcul sériel si $\sqcup^\sharp$ est associative. Mieux encore, au lieu de renvoyer $Y_j^\sharp = \bigsqcup_{i \in F_j} f_i^\sharp(X^\sharp)$, on pourra se contenter de renvoyer une description de la différence entre $X^\sharp$ et $Y_j^\sharp$.

J'ai implémenté cette stratégie dans Astrée et ai constaté que le temps d'exécution évoluait, pour de gros programmes, comme $T_1 \times (0.25 + 0.75/n)$ avec n le nombre de processeurs, ce qui donne des performances intéressantes pour $n \leq 4$.

2.2.4 Preuve formelle et définition de l'élargissement

Comme nous l'avons dit au §1.3, de nombreuses présentations de l'interprétation abstraite mettent des hypothèses fortes, par exemple l'existence d'une correspondance de Galois entre le concret et l'abstrait, ou encore la monotonie des transformateurs abstraits. Même si ces hypothèses peuvent aider à l'intuition et à la compréhension des méthodes utilisées, elles ne sont cependant pas respectées par Astrée.

La correction de l'analyse ne repose pas sur la monotonie des fonctions de transfert abstraites, mais seulement sur celle de la sémantique concrète

[13] Cette division pourra par exemple se faire avec des outils de parallélisation avec répartition dynamique, comme OpenMP [162].

et sur la relation d'abstraction :

$$\forall x, y\ x \subseteq y \Rightarrow f(x) \subseteq f(y) \tag{2.2}$$

$$\forall x^\sharp\ f \circ \gamma(x^\sharp) \subseteq \gamma \circ f^\sharp(x^\sharp) \tag{2.3}$$

Ces deux conditions passent à la composition : si $f^\sharp$ est une abstraction de f monotone et $g^\sharp$ est une abstraction de g monotone, alors $f^\sharp \circ g^\sharp$ est une abstraction de $f \circ g$ monotone.

La présentation usuelle de l'élargissement est la suivante : c'est une sur-approximation de l'union ($\gamma(a^\sharp) \cup \gamma(b^\sharp) \subseteq \gamma(a^\sharp \nabla b^\sharp)$), telle que pour tout $u_0^\sharp$ et toute suite $v_n^\sharp$, la suite définie par $u_{n+1}^\sharp = u_n^\sharp \nabla v_n^\sharp$ est ultimement stationnaire. Ceci impose notamment que $\gamma(u_n^\sharp)$ soit croissante. Pour sur-approximer le plus petit point fixe de f, on itère jusqu'à trouver $u^\sharp$ tel que $u^\sharp = u^\sharp \nabla f^\sharp(u^\sharp)$. On a alors, par hypothèse sur ∇, $\gamma \circ f^\sharp(u^\sharp) \subseteq \gamma(u^\sharp)$, et par correction de $f^\sharp$, $f \circ \gamma(u^\sharp) \subseteq \gamma(u^\sharp)$.

J'ai étudié dans l'assistant de preuve Coq la formalisation de l'interprétation abstraite et notamment de l'élargissement. Coq n'accepte de fonctions récursives que définies par induction syntaxique, ce qui a suggéré de définir l'élargissement comme un arbre bien fondé. On se donne une relation $\sqsubseteq$ telle que $u^\sharp \sqsubseteq v^\sharp \implies \gamma(u^\sharp) \subseteq \gamma(v^\sharp)$. Chaque nœud de l'arbre porte une valeur abstraite $u^\sharp$ et une fonction de « pas suivant », qui étant donné un argument $v^\sharp$ soit répond une preuve que $v^\sharp \sqsubseteq u^\sharp$, soit propose une sous-branche (structure récursive). Les suites $u_0^\sharp, \ldots$ s'obtiennent par descente dans l'arbre depuis la racine, et la terminaison est assurée par le fait que l'arbre est bien fondé. En pratique, cela veut dire que l'élargissement propose une valeur $u_n^\sharp$, l'itérateur lui renvoie $f^\sharp(u_n^\sharp)$, et l'élargissement soit fournit l'assurance à l'itérateur que $f^\sharp(u_n^\sharp) \sqsubseteq u_n^\sharp$, impliquant $f \circ \gamma(u_n^\sharp) \subseteq \gamma(u_n^\sharp)$, soit une valeur $u_{n+1}^\sharp$.

Il est intéressant de constater que la formalisation dans un assistant de preuve incite à supprimer des hypothèses inutiles.

2.2.5 SAT-solving

Plusieurs auteurs ont indiqué avoir obtenu des résultats intéressants à l'aide de techniques de SAT-solving appliquées sur une sémantique précise au niveau du bit — c'est-à-dire en codant explicitement l'arithmétique sur les entiers machines, ce qui revient grosso modo à décrire des portes logiques exécutant les calculs. Ces résultats m'intriguaient, car raisonner sur des propriétés arithmétiques au niveau du bit ignore toutes les structures algébriques et géométriques (anneau ordonné, etc.) qui permettent de construire des algorithmes efficaces de calcul.

Afin de pouvoir tester l'efficacité de ces techniques, j'ai implémenté une transformation de programme d'un sous-ensemble du langage C accepté par Astrée, ainsi que des préconditions et assertions, vers le format SAT, et

j'ai testé la résolution des problèmes générés à l'aide des outils zChaff[14] et Minisat[15] [66].

Mon scepticisme initial a été conforté par les essais auxquels j'ai procédé : malgré des algorithmes relativement astucieux de génération des clauses logiques, les temps de résolution deviennent vite insupportables dès que les calculs arithmétiques du programme analysé deviennent un tant soit peu complexes. Il est ainsi totalement impossible de prouver que la multiplication de deux mots de 32 bits est commutative, ce qui n'est d'ailleurs pas surprenant.

Ma conclusion est que ce type d'approche par codage binaire systématique est voué à l'échec, sauf à s'intéresser à des propriétés de programmes avec très peu d'arithmétique, et surtout pas d'arithmétique non linéaire. Même pour l'arithmétique linéaire, il existe des solutions plus efficaces. Les outils de SAT modulo théorie comme Yices[16] [64, 65] et Z3[17] [59] supportent maintenant des types de données « vecteurs de bits », implémentant l'arithmétique modulo 2^n. Ces procédures utilisent des techniques algébriques efficaces pour résoudre les problèmes au maximum sans avoir recours au *bit-blasting*, c'est-à-dire à l'écriture explicite sous forme propositionnelle, au niveau du bit.

2.2.6 Filtres récurrents linéaires

Le lecteur se reportera à [137] et à sa version étendue, disponible sur HAL, pour plus de détails.

Un des blocs élémentaires courants dans les logiciels de contrôle-commande est le filtre récurrent linéaire d'ordre n : étant donné un flux d'entrée $(e_k)_{k\in\mathbb{N}}$, et des valeurs initiales imposées $(s_k)_{0\leq k<n}$, le flux de sortie vérifie

$$\forall k\ s_k = \sum_{i=0}^{n_e} a_i e_{k-i} + \sum_{i=1}^{n} b_i s_{k-i} \tag{2.4}$$

Il existe une riche théorie et une riche littérature autour de ces filtres, qu'il serait difficile de résumer ici ; nous renvoyons le lecteur à, par exemple, l'ouvrage très complet d'Åström and Wittenmark [4]. Nous n'avons pas trouvé, par contre, d'étude visant à obtenir des bornes rigoureuses sur la sortie d'un tel filtre en fonction de bornes sur ses entrées, dans le cas d'une implémentation en virgule flottante.

Dans le cas d'une analyse sur les réels, le filtre est exactement défini par sa réponse impulsionnelle, à savoir la sortie R du filtre si on lui place en

[14] `http://www.princeton.edu/~chaff/zchaff.html`
[15] `http://minisat.se/`
[16] `http://yices.csl.sri.com/`
[17] `http://research.microsoft.com/en-us/um/redmond/projects/z3/`

entrée un pic de Dirac : $e_0 = 1$ et $e_i = 0$ pour $i > 0$. En effet, le flux de sortie est alors le produit de convolution $s = R \star e$, c'est-à-dire

$$s_k = \sum_{i=0}^{k} e_{k-i} R_i \tag{2.5}$$

Borner l'intervalle de variation des s_k en fonction de celui des e_k revient, dans le cas d'intervalles symétriques autour de zéro, à borner la norme subordonnée de $\|\cdot\|_\infty$ de l'application $e \mapsto R \star e$, c'est-à-dire le plus petit α tel que $\|R \star e\|_\infty \leq \alpha \|e\|_\infty$. On montre que cette norme est $\|R\|_1 = \sum_{k=0}^{\infty} |R_k|$. On montre également que les R_k sont les coefficients du développement en série entière au voisinage de zéro de la fraction rationnelle dite « transformée en z » :[18]

$$\frac{\sum_{i=0}^{n_e} a_i z^i}{1 - \sum_{i=1}^{n} b_i z^i} \tag{2.6}$$

Feret [70] a proposé (dans une formulation assez différente, il est vrai) de décomposer $\|R\|_1$ en $\sum_{k=0}^{N-1} |R_k|$ et $\sum_{k=N}^{\infty} |R_k|$, avec un N arbitraire. L'idée est que pour un filtre stable, l'essentiel de la somme tient dans les premiers termes et la « queue » décroît exponentiellement. La première somme peut être bornée explicitement par arithmétique d'intervalle sur la relation de récurrence 2.4 ; en pratique on s'arrête (et on fixe donc n) quand le signe de R_k devient inconnu, car ensuite les intervalles obtenus deviennent très imprécis. Regardons maintenant les termes R_k pour $k \geq N$. Feret donne une méthode pour borner ces termes dans le cas où $n = 2$, mais cette méthode peut être généralisée à tout ordre (au prix d'une petite complication du calcul numérique).

Les termes R_k vérifient la relation de récurrence $R_k = \sum_{i=1}^{n} b_i r_{k-i}$, donc on les obtient comme $R_{N+k} = \sum_{i=0}^{n-1} R_{N+i} H_{k-i}$ où H est la suite des coefficients de la série entière de

$$\frac{1}{1 - \sum_{i=1}^{n} b_i z^i} \tag{2.7}$$

On montre que

$$\|H\|_1 \leq \frac{1}{(|\xi_1| - 1) \ldots (|\xi_n| - 1)} \tag{2.8}$$

où les ξ_i sont les racines du polynôme $1 - \sum_{i=1}^{n} b_i z^i$, prises avec leur multiplicité (il faut que leur valeur absolue soit supérieure à 1). Il suffit donc d'obtenir une borne supérieure pour $\|H\|_1$; ensuite, on appliquera $\sum_{k=N}^{\infty} |R_k| \leq \sum_{i=0}^{n-1} |R_{N+i}| . \|H\|_1$ en utilisant les intervalles calculés pour $(R_k)_{N \leq k < N+n}$. Pour un filtre stable, la valeur réelle de ces R_k est très proche de zéro, les intervalles calculés le sont aussi, et la borne obtenue pour $\sum_{k=N}^{\infty} |R_k|$ est très faible.

[18] Suivant les auteurs, on écrira cette fraction en z ou en z^{-1}.

Nous nous sommes donc ramenés au problème de calculer une borne inférieure garantie au module des racines ξ_i d'un polynôme P de degré n et de coefficient dominant p_n. Diverses méthodes numériques permettent de calculer des valeurs approchées x_i des ξ_i, et sont d'ailleurs disponibles dans des logiciels de calcul numériques comme GSL [75], mais ces méthodes fournissent rarement de garantie sur la précision de leur résultat. Rump [179] propose des méthodes permettant, étant donné ces approximations, d'obtenir des rayons de disques r_i tels que $|\xi_i - x_i| \leq r_i$. La formule suivante convient, par exemple :

$$r_i = \left| \frac{nP(x_i)}{p_n \prod_{j \neq i} x_j - x_i} \right| \tag{2.9}$$

Feret a proposé de décomposer la sortie du filtre en la somme d'une partie exacte, comme calculée en réels, et une partie découlant du « bruit » du calcul flottant. Je reformule ici son idée. Le flux de sortie est alors $R \star e + R_1 \star \varepsilon_1 + R_2 \star \varepsilon_2$ où e est le flux d'entrée, R la réponse impulsionnelle du filtre, les ε_k représente les erreurs d'arrondi (répartis par source), et les R_i représentent les réponses impulsionnelles par rapport aux erreurs d'arrondis. En utilisant l'inégalité 3.1, on obtient que $\|\varepsilon_i\|_\infty \varepsilon_{\text{rel}} \leq \|v_i\|_\infty + \varepsilon_{\text{abs}}$ où v_i est la suite des valeurs *réelles* qui donnent lieu à l'arrondi ε_i. Mais v_i s'obtient elle-même comme résultat d'un calcul de la même forme.

Finalement, on peut obtenir que le flux de sortie s'exprime sous la forme $s = R \star e + \varepsilon$ avec $\|\varepsilon\|_\infty \leq K.\varepsilon$. R est, comme précédemment, représenté par une fraction rationnelle dont le développement en série entière au voisinage de zéro donne les coefficients de R.

Plus généralement, pour un filtre ayant des flux d'entrées $E_1, \dots, E_{n_e}$ et des flux de sorties $S_1, \dots, S_{n_s}$, on aura une matrice $R_{i,j}$ de fractions rationnelles, $R_{i,j}$ représentant la réponse de la sortie s_i par rapport à l'entrée e_j, et une matrice K, telles que

$$\begin{bmatrix} S_1 \\ \vdots \\ S_{n_s} \end{bmatrix} = R. \begin{bmatrix} E_1 \\ \vdots \\ E_{n_e} \end{bmatrix} + \varepsilon \tag{2.10}$$

Ici le produit matrice-vecteur $R.v$ où R est une matrice de fractions rationnelles et v est un vecteur de flux de réels est défini par rapport au produit de convolution entre le développement en série entière autour de zéro des fractions rationnelles et les flux. Le vecteur de flux ε vérifie alors : pour tout i, $\|\varepsilon_i\|_1 \leq M_i$ où $M = K.N$, et N est le vecteur formé des normes des flux d'entrée ($N_i = \|E_i\|_\infty$).

Ceci nous permet donc de définir une véritable sémantique compositionnelle approchée pour les filtres récurrents linéaires. Chaque filtre est défini par deux matrices exprimant l'une les relations entrées-sorties sur les réelles, via leurs transformées en Z, l'autre une borne linéaire sur les erreurs de calculs : si $S = R_1.A + \varepsilon_1$ avec $\|\varepsilon_1\|_\infty \leq K_1.\|A\|_\infty$ coordonnée par coordonnée

et $A = R_2.E + \varepsilon_2$ avec $\|\varepsilon_2\|_\infty \leq K_2.\|E\|_\infty$ $S = (R_1R_2).E + R_1.\varepsilon_2 + \varepsilon_1$. En bornant les normes-1 des développements en séries entières des fractions contenues dans A, on obtient $S = (R_1R_2).E + \varepsilon$ où $\|\varepsilon\|_\infty \leq K.\|E\|_\infty$ coordonnée par coordonnée.

2.2.7 Analyse de programmes asynchrones, cas d'un pilote de périphérique

Le lecteur se reportera à [138] pour plus de détails.

Dans tous les cas évoqués ci-dessus, le programme analysé consistait en un seul fil d'exécution. C'est le cas d'un grand nombre d'applications embarquées, notamment dans les domaines critiques. En effet, il est bien connu que la programmation parallèle, notamment par utilisation de mémoire distribuée, est délicate. Il est très facile de se tromper dans la conception du système, et le déboguage est difficile en raison du caractère non-déterministe des pannes — une panne peut se produire au bout de plusieurs mois d'exécution. Nous avons déjà cité le cas de la machine de radiothérapie Therac-25 [119], où des conditions de compétition[19] dans le logiciel pouvait produire un surdosage mortel pour le patient, et celui des pannes d'engins spatiaux *Deep Space One* et *Mars Pathfinder* [166, 169], dues à des problèmes de synchronisations et de priorités entre processus.

Il s'agit là de problèmes difficiles à modéliser et à analyser. Il est déjà difficile de formaliser exactement la sémantique des accès mémoire concurrents sur les architectures matérielles courantes comme x86, laquelle n'est pas une simple sémantique d'entrelacements comme on le pense souvent ; les documentations disponibles sont le plus souvent assez imprécises, au point qu'il est nécessaire dans certains cas d'essayer les différentes possibilités sur différents matériels [183]. Même en supposant ces problèmes de documentation et de formalisation maîtrisés, l'analyse de programmes concurrents pose des problèmes d'explosion de l'espace d'états : si les programmes sont des automates finis, leur composition asynchrone est un automate fini dont le nombre d'états est le produit des nombres d'états (bien sûr, l'utilisation de mécanismes de synchronisation fait diminuer ce nombre d'états).

Si l'objectif de modifier Astrée afin de permettre l'analyse de programmes concurrents était séduisant, il paraissait toutefois trop ambitieux de tenter l'atteindre en une étape. Nous avons donc posé comme objectif intermédiaire d'être capable d'analyser un système concurrent formé d'une

[19] Traduction peut-être peu heureuse de *race condition*, c'est-à-dire de la possibilité pour le système de parvenir dans un état indésirable dans certains séquencements particuliers des processus — par exemple, si un évènement se produit précisément dans une certaine période ou si deux fonctions activées infréquemment sont actives au même moment. Ce genre de problèmes est typiquement très difficile voir impossible à reproduire, car dépendant de conditions très précises au niveau des temps d'exécution.

part d'un pilote de périphérique, d'autre part d'un contrôleur de bus USB. Ces deux systèmes communiquent par mise à jour de structures de données dynamiques (listes, arbres) dans des zones de mémoire partagée, avec l'originalité qu'une bonne partie de ces communications se font sans usages de mécanismes d'exclusion mutuelle, simplement en tirant parti de l'atomicité de certaines opérations mémoire.

J'ai donc modélisé l'interaction d'un contrôleur hôte USB avec un pilote pour ce contrôleur destiné à un système embarqué. Le contrôleur hôte USB (*host controller*) réalise l'interface entre le bus interne (PCI, par exemple) d'un système informatique maître (un micro-ordinateur, par exemple) et le bus USB. La transmission des données sur le bus USB fait l'objet d'une norme [42], et pour chaque type de transmission standardisée (mémoire de masse, imprimante...) il existe une norme spécialisée. Quant à l'interface entre le bus PCI et le bus USB 1, il existe deux normes concurrentes : UHCI (Intel) et OHCI (consortium entre Compaq, Microsoft...) ; le contrôleur étudié était à la norme OHCI [138]. Cette norme spécifie en langue naturelle le comportement du contrôleur, et donne des exemples d'utilisation en pseudocode.

Malheureusement, comme beaucoup de documentations industrielles, la spécification OHCI est assez imprécise et permet de larges divergences d'interprétation et d'implémentation. Par exemple, il n'est pas évident de savoir quand certains évènements ont lieu, quelles actions sont atomiques, etc. On nous a ainsi mentionné le cas d'un bit de contrôle dont l'action était immédiate sur un contrôleur, et différée à la prochaine trame sur un autre. En pratique, il faut parfois procéder par essai et erreur.

Cette analyse différait de celle qu'on aurait pu faire sur un pilote destiné à un système d'exploitation généraliste, à la fois en positif et en négatif :

- Comme ce pilote est destiné à des systèmes embarqués fermés, où la liste des périphériques connectables est connue d'avance et limitative, les structures de données sont allouées dans des tableaux de taille fixe et non dans des zones d'allocation dynamique. L'énumération et l'initialisation des périphériques sont faites une fois pour toute lors de l'initialisation du pilote, et non dynamiquement lors du fonctionnement, comme c'est le cas pour les pilotes des systèmes d'exploitation d'ordinateurs personnels, qui doivent prévoir la possibilité d'ajouter ou de retrancher à tout instant une souris, un *stick* de mémoire, une imprimante...
- Il serait sans doute difficile de vendre un composant USB généraliste (par exemple, un bloc IP implémentant un contrôleur USB à la norme OHCI) qui ne serait pas compatible avec les pilotes des principaux systèmes d'exploitation (Microsoft Windows, Linux). Autrement dit, même si la norme est imprécise, les concepteurs de périphériques les testeront au moins par rapport à ces pilotes, qui auront donc une bonne chance d'être compatible avec eux. En revanche, un pilote conçu de

façon totalement indépendante courra le risque d'être compatible avec un contrôleur mais pas avec un autre.

J'ai opté pour une modélisation simplifiée du contrôleur — une modélisation exacte aurait demandé en pratique d'examiner les traces PCI, qui n'étaient pas à notre disposition au début de cette étude. À partir de la spécification OHCI, j'ai écrit un modèle C en langage C des actions possibles du contrôleur. Il s'agit d'un modèle non exécutable, car très non déterministe, qui décrit les actions atomiques possibles du contrôleur. Ce modèle choisit une des actions possibles du contrôleur (par exemple, lire ou écrire un mot en mémoire partagée, terminer avec ou sans erreur la transmission d'un bloc de données) et l'exécute (ou ne fait rien si aucune action n'est possible). Une trace d'exécution du système est un entrelacement d'actions du pilote P et de déclenchements de C : $p_1 c_1 p_2 c_2 p_3 c_3 \ldots$. Remarquons que le flot de contrôle du pilote impose le séquencement de $p_1 p_2 p_3 \ldots$, tandis que seuls les données présentes dans la zone de mémoire partagée et les registres d'état du contrôleur simulé contraignent $c_1 c_2 c_3 \ldots$.

La modélisation du contrôleur peut être limitée à ses actions observables par le processeur, c'est-à-dire à ses actions en écriture sur la mémoire partagée ; mais comme ces actions dépendent elles-mêmes de la programmation du contrôleur qui se fait par des données présentes en mémoire partagée, il faut également prendre en compte la lecture de ces données dans le modèle. En revanche, il est inutile de faire une modélisation fine du protocole USB, par exemple en ce qui concerne les délais et latences exacts de transmission, et ce d'autant plus que :

- nous devons faire une modélisation dans le pire cas (c'est-à-dire avec la possibilité d'une erreur de transmission ou de protocole à chaque instant) ;
- le modèle d'exécution abstraite du pilote sur le processeur ne tient absolument pas le compte du temps d'exécution des instructions, une information plus fine sur les délais de transmission serait donc assez inutile.

A priori, il serait possible d'analyser le couple pilote-contrôleur en simulant l'exécution d'une instruction atomique du pilote, suivie d'un nombre quelconque d'actions du contrôleur, puis l'instruction atomique suivante du pilote, etc. Ce procédé serait toutefois trop coûteux, alors qu'on peut appliquer une *réduction d'ordre partiel* tirant parti de la structure des interactions entre le pilote et le contrôleur.

L'ensemble des interactions entre processeur et contrôleur se passe via des zones de mémoire partagée, situées dans certaines structures de données (principalement des tableaux) définies dans le pilote. Le nom de ces structures est connu, et il est possible au cours de l'exécution abstraite d'obtenir, par une analyse de pointeur, l'assurance que certaines instructions ne lisent ni n'écrivent ces structures (le cas le plus simple étant celui des instructions n'opérant que sur des variables locales en pile). Les instructions

du processeur qui n'interagissent pas avec ces structures ne sont pas visibles du contrôleur, et « commutent » donc avec les actions du contrôleur. Il est donc possible de considérer comme atomiques vis-à-vis de l'analyse les suites d'instructions $i_1 \dots i_n$ telles que l'analyse de $i_1 \dots i_{n-1}$ montre qu'elles ne font pas référence à la mémoire partagée.

L'analyse implémentée dans Astrée, pour chaque instruction successive du pilote :

- Teste abstraitement si cette instruction se réfère à la mémoire partagée ; en cas de réponse positive (« l'instruction affecte peut-être la mémoire partagée »), exécuter abstraitement un nombre quelconque de fois le simulateur de contrôleur (ceci implique un calcul d'invariant par post-point fixe).
- Exécute l'instruction du pilote.

L'analyse prend en entrée, en plus du code source du pilote et du code source du simulateur de contrôleur, un fichier de configuration définissant les zones de mémoire partagée et le nom de la fonction simulant les actions du contrôleur.

Cette modélisation est toutefois imparfaite :

- L'analyse ne vérifie pas que le contrôleur limite ses accès mémoire aux zones de mémoire partagée, ce qui est une condition nécessaire pour la sûreté de la réduction d'ordre partiel expliquée plus haut. Cette limitation serait cependant très facile à lever, puisqu'il s'agirait simplement d'exploiter l'information de pointeurs.
- La spécification OHCI est assez vague sur ce qui est une action atomique du contrôleur, ou dans quel ordre certaines actions sont réalisées, par exemple les mises à jour de pointeurs d'avancement. Il est probable que le modèle utilisé est à trop gros grain et considère comme atomiques des actions qui ne le sont pas. Seul un examen des traces PCI permettrait de trancher (et on pourrait alors s'interroger sur la validité de l'analyse en cas de substitution du contrôleur par un autre également censé suivre la norme OHCI).

Il a par ailleurs fallu procéder à certaines simplifications. Par exemple, la norme OHCI utilise les bits de poids faible de certains pointeurs forcément multiples de 4 pour stocker certaines informations ; notre analyse de pointeurs ne sait pas gérer ce type d'arithmétique et il a donc fallu supprimer les actions sur ces bits.

L'analyse a permis de montrer qu'un pilote simple ne pouvait pas générer d'erreur à l'exécution, même combiné avec le contrôleur. En revanche, le passage à un pilote plus complexe, utilisant plus de fonctionnalités du contrôleur, a mis en évidence certaines limitations de l'analyse de pointeurs. En résumé, l'analyse de pointeurs d'Astrée obtient, pour chaque pointeur, un sur-ensemble des adresses sur lesquelles il peut pointer. Cette analyse est en grande partie non relationnelle. Or, pour analyser certains problèmes dans des pilotes plus complexes, il faudrait avoir une véritable analyse de

séparation entre listes chaînées, cette séparation étant une propriété dynamique (une même case mémoire appartient à au plus une seule liste parmi un ensemble, mais sa liste d'appartenance peut varier au cours du temps). Une telle analyse pourrait être basée sur la *logique de séparation* [170] ; des analyseurs basés sur celle-ci existent déjà, citons par exemple SpaceInvader.[20]

J'ai par ailleurs constaté que l'on obtient des résultats absurdes si l'on ne modélise pas les actions du contrôleur. En effet, le pilote étudié, à différentes étapes, vérifie que le contrôleur a bien réagi comme il l'attendait, et produit sur une condition d'erreur sinon (sur les systèmes embarqués étudiés, cela provoque sans doute une réinitialisation du boîtier, voire sa désactivation). Si le contrôleur n'est pas modélisé, la phase d'initialisation du pilote détecte un dysfonctionnement et produit immédiatement une condition d'erreur. L'analyse termine alors très rapidement, et pour cause : tout le code du pilote au delà de la phase d'initialisation est considéré comme inaccessible ! Ceci devrait inciter à un certain esprit critique envers les résultats donnés dans certaines publications, où l'on analyse des pilotes de périphériques complexes situés dans des systèmes d'exploitation multi-tâches en ignorant les actions des périphériques et les autres fils d'exécution...

[20] `http://www.eastlondonmassive.org/`

Chapitre 3

Analyse de programmes et calculs en virgule flottante

> *Trop nombreux sont les programmeurs enclins à tester les expressions en nombres complexes de la même façon qu'ils testent les expressions en nombres réels, en les évaluant sur une poignée d'arguments de test pour voir si les résultats coincident avec ce qui est attendu. Comme cette stratégie de test fonctionne la plupart du temps avec les expressions analytiques réelles, les programmeurs ne prêtent pas attention lorsqu'on leur dit qu'elle n'est pas fiable. Que peut-on espérer d'autre, dans une société où la conduite en état d'ivresse est encore si souvent considérée comme une simple peccadille ?*
>
> Kahan [111], traduction personnelle

> *La plupart des calculs numériques n'ont aucune importance, et donc une grande partie d'entre eux peuvent être faux sans que cela n'affecte l'Histoire.*
> *Certains calculs numériques ont une grande importance. Nous ne le savons habituellement pas avant la fin. Parfois, nous ne le savons pas avant qu'il ne soit trop tard.*
>
> Woehr [203], traduction personnelle

Les raisonnements mathématiques opèrent le plus souvent sur des structures algébriques telles que $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$. Or, en informatique, on calcule souvent sur des entiers rentrant dans un petit nombre de mots machine, fixé à l'avance (disons 32 ou 64 bits), et on approche les réels par des flottants. La facilité d'emploi de ces derniers est trompeuse. Nous pouvons distinguer deux sortes de mauvais usages des nombres en virgule flottante :

1. Les réels forment un corps ordonné, or l'addition sur les flottants n'est même pas associative. On ne peut donc pas transposer directement un raisonnement sur les réels en un raisonnement sur les flottants. Suivant le type d'algorithme considéré, les erreurs d'arrondis peuvent avoir tendance à se compenser et à rester modérées, ou au contraire s'accumuler et introduire des erreurs significatives dans le résultat du calcul. Ce phénomène est bien connu et motive la recherche d'algorithmes numériquement stables, soit une bonne partie de la recherche en mathématiques appliquées.

   On pourrait croire que tout cela est bien connu. Or, on constate encore que certains compilateurs font des optimisations « sauvages », notamment par application de règles d'associativité.

2. Les programmeurs conscients des différences entre réels et flottants peuvent avoir de fausses croyances concernant les implémentations de ces derniers. Notamment, l'existence de normes telles que IEEE-754 [101, 102] peut laisser croire que le comportement des flottants est complètement prévisible et normalisé. Or, en raison du flou ou de la permissivité des normes, notamment dans la définition de la sémantique des langages de programmation, des différences se font jour.

   Le problème est d'autant plus délicat que les différences constatées sont faibles, et parfois ne se produisent que pour certains choix bien précis de valeurs. Elles peuvent donc passer inaperçues.

Au vu de ces difficultés, certains informaticiens adoptent une attitude fataliste : pour eux, la virgule flottante est trop compliquée, trop mal définie pour que l'on puisse prouver quoi que ce soit. Cette attitude ignore les nombreux travaux d'arithmétique des ordinateurs expliquant, par exemple, comment implémenter telle ou telle opération en arrondi exact à l'aide d'opérations élémentaires IEEE-754 — citons notamment en France les travaux de Jean-Michel Müller et de son équipe — ou encore comment utiliser adroitement les opérations afin de minimiser l'erreur d'arrondi globale — citons par exemple l'algorithme de sommation de Kahan [109].

Même si l'on admet que des spécialistes puissent écrire et prouver corrects des algorithmes utilisant finement les possibilités de calcul en virgule flottante, voire les prouver à l'aide d'assistants de preuve, on peut douter de la possibilité d'analyser *automatiquement* des programmes, surtout de grande taille, pour en tirer des informations fiables. Il n'est donc pas surprenant que les prétentions du projet ASTRÉE aient été accueillies avec un certain scepticisme initial. Nous verrons que cela est pourtant possible, à condition de se contenter d'une certaine abstraction du calcul flottant.

3.1 Sémantique des opérations flottantes

3.1.1 Généralités

Les représentations en virgule flottante, qu'il s'agisse de la norme IEEE-754 [101, 102], des représentations « quasi standard » mises en œuvre dans la plupart microcontrôleurs et processeurs de traitement digital (DSP), ou de représentations à précision arbitraire comme MPFR [74, 77], sont toutes basées sur les même principes. Un ensemble de réels F, dits *réels représentables*, est distingué ; pour les types de données à précision fixée, cet ensemble est fini. La norme IEEE-754 spécifie deux formats, dits *simple précision*, de 32 bits, et *double précision*, de 64 bits, mais permet également l'existence de formats de précision étendue, par exemple de 80 ou 128 bits.

F peut également contenir les infinités $\{-\infty, +\infty\}$;[1] ceci est imposé par la norme IEEE-754 mais n'est pas supporté sur de nombreux DSP. L'expression $\infty - \infty$ n'ayant pas de sens, son résultat est la valeur spéciale NaN, ou *not a number* ; il en est de même pour le résultat d'autres *opérations invalides* comme $\sqrt{x}$ pour $x < 0$. Toute opération mettant en jeu un NaN donne un NaN, ceux-ci se propagent jusqu'à la sortie de l'algorithme. Pour certaines applications, les infinités, et parfois même les NaN, sont des valeurs attendues pour certaines utilisations. Dans d'autres, notamment dans la plupart des systèmes de contrôle-commande, infinités et NaN sont proscrits. La norme IEEE-754 prévoit heureusement que la génération de $\pm\infty$ soit remplacée par l'exception *overflow*, celle NaN par l'exception *invalid operation*.

Sur les systèmes qui nous ont été donnés à étudier, les exceptions *overflow* et *invalid operation* étaient activées et, signalant une situation imprévue et indésirable, provoquaient un passage du système dans un mode dégradé. En conséquence, Astrée devait signaler tout les points de programme pouvant provoquer ces exceptions. En revanche, les programmes analysés ne pouvant manier les valeurs NaN et $\pm\infty$, puisque leur génération aurait provoqué une exception, nous n'avons pas cherché à les gérer. Par souci de simplicité, nous ignorerons par la suite ces valeurs $\pm\infty$, et nous limiterons à signaler qu'on peut les gérer dans une analyse non relationnelle en associant à chaque variable un ensemble fini de drapeaux « peut être $+\infty$, peut être $-\infty$, peut être NaN ».

La norme IEEE-754 impose également de distinguer deux valeurs $+0$ et -0 correspondant toutes deux au réel zéro. On les obtient notamment en prenant l'inverse de $+\infty$ et $-\infty$ respectivement, ou par arrondi de quantités proches de zéro, respectivement positives et négatives. Cette distinction permet de simplifier certains algorithmes numériques, notamment en arithmétique complexe [111]. Prenons la fonction $\Im \log(z)$ (partie imaginaire

[1] On parle d'infinité *affine* ou *dirigée*. Certains microprocesseurs anciens, comme l'Intel 80287, pouvait également opérer en infinité *projective* avec un unique point à l'infini ∞.

principale du logarithme complexe) au voisinage de la demi-droite ouverte $z = x+iy$, $x < 0$: $\lim_{y \to 0^+} \Im \log(x+iy) = +\pi$ et $\lim_{y \to 0^-} \Im \log(x+iy) = -\pi$: il est raisonnable de définir par continuité de chaque côté $\log(x+0i) = +\pi$ et $\log(x-0i) = -\pi$. Ce comportement des fonctions complexes le long des discontuinités est d'ailleurs imposé en ce qui concerne l'arithmétique complexe normalisée en langage C [103, §7.3.3].

Il est cependant plus rare que des programmes mettant en œuvre de l'arithmétique réelle se reposent sur cette distinction, et cela n'était le cas d'aucun des programmes qui nous ont été soumis pour analyse. En effet, on ne peut pas distinguer $+0$ de -0 à l'aide des opérateurs de comparaison, mais seulement par des opérations mettant en jeu des infinités ($1/+0 = +\infty$, $1/-0 = -\infty$), alors que celles-ci sont prohibées comme nous l'avons dit plus haut, ou par une opération d'extraction directe du bit de signe, absente de ces programmes. Nous ignorerons donc par la suite les différences entre ± 0. Là encore, on peut les gérer dans une analyse non relationnelle à l'aide de drapeaux.

On se donne une fonction d'arrondi $r : \mathbb{R} \mapsto F$; la norme IEEE-754 impose de supporter l'arrondi au plus proche (par défaut), l'arrondi vers le haut, l'arrondi vers le bas, et l'arrondi vers zéro, mais là encore les DSP n'offrent le plus souvent que l'arrondi au plus proche. Chaque opération flottante élémentaire est définie comme la composée de l'opération exacte sur les réels et de la fonction d'arrondi. Ainsi, l'addition flottante $x \oplus y$ est définie comme $r(x + y)$.[2]

3.1.2 Sémantique des expressions

A priori, on pourrait croire que l'expression syntaxique `x*y+3+z` du langage C désigne $((x \otimes y) \oplus 3) \oplus z$, qui aurait donc comme sémantique $r(r(r(x.y) + 3) + z)$. Nous le verrons, la réalité est plus compliquée. Ainsi, certains microprocesseurs (PowerPC, en particulier) ont une instruction *fused multiply-add*, c'est-à-dire une combinaison d'une multiplication et d'une addition, souvent définie comme $\mathsf{fma}(a, b, c) = r(a.b + c)$. On voit clairement l'avantage d'une telle instruction à la fois en temps et en précision, notamment pour les calculs d'algèbre linéaire ($\sum_i a_i.b_i$ etc.). Mais que faire, par exemple, de l'expression $(a \otimes b) \oplus (c \otimes d)$? On peut l'optimiser sous la forme $\mathsf{fma}(a, b, c \otimes d)$ aussi bien que $\mathsf{fma}(c, d, a \otimes b)$, et ces deux expressions ne donnent en général pas le même résultat. Une expression $f(x).g(x) + h(x)$ où f, g et h sont expansées en ligne pourra éventuellement être compilée sous la forme $(f(x) \otimes g(x)) \oplus h(x)$ plutôt que $\mathsf{fma}(f(x), g(x), h(x))$ suivant la « pression » sur les registres (1 donnée à garder en registre au lieu de 2). Une transformation de programme aussi « innocente » que `x=a*b+c ;` remplacé par :

[2]Nous reprendrons les notations de Miné [130], distinguant les expressions flottantes $x \oplus y$, $x \ominus y$... des expressions réelles $x + y$, $x - y$...

```
y=a*b;
afficher(y);
x=c+y;
```

peut donc avoir des conséquences inattendues. En d'autres termes, suivant la façon de compiler, on peut avoir ou non des arrondis à certains points d'évaluation. Nous reviendrons d'ailleurs au §3.1.4 sur un problème lié, celui du *double arrondi*.

Notons que même si l'on sait exactement où le compilateur applique un fma, par exemple si l'on considère directement le code objet, il n'est pas non plus évident de faire certaines analyses ou tests si la machine d'analyse n'est pas la machine cible et ne dispose pas de l'instruction fma. Boldo and Melquiond [22] ont présenté une méthode astucieuse pour émuler à coût raisonnable un fma sur une machine n'en disposant pas.

3.1.3 Réécritures optimisantes

Le lecteur se reportera à [151, §4.3.2] pour plus de détails.

Il est bien connu que l'opération $\oplus$ n'est pas associative : en double précision IEEE-754, $(10^{20} \oplus 1) \oplus (-10^{20}) = 0$. C'est un problème classique, familier de tous ceux qui conçoivent des méthodes numériques en connaissance de cause : quand on fait $x \oplus y$ avec $|y|$ trop petit devant $|x|$, le résultat est x. Cela peut paraître évident, mais ceci va à l'encontre des réflexes de nombreux programmeurs. Ainsi, la façon la plus évidente de programmer $\sum_{i=0}^{\infty} u_i$ est de se limiter à une somme finie $\sum_{i=0}^{n} u_i$ (avec un critère plus ou moins rigoureux pour le choix de n), et d'écrire quelque chose comme :

```
s = 0.0;
for(int i=0; i<=n; i++) {
  s = s + u(i);
}
```

Or, sachant que $\lim_{i\to\infty} u_i = 0$, u_i sera probablement plus petit pour i grand, donc on a des chances, lorsqu'on va ajouter les derniers termes, de faire justement une addition entre des quantités de valeurs absolues très différentes. Il est donc plus judicieux de faire :

```
s = 0.0;
for(int i=n; i>=0; i--) {
  s = s + u(i);
}
```

Il existe d'ailleurs des formules de sommation plus raffinées, telles que celle de Kahan [109]. L'intérêt d'utiliser des formules de sommation astucieuses

disparaît ou est toutefois amoindri si le compilateur prend la liberté de réorganiser les calculs par application de l'associativité. La norme du langage C interdit cette transformation aux compilateurs [103, §5.1.2.3].

Certaines optimisations ne sont possibles que via de telles réorganisations. Ainsi, pour vectoriser $\sum_{i=0}^{2n-1} u_i$ sur un processeur capable de deux opérations flottantes simultanées sur des quantités connexes en mémoire, on séparera par associativité la somme en $\left(\sum_{i=0}^{n-1} u_{2i}\right) + \left(\sum_{i=0}^{n-1} u_{2i+1}\right)$.

D'autres optimisations transforment une expression en une autre, presque identique, mais qui peut se calculer par une séquence d'opérations plus simple. Ainsi, certains processeurs ont des opérations élémentaires « minimum » et « maximum », qui permettent d'éviter une suite d'opérations avec test explicite, voir saut conditionnel. Or, il y a plusieurs façons d'écrire, par exemple, le maximum de `x` et `y` :

```
if (x>y) {      if (x>=y) {      if (!(x<=y)) {      if (!(x<y)) {
  z = x;          z = x;           z = x;              z = x;
} else {        } else {         } else {            } else {
  z = y;          z = y;           z = y;              z = y;
}               }                }                   }
```

Ces quatre calculs ne se distinguent que par leur comportement sur ± 0 et NaN. L'instruction élémentaire « maximum » du processeur reproduit un seul de ces comportements ; un compilateur optimisant pourra considérer que les quatre comportements sont équivalents afin de pouvoir l'utiliser.

Certains compilateurs optimisants, comme `gcc` [76] avec l'option `-ffast-math`, réécrivent les calculs arithmétiques, notamment par associativité, afin de les optimiser. Il n'y a rien de mal à cela tant qu'il s'agit d'une option inactive par défaut ; en revanche, d'autres compilateurs appliquent ce type d'optimisations sans demander l'avis du programmeur, en violation des normes. Kahan [110] déplore vigoureusement ce choix et l'attribue au désir de certains éditeurs de compilateurs commerciaux d'afficher de bonnes performances sur les *benchmarks*.

3.1.4 Double arrondi

Le lecteur se reportera à [151, §3.1.2] pour plus de détails.

L'architecture Intel « x86[3] » a dès le départ été prévue pour offrir un support matériel des calculs en virgule flottante, initialement avec des co-processeurs arithmétiques (8087, 80287, 80387, collectivement surnommés « x87 »), puis

[3]Terme collectif pour les processeurs Intel 8088/8086, 80386, Pentium, les processeurs AMD, Cyrix etc. compatibles, utilisés notamment dans les ordinateurs personnels « compatibles PC ».

directement au sein des microprocesseurs 80486DX et des nombreuses variantes des « Pentium ». Le jeu d'instructions et de registres « x87 » supporte les formats IEEE-754 simple précision, double précision, et un format de précision étendue de 80-bits. Les registres du processeur et les calculs opèrent par défaut dans le format de précision 80-bits, bien qu'il soit possible de réduire la précision des calculs par positionnement de certains drapeaux de contrôle du processeur.

Ces particularités architecturales sont la source de nombreux problèmes pour ceux qui recherchent la reproductibilité exacte des résultats de calculs au niveau des codes sources des logiciels : par exemple, comme nous allons le voir, des instructions d'impression ou de déboguage, qui n'ont *a priori* pas d'effets de bord sur l'état du programme, peuvent avoir un effet de bord caché.

Notons r_d l'arrondi au plus proche vers le format IEEE-754 double précision, et r_e l'arrondi au plus proche vers le format précision étendue. Lorsqu'un programme demande à calculer la somme de deux variables double précision x et y et à ranger le résultat dans z, le processeur calcule d'abord la somme en précision étendue $r_e(x+y)$ puis stocke le résultat, soit $z := r_d(r_e(x+y))$ (nous faisons pour le moment l'hypothèse que le compilateur n'optimise pas les placements de variables et que chaque écriture dans le code source correspond à une écriture en machine). Un calcul direct en flottants double précision donnerait $z := r_d(x+y)$. Or, si $r_d \circ r_e$ est égale à r_d presque partout, elles diffèrent en certains points en raison de la nécessité de départager l'arrondi au plus proche entre deux flottants équidistants. C'est le problème du *double arrondi*, bien connu des spécialistes de l'arithmétique flottante [72, chapter 6][82, 4.2] mais ignoré de la quasi totalité des programmeurs,[4] qui produit différents effets surprenants.

On pourrait croire qu'il suffit de définir $x \oplus y = r_d \circ r_e(x+y)$, et ainsi de suite, pour évacuer le problème. Cet expédient n'est correct que dans le cas d'un compilateur vraiment non optimisant qui rangerait toutes les quantités flottantes en mémoire, y compris les valeurs temporaires générées lors de l'évaluation des expressions non élémentaires ; nous ne connaissons d'ailleurs aucun compilateur dans ce cas. En général, dans la mesure du possible, le compilateur va laisser dans les registres du processeur les variables temporaires, intermédiaires d'évaluation des expressions non atomiques ; mais s'il manque de registres, ou s'il doit sauver les registres par exemple en raison de l'appel d'une fonction, il les rangera temporairement en mémoire. Tous les compilateurs que nous connaissons rangent les variables temporaires dans des tampons mémoires de précision correspondant au type de la variable, et non pas de précision étendue comme les registres du processeur. La sé-

[4] Nous avons ainsi relevé des rapports pour de soi-disant « bugs » dans des compilateurs, en fait dus à des problèmes de double arrondi n'apparaissant que dans certaines configurations.

mantique d'une expression flottante dépend donc des hasards de l'allocation de registres faite par le compilateur, lesquels dépendent du compilateur, de sa version, des options d'optimisation et d'une foule de facteurs. Ainsi, la nécessité de sauvegarder les registres lors de l'appel d'une fonction fait que la sémantique du programme suivant change suivant que l'on supprime ou non la ligne d'affichage, ou suivant les optimisations :

```
double x = calcul qui donne une valeur
           très faible mais non nulle;
if (x != 0) {
  affichage();
  if (x == 0) {
    explosion();
  }
}
```

Contrairement à ce que l'on pourrait attendre, le programme ci-dessus, compilé sur x86/x87, pourra exécuter `explosion()`. Ceci s'explique ainsi : `x`, stocké dans un registre, est très faible mais non nul, le premier test est donc pris. L'appel à `affichage()` force le compilateur à sauver temporairement les registres. À notre connaissance, aucun compilateur n'utilise pour cela de variable tampon de précision étendue, une situation là encore dénoncée par Kahan [110]. Cela donne une sémantique surprenante, où les variables changent de valeur sans action du programme, et où une condition vraie à un point cesse d'être vraie trois lignes plus tard, là sans que le programme ne touche aux variables concernées. Cette sémantique rend notamment incorrectes les méthodes de preuve de Floyd-Hoare [98, 202], car celles-ci supposent qu'une variable a une valeur bien définie et constante entre deux instants où le programme agit sur elle [151, §7.4].

Tout ceci est étonnant, alors que la norme du langage C rappelle, dans un exemple non normatif, que les implémentations utilisant des registres plus larges que les variables doivent appliquer une sémantique indépendante des hasards de l'allocation des variables en registre ou en mémoire [103, 5.1.2.3, *program execution*, §12, ex. 4]. Tous les compilateurs (ou, du moins, tous les compilateurs courants) semblent donc hors norme, ce qui est peu rassurant.

Certains compilateurs[5] permettent de forcer l'écriture en mémoire des variables, mais pas des variables temporaires. Il s'agit d'un pis-aller : en nommant systématiquement les variables intermédiaires, on peut forcer le compilateur à passer systématiquement par la mémoire, et alors chaque opération encourt un double arrondi ($x \oplus y = r_d(r_e(x + y))$). Il s'agit cependant là d'une discipline d'écriture de programme assez peu naturelle.

Notons, enfin, que l'architecture x87 permet de sélectionner des modes de précision inférieure à la précision étendue : mantisses de taille correspondant

[5]Notamment `gcc` avec l'option `-ffloat-store`.

aux flottants IEEE-754 simple et double précision. Ces modes, cependant, ne contraignent pas les exposants, de sorte que le double arrondi est toujours possible pour certaines valeurs proches de zéro.

Au vu de ce qui précède, il semble que la solution est que toutes les variables et tous les registres aient un type unique. Bien sûr, cela est possible sur le x87 en utilisant systématiquement le type de précision étendue 80 bits, mais les bibliothèques de calcul ne sont souvent pas prévues pour cela, la consommation mémoire est supérieure (96 bits, voire 128 bits occupés pour chaque donnée) et il peut y avoir des pénalités d'alignement pour l'accès à la mémoire.[6]

Les processeurs récents de l'architecture x86 contiennent une seconde unité flottante, dite SSE. SSE était à l'origine destinée au calcul vectoriel de traitement de signal, mais a ensuite été étendue aux calculs généraux en flottants simple et double précision. En mode 64 bits,[7] que ce soit sous Linux ou sous Windows, et sur MacOS X quel que soit le mode, le SSE est utilisé par défaut et le x87 est en voie d'obsolescence. SSE calcule uniquement avec des flottants IEEE-754 simple ou double précision, il n'y a donc pas de problème de double arrondi. Remarquons par contre qu'on ajoute de la confusion sur une situation déjà compliquée, car le même programme pourra avoir un comportement différent sur une même machine x86 32-bits en passant d'un système d'exploitation à l'autre, ou sur une même machine en passant d'un mode de compilation à l'autre. [8]

3.1.5 Des fonctions mal spécifiées et peu sûres

Le coprocesseur flottant 80387 proposait un jeu d'instructions « sophistiquées », y compris des fonctions transcendantes (sinus, cosinus, logarithme...), mais rien ne garantit que les processeurs compatibles du même fabricant, *a fortiori* ceux d'autres fabricants, reproduisent son comportement : le comportement de ces instructions est spécifié de façon floue ; les bibliothèques de calcul ne sont pas en reste [151, §4.1]. Par ailleurs, il n'est pas rare que les bibliothèques de calcul livrées avec les systèmes d'exploitation aient un comportement fantaisiste quand elles sont utilisées dans des modes ou sur des arguments inhabituels [151, §4.2].

[6] Même si les registres font 80 bits, soit 5 mots de 16 bits, l'ABI courante sur processeurs IA32 est de les aligner sur 3 mots de 32 bits ; mais cette ABI est inadaptée pour les processeurs modernes, qui préfèrent un alignement sur 16 octets. [76, *Intel 386 and AMD x86-64 Options*].

[7] Nous faisons allusion ici aux architectures récentes compatibles x86 mais offrant notamment un adressage sur 64 bits, dites AMD64, EM64T, x64 ou encore x86-64, à ne pas confondre avec l'architecture IA64 des processeurs Itanium.

[8] Citons le cas d'un programme Objective Caml dont le comportement changeait sur MacOS X entre l'interpréteur *bytecode*, destin et la compilation en code natif, le premier utilisant SSE (choix par défaut du compilateur C utilisé pour compiler l'interpréteur de *bytecode*), le second utilisant x87. L'utilisateur était tombé sur une des rares valeurs produisant un double arrondi sur le x87.

3.1.6 Une situation difficile

Pour reprendre à l'égard de la virgule flottante une expression provocante formulée à propos d'autres aspects de la programmation, *"What you see is not what you execute"* [8, 10] : ce que semble dire le code source ne correspond pas à ce qui est effectivement exécuté en machine. Les résultats que nous dérivons d'une analyse de code source peuvent être faux sur le code objet en raison de modifications sémantiques introduites par la compilation.

Une solution radicale est d'analyser ou de raisonner sur le code « assembleur ». Il existe déjà des infrastructures [8, 9, 10, 84], on peut leur adjoindre des fonctionnalités adaptées à la virgule flottante. Même dans ce cas, on est encore à la merci des problèmes de spécification et d'implémentation de certaines fonctions (§3.1.5).

Le danger des imprécisions sémantiques que nous avons évoquées est d'autant plus grand que l'on se base sur une sémantique précise. Ainsi, si l'on a décidé que le résultat de $x \oplus y$ a une valeur précise, indépendante du temps, on est vulnérable aux problèmes de double arrondi ou de transformation d'expression (fma). Si l'on prend une sémantique exacte $x \oplus y = r(o + y)$, on sera vulnérable au moindre écart sémantique. Les méthodes qui partent d'une sémantique lâche et approximative seront par contre moins sensibles à ces problèmes.

3.2 Abstraction

Notre but a été de dégager des méthodes permettant d'obtenir efficacement des résultats sûrs portant sur les résultats finaux ou les valeurs intermédiaires des calculs en virgule flottante. En revanche, nous ne nous sommes pas intéressés aux propriétés plus fines, mais plus coûteuses à établir, étudiées par l'équipe d'Éric Goubault, concernant notamment la détermination et la quantification de l'origine des erreurs d'arrondi [85, 86, 125, 126, 127].

3.2.1 Abstractions non sûres

De même que certains outils raisonnent sur les entiers des programmes en langage C (ou autres langages similaires) comme s'ils s'agissaient d'entiers relatifs ($\mathbb{Z}$) et non d'entiers modulaires ($[0, 2^n - 1]$ ou $[-2^n, 2^{n-1}]$ considérés modulo 2^n), certains outils raisonnent sur les nombres en virgule flottante comme s'ils s'agissaient de nombres réels ($\mathbb{R}$). Dans un cas comme dans l'autre, il s'agit de substituer à des êtres aux propriétés mathématiques mal maîtrisés des structures algébriques fortes, quitte à introduire une incorrection.

Dans certains cas, il est possible de borner *a priori* les quantités rencontrées (au besoin, on vérifiera *a posteriori* que les bornes sont correctes). On peut alors construire une abstraction non sûre des réels ou des flottants par

des entiers bornés représentant des réels en virgule fixe. L'intérêt d'une telle abstraction est la possibilité d'utiliser des procédures de décision pour les entiers bornés, fondées sur les techniques de *bit-vectors* [14].

3.2.2 Raisonnements corrects

Il existe plusieurs façons de raisonner sur les calculs en virgule flottante :
- On peut partir d'une spécification au bit près, éventuellement exécutable, des opérations élémentaires [73]. Au vu des problèmes évoqués au §3.1.1, cela n'a probablement de sens que si l'on opère au niveau du code objet ou si l'on est absolument sûr de ce que le compilateur fait. En pratique, on commencera par démontrer quelques propriétés de plus haut niveau, comme par exemple le fait que $x \oplus y = r(x + y)$ avec r la fonction d'arrondi spécifiée par la norme. On pourra même démontrer que cette spécification au bit près est une implémentation de la norme. Cette approche est la plus puissante (elle permet de démontrer toutes les propriétés démontrables), mais elle est délicate — notamment, il faut être sûr de comment sont implémentées les fonctions élémentaires, ce qui est loin d'être évident quand on programme dans un langage de haut niveau dont le compilateur peut procéder à des réorganisations.
- On peut partir des propriétés garanties par la norme et laisser non spécifiées les points qu'elle ne fixe pas. Là encore, en pratique, on commencera par prouver quelques lemmes, par exemple celui de Sterbenz [155, §2.1.3, Th 2].
- On peut partir de lemmes de plus haut niveau impliqués par la norme (après les avoir démontrés, bien sûr). C'est ce que nous avons fait dans tous nos travaux.

Les propriétés dont nous avons besoin sont :

1. La monotonie des opérations élémentaires ; par exemple, $x \mapsto a \otimes x$ est croissante pour $a \geq 0$. Ceci permet d'utiliser une arithmétique d'intervalles.
2. Des relations simples entre x et $r(x)$, par exemple une borne sur $|x - r(x)|$.

L'arithmétique d'intervalle s'implémente simplement, du moins pour les opérations élémentaires $+, -, \times, /, \sqrt{}$, si l'analyseur fonctionne sur une plateforme disposant d'arrondis dirigés (vers $\pm\infty$) des mêmes types flottants que la plate-forme d'exécution du programme à analyser. Ainsi, l'addition s'implémente par $[a, b] + [c, d] = [a +_{-\infty} c, b +_{+\infty} d]$, la soustraction est très semblable ; la multiplication et la division sont plus compliquées. Comme le changement de mode d'arrondi est coûteux sur certains processeurs, on emploie volontiers l'astuce $a +_{-\infty} b = -((-a) +_{+\infty} (-b))$.

L'arithmétique d'intervalles est cependant insuffisante pour démontrer un grand nombre de propriétés. Il faut donc utiliser des domaines relationnels. Or, le domaines relationnels supposent en général des relations algébriques fortes sur les valeurs concrètes manipulées, relations absentes sur les flottants.

Par souci de simplicité, nous ferons abstraction par la suite des dépassements de capacité dans les flottants. Ceux-ci peuvent être gérés en testant systématiquement des conditions auxiliaires vérifiant que les nombres générés sont bien entre les bonnes bornes.

Miné [130, 131, 134] a proposé de remplacer le résultat $r(x)$ de chaque opération flottante (par exemple, $x \otimes b$) par la somme du résultat x de l'opération correspondante dans les réels et d'une erreur, erreur qu'il est possible de borner en fonction de $|x|$. Si x ne subit pas l'arrondi vers zéro pour cause de trop petite valeur absolue (parfois nommé *underflow*), alors la différence entre x et $r(x)$ se fait au niveau du bit de mantisse le moins significatif et on a donc au maximum une certaine erreur relative ε_{rel} : $|x - r(x)| \leq \varepsilon_{\text{rel}}|x|$. Si l'on veut être rigoureux, il ne faut cependant pas oublier le cas où $|x|$ est tellement faible que x est arrondi vers zéro, de sorte qu'on commet alors une erreur absolue au plus de ε_{abs} (dans le cas de l'arrondi au plus proche, ε_{abs} est la moitié du plus petit flottant strictement positif ε_0). Miné, Miné, Miné propose donc l'inégalité « quasi linéaire » :

$$|x - r(x)| \leq \varepsilon_{\text{rel}}|x| + \varepsilon_{\text{abs}} \tag{3.1}$$

On peut cependant faire mieux. Dans le cas d'une arithmétique flottante avec dénormalisés, tout opérande flottant est un multiple du plus petit strictement positif ε_0. Donc, si x et y sont deux flottants, leur différence ou leur somme est donc elle aussi un multiple d'ε_0. Il ne peut donc y avoir d'arrondi vers zéro, et la relation s'affine :

$$|(a \oplus b) - (a + b)| \leq \varepsilon_{\text{rel}}|a + b| \tag{3.2}$$

Au Ch. 4, nous utilisons des contraintes linéaires sur les réels. Nous pouvons par exemple réécrire l'inégalité 3.2 comme :

$$\begin{aligned}(\leq (1 - \varepsilon_{\text{rel}})(a + b) \leq a \oplus b \leq (1 + \varepsilon_{\text{rel}})(a + b) \wedge a + b \geq 0)\\ \vee (\leq (1 + \varepsilon_{\text{rel}})(a + b) \leq a \oplus b \leq (1 - \varepsilon_{\text{rel}})(a + b) \wedge a + b \leq 0)\end{aligned} \tag{3.3}$$

Quant à l'inégalité 3.1, nous pouvons la raffiner pour les réels très proches de zéro, par exemple comme :

$$\begin{aligned}(\leq (1 - \varepsilon_{\text{rel}})x \leq r(x) \leq (1 + \varepsilon_{\text{rel}})x \wedge x \geq \epsilon_0)\\ \vee (r(x) = \epsilon_0 \wedge \epsilon_0/2 < x \leq \epsilon_0)\\ \vee (r(x) = 0 \wedge -\epsilon_0/2 \leq x \leq \epsilon_0/2)\\ \vee (r(x) = -\epsilon_0 \wedge -\epsilon_0 \leq x < -\epsilon_0/2)\\ \vee (\leq (1 + \varepsilon_{\text{rel}})x \leq r(x) \leq (1 - \varepsilon_{\text{rel}})x \wedge x \leq \epsilon_0)\end{aligned} \tag{3.4}$$

Chapitre 4

Élimination des quantificateurs et abstractions

Éliminez tous les autres facteurs, et celui qui subsiste doit être la vérité.

Arthur Conan-Doyle, *Le Signe des Quatre*

Nous considérons des formules écrites à l'aide des connecteurs logiques $\vee$, $\wedge$, $\neg$, des quantificateurs $\forall$ et $\exists$ sur un ensemble V de variables, et des formules atomiques $l_1 \leq l_2$, $l_1 < l_2$, $l_1 = l_2$, $l_1 \geq l_2$, $l_1 > l_2$ et $l_1 \neq l_2$ où l_1 et l_2 sont des expressions affines linéaires à coefficients entiers à variables dans V. Par souci de simplicité, nous nous restreindrons sans perte de généralité aux cas où les formules atomiques sont de la forme $l \geq 0$, les inégalités strictes s'obtenant par négation d'inégalités larges.

Ces formules s'interprètent directement sur $\mathbb{Q}$ ou $\mathbb{R}$: une formule à n variables libres a pour modèle un sous-ensemble de $\mathbb{Q}^n$ ou $\mathbb{R}^n$. Il est évident qu'une formule est satisfiable sur $\mathbb{Q}$ si et seulement si elle l'est sur $\mathbb{R}$. Nous parlerons de « théorie linéaire des réels ».

Cette théorie admet l'élimination des quantificateurs : étant donné une formule F avec des quantificateurs, il existe une formule F' sans quantificateurs et avec les mêmes modèles. Cette transformation est algorithmique.

Tous les algorithmes d'élimination des quantificateurs pour cette théorie se ramènent à l'élimination d'un quantificateur existentiel $\exists x\ F$, où F est une formule sans quantificateurs, parfois en permettant l'élimination simultanée d'un bloc $\exists x_1, \ldots, x_n\ F$. Pour se ramener à ce cas, on procède par récurrence sur la structure syntaxique de la formule : si l'on rencontre un quantificateur existentiel, on l'élimine, et pour un quantificateur universel $\forall x\ F$ on le réécrit en $\neg \exists x\ \neg F$ et on procède de même. Par la suite, nous ne discuterons donc que le cas du quantificateur existentiel unique.

4.1 Algorithmes

4.1.1 Élimination de quantificateurs existentiels sur les conjonctions

Pour nous convaincre qu'il est possible d'éliminer algorithmiquement les quantificateurs, nous allons d'abord donner un algorithme naïf et inefficace. On passe d'abord F en forme normale disjonctive et on se ramène donc à éliminer les quantificateurs dans $\exists x\ (F_1 \vee \dots \vee F_n)$ où les F_i sont des conjonctions d'inégalités linéaires, donc à $(\exists x\ F_1) \vee \dots \vee (\exists x\ F_n)$. On se ramène donc à l'élimination de quantificateurs sur des conjonctions d'inégalités linéaires (dans le pire cas, en nombre exponentiel).

Algorithme de Fourier-Motzkin

Considérons une conjonction C d'inégalités linéaires, larges dans un premier temps ($l \geq 0$). En séparant les inégalités où x apparaît avec un coefficient strictement positif, strictement négatif ou nul, on peut mettre cette conjonction sous la forme :

$$\begin{cases} x \geq l_1^-(y,z,\dots) \\ \vdots \\ x \geq l_a^-(y,z,\dots) \\ x \leq l_1^+(y,z,\dots) \\ \vdots \\ x \leq l_b^+(y,z,\dots) \\ l_1(y,z,\dots) \geq 0 \\ \vdots \\ l_c(y,z,\dots) \geq 0 \end{cases} \tag{4.1}$$

Autrement dit :

$$\begin{cases} x \geq \max\left(l_1^-(y,z,\dots),\dots,l_a^-(y,z,\dots)\right) \\ x \leq \min\left(l_1^+(y,z,\dots),\dots,l_b^+(y,z,\dots)\right) \\ l_1(y,z,\dots) \geq 0 \\ \vdots \\ l_c(y,z,\dots) \geq 0 \end{cases} \tag{4.2}$$

$\exists\ xC$ est donc équivalent à :

$$\max\left(l_1^-(y,z,\dots),\dots,l_a^-(y,z,\dots)\right) \leq \min\left(l_1^+(y,z,\dots),\dots,l_b^+(y,z,\dots)\right) \\ \wedge\, l_1(y,z,\dots) \geq 0 \wedge \dots l_c(y,z,\dots) \geq 0 \tag{4.3}$$

$\max(\alpha_1, \ldots, \alpha_a) \leq \min(\beta_1, \ldots, \beta_b)$ est équivalent à la conjonction de tous les $\alpha_i \leq \beta_j$. On obtient ainsi un système de $ab + c$ inégalités linéaires. Si $n = a + b + c$ est le nombre total d'inégalités avant élimination, le nombre d'inégalités après élimination est inférieur à $n^2/4$. Notons qu'une grande partie de ces inégalités peut être superflue (c'est-à-dire qu'il s'agit d'inégalités impliquées par les autres et que l'on pourrait supprimer sans changer le résultat).

Le procédé se généralise aux systèmes d'inégalités mixtes larges et strictes : quand on combine deux inégalités large on obtient une inégalité large, sinon on obtient une inégalité stricte.

Si l'on fait q éliminations successives, la borne citée plus haut nous permet de conclure qu'on obtient au maximum $n^{2^q}/4^q$. Cette double exponentielle est de mauvais augure, et on peut se demander si une passe de simplification, supprimant les inégalités superflues, ne permettrait pas d'améliorer la complexité.

Polyèdres convexes

L'ensemble des solutions d'un système d'inégalités linéaires à n variables est un polyèdre convexe dans $\mathbb{Q}^n$ ou $\mathbb{R}^n$. Si le polyèdre est d'intérieur non vide, chaque face (de dimension $n-1$) du polyèdre correspond à une inégalité du système d'origine ; et les inégalités qui ne correspondent à aucune face sont superflues. Un polyèdre non vide mais d'intérieur vide s'obtient si certaines contraintes impliquent des relations d'égalité.

Un polyèdre convexe peut être représenté comme solution d'un système d'inégalités linéaires (représentation par contraintes), ou encore comme enveloppe convexe d'un système de points (sommets du polyèdre), et, pour les polyèdres infinis, de droites et de demi-droites (représentation par générateurs). Passer d'une représentation à l'autre est coûteux, car l'une peut être exponentiellement plus grande que l'autre : il suffit de considérer par exemple un hypercube à n dimensions, qui est donné par $2n$ contraintes mais a 2^n sommets.

Il existe une riche littérature sur l'algorithmique sur les polyèdres convexes, notamment stimulée par les applications en analyse de programmes [49] ; Bagnara et al. [7] en donne un panorama. Il existe plusieurs bibliothèques libres manipulant des polyèdres en double représentation (par contraintes et par générateurs),[1] avec des calculs sur des entiers en précision étendue (indispensable dans un contexte où l'on ne peut borner d'avance les coordonnées) ; citons notamment NewPolka, maintenant intégrée dans

[1] On appelle parfois « algorithme de Chernikova » l'algorithme permettant de passer d'une représentation à l'autre, du nom de l'auteur russe Черникова, parfois aussi translittéré comme Černikova, qui a laissé plusieurs articles sur la résolution complète de systèmes d'équations ou d'inéquations linéaires. Il semble que cet algorithme ait été plusieurs fois redécouvert, par exemple par Halbwachs [91, §2.3].

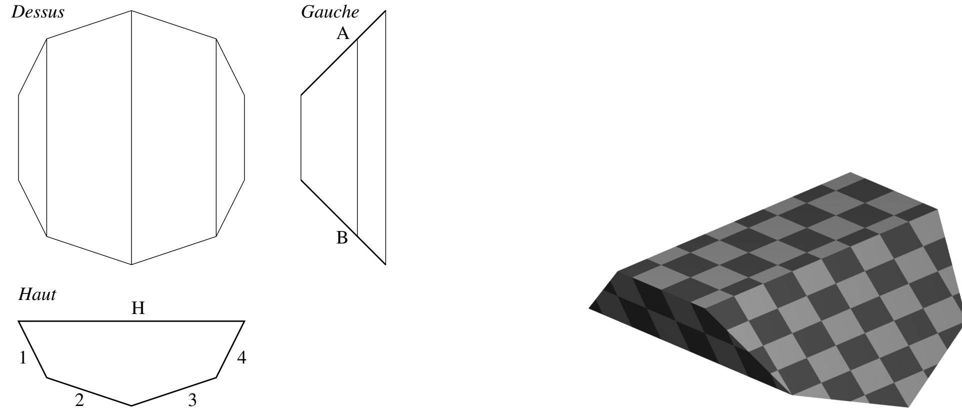


FIG. 4.1 – Ce polyèdre à 7 faces dans l'espace de dimension 3, dessiné selon les conventions du dessin technique, se projette en dimension 2 en un polygone à 10 côtés. On généralise dans le lemme 1 cette construction à un polyèdre à $n+3$ faces mais dont la projection a $2n+2$ côtés.

APRON,[2] et la *Parma Polyhedra Library* (PPL).[3] Nous avons utilisé ces deux bibliothèques dans nos expérimentations.

L'élimination de q variables quantifiées existentiellement revient géométriquement à projeter q dimensions du polyèdre et à prendre la représentation par contraintes du polyèdre projeté. Une intuition fausse est que la projection « simplifie » forcément les polyèdres, qu'on obtient forcément un nombre de faces inférieur. En fait, même pour une projection entre la dimension 3 et la dimension 2, on peut doubler le nombre de faces :

Lemme 1. *Il existe une famille p_n de polyèdres bornés dans $\mathbb{R}^3$, indexée par n, telle que p_n a $n+3$ faces mais dont la projection sur un plan a $2n+2$ côtés.*

Démonstration. Nous avons dessiné la construction pour $n = 4$ (Fig. 4.1). On trace d'abord un polygone convexe à $n+1$ côtés dans le plan (x, z) (en bas de la figure : côtés H, $1, \dots, n$), avec un côté $z = 0$ et les autres côtés en une sorte de forme semicirculaire (par exemple en prenant la moitié d'un polygone régulier à $2n$ côtés). Ce polygone est ensuite extrudé le long de l'axe y, et le prisme ainsi formé est tranché à angle aigu en ses deux extrémités (lignes en gras A and B à droite de la figure), produisant ainsi un polyèdre à $n+3$ faces. Sa projection sur le plan (x, y) a $2n+2$ faces, comme on le voit bien sur la figure. □

Le calcul de projection est très simple sur la représentation par générateurs (il suffit de projeter les sommets, droites et demi-droites), l'algorithme

[2] `http://apron.cri.ensmp.fr/`
[3] `http://www.cs.unipr.it/ppl/`

usuel de projection est donc de passer en représentation par générateurs, de projeter et de repasser en représentation par contraintes. Ceci nous donne une borne sur le nombre de faces du polyèdre projeté : 2^{2^n}, encore une double exponentielle. Peut-on vraiment atteindre cette borne doublement exponentielle ? Nous n'avons pas trouvé d'exemple dans la littérature [115].

4.1.2 Méthodes par substitution

L'idée de ces méthodes est de remplacer une disjonction infinie (quantification existentielle) $\exists x\ F(x)$ par une disjonction finie $F(x_1) \vee \cdots \vee F(x_m)$ où $x_1, \ldots, x_m$ soit des « témoins » bien choisis, dont l'expression dépend en général des variables libres de la formule.

Considérons une formule F sans quantificateurs dont les variables libres sont $x, y, z, \ldots$. Sans perte de généralité, nous pouvons repousser les négations aux feuilles de l'arbre syntaxique en $\wedge$ et $\vee$, de sorte que ces feuilles sont de la forme $l \geq 0$ ou $\neg(l \geq 0)$, autrement dit $l < 0$ (*forme normale négative*).

Fixons $y, z, \ldots$ et étudions l'ensemble des modèles de F relativement à x. Cet ensemble est une union (éventuellement vide) d'intervalles dont les bornes sont soit $\pm\infty$, soit les solutions ρ_i pour x des équations $l_i(x, y, z, \ldots) = 0$ où les $l_1, \ldots, l_m$ sont les expression linéaires présentes dans F. L'idée des algorithmes de Ferrante and Rackoff [71][25, §7.3][159, §4.2] et Loos and Weispfenning [123][159, §4.4] est d'utiliser le fait que la valeur de vérité de F ne peut changer qu'en cet ensemble de m points $\rho_1, \ldots, \rho_m$. L'idée est de choisir au moins un représentant x_i par intervalle où la fonction est constante (Fig. 4.2).

Dans les deux cas, on commence par distinguer les intervalles extrêmaux (ceux de la forme $(-\infty, \rho_i)$ ou $(\rho_i, +\infty)$). On obtient la valeur de $F(-\infty)$ et $F(+\infty)$ par simple substitution arithmétique (si on admet l'arithmétique sur les infinités) ou, plus simplement, par *substitution virtuelle* : si la formule atomique $l \geq 0$ ne fait pas intervenir x, on la laisse intacte, si x a un coefficient strictement positif dans l on remplace par vrai (pour $x \mapsto +\infty$) ou faux (pour $x \mapsto -\infty$), et l'inverse pour un coefficient strictement négatif. Le choix des représentants des autres intervalles diffère entre les deux algorithmes.

Points intérieurs Ferrante and Rackoff [71] remarquent que comme $F(x)$ est constante sur les intervalles (ρ_i, ρ_j), il suffit de prendre comme témoins les milieux de ces intervalles. En général, cela va donner $m(m+1)/2$ valeurs différentes où m est le nombre de formules atomiques. À chaque élimination de variable, la taille de la formule est donc élevée au cube (à constante multiplicative près). Ceci donne une complexité globale en $|F|^{2^{cq}}$ où $|F|$ est la taille de la formule d'origine et q est le nombre de quantificateurs à éliminer.

On rajoute ensuite $F(+\infty)$ et $F(-\infty)$.

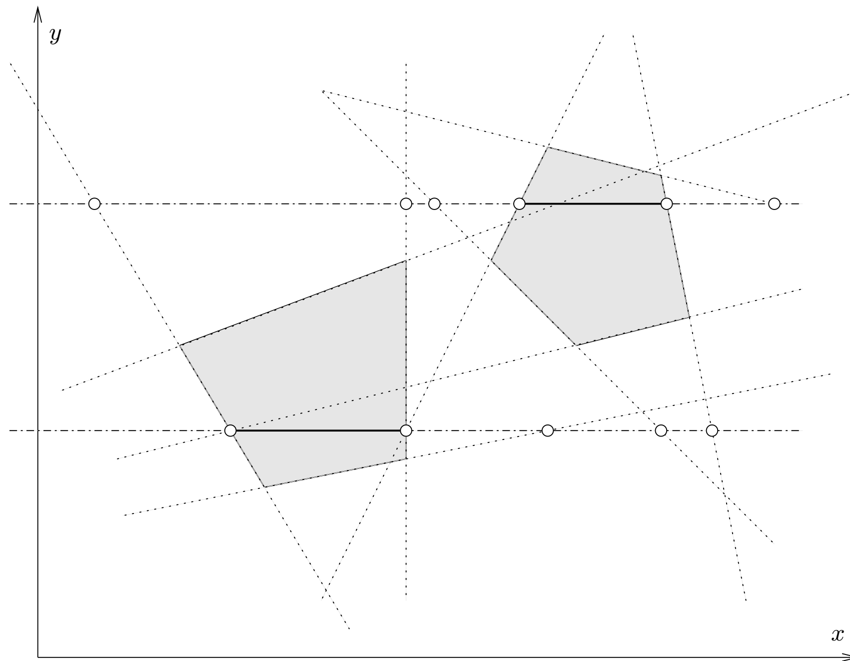


FIG. 4.2 – La zone grisée S est l'ensemble des solutions (x, y) de la formule F, dont les atomes sont les inégalités linéaires correspondant aux droites Δ tracées en pointillés. Pour $y = y_0$ fixé, l'ensemble des x tels que $F(x, y)$ est vrai est formé d'intervalles reliant les intersections I de la droite $y = y_0$ avec les droites de Δ, figurées avec un petit rond. $y = y_0$ a donc une intersection avec S si et seulement si un élément de I, ou un intervalle borné par un ou deux éléments de I, est dans S. Ce dernier point peut être testé soit en testant, outre les directions $x \to \pm\infty$, tous les milieux de segments entre deux éléments de I (méthode de Ferrante and Rackoff) ou, outre $x \to -\infty$, pour chaque élément de I un point situé infinitésimalement proche à sa droite (méthode de Loos and Weispfenning). Ces deux méthodes se basent sur le fait que chaque points de I (intersection de $y = y_0$ et d'une droite de Δ) peut s'exprimer comme une fonction linéaire affine explicite de y_0.

Infinitésimaux Chaque intervalle de positivité de F a pour extrémité gauche un des ρ_i, qui doit qui plus est correspondre :

- à une formule atomique de la forme $l \geq 0$ où x a un coefficient strictement positif si l'intervalle est fermé en son extrémité gauche ;
- à une formule atomique de la forme $l < 0$ où x a un coefficient strictement négatif si l'intervalle est ouvert en son extrémité gauche.

Pour obtenir les extrémités gauches fermées, on prend donc dans les x_i tous les ρ_j correspondant à des contraintes $l \geq 0$ où x a un coefficient strictement positif. Pour les extrémités gauches ouvertes, on prend $\rho_j + \epsilon$, où les ρ_j correspondent à des contraintes $l < 0$ où x a un coefficient strictement négatif et ϵ est un infinitésimal, c'est à dire un élément rajouté strictement positif et plus petit que tous les réels strictement positif.[4]

On pourrait bien sûr procéder par calcul sur les infinitésimaux, mais là encore on peut procéder par substitution virtuelle. Considérons une formule atomique de la forme $cx + d(y, z, \dots) \geq 0$, où l'on veut remplacer x par $\rho + \epsilon$. Si $c > 0$, alors on obtient $c\rho + d(y, z, \dots) \geq 0$. Si $c < 0$, on obtient $c\rho + d(y, z, \dots) > 0$.

On rajoute ensuite $F(-\infty)$. La méthode fonctionne également si on considère les extrémités droites d'intervalles (substituer x par $-x$). On pourra choisir la direction qui nécessite le moins de points.

L'avantage de cette méthode est que le nombre de points x_i est linéaire en la taille de la formule, alors que la méthode de Ferrante and Rackoff utilise un nombre quadratique de points. On a donc une méthode là aussi en $|F|^{2^{cq}}$ mais avec une meilleure constante.

Ces méthodes sont des méthodes syntaxiques, qui conservent (en la dupliquant) la structure de la formule d'origine sans jamais essayer de la simplifier. La géométrie du problème est ignorée.

4.1.3 Méthode géométrique

Au §4.1.1, nous avions introduit une méthode naïve pour l'élimination des quantificateurs : tout passer d'abord en forme normale disjonctive, puis projeter séparément chaque conjonction. Géométriquement, cela revient à représenter l'ensemble des solutions comme une union explicite de polyèdres convexes, puis à projeter chaque polyèdre (§4.1.1).

Nous avons proposé un algorithme améliorant cette idée naïve par deux idées [141] :

1. On génère les éléments de la disjonction et on les projette au fur et à mesure, en accumulant le résultat dans la solution. On ne génère pas

[4] On peut définir ainsi le calcul sur les infinitésimaux : les quantités sont de la forme $a + b\epsilon$ où a et b sont des rationnels, et on les ordonne en prenant l'ordre lexicographique sur (a, b).

les polyèdres qui se projettent totalement dans la solution déjà générée (ils n'ajoutent rien).

2. Pour obtenir les éléments de la disjonction, on utilise non pas des moyens syntaxiques, mais un algorithme de satisfiabilité modulo théorie (SMT).

Le problème de satisfiabilité booléenne (SAT) est bien connu comme problème NP-complet [43]. Il existe pour ce problème divers algorithmes, souvent dérivés de l'algorithme DPLL [57, 58], de nombreux outils efficaces de résolution, un format de fichier standard [63], des bibliothèques de *benchmarks* [99] et même une compétition.[5] Le problème est simple : étant donné une formule propositionnelle, dire si elle est satisfiable et, si elle l'est, donner un modèle.

Ce cadre a été par la suite étendu à des formules non propositionnelles, c'est-à-dire dont les atomes sont pris dans une théorie, par exemple, ce qui nous intéresse ici, la théorie des inégalités linéaires sur les réels. Il s'agit d'un sujet de recherche actif, et les algorithmes usuels [80] mêlent une recherche SAT et une procédure de décision pour les conjonctions de la théorie considérée. Nous avons principalement utilisé l'outil YICES[6] mais nous avons également implémenté un algorithme SMT naïf et « paresseux », permettant notamment de vérifier à quel point l'efficacité de notre algorithme d'élimination des quantificateurs dépendait de l'excellence de l'algorithme SMT utilisé. Cet algorithme naïf fonctionne ainsi :

1. Remplacer chaque inégalité linéaire $l \geq 0$ par une variable propositionnelle, à l'aide d'un dictionnaire.
2. Résoudre le problème SAT propositionnel. S'il n'a pas de solution, le problème d'origine n'en a pas.
3. S'il a une solution, celle-ci définit une conjonction de littéraux, donc un polyèdre. Si celui-ci contient un point, on le propose comme solution.
4. Sinon, on calcule un témoin de vide (§5.1.2) : une conjonction insatisfiable, donc indésirable, de littéraux et on rajoute sa négation au problème.

Ayant à notre disposition un SMT-solveur et une bibliothèque de calcul sur les polyèdres, nous pouvons maintenant proposer un algorithme pour éliminer le quantificateur dans $\exists\ x_1, \ldots, x_n\ F$ et fournir une solution S en forme normale disjonctive :

1. Le SMT-solveur propose un modèle de F ; s'il n'y en a pas, F est insatisfiable et on termine.
2. Ce modèle se généralise immédiatement à tous les points de même valuation par rapport aux formules atomiques de F (les points qui

[5] `http://www.satcompetition.org/`
[6] `http://yices.csl.sri.com/`

vérifient les mêmes inégalités). On obtient ainsi une conjonction C de littéraux telle que $C \implies F$.

3. On généralise cette conjonction en supprimant des littéraux de C tant que $C \implies F$.
4. On calcule la projection P de C (élimination de $\exists\ x_1, \dots, x_n\ P$). On rajoute P à S ($S := S \vee P$) et on retranche P de F ($F := F \wedge \neg P$). On reprend au départ.

Cet algorithme termine forcément, en au maximum 2^n itérations où n est le nombre d'inégalités dans F : chaque itération « consomme » un valuation de ces inégalités.

Notons également qu'il est possible d'appliquer l'algorithme pour éliminer 0 quantificateurs, ce qui équivaut à demander une mise sous forme normale disjonctive. En appliquant le même algorithme à la négation de la formule, on obtient une mise en forme normale conjonctive. On peut simplifier la formule en alternant les mises sous les deux formes normales.

Nous avons implémenté notre algorithme, ainsi que ceux de Ferrante and Rackoff et Loos and Weispfenning dans notre outil Mjollnir, que nous avons ensuite comparé à des outils existants, d'une part sur quelques instances isolées (table 4.1) et sur des instances aléatoires (table 4.2).

Mjollnir est l'algorithme décrit ici, implémenté avec le SMT-solveur Yices et la bibliothèque de polyèdres NewPolka via Apron, ou optionnellement la *Parma Polyhedra Library*. L'analyse de performances montre que la grande majorité du temps est passé en résolution SMT, de sorte que les éventuelles différences entre bibliothèques de polyèdres sont négligeables.

Naïf est une version préliminaire du même algorithme, implémentée avec l'algorithme de SMT naïf et paresseux vu plus haut.

Mjollnir (mod1) calcule une forme normale disjonctive par ALL-SAT, avant de projeter (option `-no-block-projected-model`).

Mjollnir (mod2) est une variante de notre algorithme censée, au prix d'une complication, être plus efficace [141, §4.2] (option `-add-blocking-to-g`).

Mjollnir Ferrante-Rackoff , voir §4.1.2.

Mjollnir Loos-Weispfenning , voir §4.1.2.

Lira[7] est un outil basé sur les automates de Büchi qui traite les problèmes linéaires mixtes rationnels et entiers (donc l'arithmétique de Presburger).

Mathematica[8] est un logiciel d'algèbre symbolique. Sa fonction `Reduce` implémente, semble-t-il, la décomposition cylindrique algébrique [35], un

[7] `http://lira.gforge.avacs.org/`
[8] `http://www.wolfram.com/`

algorithme d'élimination des quantificateurs dans la théorie des corps réels clos.

Redlog[9] est une extension du logiciel d'algèbre symboliqueREDUCE 3.8.[10] REDLOG implémente divers algorithmes dûs à Volker Weispfenning et son équipe [123].

Test	r. lim. $\mathbb{R}$	r. lim. float	`prsb23`	`blowup5`
MJOLLNIR	1.4	17	0.06	négligeable
MJOLLNIR (mod1)	1.6	77[a]	0.06	négligeable
MJOLLNIR (mod2)	1.5	34	0.07	négligeable
MJOLLNIR Loos-Weispfenning	o-o-m	o-o-m	o-o-m	négligeable
Preuve de concept	-	823	-	-
MJOLLNIR Ferrante-Rackoff	o-o-m	o-o-m	o-o-m	négligeable
Preuve de concept	-	823	-	-
LIRA	o-o-m	o-o-m	8.1	0.6
REDLOG `rlqe`	182	o-o-m	1.4	négligeable
REDLOG `rlqe`+`rldnf`	o-o-m	o-o-m	-	-
MATHEMATICA `Reduce`	(> 12000)	o-o-m	(> 780)	7.36

[a]La consommation mémoire monte à 1.1 GiB.

TAB. 4.1 – Comparaison entre les différents logiciels d'élimination de quantificateurs sur des instances isolées provenant de vérification de programmes (deux premières colonnes) et des exemples de LIRA.

	profondeur 14			profondeur 15			profondeur 16		
	Résolus	Moy	O-o-m	Résolus	Moy	O-o-m	Résolus	Moy	O-o-m
MJOLLNIR	100	1.6	0	94	9.8	0	73	35.3	0
MJOLLNIR (mod1)	94	8.2	3	80	27.3	7	39	67.1	25
MJOLLNIR (mod2)	100	3.8	0	91	13.9	0	65	39.2	0
MJOLLNIR Loos-W.	93	1.77	4	90	6.42	5	62	17.65	27
Naïf	94	1.4	0	86	2.2	0	55	17.7	0
MJOLLNIR Ferrante-R.	51	18.2	41	23	23.2	65	3	7.3	85
Naïf	94	1.4	0	86	2.2	0	55	17.7	0
LIRA	14	102.4	83	3	77.8	94	1	8	95
REDLOG (`rlqe`)	92	13.7	0	53	27.4	0	27	33.5	0
MATHEMATICA	6	30.2	0	1	255.7	0	1	19.1	0

TAB. 4.2 – Comparaisons de performances sur 3×100 instances aléatoires générées par `randprsb`, un outil associé à LIRA, avec des profondeurs de formules n respectivement 14, 15 and 16 (obtenues par `randprsb 0 7 -10 10` n i) avec i parcourant $[0, 99]$. Le tableau donne le nombre d'instances résolues sur les 100, le temps moyen par instance résolue, et le nombre moyen d'instances provoquant une insuffisance de mémoire (o-o-m).

Notre algorithme fournit des formules en forme normale disjonctive dont les atomes sont pris parmi un ensemble à au maximum $s^{2^q}/4^q$ éléments (voir

[9]`http://www.algebra.fim.uni-passau.de/~redlog/`
[10]`http://www.uni-koeln.de/REDUCE/`

§4.1.1), où q est le nombre de variables à éliminer et s est le nombre d'inégalités dans la formule F d'origine. La disjonction fournie a donc au maximum $2^{s^{2^q}/4^q}$ termes. Au final, nous obtenons une borne de complexité en $2^{2^{2^{|F|}}}$. La complexité des algorithmes de Ferrante and Rackoff et Loos and Weispfenning est seulement en $2^{2^{c|F|}}$; ces algorithmes ne fournissent cependant pas une forme normale disjonctive, dont le calcul rajouterait une exponentielle supplémentaire.

Même si notre algorithme a une borne de complexité supérieure à celle des algorithmes par substitution, les essais montrent qu'il se comporte mieux en pratique. Nous pensons que cela est dû au fait que les algorithmes par substitution ne simplifient jamais les formules qu'ils créent, qui grossissent à chaque étape, alors que notre algorithme restructure les formules à chaque étape et notamment élimine certaines disjonctions inutiles.

4.2 Application à l'analyse modulaire de programmes

Le lecteur se reportera à [143] pour plus de détails.

4.2.1 Domaines de modèles de contraintes linéaires

Sankaranarayanan et al. [182] ont proposé des domaines de contraintes linéaires de la forme :

$$\begin{cases} a_{1,1}v_1 + \cdots + a_{1,n}v_n \leq b_1 \\ \vdots \\ a_{m,1}v_1 + \cdots + a_{m,n}v_n \leq b_m \end{cases} \tag{4.4}$$

où les rationnels $a_{i,j}$ sont fixés et définissent le domaine, tandis que le vecteur des $b_i \in \mathbb{Q} \cup \{+\infty\}$ définit l'élément dans le domaine. De façon équivalente et plus concise, on pourra écrire $AV \leq B$. Intuitivement, il s'agit pour chaque domaine ainsi défini de considérer des polyèdres convexes dont on a fixé d'avance la direction des faces.

Il est aisé de constater que parmi les domaines exprimables de cette façon, on trouve notamment :

- les intervalles,
- les matrices de différences de potentiels — contraintes $v_1 - v_2 \leq C$,
- les octogones — contraintes $\pm v_1 \pm v_2 \leq C$ [129, 131, 132].

Pour chacun de ces domaines, il y a une correspondance de Galois : $\gamma(B) = \{V \mid AV \leq B\}$ et $\alpha(S)$ est le vecteur des $b_i = \sup_{V \in S} AV$. Remarquons que γ n'est pas injective : on retrouve la situation des octogones, à savoir qu'on peut avoir des contraintes trop lâches. Par exemple,

$x - y \leq 1 \wedge y - z \leq 2 \wedge x - z \leq 4$ et $x - y \leq 1 \wedge y - z \leq 2 \wedge x - z \leq 3$ définissent exactement le même ensemble d'états (x, y, z), mais dans le deuxième système toutes les inégalités sont « reserrées ».

L'opération de *clôture* consiste à calculer en fonction de $x^\sharp$ un $y^\sharp$ minimal tel que $y^\sharp = x^\sharp$, ce qui correspond à calculer $y^\sharp = \alpha \circ \gamma(x^\sharp)$. Dans le cas des matrices de différences de potentiel et des octogones, la clôture se fait par des algorithmes de plus court chemin (Floyd-Warshall). Dans le cas des domaines de contraintes linéaires en général, on procède par programmation linéaire (il suffit d'écrire les définitions de α et de γ pour obtenir un problème d'optimisation linéaire définissant $\alpha \circ \gamma$).

Par programmation linéaire encore, il est possible de définir des fonctions de transfert abstraites optimales pour l'analyse en avant ou en arrière des opérations comme :

- Les affectations linéaires affines $v_i := \sum_k h_k v_k + C$.
- Les gardes linéaires, tests $\sum_k h_k v_k \leq C$.

Comme d'habitude, ce sont les calculs d'invariants inductifs qui sont la partie la plus délicate de l'analyse. On peut, bien sûr, définir un opérateur d'élargissement coordonnée par coordonnée, comme le fait Miné [129, 131, 132], mais les opérateurs d'élargissement introduisent souvent des surapproximations néfastes. On a donc cherché à calculer des invariants « exacts » dans ces domaines.

Gaubert et al. [81] ont proposé de calculer des invariants inductifs dans certains domaines (octogones) par itération de politiques (terme provenant de la théorie des jeux). En résumé, il s'agit de formuler la fonction de transfert abstraite $f^\sharp(x^\sharp)$ sous la forme $x^\sharp \mapsto \inf_i \sup_j f^\sharp_{i,j}(x^\sharp)$. On cherche à calculer une surapproximation sûre du plus petit point fixe de $f^\sharp(x^\sharp)$. Clairement, pour tout i, le plus petit point fixe de $x^\sharp \mapsto \inf_i \sup_j f^\sharp_{i,j}(x^\sharp)$ convient, mais il est peut-être très surapproximé. Le choix de i est une *politique*. Lorsqu'une politique choisie est clairement sous-optimale, on l'altère pour passer à une autre politique, et on itère le procédé jusqu'à obtention d'une politique sans sous-optimalité évidente. Le plus petit point fixe de $f^\sharp(x^\sharp)$ correspond à au moins une politique, mais il n'est pas sûr que ce soit celle-ci qui soit obtenue par ce procédé. Adjé et al. [1] ont donc proposé une méthode permettant de continuer à descendre dans certains cas sous-optimaux. Ces techniques sont cependant très nouvelles en analyse de programmes et il reste à voir si elles produisent de bons résultats en pratique.

On a proposé des méthodes de calculs d'invariants se ramenant à des résolutions sous contraintes non linéaires, utilisant le fait que la théorie des corps réels clos admet l'élimination des quantificateurs : pour toute formule logique forméee à partir d'inégalités polynomiales à coefficients entiers, de connecteurs $\wedge$, $\vee$, $\neg$ et de quantificateurs $\forall$ et $\exists$, on peut construire une formule logique possédant les mêmes variables libres (à suppression près) et les mêmes modèles sur le corps des réels (ou, plus généralement, sur tout

corps réel clos), mais sans quantificateurs [16, 17, 40, 195, 196, 200]. Ainsi, $\exists x\ x^2 + bx + c = 0$ donne, après élimination, $b^2 - 4c \geq 0$.

Colón et al. [41], Sankaranarayanan et al. [181] proposent de générer des invariants linéaires $\sum c_i v_i \leq d$ par réduction de la recherche des coefficients c_i et d à un problème de résolution de contraintes non linéaires sur ces coefficients. Pour ma part, j'ai montré que même si on relâche les conditions ci-dessus et que l'on permet des inégalités et des opérations de programme *polynomiales*, alors il est toujours possible, pour un domaine de contraintes donné, de calculer avec une précision arbitraire le plus petit invariant inductif dans ce domaine [112, 147] par réduction à des problèmes d'élimination de quantificateurs.

Malheureusement, les techniques d'élimination de quantificateurs sur des problèmes non linéaires sont très coûteuses (double exponentielle même en pratique). Nous avons donc cherché des techniques moins coûteuses en nous restreignant au cas linéaire.

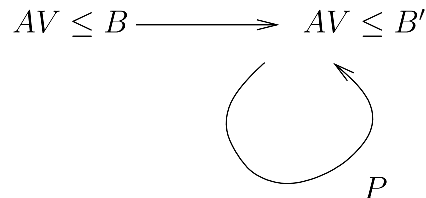
4.2.2 Formulation de la recherche de postconditions et d'invariants

Soit P un fragment de programme ne comprenant que des opérations (tests, assignements) linéaires, mais sans boucle. La sémantique dénotationnelle de ce programme $[\![P]\!]$ s'exprime, compositionnellement par transformation syntaxique de P, sous la forme d'une formule logique dont les variables libres sont les variables du programme au début et à la fin de P (en notant x' la valeur de x à la fin de l'exécution), et contenant éventuellement des quantificateurs existentiels portant sur les valeurs intermédiaires des variables.

Nous nous intéressons ici à l'analyse en avant, mais on pourrait sans difficulté formuler des résultats semblables pour l'analyse en arrière. Si $AV \leq B$ est une précondition, alors $AV' \leq B'$ est une postcondition correcte si :

$$\forall V\, \forall V'\ AV \leq B \wedge [\![P]\!] \Rightarrow AV' \leq B' \tag{4.5}$$

Prenons maintenant l'exécution en boucle de $[\![P]\!]$: on a une précondition avant la boucle $AV \leq B$ et on cherche un invariant inductif $AV' \leq B'$:



Il est clair qu'un tel invariant doit vérifier :

$$B' \leq B \wedge \forall V \forall V'\ AV \leq B' \wedge [\![P]\!] \Rightarrow AV' \leq B' \tag{4.6}$$

Dans les deux cas (relations 4.5 et 4.6), l'ensemble des valeurs correctes de B' est défini par une formule logique quantifiée dont les variables libres sont les coefficients de B et B'. Notons F une telle formule. Si nous voulons non pas définir tous les B' possibles, mais seulement le B' optimal, nous voulons $B'_{\text{opt}} = \min_{F[B']} B'$. Autrement dit, nous obtenons une formule :

$$G \stackrel{\triangle}{=} F[B'_{\text{opt}}] \wedge (\forall B' \; F[B'] \Rightarrow B'_{\text{opt}} \leq B') \quad (4.7)$$

Considérons maintenant $G_i = \exists b'_1 \ldots \exists b'_{i-1} \exists b'_{i+1} \exists b'_m \; G$ (G où toutes les coordonnées de B'_{opt} sont existentiellement quantifiées sauf la coordonnée i). G_i a pour variables libres les coordonnées de B et b'_i, et définit une fonction partielle (au sens mathématique) envoyant B sur b'_i, c'est-à-dire une relation telle qu'en fixant B, il y a au plus une valeur possible pour b'_i.

Limitons-nous maintenant à des valeurs *finies* pour les coordonnées de B et B'. En appliquant l'algorithme vu au 4.1.3, on peut obtenir une formule H_i sans quantificateurs, en forme normale disjonctive, équivalente à G_i. H_i définit alors une fonction affine par morceaux : pour chaque valeur de B, on regarde dans quels éléments de la disjonction H_i B peut rentrer. S'il n'y en a aucun, c'est que pour cette valeur de B il n'existe pas de meilleur invariant inductif ou de meilleure postcondition de la forme spécifiée. Sinon, en résolvant pour b_i dans un quelconque des termes de la disjonction convenant à B, on obtient la valeur de b_i. En fait, il est possible de transformer par manipulations syntaxiques la formule H_i en un arbre de tests linéaires dont les feuilles sont les valeurs de b_i, ce qui donne un programme fonctionnel calculant explicitement B' en fonction de B.

Nous avons donc défini un processus entièrement automatique prenant en entrée un fragment de programme P sans boucle (respectivement, une unique boucle autour d'un fragment de programme) et une définition de domaine abstrait et produisant en sortie la fonction de transfert abstraite optimale $\alpha \circ [\![P]\!] \circ \gamma$ (respectivement, la fonction de calcul du point fixe abstrait optimal $\text{lfp}(\alpha \circ [\![P]\!] \circ \gamma)$).

Illustrons cela par un exemple simple : précondition $x \in [x_{\min}, x_{\max}]$, programme $y := |x|$, postcondition $y \in [y_{\min}, y_{\max}]$. L'élimination de quantificateurs produit :

$$(x_{\min} + x_{\max} \geq 0 \wedge y_{\max} = x_{\max}) \vee \\ (x_{\min} + x_{\max} < 0 \wedge y_{\max} = -x_{\min}) \quad (4.8)$$

Chaque partie de la forme normale disjonctive représente une partie du domaine de définition de la fonction de transfert abstraite :

- sur $x_{\min} + x_{\max} \geq 0$ alors $y_{\max} = x_{\max}$,
- sur $x_{\min} + x_{\max} < 0$ alors $y_{\max} = -x_{\min}$.

et la conversion en une fonction exécutable donne :

```
if (xmin + xmax >= 0) {
  ymax = xmax;
} else {
  ymax = -xmin;
}
```

4.2.3 Généralisation au cas non-linéaire

Seidenberg [187], Tarski [195, 196] ont montré que la théorie des corps réels clos (c'est-à-dire, grossièrement, l'arithmétique polynomiale sur les réels) admet l'élimination des quantificateurs. Par exemple, la formule $\exists x\ x^2 + bx + c = 0$ fournit, après élimination, $b^2 - 4c \geq 0$. Des algorithmes plus efficaces, mais tout de même de complexité doublement exponentielle, ont été suggérés par la suite, notamment la décomposition cylindrique algébrique [35, 40]. Dans certains cas, il est possible d'utiliser des techniques moins coûteuses [200].

À l'aide de ces résultats, nous avons montré une généralisation de la caractérisation donnée au §4.2.2 au cas polynomial, à savoir que l'on peut caractériser exactement les coefficients $C_1, \ldots, C_m$ optimaux pour des post-conditions optimales ou des plus petits invariants dans des domaines de contraintes polynomiales [147] :

$$\begin{cases} P_1(v_1, \ldots, v_n) & \leq C_1 \\ \vdots & \\ P_m(v_1, \ldots, v_n) & \leq C_m \end{cases} \tag{4.9}$$

À partir de cette caractérisation, il est possible de calculer des surapproximations des valeurs numériques de $C_1, \ldots, C_m$ avec une précision arbitraire. En raison du coût de l'élimination de quantificateurs, cette méthode est sans doute plutôt d'un intérêt théorique que pratique — reste maintenant à trouver des méthodes plus efficaces, peut-être au prix de surapproximations supplémentaires.

4.2.4 Correction

La correction des fonctions de transfert produites est garantie par construction, puisque l'on part d'une définition logique de fonction de transfert correcte et que l'on obtient une implémentation par raffinement automatique de cette définition. Le problème est que ce raffinement, et notamment la procédure d'élimination de quantificateurs, pourrait être défectueux. On pourrait vouloir garantir la correction de l'élimination de quantificateurs, soit en prouvant correcte l'implémentation de l'algorithme, soit en lui faisant fournir, en sus de chaque résultat, un certificat de correction de celui-ci (voir Ch. 5).

Comment parvenir à un tel résultat ? Chaieb and Nipkow [36], Nipkow [159] proposent une implémentation certifiée d'algorithmes d'élimination basés sur les substitutions (§4.1.2)Pour l'algorithme décrit au 4.1.3, on peut distinguer deux points délicats : le *SMT-solver* (notamment, donner une preuve de l'insatisfiabilité de la formule dans le test de terminaison de la boucle d'énumération de la forme normale disjonctive) et la projection polyédrale.

Il n'est sans doute pas difficile de prouver correcte, une implémentation de l'algorithme de Fourier-Motzkin (§4.1.1), ou de modifier celle-ci pour qu'elle produise des témoins de correction, même en incorporant une phase d'élimination de contraintes redondantes : une contrainte C est redondante par rapport à une conjonction D si $D \wedge C \equiv D$, c'est-à-dire si $\neg C \wedge D$ est insatisfiable, on se ramène donc à un problème de recherche de témoin d'insatisfiabilité d'un système d'inégalités linéaires (§5.1.2). Cependant, en pratique, on utilise plutôt des algorithmes basés sur la double description des polyèdres par contraintes et générateurs (§4.1.1) : on part d'un polyèdre sous forme de contraintes (système d'inégalités linéaires), on obtient un système de générateurs (points, demi-droites, droites) dont le polyèdre est l'enveloppe convexe, on projette les générateurs, leur projection génère la projection du polyèdre, et on fait la conversion inverse vers un système d'inégalités. Il est probable que la preuve de correction d'une telle implémentation soit assez complexe et fastidieuse.

Remarquons cependant qu'il n'y a pas besoin, pour prouver la correction de l'analyse présentée ici, de prouver celle de l'élimination de quantificateurs. En effet, l'élimination de quantificateurs sert à trouver une solution optimale à un problème d'optimisation sous contraintes, alors que la correction du système est garantie par simple vérification qu'il s'agit d'une solution. La correction d'une fonction de transfert s'obtient simplement en vérifiant la relation 1.3, ce qui revient ici à montrer les propriétés 4.5 et 4.6, qui sont des formules d'algèbre linéaire universellement quantifiées, de la forme $\forall V \forall V'\ F$. Il suffit donc de montrer que $\neg F$ est insatisfiable, ce qui est un problème de *SMT-solving*.

Dans tous les cas, on s'est donc ramené au problème de la correction d'un algorithme de *SMT-solving*. Nous nous réjouissons donc des efforts récents visant à modifier des SMT-solvers pour qu'ils produisent des preuves de correction de l'insatisfiabilité de formules [60, 154].

4.2.5 Extensions et perspectives

La méthode présentée au 4.2.2 peut être étendue à une classe plus grande d'invariants. La première est de permettre de considérer des valeurs infinies pour les coordonnées de B et B'. Pour cela, on remplace chaque variable v dans $\mathbb{R} \cup \{+\infty\}$ par une paire de variables $(v_{\mathbb{R}}, v_{\infty})$, la première réelle (donnant la valeur de v si v est finie), l'autre booléenne (vraie si et seulement

si v est infinie). On peut réécrire les opérateurs de comparaison en utilisant ces nouvelles variables. Le défaut de cette technique est qu'elle augmente la complexité booléenne des formules, ce qui peut entraîner des mauvaises performances de l'élimination de quantificateurs.

Nous avons présenté ici la méthode d'abstraction sur une boucle unique, mais on la généralise sans difficulté à un graphe de flot de contrôle arbitraire. Il suffit de sélectionner dans ce graphe des *points de coupure* tels que tout cycle du graphe passe au moins par un point de coupure Gulwani et al. [88, §2] ; aux points de coupure, les valeurs sont abstraites, et la correction de l'invariant s'exprime par la stabilité par tout chemin reliant deux points de coupures sans passer par un autre point de coupure. Une méthode simple est de mettre un point de coupure sur la cible de toute arête arrière dans un parcours du graphe en profondeur d'abord en partant du début du programme ; pour un graphe de flot de contrôle découlant de la compilation d'un programme structuré en blocs, ceci revient à mettre un point de coupure en tête de chaque boucle.[11] Notre cas à une seule boucle est une instance de ce cas général, où l'on choisit la tête de la boucle comme unique point de coupure.

Les calculs en virgule flottante peuvent être analysés avec notre méthode données sur les réels. Il suffit pour cela d'introduire pour chaque calcul flottant une variable supplémentaire $r(x)$, reliée à la valeur exacte de l'opération dans les réels par une relation telle que 3.4. Il s'agit d'une abstraction stricte du problème : nous ne représentons pas exactement le comportement des flottants (voir §3.2.2). Là encore, ces relations compliquent beaucoup la structure booléenne des formules, puisqu'elles introduisent des disjonctions pour chaque calcul arithmétique. Un autre inconvénient est que les fonctions de transfert abstraites optimales par rapport à cette abstraction des flottants peuvent être très compliquées par rapport à celles obtenues en considérant les flottants comme des réels — avec des disjonctions de cas entre des formules différant de quantités minimes. Nous envisageons donc soit de simplifier les fonctions de transfert a posteriori, soit, ce qui serait sans doute préférable, d'intégrer la surapproximation dans le processus de calcul.

La plupart des variables dans les programmes informatiques sont des entiers bornés. Notre technique peut-elle s'appliquer aux entiers relatifs ($\mathbb{Z}$) ? La théorie des inégalités linéaires sur les entiers, aussi appelée arithmétique de Presburger, est décidable et admet l'élimination des quantificateurs. Celle-ci a cependant un coût triplement exponentiel (borne inférieure et supérieure) [201], et est coûteuse en pratique. Nous avons obtenu des résultats intéressants en appliquant notre technique à des problèmes en nombres entiers, en prenant la précaution suivante : quand une contrainte $x < y$ était générée, celle-ci était remplacée par $x \leq y - 1$. Bien entendu, il s'agit ici d'une abs-

[11] Cette méthode n'est cependant pas nécessairement la meilleure pour la précision de l'analyse [88, §2.3]. Bourdoncle [24, Sec. 3.6] a donné différentes méthodes alternatives.

traction, ou *relaxation* imparfaite du problème en nombre entiers — il n'est pas possible de représenter ainsi des contraintes de congruence ($\exists k\ x = 2k$, soit $x \equiv 0 \pmod 2$). Nous avons cependant constaté sur des exemples qu'on peut arriver rapidement à des invariants, par exemple sur la fonction 91 de McCarthy [124].

Nous n'avons pas évoqué les variables de choix discret, les variables booléennes ou énumérées, qui pourtant sont courantes dans les logiciels de contrôle-commande. On peut bien sûr les considérer comme des entiers, et appliquer les abstractions ci-dessus ; mais, sachant que les choix de variables énumérées peuvent être assez arbitraires et indépendants de toute relation numérique, ces abstractions peuvent donner de mauvais résultats. Une technique classique est de rentrer les variables de choix discret dans le flot de contrôle [92], mais, en présence de n booléens, on peut devoir multiplier par 2^n le nombre de points de programme. Bien sûr, une pré-analyse d'accessibilité sur les états discrets peut éviter d'en créer qui soient totalement inutiles [90]. Jeannet [105, 106, 107] a proposé des techniques de *partitionnement dynamique* dans des schémas d'analyse statique par interprétation abstraite « itérative » (calculs d'invariants par élargissements) ; nous ne savons cependant pas comment intégrer ce type de techniques dans un calcul direct d'invariant comme nous avons décrit ici.

Le sujet des relaxations entre flottants et réels, entiers et réels, et des interactions avec les variables de choix discret, devra être exploré plus avant pour obtenir des méthodes plus performantes.

Chapitre 5

Témoins d'insatisfiabilité

La preuve d'une négative ou d'un fait purement négatif est impossible, & conséquemment ne doit point être admise

Encyclopédie ou dictionnaire raisonné des sciences des arts et des métiers, Diderot & d'Alembert, ed., entrée « Preuve »

Negativa non sunt probanda.

5.1 Insatisfiabilité de formules

En matière de vérification de programmes, il arrive fréquemment que l'on doive prouver qu'une formule logique est non satisfiable :

- Montrer que I est un invariant pour la relation τ revient à montrer que $I \wedge \tau \wedge \neg I[x \mapsto x']$ n'a pas de solution (x un état dans l'invariant qui permet de sauter vers un état x' hors invariant).
- Étant donné un invariant I, montrer que celui-ci exclut une condition de panne C revient à montrer que $I \wedge C$ est non satisfiable.

5.1.1 Procédures de décision

Le problème de déterminer si une formule F, bâtie à partir de formules atomiques, de connecteurs $\wedge$ et $\vee$, est satisfiable, peut être ramené au problème de déterminer si une *conjonction* de formules atomiques est satisfiable :

- Soit par mise en forme normale disjonctive par une méthode syntaxique (réécrire $(a \vee b) \wedge c$ en $(a \wedge c) \vee (b \wedge c)$), ce qui est en général très coûteux.
- Soit par des techniques de SMT-solving (voir §4.1.3). Ces techniques font appel en interne à une procédure de décision pour les conjonctions de la théorie logique concernée, laquelle devra idéalement proposer, en sus d'une réponse négative à la satisfiabilité de la conjonction $C_1 \wedge \dots \wedge C_n$, une sous-conjonction contradictoire $C_{i_1} \wedge \dots \wedge C_{i_m}$.

Les procédures de décision sont, en général, des algorithmes assez complexes. On peut donc s'interroger sur la fiabilité d'un système de vérification de logiciel basé sur des procédures de décision dont il faut croire le résultat. C'est d'ailleurs une des objections fréquentes à l'usage des méthodes formelles : que gagne-t-on à faire reposer la preuve de fiabilité d'un logiciel sur un autre logiciel ? À cette question on peut apporter plusieurs réponses :

1. Ces *bugs* de procédures de décision seront probablement assez systématiques, et fourniraient des résultats faux sur plusieurs occurences (par opposition à des résultats faux sur exactement le problème posé par un programme donné). Il suffirait donc de bien tester ces procédures. Cet argument n'est, selon moi, pas suffisamment étayé — il semble que plusieurs outils basés sur des procédures de décision aient contenu des *bugs* qui ont perduré jusqu'à être ce qu'ils aient été mis en évidence.
2. La procédure de décision peut être prouvée correcte à l'aide d'un assistant de preuve tel que Coq. La preuve assistée est une technique puissante, qui a notamment permis une preuve entièrement vérifiée du *théorème des quatre couleurs* en théorie des graphes [83] ; on tente actuellement d'en faire de même à l'égard de la conjoncture de Kepler.[1] Dans les deux cas, il s'agissait de théorèmes dont les preuves publiées reposaient sur de grandes quantités de calculs informatiques, dont il s'agissait de prouver la correction. Clairement, il est donc possible de prouver correctes des procédures de calcul assez compliquées.[2]

   Les problèmes sont d'une part que ces preuves sont très longues à réaliser, d'autre part que les implémentations vérifiées par les assistants de preuve peuvent être inefficaces, notamment en raison des limitations des types de programme que ces assistants considèrent.
3. Dans certains cas (notamment, certains problèmes NP-complets), si la recherche du résultat est malaisée, le résultat est par contre aisément vérifiable. Par exemple, Leroy [117, §4.4], vérifiant la correction d'un compilateur, n'a pas prouvé correcte l'algorithme de coloriage de registre (compliqué), mais seulement la phase de vérification de la correction du coloriage fourni (simple). On obtient ainsi un outil qui peut soit répondre la vérité et fournir un *témoin* de son bon fonctionnement, soit renvoyer un message d'erreur interne.

[1] Thomas Hales, qui a proposé en 1998 une preuve de cette conjecture utilisant de grandes quantités de calculs, anime maintenant le projet *Flyspeck*, qui vise à produire une preuve de ce théorème dans l'assistant HOL light.

[2] Une objection courante à cette approche est que l'on se repose alors sur la correction d'un autre programme, à savoir l'assistant de preuve. Dans le cas de Coq, on se repose essentiellement sur la correction de la procédure de vérification de type pour le Calcul des constructions inductives (CIC), qui est de taille modérée. Une procédure de vérification de type pour le Calcul des constructions (CoC), un sous-ensemble du CIC, a d'ailleurs été prouvée correcte à l'intérieur de Coq [13]. En revanche, dans d'autres assistants de preuve, comme PVS, on doit se reposer sur la correction d'un ensemble assez complexe de procédures de décision.

4. Enfin, même si le résultat final de l'algorithme n'était pas *a priori* vérifiable, on peut transformer l'algorithme ou en choisir un différent afin qu'il produise un témoin. C'est l'approche que j'ai choisie.

5.1.2 Exemples de théories avec témoins d'insatisfiabilité

Inégalités linéaires

Chaque fois que l'on élimine une variable par l'algorithme de Fourier-Motzkin (voir §4.1.1), les inégalités résultantes sont des combinaisons linéaires à coefficients entiers positifs ou nuls des inégalités d'origine. Si l'on élimine toutes les variables $\exists x_1, \dots, x_n\ C$ d'une conjonction C, on obtient une conjonction d'inégalités triviales, à zéro variables, équivalente à $\exists x_1, \dots, x_n\ C$, ces inégalités étant combinaisons linéaires à coefficients positifs de celles de C. Si C est non satisfiable, alors une de ces inégalités va être, à une constante multiplicative près, $-1 \geq 0$ ou $0 > 0$. On obtient ainsi une version du lemme de Farkas :

Lemme 2. *Un système d'inégalités linéaires, strictes ($l > 0$) ou larges ($l \geq 0$), n'a pas de solution si et seulement si il existe une combinaison linéaire à coefficients positifs ou nuls de ces inégalités qui soit trivialement fausse ($-1 \geq 0$ ou $0 > 0$).*

Notons que le vecteur de coefficients ainsi produit est un *témoin* du fait que le système d'origine n'ait pas de solution. Ce témoin permet de vérifier rapidement ce fait sans avoir à relancer un algorithme complexe, et fournit une *preuve* simple d'absence de solutions.

Soit n le nombre d'inégalités et m le nombre de variables, $\alpha_1, \dots, \alpha_n$ les coefficients de la combinaison linéaire. La contrainte que la combinaison donne $-1 \geq 0$ s'exprime par m relations d'égalités, auxquelles on adjoint n relations de la forme $\alpha_i \geq 0$. La contrainte que la combinaison donne $0 > 0$ s'exprime par m relations d'égalités, auxquelles on adjoint n relations de la forme $\alpha_i \geq 0$ et une relation $\sum \alpha_{s_i} = 1$ où les s_i sont les indices des inégalités strictes.

Ainsi, le témoin de l'absence de solutions d'un système d'inégalités linéaires s'exprime lui-même comme la solution d'un système d'inégalités linéaires. La recherche d'un point solution d'un système d'inégalités linéaires est un cas particulier de *programmation linéaire*, problème classique de recherche opérationnelle sur lequel existe une ample littérature [56].

Notons le phénomène de dualité suivant : le fait que le système d'inégalités linéaires primal n'a pas de solution est équivalent au fait qu'un système dual d'inégalités a au moins une solution.

Égalités polynomiales complexes

La géométrie algébrique sur les corps algébriquement clos (par exemple, le corps $\mathbb{C}$ des nombres complexes) est relativement simple. Hilbert a notamment donné une caractérisation algébrique simple de l'ensemble des polynômes (en fait, un idéal de l'anneau des polynômes) s'annulant sur les zéros communs d'une famille de polynômes, le *théorème des zéros de Hilbert* ou *Nullstellensatz*. Nous en donnerons une version simplifiée :

Théorème 3. *Soit une famille $P_1, \dots, P_m$ de polynômes de $\mathbb{C}[X_1, \dots, X_n]$, et soit C tel que $C = 0$ soit une « conséquence géométrique » de $P_1 = 0 \wedge \dots P_m = 0$, autrement dit :*

$$\forall x_1, \dots, x_n \in \mathbb{C}^n \; P_1(x_1, \dots, x_n) = 0 \wedge \dots \wedge P_m(x_1, \dots, x_n) = 0 \implies C(x_1, \dots, x_n) = 0 \quad (5.1)$$

Alors il existe $k \geq 0$ et $Q_1, \dots, Q_m \in \mathbb{C}[X_1, \dots, X_n]$ tels que $C^k = \sum_i Q_i P_i$.

La réciproque de ce théorème est bien sûr triviale quelle que soit le corps. Un cas particulier particulièrement intéressant de ce théorème est avec $C = 1$:

Corollaire 4. *Soit une famille $P_1, \dots, P_m$ de polynômes de $\mathbb{C}[X_1, \dots, X_n]$. Il n'existe aucune solution de $P_1 = 0 \wedge \dots \wedge P_m = 0$ si et seulement s'il existe $Q_1, \dots, Q_m \in \mathbb{C}[X_1, \dots, X_n]$ tels que $\sum_i Q_i P_i = 1$.*

Ce corollaire est faux sur le corps des réels, par exemple $X^2 + 1 = 0$ n'a pas de solution, mais on ne peut obtenir le polynôme 1 comme multiple du polynôme $X^2 + 1$.

Soit $P_1, \dots, P_m \in \mathbb{Q}[X_1, \dots, X_n]$ (on peut remplacer $\mathbb{Q}$ par tout autre sous-corps de $\mathbb{C}$). S'il existe $Q_1, \dots, Q_m \in \mathbb{C}[X_1, \dots, X_n]$ tel que $\sum Q_i P_i = 1$, alors forcément, quitte à projeter, on peut choisir $Q_1, \dots, Q_m \in \mathbb{Q}[X_1, \dots, X_n]$. On montre que de tels $Q_1, \dots, Q_m \in \mathbb{Q}[X_1, \dots, X_n]$ s'obtiennent par réduction de 1 sur une base de Gröbner obtenue à partir de $P_1, \dots, P_m$, procédé algorithmique quoique parfois coûteux [53]. Les $Q_1, \dots, Q_m$ forment donc un témoin d'insatisfiabilité.

Notons qu'un témoin de réfutation fourni par l'algorithme ci-dessus est *correct* quel que soit le sous-corps de $\mathbb{C}$ utilisé : s'il fournit un témoin, alors celui-ci assure que le système original n'a pas de solutions. En revanche, l'algorithme n'est *complet* sur si le corps est algébriquement clos.

5.2 Cas des inégalités polynomiales réelles

Harrison [95], Parrilo [164] ont suggéré de rechercher des témoins d'incompatibilité dont l'existence est garantie par un théorème de géométrie

algébrique réelle, le *real Nullstellensatz* [192]. Ces témoins sont des sommes de carrés de polynômes, obtenus comme solutions de problèmes de *programmation semidéfinie.*

5.2.1 Élimination de quantificateurs et décision

La geométrie algébrique réelle est beaucoup moins maniable que la géométrie algébrique complexe. En sus des ensembles algébriques (lieu d'annulation des polynômes), on doit étudier les ensembles semi-algébriques, c'est-à-dire ceux que l'on peut former par combinaison d'inégalités polynomiales :

Définition 5. *L'ensemble des ensembles semi-algébriques de $\mathbb{R}^n$ est le plus petit ensemble clos par union et intersection finies et contenant les ensembles de solutions de systèmes d'inégalités de la forme $P(x_1, \dots, x_n) \bowtie 0$ où $\bowtie$ est un opérateur de comparaison $<, >, \neq, \leq, \geq, =$.*

Il est clair que cet ensemble est stable par union, intersection, et passage au complémentaire, mais il est également stable par projection — c'est le théorème de Tarski-Seidenberg. En d'autres termes, la théorie des inégalités polynomiales sur les réels admet l'élimination des quantificateurs, la projection correspondant à l'élimination des quantificateurs existentiels : à chaque formule quantifiée on peut associer au moins une formule équivalente sans quantificateurs.

Il existe plusieurs algorithmes calculant effectivement une telle formule équivalente. Les premiers algorithmes [187, 196] avaient une complexité prohibitive, non élémentaire [122], alors que de meilleurs algorithmes ne seraient qu'en double exponentielle « bien parallélisable » [97], laquelle est d'ailleurs une borne inférieure de complexité [17, §11.4][32]. Basu et al. [17, ch. 14] proposent un algorithme dont la complexité est doublement exponentielle, mais seulement simplement exponentielle en la taille de la formule une fois que l'on a fixé la structure des quantificateurs.

L'algorithme le plus utilisé en pratique est la décomposition cylindrique algébrique [40][17, ch. 11]. À notre connaissance, seuls deux logiciels l'implémentent : QepCad et Mathematica. Notre expérience avec ces deux logiciels que le temps et l'espace nécessaires pour les calculs sont vite prohibitifs.

Tout algorithme d'élimination des quantificateurs peut être utilisé pour le test de satisfiabilité : il suffit de quantifier existentiellement toutes les variables de la formule et de lui passer en paramètre la formule ainsi quantifié, et l'algorithme va produire une formule sans variables trivialement décidable (en pratique, l'algorithme produira directement « vrai » ou « faux »). Il est cependant probable que la décomposition cylindrique algébrique n'est pas la meilleure méthode pratique pour le test de satisfiabilité ; Basu et al. [17, ch. 13] donnent un algorithme simple exponentiel en le nombre de variables, dont nous ne connaissons malheureusement aucune implémentation.

5.2.2 Incompatibilités algébriques

Il existe pour la géométrie algébrique réelle l'analogue du *Nullstellensatz* ou du lemme de Farkas, je veux dire par là un théorème qui caractérise précisément l'ensemble des conséquences d'un système d'inégalités et d'égalités polynomiales. Stengle [192] a donné des théorèmes caractérisant les conséquences de systèmes d'inégalités larges, puis larges et strictes (*real Nullstellensatz* et *Positivstellensatz*). Nous reprendrons ici la formulation de ces théorèmes de Lombardi [122].

K désigne un corps ordonné, X un ensemble fini de variables $K[X_1, \ldots, X_n]$, $K[X]$ désigne l'anneau de polynômes $K[X_1, \ldots, X_n]$. Nous noterons :

- $M(F)$ le monoïde multiplicatif engendré par F (c'est-à-dire l'ensemble des produits de 0 ou plus éléments de F),
- $M_1(F)$ est l'ensemble des produits d'éléments de F où chaque élément intervient au plus une fois,
- $C(F)$ le cône positif engendré par F est l'ensemble des sommes d'éléments de la forme $p.P.Q^2$ où $p \geq 0$ dans K, $P \in M(F)$ et $Q \in K[X]$. Remarquons qu'il est équivalent de supposer $P \in M_1(F)$, quitte à passer les facteurs carrés dans Q,
- $I(F)$ l'idéal engendré par F.

Définition 6. *Étant données 4 parties finies de $K[X]$: $F_>$, $F_\geq$, $F_=$, $F_\neq$, contenant des polynômes auxquels on souhaite imposer respectivement les conditions de signes > 0, ≥ 0, $= 0$, $\neq 0$, on dira que $F = [F_>; F_\geq; F_=; F_\neq]$ est fortement incompatible dans K si on a une égalité dans $K[X]$ du type suivant : $S+P+Z = 0$ avec $S \in M(F_> \cup F^{*2})$, $P \in C(F_\geq \cup F_>)$, $Z \in I(F_=)$.*

Toute incompatibilité forte écrite sous la forme ci-dessus peut être ramenée à un incompatibilité forte écrite sous la forme suivante : $S + P + Z = 0$ avec $S \in M(F_>^{*2} \cup F_\neq^{*2})$, $P \in C(F_\geq \cup F_>)$, $Z \in I(F_=)$.

Le *real Nullstellensatz* établit l'équivalence entre :

1. L'incompatibilité forte décrite ci-dessus.
2. L'absence de solutions réelles.
3. L'absence de solutions dans toutes les extensions ordonnées de K.

Là encore, une des directions est triviale (l'incompatibilité forte implique l'absence de solutions).

Si l'on ne considère que des inégalités larges $F_\geq = \{L_1, \ldots, L_n\}$, la condition devient

$$\sum_{S \subseteq \{1,\ldots,n\}} Q_S \left(\prod \{s \mid s \in S\}\right) + 1 = 0 \tag{5.2}$$

où les Q_S sont des combinaisons linéaires à coefficients positifs de carrés de polynômes dans K.

5.2.3 Sommes de carrés

Définition 7. *Une matrice symétrique M est dite* définie positive *(noté $M \succ 0$) si pour tout vecteur x non nul, $x^t M x > 0$, ou si, de façon équivalente, toutes les valeurs propres de M sont strictement positives. On parle de matrice* semidéfinie positive *(noté $M \succeq 0$) si pour tout vecteur x, $x^t M x \geq 0$, ou si, de façon équivalente, toutes les valeurs propres de M sont positives ou nulles.*

Lemme 8. *Soit S une combinaison linéaire à coefficients positifs de carrés de polynômes pris dans le sous-espace vectoriel de $K[X]$ engendré par une famille finie $m_1, \ldots, m_d$. Alors il existe une matrice symétrique $n \times n$ semidéfinie positive Q à coefficients dans K telle que $S = [m_1, \ldots, m_d] Q [m_1, \ldots, m_d]^t$. Réciproquement, s'il existe une telle matrice, alors S est une telle combinaison.*

Reprenons l'équation 5.2. Supposons que les Q_S sont des combinaisons linéaires à coefficients positifs de carrés de polynômes, ces polynômes étant eux-mêmes combinaisons linéaires d'une famille finie $m_1 \ldots, m_d$ (par exemple, leurs monômes). L'équation se réécrit alors :

$$\sum_{S \subseteq \{1,\ldots,n\}} ([m_1, \ldots, m_d] Q_S [m_1, \ldots, m_d]^t) \left(\prod \{s \mid s \in S\} \right) + 1 = 0 \qquad (5.3)$$

Cette équation entre polynômes peut être projetée sur la base des monômes, fournissant un système d'équations scalaires sur K dont les inconnues sont les coefficients des matrices Q_S. Une solution de ce système est un 2^n-uplet de matrices symétriques, indexées par $S \subseteq \{1, \ldots, n\}$; chacune de ces matrices doit être semidéfinie positive. Si nous posons Q la matrice diagonale par blocs formée des Q_S, l'ensemble des solutions de l'égalité est une variété linéaire affine de la forme $-W_0 + \sum_{i \geq 1} \lambda_i W_i$ où les W_i sont des matrices symétriques, $-W_0$ étant une solution particulière. Nous recherchons donc des λ_i tels que $-W_0 + \sum_{i \geq 1} \lambda_i W_i \succeq 0$.

Ce problème de recherche d'une matrice semidéfinie positive au sein d'une variété linéaire affine de matrices symétriques est un cas particulier de *programmation semidéfinie* (le problème général demandant également l'optimisation d'une forme linéaire, ce qui ne nous intéresse pas).

Nous avons ainsi réduit le problème de la recherche du témoin d'insatisfiabilité à celle d'une solution quelconque d'un problème de programmation semidéfinie. Cette réduction est cependant imparfaite :

– Il faut se fixer a priori une base de monômes et chercher une solution par rapport à cette base. Cela revient à se donner a priori une borne sur les degrés des polynômes apparaissant dans les sommes de carrés. Or, on ne connaît pas de bornes « pratiques » sur les degrés des polynômes nécessaires — les bornes connues sont non élémentaires donc

inutilisables [122]. Il faut donc restreindre arbitrairement l'espace de recherche.
- Le nombre de termes dans la combinaison linéaire 5.2 est 2^n où n est le nombre d'inégalités considérées. On peut éventuellement là encore restreindre arbitrairement l'espace de recherche, en se restreignant par exemple à des S de cardinal inférieur à une certaine limite k, mais cette solution est peu satisfaisante, comme on le voit bien sur l'exemple suivant :

$$\begin{cases} x-1 & \geq 0 \\ y-3 & \geq 0 \\ z-5 & \geq 0 \\ -xyz+14 & \geq 0 \end{cases}$$

Il est clair que pour obtenir une incompatibilité forte à partir de ces 4 équations, il va falloir considérer le produit des 3 premiers polynômes. Plus généralement, on peut exhiber des systèmes « simples » de n inégalités où il faut prendre le produit de $n-1$ d'entre elles...

5.2.4 Programmation semidéfinie

Soient $W_0, W_1, \ldots, W_p$ des matrices symétriques. On s'intéresse à l'ensemble S des vecteurs $\lambda_1 \ldots, \lambda_p$ tels que $-W_0 + \sum_{i=1}^{p} \lambda_i W_i \succeq 0$. Cet ensemble est :

- *convexe*, car intersection des demi-espaces définis par $-x^t W_0 x + \sum_{i=1}^{p} \lambda_i x^t W_i x \geq 0$ pour tout vecteur x ;
- *semialgébrique*, car l'ensemble des matrices symétriques semidéfinies positives est semialgébrique (il est défini par les signes des coefficients du polynôme caractéristique) ; on a donc là encore un phénomène de dualité où l'existence de solution d'un problème semialgébrique dérivé implique l'absence de solution du problème d'origine.

Son intérieur (au sens topologique) est l'ensemble des $\lambda_1 \ldots, \lambda_p$ tels que $-W_0 + \sum_{i=1}^{p} \lambda_i W_i \succ 0$.

Notre problème est de trouver un point dans l'ensemble des solutions S. On pourrait bien sûr utiliser des méthodes algébriques, mais cela risque d'être coûteux étant données les dimensions du problème. Il existe par contre des méthodes numériques, notamment basées sur un calcul de *points intérieur* [197]. Ces méthodes considèrent un problème primal et un problème dual, et leur bon fonctionnement est démontré si les deux ensembles de solution sont non vides. Il n'est donc pas surprenant qu'expérimentalement il y ait des problèmes dans des cas dégénérés où S est d'intérieur vide.

Lorsque l'intérieur de S est non vide, il existe autour de tout point solution $\lambda_1, \ldots, \lambda_p$ une boule de solutions.[3] Si l'on ne commet pas une erreur

[3] Les méthodes de recherche par points intérieurs ont tendance à rechercher un certain « centre » de l'espace de solutions [197], et fournissent donc naturellement des points

trop grosse en approchant la solution numérique $\lambda_1, \dots, \lambda_p$ par des rationnels, cette valeur en rationnels sera également solution, ce qui conclut notre recherche.

L'intérieur de S est vide si et seulement si la variété linéaire affine engendrée n'est pas tout l'espace ; autrement dit, S est inclu dans un hyperplan affine ou dans une variété linéaire affine de dimension inférieure (qui est sa clôture de Zariski). On comprend qu'il est difficile de tomber numériquement pile sur une surface infiniment fine de l'espace. Aussi, même si la méthode numérique converge (ce qui n'arrive pas toujours), elle le fait vers une matrice seulement « presque » semidéfinie positive, typiquement une matrice dont toutes les valeurs propres sont positives sauf une qui est de l'ordre de -10^{-8}.

Il est sans doute exclu de calculer à l'avance, par des méthodes algébriques générales, la clôture de Zariski de S : ces méthodes seront en général au moins aussi compliquées que de chercher un point dans S, ce qui permettrait de conclure le problème d'origine. Le problème reste donc entier.

éloignés du bord.

Chapitre 6

Conclusion et perspectives

En effet, comment les conséquences pourroient-elles être claires & certaines, si les principes étoient obscurs ?]

Encyclopédie ou dictionnaire raisonné des sciences des arts et des métiers, Diderot & d'Alembert, ed., entrée « Élémens des sciences »

Il y a encore 10 ans, on disait volontiers que l'analyse statique précise et sûre ne passerait pas à l'échelle, et serait donc réservée à des programmes de taille modeste. Weise [199] expliquait que seules des analyses assez imprécises pourraient passer à l'échelle. Plus récemment, quand Patrick Cousot avait soumis le projet ASTRÉE pour un financement, on lui avait tout d'abord répondu que ses objectifs étaient irréalistes. On a vu au contraire qu'il était possible d'obtenir des résultats de haute précision (pas ou très peu de fausse alarme) sur des programmes d'assez grande taille, à condition de restreindre la classe des programmes analysés (notamment, en excluant les manipulations trop compliquées de la mémoire). En revanche, nous reprenons certaines des conclusions de Weise, notamment le fait que lorsqu'on parle de programmes de grande taille, des analyses « peu coûteuses, car en temps polynomial » sont en fait de coût prohibitif, que seules des analyses quasi-linéaires passent à l'échelle, et qu'il faut faire attention aux structures et aux algorithmes qu'on utilise (§2.2.1).

Le succès d'outils comme ASTRÉE, SLAM,[1] ou AIT[2] a permis de démontrer que l'analyse statique n'était pas un jouet universitaire inapplicable en pratique, mais bien une technologie qui permet de répondre à de vrais défis

[1] SLAM est un outil d'analyse statique basé sur l'abstraction de prédicats, produit par la division de la recherche de Microsoft et depuis industrialisé en tant que *Static Driver Verifier* au sein du *Visa Windows Driver Kit*. `http://research.microsoft.com/en-us/projects/slam/`

[2] AIT, de la société AbsInt, est un outil calculant une surapproximation garantie du temps d'exécution dans le pire cas d'un programme sur une architecture donnée, en tenant compte des effets du cache et du pipe-line du processeur. `http://www.absint.com/ait/`

industriels. D'autres outils d'analyse, certes parfois moins sémantiquement sûrs, ont montré qu'ils pouvaient contribuer à chercher des bugs dans des systèmes d'exploitation et des programmes — citons par exemple les travaux d'Engler et al. [68] et leurs débouchés industriels dans l'entreprise COVERITY —, voire à les éviter en simplifiant leur maintenance — citons par exemple COCCINELLE.[3]

Il n'est donc pas surprenant que, dans certains secteurs industriels du moins, on appelle à passer d'une assurance-qualité basée sur les processus, c'est-à-dire le respect de normes et de processus de programmation censés garantir la qualité du produit fini, à une assurance-qualité basée sur l'analyse directe du produit fini [104].

Cependant, les techniques actuellement connues sont encore insuffisantes pour une utilisation plus large de l'analyse statique sémantiquement correcte. Voyons quelques limitations et les perspectives correspondantes :[4]

1. L'usage de langages de bas niveau tels que C, et notamment d'« astuces » de programmation que ce langage permet (transtypages sauvages via les pointeurs, arithmétique de pointeurs, etc.), complique considérablement la tâche de l'analyse. Nous pouvons espérer d'une part l'amélioration des méthodes d'analyse sur les modèles mémoire à arithmétique de pointeurs, soit l'usage de langages de plus haut niveau. Sur ce dernier point, nous constatons deux directions :
   - L'usage de langages à typage fort (comme CAML ou F#) ou à typage semi-fort (comme JAVA ou C#) évite largement ces problèmes, bien que l'analyse de ces langages, par exemple pour démontrer l'absence de lancement de certaines exceptions, n'est pas triviale. On peut bien sûr objecter que ces langages, impliquant des environnements d'exécution assez complexes (*garbage collection* voire compilation juste-à-temps), sont inadaptés pour les applications embarquées critiques.
   - Les programmes de bas niveau peuvent être générés automatiquement à partir d'une spécification de haut niveau, par exemple en SCADE, SIMULINK, STATEFLOW etc. On parle alors souvent de *model-based development*.

     Pour autant que le langage de haut niveau a une sémantique précise (ce qui n'est pas le cas avec certains langages industriels), des propriétés fonctionnelles (par exemple, que tel booléen n'est jamais vrai

[3] COCCINELLE permet de simplifier la maintenance de logiciels complexes tels que le noyau de système d'exploitation LINUX à l'aide de « *patches* sémantiques » : une modification est rarement isolée et doit souvent s'accompagner de petites modifications ailleurs dans le code, que ce projet se propose de largement automatiser [33]. `http://www.emn.fr/x-info/coccinelle/`

[4] Mon programme de recherche au cours des prochaines années concernera notamment l'analyse modulaire de propriétés mêlant discret et numérique, et l'utilisation de techniques de recherche sous contrainte pour la recherche d'invariants.

si telle variable est supérieure à telle borne) peuvent être démontrées plus facilement sur ces spécifications que sur le code bas niveau. On objectera que des différences entre ce qui est écrit dans la spécification et ce qui est réellement exécuté peuvent être introduites par le compilateur, mais il existe des projets de compilateurs certifiés, par exemple celui de Marc Pouzet.

2. Les analyses réalisées par Astrée sont monolithiques : si on change une ligne dans le code, on doit refaire toute l'analyse. On aimerait plutôt une analyse *modulaire*, qui permettrait d'obtenir des informations sur des modules indépendamment les uns des autres, et *compositionnelle*, c'est-à-dire que l'information sur un module composé de sous-modules pourrait s'obtenir par composition des informations analysées sur les sous-modules. Il est probable qu'il soit très utile d'exploiter les divisions et hiérarchies naturellement introduites au niveau du langage de programmation, par exemple l'encapsulation fournie par les langages à objets, ou encore le découpage en nœuds hiérarchisés et composables fournie par les langages comme Lustre.
3. La technique de recherche d'invariants par itérations (éventuellement avec élargissement) jusqu'à obtention d'un post-point fixe est à la fois souvent imprécise, en raison de l'élargissement, et coûteuse, en raison du nombre parfois élevé d'itérations nécessaires. Différentes directions sont explorées : réduction à un problème de recherche de solution ou d'optimisation d'un problème numérique [41, 47, 181], booléen [89] ou mixte [88], itération de politique [45]...
4. L'analyse de programmes parallèles communiquant par mémoire partagée est un sujet très difficile. La combinatoire de l'analyse est défavorable. Peut-être, là encore, l'erreur est de vouloir à toute force analyser un programme de bas niveau, manipulant directement des primitives de synchronisation ou d'exclusion mutuelle, et dans lequel il est difficile de retrouver les invariants qui garantissent la correction, au lieu de synthétiser l'usage de ces primitives depuis des définitions de haut niveau.

   L'inclusion dans des langages largement utilisés de primitives « syntaxiques » d'exclusion mutuelle (par exemple le `synchronized` de Java) a déjà permis de simplifier ce problème : par exemple, dans la plupart des cas, le verrou qui sert à garantir l'exclusion mutuelle de l'accès à un objet est justement celui associé à cet objet par le langage. Cependant, cela ne supprime pas le problème de l'explosion combinatoire exponentielle en le nombre de fils d'exécution.

   Il est possible que, là encore, l'usage de primitives de plus haut niveau, telles que les transactions,[5] permette de diminuer les difficultés d'écriture et d'analyse des programmes.

[5] Ce concept de *transaction* est repris des bases de données. Une transaction est une

Plus généralement, nous pouvons nous interroger sur le futur de l'analyse de programme et, plus généralement, de systèmes, au delà des points purement techniques. Sur ce point, j'ai fait des constats tant positifs que négatifs. Le fait que des industriels reconnus de l'aéronautique comme Airbus ou Rockwell-Collins, ou un grand industriel du logiciel grand public et d'entreprise comme Microsoft, aient déployé à l'échelle industrielle ce type de techniques nous laisse espérer une plus large diffusion. En revanche, il est clair que les techniques permettant de prouver l'absence d'erreur à l'exécution n'intéressent que les domaines les plus critiques de l'industrie, ceux où une panne à des conséquences dramatiques, humaines, commerciales ou financières. Or, la majorité des produits informatisés actuellement vendus ont des pannes dues à des bugs sans que cela n'ait de conséquences graves pour leurs fournisseurs : ainsi, chacun est habitué à ce que son micro-ordinateur ait des dysfonctionnements, ou que son baladeur MP3 ou son téléphone portable aient parfois des ratés.[6] On peut donc craindre que ces techniques ne soient réservées à un marché « de niche ».

En ce qui concerne les systèmes embarqués, l'augmentation de la puissance de calcul des processeurs a permis de se passer pour l'essentiel de la programmation en assembleur, au profit notamment du langage C (ou d'autres langages comme Ada). Ces langages sont cependant encore très bas niveau, et on propose donc souvent l'utilisation de langage de plus haut niveau. Le langage C++, mêlant constructions de très haut niveau (objets à méthodes virtuelles et héritage multiple, modèles, surcharge d'opérateurs etc.) et de bas niveau (gestion manuelle de la mémoire, etc.) est redoutable par la facilité qu'il donne au programmeur de faire des erreurs difficiles à détecter. On peut en grande partie éviter ces problèmes avec des langages empêchant la corruption de la mémoire par le typage statique ou dynamique et un système de gestion de la mémoire, mais ceci implique en général un système de support de l'exécution lourd et complexe, dont on peut raisonnablement se demander comment l'utiliser dans un système critique. Citons, parmi les difficultés, le problème encore d'actualité d'utiliser la *garbage collection* dans un système temps réel — sans aller jusqu'à stopper tout le système pendant leur exécution comme les *garbage collectors* d'il y a 25 ans, la plupart des *garbage collectors* introduisent des délais assez imprévisibles

suite d'actions sur la mémoire qui, du point de vue du processus qui effectue la transaction, est atomique. Une transaction peut échouer, et dans ce cas l'état de la mémoire n'est finalement pas modifié, ou peut devoir être recommencée en cas d'accès concurrents. Le lecteur pourra par exemple se reporter à l'ouvrage de Larus and Rawjar [116] qui résume l'état de l'art en la matière.

[6] À un moment au cours de la rédaction de ce mémoire, mon téléphone portable a décidé d'afficher son écran décalé vers le bas d'un certain nombre de pixels, les pixels disparus par le bas réapparaissant par le dessus. J'avoue une certaine inquiétude quand j'apprends que l'on propose de stocker des moyens de paiement dans ce type d'appareil...

dans l'exécution des programmes.[7]

Une autre source d'inquiétude est la complexité croissante des architectures matérielles. Nous sommes passés en vingt ans de processeurs au comportement plutôt prévisible (disons, le Motorola 68000) à des processeurs munis de caches, de pipe-lines, d'exécution spéculative. Maintenant que les problèmes de dissipation d'énergie limitent la vitesse d'horloge des processeur, on propose, dans des ordinateurs grand public, des processeurs multi-cœurs, qui imposent, pour être exploités au mieux de leur capacité, la programmation parallèle. D'une part, la sémantique des séquencements d'opération à bas niveau sur ces architectures est très complexe et mal documentée [183], d'autre part il est difficile de programmer en parallèle sans se tromper, notamment en mémoire partagée. Là encore, on peut se demander s'il est raisonnable d'utiliser des systèmes aussi complexes dans des applications critiques.

Peut-être des changements sont-ils nécessaires dans les processus industriels de conception des logiciels de contrôle des systèmes embarqués. Mon impression, suite à des contacts avec différents industriels, est que ceux-ci considèrent souvent que l'analyse statique s'applique en bout de chaîne de conception, une fois le système totalement réalisé, comme une sorte de « coup de tampon » final — en quelque sorte en supplément des tests d'intégration. Par ailleurs, les industriels séparent souvent les équipes qui pratique ces tests d'intégration des équipes qui conçoivent le logiciel, et ce avec raison : il importe que les présupposés et préjugés des concepteurs du logiciel n'influencent pas les testeurs.[8] Cette séparation est non seulement organisationnelle (services différents) mais aussi intellectuelle : les testeurs n'ont pas accès aux documents expliquant le fonctionnement interne des applications à tester, mais seulement à des spécifications d'assez haut niveau.

Si cette division se justifie en ce qui concerne le test, elle est moins pertinente en ce qui concerne la vérification à l'aide de méthodes formelles. En effet, un outil de vérification formelle ne fait *a priori* confiance à rien et ne se laissera pas induire en erreur par les préjugés des personnes qui l'utilisent.[9]

[7] Le système d'exécution pour Java Metronome, d'IBM [6], inclut un *garbage collector* dont le coût en temps est limité et borné. Ce type de technologie n'est toutefois pas encore répandu.

[8] En effet, le concepteur d'un logiciel finit souvent par confondre ce qu'il pense que son logiciel fait, d'une part, et ce qu'il fait vraiment, d'autre part. Année après année, je rencontre des étudiants qui me justifient qu'une fonction calcule vraiment ce qu'elle doit calculer en faisant l'hypothèse qu'une autre fonction est correcte, sans avoir testé ou vérifié cette dernière, et qui me soutiennent mordicus son bon fonctionnement. Il est donc important d'avoir un regard externe — ici, celui de l'enseignant.

[9] À moins que celles-ci ne décident d'ajouter en axiomes du système des propriétés qu'elles pensent vraies, mais qui ne le sont pas. Par exemple, avec les systèmes de preuves déductives, quand une preuve d'un « résultat évident » devient trop laborieuse, il est tentant de déclarer ce résultat comme axiome. Nous avons entendu des rumeurs selon lesquelles certains projets industriels faisant usage de preuve formelle ont parfois procédé ainsi...

Il n'y a donc pas d'inconvénient à ce que les informations de conception précises soient transmises, par exemple les invariants ou propriétés attendus. *A contrario*, il serait même avantageux que ceux-ci soient connus de l'outil de vérification :

- Il est en général moins difficile de prouver qu'un invariant donné est vrai plutôt que de rechercher un invariant correct.
- Il est intéressant, du point de vue du processus de développement, de vérifier que si des propriétés « locales » attendues sont vraiment vérifiées par le système. En effet, quand un outil automatique n'arrive pas à démontrer une propriété globale, il est souvent difficile de déterminer s'il s'agit d'une insuffisance de l'outil (abstraction trop grossière, incomplétude de la méthode de preuve...), d'hypothèses trop larges, ou d'un vrai problème dans le programme. Bien sûr, la solution la plus confortable est que l'outil essaye de produire une trace de contre-exemple, mais il est également appréciable de pouvoir contrôler que des propriétés intermédiaires sont vérifiées.

Par ailleurs, cette isolation entre les concepteurs et les « testeurs » responsables de l'utilisation de l'analyse statique, complique la tâche des outils parce que les logiciels sont très rarement écrits dans l'optique d'une vérification aisée. Nous avons listé plus haut certaines fonctionnalités de langages industriels (comme C) qui contribuent à rendre difficile les analyses (par exemple, arithmétique de pointeurs). Si l'analyse arrive très tard dans le processus industriel, comme une arrière-pensée, il est devenu impossible de changer le logiciel, sauf à trouver une erreur grave ; en tout cas, on ne peut demander de changements destinés à la faciliter. Parfois même, l'utilisation de l'analyse arrive tellement tard dans le processus de développement et de maintenance, que les concepteurs originaux ne sont déjà plus là, et qu'il est impossible d'obtenir des informations, même informelles, sur les raisons pour lesquelles tel ou tel programme fonctionne. Il va de soit que dans ce cas il est en général vain d'attendre qu'un outil automatisé retrouve ces raisons.

Nous pensons donc que pour que l'analyse statique prenne toute son utilité industrielle, il est nécessaire qu'en sus d'améliorations techniques (analyse modulaire, etc.), elle soit mieux intégrée au processus industriel :

- Les logiciels devraient être conçus avec l'analyse statique en vue : les constructions difficiles à analyser et non indispensables devraient être écartées.
- L'analyse devrait être utilisée tout au long du processus de conception et non pas seulement sur un produit « fini ».
- L'analyse devrait être intégrée dans les environnements de développement.

Bibliographie

[1] Assalé Adjé, Stéphane Gaubert, et Éric Goubault. Computing the smallest fixed point of nonexpansive mappings arising in game theory and static analysis of programs. preprint, arXiv :0806.1160v2, 2008. (Web).

[2] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6) :509–516, June 1978. ISSN 0018-9340. doi : 10.1109/TC.1978.1675141.

[3] Arden Albee, Steven Battel, Richard Brace, Garry Burdisk, Peter Burr, John Casani, Duane Dipprey, Jeffrey Lavell, Charles Leising, Duncan MacPherson, Wesley Menard, Richard Rose, Robert Sackheim, Al Schallenmuller, et Charles Whetsel. Report on the loss of the Mars Polar Lander and Deep Space 2 missions. Technical Report JPL D-18709, Jet Propulsion Laboratory, California Institute of Technology, March 2000.

[4] Karl Johan Åström et Björn Wittenmark. *Computer-controlled systems*. Prentice-Hall, 1997. ISBN 0-13-314899-8.

[5] ASW. Thrice almighty — the virtues of triple redundancy. Air Safety Week, April 4 2005. (Web).

[6] David F. Bacon, Perry Cheng, et V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Symposium on Principles of programming languages (POPL)*, pages 285–298. ACM, 2003. ISBN 1-58113-628-5. doi : 10.1145/604131.604155. (PDF).

[7] Roberto Bagnara, Patricia M. Hill, et Enea Zaffanella. *The Parma Polyhedra Library, version 0.9.* (Web).

[8] Gogul Balakrishnan. *WYSINWYX : What You See Is Not What You eXecute*. Thèse de doctorat, University of Wisconsin, Madison, Wisconsin, USA, August 2007. (PDF). Available as computer science TR-1603.

[9] Gogul Balakrishnan et Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, éd. sci., *Compiler Construction (CC)*, volume 2985 de *LNCS*, pages 5–23. Springer, 2004. ISBN 3-540-21297-3. doi : 10.1007/b95956.

[10] Gogul Balakrishnan, Thomas Reps, David Melski, et Tim Teitelbaum. WYSINWYX : What You See Is Not What You eXecute. In *Verified Software : Theories, Tools, Experiments (VSTTE)*, volume 4171 de *LNCS*, pages 202–213. Springer, 2008. ISBN 978-3-540-69147-1. doi : 10.1007/978-3-540-69149-5. (PDF).

[11] Thomas Ball et Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN (Workshop on Model Checking of Software)*, volume 2057 de *LNCS*, pages 103–122. Springer, 2001. ISBN 3-540-42124-6. doi : 10.1007/3-540-45139-0_7.

[12] J. J. Barnett et al. Report on the incident to Airbus A340-642, registration G-VATL en-route from Hong Kong to London Heathrow on 8 February 2005. Technical Report 4/2007, Air Accidents Investigation Branch, 2007. (Web).

[13] Bruno Barras. Coq en Coq. Rapport de Recherche 3026, INRIA, October 1996. (Web). (PDF).

[14] Clark W. Barrett, David L. Dill, et Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC (Design Automation Conference)*, pages 522–527, 1998. ISBN 0-89791-964-5. doi : 10.1145/277044.277186. (Postscript).

[15] Gregg F. Bartley. Boeing B-777 : Fly-by-wire flight controls. In Spitzer [190], chapitre 11. ISBN 084938348X.

[16] Saugata Basu, Richard Pollack, et Marie-Françoise Roy. On the combinatorial and algebraic complexity of quantifier elimination. *Journal of the ACM (JACM)*, 43(6) :1002–1045, 1996. ISSN 0004-5411. doi : 10.1145/235809.235813.

[17] Saugata Basu, Richard Pollack, et Marie-Fran coise Roy. *Algorithms in real algebraic geometry*. Algorithms and computation in mathematics. Springer, deuxième édition, 2006. ISBN 3-540-33098-4. (Web).

[18] Yves Bertot et Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004. ISBN 3-540-20854-2.

[19] Michael Blair, Sally Obenski, et Paula Bridickas. Patriot missile defense, software problem led to system failure at Dharhan, Saudi Arabia. Technical Report IMTEC-92-26, US Government Accountability Office (GAO), February 1992. (Web). (PDF).

[20] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, et Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In Torben Æ. Mogensen, David A. Schmidt, et I. Hal Sudborough, éds. sci., *The Essence of Computation : Complexity, Analysis, Transformation*, number 2566 in LNCS, pages 85–108. Springer, 2002. ISBN 3-540-00326-6. doi : 10.1007/3-540-36377-7_5. (PDF). (Postscript).

[21] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, et Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, 2003. ISBN 1-58113-662-5. doi : 10.1145/781131.781153. (PDF). (Postscript).

[22] Sylvie Boldo et Guillaume Melquiond. Emulation of FMA and correctly-rounded sums : Proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4) :462–471, 2008. ISSN 0018-9340. doi : 10.1109/TC.2007.70819. (Web). (PDF).

[23] Cari Borrás. Overexposure of radiation therapy patients in Panama : problem recognition and follow-up measures. *Pan-American J. of public health*, 20(2/3) :173–187, September 2006. ISSN 1020-4989. (Web). (PDF). (revue également connue comme *Revista panamericana de salud pública*).

[24] François Bourdoncle. *Sémantique des langages impératifs d'ordre supérieur et interprétation abstraite.* Thèse de doctorat, École polytechnique, Palaiseau, 1992. (PDF).

[25] Aaron R. Bradley et Zohar Manna. *The Calculus of Computation : Decision Procedures with Applications to Verification.* Springer, October 2007. ISBN 3-540-74112-7.

[26] Guillaume P. Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Michael R. Lowry, Corina S. Pasareanu, Arnaud Venet, Willem Visser, et Richard Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2-3) :167–198, 2004. ISSN 0925-9856. doi : 10.1023/B :FORM.0000040027.28662.a4. (PDF).

[27] Dominique Brière et Pascal Traverse. Airbus A320/A330/A340 electrical flight controls — a family of fault-tolerant systems. In *FTCS-23 (Symposium on Fault-Tolerant Computing)*, pages 616–623. IEEE, June 1993. ISBN 0-8186-3680-7. doi : 10.1109/FTCS.1993.627364.

[28] Dominique Brière, Christian Favre, et Pascal Traverse. Electrical flight controls, from Airbus A320/330/340 to future military transport aircraft : A family of fault-tolerant systems. In Spitzer [190], chapitre 12. ISBN 084938348X. Aussi dans [29].

[29] Dominique Brière, Christian Favre, et Pascal Traverse. Electrical flight controls, from Airbus A320/330/340 to future military transport aircraft : A family of fault-tolerant systems. In Cary R. Spitzer, éd. sci., *Avionics : Elements, Software and Functions (Avionics Handbook)*. CRC Press, 2007. ISBN 0-8493-8438-9.

[30] Susan Brilliant, John Knight, et Nancy Leveson. Analysis of faults in an N-version software experiment. *IEEE Trans. on Software Engineering*, SE-16(2) :238–247, February 1990. ISSN 0098-5589. doi : 10.1109/32.44387. (PDF).

[31] Frederick Philipps Brooks. *The Mythical Man-Month : Essays on Software Engineering, Anniversary Edition.* Addison-Wesley, deuxième édition, 1995. ISBN 0-201-83595-9.

[32] Christopher W. Brown et James H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *ISSAC (Symposium on Symbolic and algebraic computation)*, pages 54–60. ACM, 2007. ISBN 978-1-59593-743-8. doi : 10.1145/1277548.1277557.

[33] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, et Gilles Muller. A foundation for flow-based program matching : using temporal logic and model checking. In Pierce [167], pages 114–126. ISBN 978-1-60558-379-2. doi : 10.1145/1480881.1480897. (Web). (PDF).

[34] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, et John A. Plaice. LUSTRE : a declarative language for real-time programming. In *POPL (Symposium on Principles of programming languages)*, pages 178–188. ACM, 1987. ISBN 0-89791-215-2. doi : 10.1145/41625.41641.

[35] Bob F. Caviness et Jeremy R Johnson, éds. sci. *Quantifier elimination and cylindrical algebraic decomposition.* Springer, 1998. ISBN 3-211-82794-3.

[36] Amine Chaieb et Tobias Nipkow. Proof synthesis and reflection for linear arithmetic. *J. of Automated Reasoning*, 41 :33–59, 2008. ISSN 0168-7433. doi : 10.1007/s10817-008-9101-x. (PDF).

[37] Barry Cipra. How number theory got the best of the Pentium chip. *Science*, 267(5195) :175, 1995. ISSN 0036-8075. doi : 10.1126/science.267.5195.175.

[38] Edmund Clarke et Helmut Veith. Counterexamples revisited : Principles, algorithms, applications. In Nachum Derschowitz, éd. sci., *Verification (Manna* Festschrift*)*, volume 2772 de *LNCS*, pages 208–224. Springer, 2003. ISBN 3-540-21002-4. doi : 10.1007/b12001.

[39] Edmund M. Clarke, Jr, Orna Grumberg, et Doron A. Peled. *Model Checking*. MIT Press, 1999. ISBN 0-262-03270-8.

[40] George Collins. Quantifier elimination for real closed fields by cylindric algebraic decomposition. In *Automata theory and formal languages (2nd GI conference)*, LNCS, pages 134–183. Springer, 1975. ISBN 0-387-07407-4. réimprimé dans [35].

[41] Michael A. Colón, Sriram Sankaranarayanan, et Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification (CAV)*, number 2725 in LNCS, pages 420–433. Springer, 2003. ISBN 3-540-40524-0. doi : 10.1007/b11831.

[42] *Universal Serial Bus Specification, version 2.0.* Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, April 2000. (Web).

[43] Stephen A. Cook. The complexity of theorem-proving procedures. In *Symposium on theory of computing (STOC)*, pages 151–158. ACM, 1971. doi : 10.1145/800157.805047.

[44] Steven A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1) :70–90, 1978. ISSN 0097-5397. doi : 10.1137/0207005.

[45] Alexandru Costan, Stephane Gaubert, Éric Goubault, Matthieu Martel, et Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In Etessami and Rajamani [69], pages 462–475. ISBN 3-540-27231-3. doi : 10.1007/11513988_46. (PDF).

[46] P. Cousot et R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming (1976)*, pages 106–130, Paris, 1977. Dunod. ISBN 2-04-005185-6. Aussi connu comme *Actes du deuxième colloque international sur la programmation.*

[47] Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In Radhia Cousot, éd. sci., *Verification, Model Checking and*

Abstract Interpretation (VMCAI), number 3385 in LNCS, pages 1–24. Springer, 2005. ISBN 3-540-24297-X. doi : 10.1007/b105073.

[48] Patrick Cousot et Radhia Cousot. Abstract interpretation frameworks. *J. of Logic and Computation*, pages 511–547, August 1992. ISSN 0955-792X. doi : 10.1093/logcom/2.4.511. (PDF). (Postscript).

[49] Patrick Cousot et Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96. ACM, 1978. doi : 10.1145/512760.512770.

[50] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, et Xavier Rival. The ASTRÉE analyzer. In Sagiv [180], pages 21–30. ISBN 3-540-25435-8. doi : 10.1007/b107380. (PDF). (Postscript).

[51] Patrick Cousot, Radhia Cousot, Jerome Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, et Xavier Rival. Varieties of static analyzers : A comparison with ASTRÉE. In *Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2007. ISBN 0-7695-2856-2.

[52] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, et Xavier Rival. Combination of abstractions in the astrée static analyzer. In Mitsu Okada et Ichiro Satoh, éds. sci., *Advances in Computer Science — ASIAN 2006*, volume 4435 de *LNCS*, pages 272–300. Springer, 2008. ISBN 978-3-540-77504-1. doi : 10.1007/978-3-540-77505-8_23. (Web). (PDF).

[53] David Cox, John Little, et Donal O'Shea. *Ideals, Varieties, and Algorithms : An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, troisième édition, 2007. ISBN 0-387-35650-9.

[54] Amy Dahan Dalmenico. *Jacques-Louis Lions, un mathématicien d'exception*. La Découverte, Paris, 2005. ISBN 2-7071-4709-5.

[55] Werner Damm et Holger Hermanns, éds. sci. *Computer Aided Verification (CAV)*, volume 4590 de *LNCS*, 2007. Springer. ISBN 978-3-540-73367-6.

[56] George Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1998. ISBN 0-691-05913-6.

[57] Martin Davis et Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, 1960. ISSN 0004-5411. doi : 10.1145/321033.321034.

[58] Martin Davis, George Logemann, et Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962. ISSN 0001-0782. doi : 10.1145/368273.368557.

[59] Leonardo de Moura et Nikolaj Bjørner. Z3 : An efficient SMT solver. In Ramakrishnan and Rehof [168], pages 337–340. ISBN 3-540-78799-2. doi : 10.1007/978-3-540-78800-3_24.

[60] Leonardo de Moura et Nikolaj Bjørner. Proofs and refutations, and Z3. Technical report, Microsoft Research, 2008. (PDF). Presented at IWIL'08.

[61] Rocco de Nicola, éd. sci. *Programming Languages and Systems (ESOP)*, volume 4421 de *LNCS*, 2007. Springer. ISBN 978-3-540-71316-6.

[62] David Delmas et Jean Souyris. Astrée : From research to industry. In Nielson and Filé [158], pages 437–451. ISBN 978-3-540-74060-5. doi : 10.1007/978-3-540-74061-2_27.

[63] *Satisfiability Suggested Format.* DIMACS, 1993. (Postscript).

[64] Bruno Dutertre et Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball et Robert B. Jones, éds. sci., *Computer-aided verification (CAV)*, volume 4144 de *LNCS*, pages 81–94. Springer, 2006. ISBN 3-540-37406-X. doi : 10.1007/11817963_11. (PDF).

[65] Bruno Dutertre et Leonardo de Moura. Integrating simplex with DPLL(T). Technical Report SRI-CSL-06-01, SRI International, May 2006. (PDF).

[66] Niklas Eén et Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT '03)*, volume 2919 de *LNCS*, pages 333–336. Springer, 2004. ISBN 3-540-20851-8. doi : 10.1007/b95238. (Postscript).

[67] Dawson R. Engler et Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In Steffen and Levi [191], pages 191–210. ISBN 3-540-20803-8. doi : 10.1007/b94790.

[68] Dawson R. Engler, Benjamin Chelf, Andy Chou, et Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Operating System Design and Implementation (OSDI)*, pages 1–16. USENIX, 2000. ISBN 1-880446-16-2. (PDF).

[69] Kousha Etessami et Sriram K. Rajamani, éds. sci. *Computer Aided Verification (CAV)*, number 4590 in LNCS, 2005. Springer. ISBN 3-540-27231-3. doi : 10.1007/b138445.

[70] Jérôme Feret. Static analysis of digital filters. In Schmidt [186], pages 33–48. ISBN 3-540-21313-9. doi : 10.1007/b96702.

[71] Jeanne Ferrante et Charles Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. on Computing*, 4 (1) :69–76, March 1975. ISSN 0097-5397. doi : 10.1137/0204006.

[72] Samuel A. Figueroa del Cid. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages.* Thèse de doctorat, New York University, 2000. (PDF).

[73] Jean-Christophe Filliâtre et Sylvie Boldo. Formal verification of floating-point programs. In *Computer arithmetic (ARITH 18)*, pages 187–194. IEEE, 2007. ISBN 0-7695-2854-6. doi : 10.1109/ARITH.2007.20.

[74] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, et Paul Zimmermann. MPFR : A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2) : 13, 2007. ISSN 0098-3500. doi : 10.1145/1236463.1236468.

[75] *GSL — GNU scientific library.* Free Software Foundation, 2004.

[76] gcc. *The GNU compiler collection.* Free Software Foundation, 2005.

[77] mpfr. *MPFR.* Free Software Foundation, Inc., 2008. `http://www.mpfr.org`.

[78] Pierre Froment. L'architecture avionique de l'A380. *Annales des Mines*, RI-2005-IV, November 1985. ISSN 1148-7941. (PDF).

[79] Deborah Gage et John McCormick. 'We did nothing wrong' — Why software quality matters. *Baseline (Ziff-Davis)*, pages 4–21, March 2004. (PDF).

[80] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, et Cesare Tinelli. DPLL(T) : Fast Decision Procedures. In Rajeev Alur et Doron Peled, éds. sci., *Computer Aided Verification (CAV)*, volume 3114 de *LNCS*, pages 175–188. Springer, 2004. ISBN 3-540-22342-8. doi : 10.1007/b98490.

[81] Stéphane Gaubert, Éric Goubault, Ankur Taly, et Sarah Zennou. Static analysis by policy iteration on relational domains. In de Nicola [61], pages 237–252. ISBN 978-3-540-71316-6.

[82] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1) :5–48, 1991. ISSN 0360-0300. doi : 10.1145/103162.103163.

[83] Georges Gonthier. A computer-checked proof of the four colour theorem. Technical report, Microsoft Research, Cambridge, England, 2005. (PDF).

[84] Denis Gopan et Thomas W. Reps. Low-level library analysis and summarization. In Damm and Hermanns [55], pages 68–81. ISBN 978-3-540-73367-6. doi : 10.1007/978-3-540-73368-3_10.

[85] Éric Goubault. Static analyses of floating-foint operations. In *SAS*, volume 2126 de *LNCS*. Springer, 2001.

[86] Eric Goubault et Sylvie Putot. Static analysis of numerical algorithms. In Kwangkeun Yi, éd. sci., *Static analysis (SAS)*, volume 4134 de *LNCS*, pages 18–34. Springer, 2006. ISBN 3-540-37756-5. doi : 10.1007/11823230_3.

[87] Susanne Graf et Hassan Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, éd. sci., *Computer-Aided Verification (CAV)*, number 1254 in LNCS, pages 72–83. Springer, 1997. ISBN 3-540-63166-6. doi : 10.1007/3-540-63166-6_10.

[88] Sumit Gulwani, Saurabh Srivastava, et Ramarathnam Venkatesan. Program analysis as constraint solving. In *Programming Language Design and Implementation (PLDI)*. ACM, 2008. ISBN 978-1-59593-860-2. doi : 10.1145/1375581.1375616.

[89] Sumit Gulwani, Saurabh Srivastava, et Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403 de *LNCS*, pages 120–135. Springer, 2009. ISBN 978-3-540-93899-6. doi : 10.1007/978-3-540-93900-9_13.

[90] Nicolas Halbwachs. Delay analysis in synchronous programs. In Costas Courcoubetis, éd. sci., *Computer Aided Verification (CAV)*, volume 697 de *LNCS*, pages 333–346. Springer, 1993. ISBN 3-540-56922-7. doi : 10.1007/3-540-56922-7_28. (Postscript).

[91] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. Thèse de doctorat, Université scientifique et médicale de Grenoble, 1979.

[92] Nicolas Halbwachs, Pascal Raymond, et Christophe Ratel. Generating efficient code from data-flow programs. In *Programming Language Implementation and Logic Programming (PLILP)*, volume 528 de *LNCS*, pages 207–218, 1991. ISBN 3-540-54444-5. doi : 10.1007/3-540-54444-5_100.

[93] Nicolas Halbwachs, Yann-Erick Proy, et Pascal Raymond. Verification of linear hybrid systems by means of convex approximations. In *Static analysis (SAS)*, September 1994. ISBN 3-540-58485-4. doi : 10.1007/3-540-58485-4_43. (Postscript).

[94] Nicolas Halbwachs, Yann-Erick Proy, et Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Form. Methods Syst. Des.*, 11(2) :157–185, 1997. ISSN 0925-9856. doi : 10.1007/s10703-006-0013-2. (Web).

[95] John Harrison. Verifying nonlinear real formulas via sums of squares. In Klaus Schneider et Jens Brandt, éds. sci., *Theorem Proving in Higher Order Logics (TPHOL)*, volume 4732 de *LNCS*, pages 102–118. Springer, 2007. ISBN 978-3-540-74591-4. doi : 10.1007/978-3-540-74591-4_9.

[96] Les Hatton. *Safer C : Developing Software for High-Integrity and Safety-Critical Systems.* McGraw-Hill, 1995. ISBN 0077076400.

[97] Joos Heintz, Marie-Françoise Roy, et Pablo Solernó. Sur la complexité du principe de Tarski-Seidenberg. *Bulletin de la Société mathématique de France*, 118(1) :101–126, 1990. ISSN 0037-9484. (Web). (PDF). (Postscript).

[98] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969. ISSN 0001-0782. doi : 10.1145/363235.363259.

[99] Holger H. Hoos et Thomas Stützle. SATLIB : An online resource for research on SAT. In *I. P. Gent and van Maaren, H. and T. Walsh*, pages 283–292. IOS Press, 2000. (Web).

[100] IAEA. Investigation of an accidental exposure of radiotherapy patients in panama. Rapport d'un groupe d'experts STI/PUB/1114, International atomic energy agency, 2001. (PDF).

[101] *International standard – binary floating-point arithmetic for microprocessor systems.* IEC, 2nd édition, 1989. IEC-60559.

[102] *IEEE standard for Binary floating-point arithmetic for microprocessor systems.* IEEE, 1985. ANSI/IEEE Std 754-1985.

[103] *International standard – Programming languages – C.* ISO/IEC, 1999. standard 9899 :1999.

[104] Daniel Jackson. A direct path to dependable software. *Communications of the ACM*, 52(4) :78–88, April 2009. ISSN 0001-0782. doi : 10.1145/1498765.1498787.

[105] Bertrand Jeannet. Dynamic partitioning in linear relation analysis : Application to the verification of synchronous programs. Technical Report RS-00-38, BRICS, December 2000. (Web). (PDF). (Postscript).

[106] Bertrand Jeannet. *Partitionnement dynamique dans l'analyse de relations linéaires et application à la vérification de programmes synchrones.* Thèse de doctorat, Institut National Polytechnique de Grenoble, September 2000.

[107] Bertrand Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1) :5–37, July 2003. ISSN 0925-9856. doi : 10.1023/A :1024480913162.

[108] Yungbum Jung, Jaehwang Kim, Jaeho Shin, et Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In *Static analysis (SAS)*, volume 3672 de *LNCS*, pages 203–217. Springer, 2005. ISBN 3-540-28584-9. doi : 10.1007/11547662. (Postscript).

[109] William Kahan. Further remarks on reducing truncation errors. *Commun. ACM*, 8(1) :40, 1965. ISSN 0001-0782. doi : 10.1145/363707.363723.

[110] William Kahan. The baleful influence of SPEC benchmarks upon floating-point arithmetic. presentation available through the author's Web page, 2005. (PDF).

[111] William Kahan. Branch cuts for complex elementary functions, or much ado about nothing's sign bit. In A. Iserles et M.J.D. Powell, éds. sci., *The State of the Art in Numerical Analysis*, pages 165–211, Oxford, 1987. Clarendon Press. ISBN 0-19-853614-3.

[112] Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *Applications of Computer Algebra (ACA)*, 2004. (PDF).

[113] Matt Kaufmann, Panagiotis Manolios, et J Strother Moore. *Computer-Aided Reasoning : An Approach.* Kluwer Academic Publishers, 2000. ISBN 0-7923-7744-3.

[114] John Knight et Nancy Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. on Software Engineering*, SE-12(1) :96–109, January 1986. ISSN 0098-5589. (PDF).

[115] Steven Knox. *The Number of Facets of a Projection of a Convex Polytope.* Thèse de doctorat, University of Illinois, 1996.

[116] Jim Larus et Ravi Rawjar. *Transactional Memory.* Morgan & Claypool, 2007. ISBN 1-59829-124-6.

[117] Xavier Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*, pages 42–54. ACM, 2006. ISBN 1-59593-027-2. doi : 10.1145/1111037.1111042. (PDF).

[118] Nancy G. Leveson. The role of software in spacecraft accidents. *J. of Spacecraft and Rockets*, 41(4) :564–575, 2004. ISSN 0022-4650. (PDF).

[119] Nancy G. Leveson et Clark Savage Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7) :18–41, 1993. ISSN 0018-9162. doi : 10.1109/MC.1993.274940. (PDF).

[120] Jacques-Louis Lions et al. Ariane 5 : flight 501 failure, report by the inquiry board. Technical report, European Space Agency (ESA) and Centre national d'études spatiales (CNES), 1996. (Web).

[121] Henri Lombardi. *Mathématiques constructives et complexité en temps polynomial : 2 exemples : L'Algèbre réelle discrète, les différentes représentations des nombres réels.* Habilitation à diriger des recherches, Université de Franche-Comté, Besançon, France, 1990.

[122] Henri Lombardi. Théorème des zéros réels effectif et variantes. Technical report, Université de Franche-Comté, 1990. (PDF). partie de [121].

[123] Rüdiger Loos et Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5) :450–462, 1993. Special issue on computational quantifier elimination.

[124] Zohar Manna et John McCarthy. Properties of programs and partial function logic. In Bernard Meltzer et Donald Michie, éds. sci., *Machine Intelligence*, 5, pages 27–38. Edinburgh University Press, 1969. ISBN 0-85224-176-3.

[125] Matthieu Martel. Propagation of roundoff errors in finite precision computations : a semantics approach. In Daniel Le Métayer, éd. sci., *Programming Languages and Systems (ESOP)*, volume 2305 de *LNCS*. Springer, 2002. ISBN 3-540-43363-5. doi : 10.1007/3-540-45927-8_14.

[126] Matthieu Martel. Semantics of roundoff error propagation in finite precision computations. *Journal of Higher Order and Symbolic Computation*, 19(1) :7–30, 2006.

[127] Matthieu Martel. Static analysis of the numerical stability of loops. In *SAS*, number 2477 in LNCS. Springer, 2002.

[128] Laurent Mauborgne et Xavier Rival. Trace partitioning in abstract interpretation based static analyzer. In Sagiv [180], pages 5–20. ISBN 3-540-25435-8. doi : 10.1007/b107380. (Web). (PDF).

[129] Antoine Miné. The octagon abstract domain. In Axel Simon, Andy King, et Jacob M. Howe, éds. sci., *WCRE (Analysis, Slicing, and Transformation)*, pages 310–319. IEEE, 2001. doi : 10.1109/WCRE.2001.957836. (PDF).

[130] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In Schmidt [186], pages 3–17. ISBN 3-540-21313-9. doi : 10.1007/b96702.

[131] Antoine Miné. *Domaines numériques abstraits faiblement relationnels*. Thèse de doctorat, École polytechnique, 2004.

[132] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1) :31–100, 2006. doi : 10.1007/s10990-006-8609-1. (PDF).

[133] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Proc. of the 2d Symp. on Programs as Data Objects (PADO II)*, volume 2053 de *Lecture Notes in Computer Science*, pages 155–172, Aarhus, Danemark, May 2001. Springer. (PDF).

[134] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 de *LNCS*, pages 348–363. Springer, January 2006. ISBN 3-540-31139-4. doi : 10.1007/11609773. (PDF).

[135] *MISRA-C : 2004 Guidelines for the use of the C language in critical systems*. MIRA Ltd, October 2004. (Web).

[136] David Monniaux. The parallel implementation of the Astrée static analyzer. In *Programming Languages and Systems (APLAS)*, volume 3780 de *LNCS*. Springer, 2005. ISBN 3-540-29735-9. doi : 10.1007/11575467_7. (PDF). (Postscript).

[137] David Monniaux. Compositional analysis of floating-point linear numerical filters. In Etessami and Rajamani [69], pages 199–212. ISBN 3-540-27231-3. doi : 10.1007/b138445. (PDF).

[138] David Monniaux. Verification of device drivers and intelligent controllers : a case study. In Christoph Kirsch et Reinhard Wilhelm, éds. sci., *International conference on embedded software (EMSOFT)*, pages 30–36. ACM & IEEE, 2007. ISBN 978-1-59593-825-1. (PDF). (Postscript).

[139] David Monniaux. Backwards abstract interpretation of probabilistic programs. In *Programming Languages and Systems (ESOP)*, number 2028 in LNCS, pages 367–382. Springer, 2001. ISBN 3-540-41862-8. doi : 10.1007/3-540-45309-1_24. (PDF). (Postscript).

[140] David Monniaux. Analysis of cryptographic protocols using logics of belief : an overview. *Journal of Telecommunications and Information Technology*, 4 :57–67, 2002. ISSN 1509-4553. (PDF).

[141] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In Iliano Cervesato, Helmut Veith, et Andrei Voronkov, éds. sci., *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 5330 de *LNAI*, pages 243–257. Springer, 2008. ISBN 978-3-540-89438-4. doi : 10.1007/978-3-540-89439-1_18. (PDF).

[142] David Monniaux. An abstract Monte-Carlo method for the analysis of probabilistic programs (extended abstract). In *28th Symposium on Principles of Programming Languages (POPL '01)*, pages 93–101. Association for Computer Machinery, 2001. (PDF). (Postscript).

[143] David Monniaux. Automatic modular abstractions for linear constraints. In Pierce [167]. ISBN 978-1-60558-379-2. doi : 10.1145/1480881.1480899. (PDF).

[144] David Monniaux. Abstract interpretation of probabilistic semantics. In *Static analysis (SAS)*, number 1824 in LNCS, pages 322–339. Springer, 2000. ISBN 3-540-67668-6. doi : 10.1007/b87738. (PDF). (Postscript). Extended version on the author's web site.

[145] David Monniaux. An abstract analysis of the probabilistic termination of programs. In *Static analysis (SAS)*, number 2126 in LNCS, pages 111–126. Springer, 2001. (PDF). (Postscript).

[146] David Monniaux. Abstract interpretation of programs as Markov decision processes. In *Static analysis (SAS)*, number 2694 in LNCS, pages 237–254. Springer, 2003. ISBN 3-540-44898-5. doi : 10.1007/3-540-44898-5_13. (PDF). (Postscript).

[147] David Monniaux. Optimal abstraction on real-valued programs. In Nielson and Filé [158], pages 104–120. ISBN 978-3-540-74060-5. doi : 10.1007/978-3-540-74061-2_7. (PDF). (Postscript).

[148] David Monniaux. Abstracting cryptographic protocols with tree automata. In *Static analysis (SAS)*, number 1694 in LNCS, pages 149–163. Springer, 1999. ISBN 3-540-48294-6. doi : 10.1007/3-540-48294-6_10. (PDF). (Postscript).

[149] David Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2–3) :177–202, 2003. ISSN 0167-6423. doi : 10.1016/S0167-6423(02)00132-6. (PDF). (Postscript). Journal version of [148].

[150] David Monniaux. Abstract interpretation of programs as Markov decision processes. *Science of Computer Programming*, 58(1–2) :179–205, October 2005. ISSN 0167-6423. doi : 10.1016/j.scico.2005.02.008. (PDF). (Postscript). Journal version of [146].

[151] David Monniaux. The pitfalls of verifying floating-point computations. *Transactions on programming languages and systems (TOPLAS)*, 30 (3) :12, May 2008. ISSN 0164-0925. doi : 10.1145/1353445.1353446. (Web). (PDF).

[152] David Monniaux. Abstraction of expectation functions using gaussian distributions. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, et Supratik Mukhopadhyay, éds. sci., *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, number 2575 in LNCS, pages 161–173. Springer, 2003. ISBN 3-540-00348-7. doi : 10.1007/3-540-36384-X_15. (PDF). (Postscript).

[153] David Monniaux. *Analyse de programmes probabilistes par interprétation abstraite*. Thèse de doctorat, Université Paris IX Dauphine, 2001. (PDF).

[154] Michal Moskal. Rocket-fast proof checking for smt solvers. In Ramakrishnan and Rehof [168], pages 486–500. ISBN 3-540-78799-2. (PDF).

[155] Jean-Michel Muller. *Elementary functions, algorithms and implementation*. Birkhäuser, deuxième édition, 2006. ISBN 0-8176-4372-9.

[156] George C. Necula, Scott McPeak, Shree P. Rahul, et Westley Weimer. CIL : Intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, éd. sci., *Compiler Construction (CC)*, volume 2304 de *LNCS*, pages 209–265. Springer, 2002. ISBN 3-540-43369-4. doi : 10.1007/3-540-45937-5_16. (PDF).

[157] Thomas R. Nicely. Pentium FDIV FAQ, February 2008. (Web).

[158] Hanne Riis Nielson et Gilberto Filé, éds. sci. *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 de *LNCS*, 2007. Springer. ISBN 978-3-540-74060-5.

[159] Tobias Nipkow. Linear quantifier elimination. In Alessandro Armando, Peter Baumgartner, et Gilles Dowek, éds. sci., *Automated reasoning*

(IJCAR), volume 5195 de *LNCS*, pages 18–33. Springer, 2008. ISBN 978-3-540-71069-1. doi : 10.1007/978-3-540-71070-7_3.

[160] Tobias Nipkow, Lawrence C. Paulson, et Markus Wenzel. *Isabelle / HOL, a proof assistant for higher-order logic*. Number 2283 in LNCS. Springer, 2002. ISBN 3-540-43376-7.

[161] James Oberg. Why the Mars probe went off course [accident investigation]. *IEEE Spectrum*, 36(12) :34–39, December 1999. ISSN 0018-9235. doi : 10.1109/6.809121.

[162] *OpenMP Application Program Interface, version 2.5*. OpenMP architecture review board, 2005. (PDF).

[163] Sam Owre, John M. Rushby, , et Natarajan Shankar. PVS : A prototype verification system. In Deepak Kapur, éd. sci., *Automated deduction (CADE)*, volume 607 de *LNAI*, pages 748–752. Springer, June 1992. ISBN 3-540-55602-8. doi : 10.1007/3-540-55602-8_217. (Web).

[164] Pablo Parrilo. *Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization*. Thèse de doctorat, California Institute of Technology, 2000.

[165] J. Gregory Pavlovich et Charlotte L. Rea-Dix. Formal report of investigation, accident investigation board. Technical report, United States Air Force, June 1999. (Web).

[166] Charles Pecheur. Verification and validation of autonomy software at NASA. Technical Report NASA/TM-2000-209602, NASA, Ames Research Center, April 2000. (PDF).

[167] Benjamin C. Pierce, éd. sci. *Symposium on Principles of programming languages (POPL)*, 2009. ACM. ISBN 978-1-60558-379-2.

[168] C. R. Ramakrishnan et Jakob Rehof, éds. sci. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 de *LNCS*, 2008. Springer. ISBN 3-540-78799-2.

[169] Glenn E Reeves. What really happened on Mars ? Public email, December 1997. (Web).

[170] John C. Reynolds. Separation logic : A logic for shared mutable data structures. In *Logic in computer science (LICS)*, pages 55–74. IEEE Computer Society, 2002. ISBN 0-7695-1483-9. doi : 10.1109/LICS.2002.1029817. (Postscript).

[171] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) : 358–366, March 1953.

[172] Xavier Rival. Invariant translation-based certification of assembly code. *International Journal on Software and Tools for Technology Trasnfer*, 6(1) :15–37, july 2004. (PDF).

[173] Xavier Rival et Laurent Mauborgne. The trace partitioning abstract domain. *Transactions on Programming Languages and Systems (TOPLAS)*, 29(5) :26, 2007. ISSN 0164-0925. doi : 10.1145/1275497.1275501. (PDF).

[174] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* MIT Press, 1987. ISBN 0-262-68052-1.

[175] William P. Rogers, Neil A. Armstrong, David C. Acheson, Eugene E. Covert, Richard P. Feynman, Robert B. Hotz, Donald J. Kutyna, Sally K. Ride, Robert W. Rummel, Joseph F. Sutter, Arthur B. C. Walker, Jr., Albert D. Wheelon, Charles Yeager, et Alton G. Keel, Jr. Report to the President. Technical report, Presidential commission on the Space Shuttle Challenger accident, June 1986. (Web).

[176] M. Romdhani, P. Chambert, A. Jeffroy, P. de Chazelles, et A. A. Jerraya. Composing ActivityCharts/StateCharts, SDL and SAO specifications for codesign in avionics. In *EURO-DAC / EURO-VHDL (Conference on European design automation)*, pages 585–590. IEEE Computer Society, 1995. ISBN 0-8186-7156-4. doi : 10.1109/EURDAC.1995.527465.

[177] DO-178B. *Software considerations in airborne systems and equipment certification (RTCA/DO-178B, EUROCAE ED-12B).* RTCA, Inc., December 1992.

[178] RTI Health, Social, and Economics Research. The economic impacts of inadequate infrastructure for software testing. planning report 02-3, NIST (US National Institute of Standards and Technology), May 2002. (PDF).

[179] Siegfried M. Rump. Ten methods to bound multiple roots of polynomials. *J. of Computational and Applied Math.*, 156(2) :403–432, 2003.

[180] Shmuel "Mooly" Sagiv, éd. sci. *Programming Languages and Systems (ESOP)*, number 3444 in LNCS, 2005. Springer. ISBN 3-540-25435-8. doi : 10.1007/b107380.

[181] Sriram Sankaranarayanan, Henny Sipma, et Zohar Manna. Constraint-based linear-relations analysis. In *SAS*, number 3148 in LNCS, pages 53–68. Springer, 2004.

[182] Sriram Sankaranarayanan, Henny B. Sipma, et Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, volume 3385 de *LNCS*, pages 21–47. Springer Verlag, 2005.

[183] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, et Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Symposium on Principles of programming languages (POPL)*, pages 379–391. ACM, 2009. ISBN 978-1-60558-379-2. doi : 10.1145/1480881.1480929.

[184] Norman Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, et Florence Maraninchi. Defining and translating a “safe” subset of Simulink/Stateflow into Lustre. In *International conference on Embedded software (EMSOFT)*, pages 259–268, 2004. ISBN 1-58113-860-1. doi : 10.1145/1017753.1017795.

[185] Norman Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, et Florence Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. Technical Report TR-2004-16, VERIMAG, 2004. (PDF). Full version of [184].

[186] David Schmidt, éd. sci. *Programming Languages and Systems (ESOP)*, number 2986 in LNCS, 2004. Springer. ISBN 3-540-21313-9. doi : 10.1007/b96702.

[187] Abraham Seidenberg. A new decision method for elementary algebra. *Annals of Mathematics*, 60(2) :365–374, September 1954. (Web).

[188] Robert Skeel. Roundoff error and the Patriot missile. *SIAM News*, 25 (4) :11, July 1992.

[189] Jean Souyris et David Delmas. Experimental assessment of Astrée on safety-critical avionics software. In Francesca Saglietti et Norbert Oster, éds. sci., *SAFECOMP*, volume 4680 de *LNCS*, pages 479–490. Springer, 2007. ISBN 978-3-540-75100-7. doi : 10.1007/978-3-540-75101-4_45.

[190] Cary R. Spitzer, éd. sci. *The Avionics Handbook*. CRC Press, December 2000. ISBN 084938348X.

[191] Bernhard Steffen et Giorgio Levi, éds. sci. *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 de *Lecture Notes in Computer Science*, 2004. Springer. ISBN 3-540-20803-8.

[192] Gilbert Stengle. A *Nullstellensatz* and a *Positivstellensatz* in semialgebraic geometry. *Mathematische Annalen*, 2(207) :87–97, 1973.

[193] Arthur G. Stephenson, Lia S. LaPiana, Daniel R. Mulville, Peter J. Rutledge, Frank H. Bauer, David Folta, Greg A. Dukeman, Robert Sackheim, et Peter Norvig. Mars climate orbiter mishap investigation board phase I report. Technical report, NASA, November 1999. (PDF).

[194] Norihisa Suzuki et Kiyoshi Ishihata. Implementation of an array bound checker. In *Symposium on Principles of programming languages (POPL)*, pages 132–143, New York, NY, USA, 1977. ACM. doi : 10.1145/512950.512963.

[195] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951. reprinted as [196].

[196] Alfred Tarski. A decision method for elementary algebra and geometry. Technical Report 109, The RAND Corporation, 1957. (PDF). deuxième édition (originale 1948, révisée 1951), réimpression de [195].

[197] Lieven Vandenberghe et Stephen Boyd. Semidefinite programming. *SIAM Review*, 38(1) :49–95, March 1996. (PDF).

[198] John von Neumann. Various techniques used in connection with random digits. *National Bureau of Standards Applied Mathematics Series*, 12 :36–38, 1951.

[199] Daniel Weise. Static analysis of mega-programs. In Agostino Cortesi et Gilberto Filé, éds. sci., *Static Analysis (SAS)*, volume 1694 de *Lecture Notes in Computer Science*, pages 300–302. Springer, 1999. ISBN 3-540-66459-9. doi : 10.1007/3-540-48294-6_19.

[200] Volker Weispfenning. Quantifier elimination for real algebra : the quadratic case and beyond. *Applicable Algebra in Engineering Communication and Computing*, 8(2) :85–101, 1997.

[201] Volker Weispfenning. Complexity and uniformity of elimination in Presburger arithmetic. In *Universitat Passau*, pages 48–53. ACM Press, 1997. (Postscript).

[202] Glynn Winskel. *The Formal Semantics of Programming Languages : An Introduction*. Foundations of Computing. MIT Press, 1993. ISBN 0-262-23169-7.

[203] Jack Woehr. A conversation with William Kahan. *Dr Dobb's Journal*, Nov. 1 1997. (Web).

[204] Yichen Xie et Alex Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE '05*, pages 115–125. ACM, 2005. ISBN 1-59593-014-0. doi : 10.1145/1081706.1081728.

[205] Misha Zitser. Securing software : an evaluation of static source code analyzers. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., 2003. (Web).

[206] Misha Zitser, Richard Lippmann, et Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6) :97–106, 2004. ISSN 0163-5948. doi : 10.1145/1041685.1029911.

Index

91, fonction, voir McCarthy, John

0, voir zéro
387, unité virgule flottante, 44
80387, voir 387

A320, Airbus, 3
A340, Airbus, 3
ACL2, assistant de preuve, 7
Airbus, 3
Analyse de programmes
  dynamique, 6
  statique, **6–12**
arbre
  binaire équilibré, 24
  de Patricia, 24
Ariane, fusée
  accident, 2, 5
arithmétique
  dépassement, 22
arrondi
  double, 44
arrondi en virgule flottante, **41**
  axiomatisation, **49**
assistant de preuve, 7
Astrée, **19–23**

B777, Boeing, 3
BDD, 8
Boeing, 3

CAD,
  see décomposition cylindrique algébrique65
calculabilité, 7
Caveat, 21
Challenger, navette spatiale, 4
clôture
  d'un élément abstrait, 62
commandes de vol électriques, **3**
complexité, 7
contre-exemple
  raffinement d'abstraction guidé par, 9
Cook, Stephen A.
  théorème de complétude relative de, 7
Coq, assistant de preuve, 7, 25
corps (mathématiques)
  réel clos
    élimination des quantificateurs, 65
Coverity, outil d'analyse statique, 8
critique, système, voir système critique

Deep Space 1, 2
Deep Space 2, 2
DO-178B, 3
DPLL, 8
dysfonctionnement
  informatique, **1**
Décomposition
  cylindrique algébrique, 65, 73

erreur
  à l'exécution, 10

Farkas, lemme de, 71
FDIV
  bug de l'instruction, 3

Ferrante-Rackoff
  algorithme de, **55**
Feynman, Richard P., 4
Filtre
  linéaire, 30
flottants
  abstraction, 48–50
  arrondi, voir arrondi en virgule flottante
Floyd-Washall, algorithme de, 62
fly-by-wire, voir commandes de vol électriques
Fourier-Motzkin, algorithme de, **52**, 66, 71

Gödel, Kurt
  théorème d'incomplétude de, 7

IEEE-754, **41**
insatisfiabilité
  de contraintes arithmétiques, 69–74
interprétation abstraite, **12–17**
intervalle, 62
ISABELLE, assistant de preuve, 7

Kahn, Gilles, 5

Loos-Weispfenning
  algorithme de, **55**
LUSTRE, 20

Mars Climate Orbiter, 2
Mars Pathfinder, 2
Mars Polar Lander, 2
McCarthy, John
  Fonction 91 de, 68
Milstar, satellite, 2, 5
MISRA-C, 6

Nullstellensatz, 71
  réels, 73
NuSMV, *model-checker*, 8

octogone, 62
OHCI, norme de contrôleurs de bus USB, 34–36
optimisation
  compilation
    et flottants, 43
ordre
  partiel
    réduction, 35

parallèle
  analyse de programme, 33
  implémentation d'interpréteur abstrait, 26
Patriot, missile, 2
Pentium, processeur
  bug de la multiplication flottante, 3
pointeur
  nul ou incorrect, 22
Polyspace Verifier, outil d'analyse statique, 8
polyèdre, **53–55**
Positivstellensatz, 73
preuve
  d'implémentation d'analyseur, 25, 28
preuve mathématique
  assistant de, voir assistant de preuve
Programmation (recherche opérationnelle)
  semidéfinie, 76
programmation (recherche opérationnelle)
  linéaire, 62, 71
PVS, assistant de preuve, 7
Péano, Giuseppe
  arithmétique de, 7

quantificateur
  élimination de, 51
    arithmétique réelle linéaire, 52–61

arithmétique réelle polynomiale, 65

radiothérapie, 2
redondance, 3–4
Rice, théorème de, 7

SAO, 20
SAT, 29, 58
Satisfiabilité booléenne, voir SAT
Satisfiabilité modulo théorie, voir SMT
SCADE, 20
Scud, missile, 2
SIMULINK, 20
SMT, 59, 66
Somme
  de carrés, 74
spécification, **9**
SSE, unité virgule flottante, 47
STATEFLOW, 20
système
  critique, 2
sémantique
  de langage de programmation, **9**
  de programme, 7

tableau
  débordement de, 22
Therac-25, machine de radiothérapie, 2, 5
transformée
  en Z, 31
Turing, Alan M.
  théorème de l'arrêt, 7

USB, 34–36

x86, architecture
  flottants, 44
  multi-cœur, 33
  parallélisme, 33

Zariski, clôture de, 77

zéro
  en IEEE-754, 41

élargissement, 28

Résumé

Il est important que les logiciels pilotant les systèmes critiques (avions, centrales nucléaires, etc.) fonctionnent correctement — alors que la plupart des systèmes informatisés de la vie courante (micro-ordinateur, distributeur de billets, téléphone portable) ont des dysfonctionnements visibles. Il ne s'agit pas là d'un simple problème d'ingéniérie : on sait depuis les travaux de Turing et de Cook que la preuve de propriétés de bon fonctionnement sur les programmes est un problème intrinsèquement difficile.

Pour résoudre ce problème , il faut des méthodes à la fois efficaces (coûts en temps et en mémoire modérés), sûres (qui trouvent tous les problèmes possibles) et précises (qui fournissent peu d'avertissements pour des problèmes inexistants). La recherche de ce compromis nécessite des recherches faisant appel à des domaines aussi divers que la logique formelle, l'analyse numérique ou l'algorithmique « classique ».

De 2002 à 2007 j'ai participé au développement de l'outil d'analyse statique Astrée. Ceci m'a suggéré quelques développements annexes, à la fois théoriques et pratiques (utilisation de techniques de preuve formelle, analyse de filtres numériques...). Plus récemment, je me suis intéressé à l'analyse modulaire de propriétés numériques et aux applications en analyse de programme de techniques de résolution sous contrainte (programmation semidéfinie, techniques SAT et SAT modulo théorie).

Summary

Software operating critical systems (aircraft, nuclear power plants) should not fail — whereas most computerised systems of daily life (personal computer, ticket vending machines, cell phone) fail from time to time. This is not a simple engineering problem: it is known, since the works of Turing and Cook, that proving that programs work correctly is intrinsically hard.

In order to solve this problem, one needs methods that are, at the same time, efficient (moderate costs in time and memory), safe (all possible failures should be found), and precise (few warnings about nonexistent failures). In order to reach a satisfactory compromise between these goals, one can research fields as diverse as formal logic, numerical analysis or "classical" algorithmics.

From 2002 to 2007 I participated in the development of the Astrée static analyser. This suggested to me a number of side projects, both theoretical and practical (use of formal proof techniques, analysis of numerical filters...). More recently, I became interested in modular analysis of numerical property and in the applications to program analysis of constraint solving techniques (semidefinite programming, SAT and SAT modulo theory).