Conception et Implantation d'un Analyseur Statique Spécialisé pour l'Analyse de Code Critique Temps-Réel Embarqué

Bruno Blanchet[1] , Patrick Cousot[1] , Radhia Cousot[2] , Jérôme Feret[1] , Laurent Mauborgne[1] , Antoine Miné[1] , David Monniaux[1] , Xavier Rival[1]

Résumé

Nous rapportons sur une expérience préliminaire fructueuse sur la conception et l'implantation d'un analyseur statique de programmes, basé sur l'interprétation abstraite, pour la vérification de l'absence d'erreurs à l'exécution dans des logiciels critiques embarqués temps-réel. L'analyseur est à la fois précis (aucune fausse alarme dans l'expérimentation considérée) et efficace (moins d'une minute de temps d'analyse pour 10 000 lignes de code). Même s'il est basé sur une simple analyse d'intervalles, de nombreuses extensions à cette analyse classique ont été introduites pour obtenir de telles performances : expansion des petits tableaux, élargissements étagés, dépliages de boucles, partitionnement de traces, relation entre compteurs de boucle et autres variables. L'efficacité de l'outil vient d'une représentation astucieuse des environnements abstraits basée sur des applications fonctionnelles implantées par des arbres binaires de recherche équilibrés.

Dédicacé à Neil Jones, pour son $60^{ième}$ anniversaire.

1 Introduction

1.1 Analyseurs statiques généraux

L'objectif de l'analyse statique de programmes est de déterminer automatiquement et statiquement (c'est-à-dire à la compilation) des propriétés relatives à l'exécution du programme (comme l'absence d'erreurs à l'exécution). Ce problème est en général indécidable, et donc toute analyse statique est basée sur une approximation formalisée par la théorie de l'Interprétation Abstraite [8, 9]. Par exemple, les programmes bien typés ne peuvent pas produire certaines formes d'erreurs à l'exécution mais certains programmes non typables ne produisent jamais aucune de ces erreurs à l'exécution.

Dans de nombreux contextes, comme la transformation de programmes, l'incertitude induite par cette approximation est acceptable. Par exemple l'analyse d'intervalles peut être utilisée pour éliminer statiquement les tests de bornes de tableaux inutiles à l'exécution [7]. Le taux de sélection (c'est-à-dire la proportion du nombre d'alarmes potentielles qui sont définitivement résolues positivement ou négativement) est souvent compris entre 80 et 95%, de sorte que l'optimisation est

1. CNRS & École normale supérieure, 75005 Paris, France
2. CNRS & École polytechnique, 91128 Palaiseau cedex, France

efficace. De plus elle est correcte car les 5 à 20% de cas pour lesquels les tests ne peuvent pas être éliminés à la compilation seront faits à l'exécution. Parfois, l'analyse statique de programmes permet de découvrir des erreurs certaines dès la compilation (par exemple des défauts d'initialisation), ce qui est utile pour le test de programmes. L'objectif de tels *analyseurs statiques généraux* de programmes sont les suivants :

1. de traiter complètement un langage de programmation général (comme Ada ou C), y compris les librairies standards ;
2. de ne requérir aucune spécification ou annotation de l'utilisateur (excepté, peut-être, des spécifications légères comme par exemple les intervalles de valeurs des variables d'entrée, ou l'effet des fonctions de la librairie dont le source n'est pas disponible) ;
3. d'être assez précis pour fournir une information intéressante pour la plupart des programmes (par exemple des informations pertinentes pour la manipulation statique ou la mise au point de programmes) ;
4. d'être assez efficace pour traiter de grands programmes (avec un coût de quelques mégaoctets de mémoire et minutes de temps de calcul à quelques gigaoctets et heures voire jours de calculs pour traiter des dizaines de milliers de lignes de code source).

De tels analyseurs statiques généraux de programmes sont très difficiles à concevoir à cause de la complexité des langages de programmation et des systèmes modernes. Certains sont disponibles commercialement et ont eu des succès pour repérer des erreurs ou en prévenir dès les premières phases de développement des programmes. Puisque la couverture est de 100%, les fausses alarmes peuvent être traitées par d'autres techniques de mise au point, comme par exemple les techniques classiques de test, ce qui réduit d'autant la nécessité de faire des tests d'absence d'erreurs à l'exécution par un facteur de 80 à 95% correspondant aux cas traités statiquement, ce qui est un facteur économique très significatif.

1.2 Vérification de programmes

Dans le contexte de logiciels temps-réel embarqués critiques pour la sécurité, comme il s'en trouve dans les industries des transports, le test d'erreurs à l'exécution n'est, en général, pas du tout acceptable. Dans ce cas les coûts de mise au point par test sont extrêmement élevés and parfois significativement plus élevés que le coût de développement du logiciel. Dans ce contexte industriel particulier, où la correction est requise pour des raisons de criticité et de sécurité, des méthodes formelles rigoureuses devraient être applicables et économiquement rentables.

1.2.1 Méthodes déductives.

En pratique les méthodes déductives (voir par exemple [1, 17]) sont difficiles à appliquer quand on ne dispose pas d'une spécification formelle du programme ou quand la taille du programme est très grande. En fait le coût de développement de la spécification et de la preuve de correction, même avec un assistant de preuve ou un démonstrateur de théorème, est en général bien plus grand que le coût de développement et de test du programme lui-même (des chiffres de 600 personnes-années pour 80 000 lignes de code C ont été avancés). Seules des petites parties

du logiciel ou des parties extrêmement critiques du logiciel peuvent être traitées formellement auquel cas les erreurs apparaissent inévitablement ailleurs (par exemple aux interfaces). De plus, pour du logiciel embarqué dont la durée de vie peut être de 10 à 20 ans, le programme et sa preuve doivent être maintenus pendant toute cette longue période de temps.

1.2.2 Vérification exhaustive de modèle (*model checking*).

La vérification exhaustive de modèle (voir, par exemple, [12]) est également difficile à appliquer quand on ne dispose pas de spécifications formelles ou quand la taille des programmes est très grande. Ceci vient du fait qu'au coût de développement du programme, il faut ajouter le coût de développement de la spécification et également celui de développement du modèle. Des problèmes bien connus sont la difficulté de fournir des spécifications temporelles compréhensibles et des modèles évitant l'explosion combinatoire quand ils sont explorés exhaustivement. Si le modèle n'est pas prouvé correct, alors la correction du programme n'est pas totalement vérifiée et cette technique s'apparente alors plus aux techniques de test par prototypage. Si par contre, le modèle du programme est prouvé correct, soit manuellement, soit par des méthodes déductives, alors la preuve du modèle est logiquement équivalente à la preuve de correction du programme original [6] ce qui par conséquent requiert un immense effort. De plus le modèle du programme et sa preuve de correction doivent être maintenus en même temps que le programme lui-même, ce qui peut induire un coût additionnel significatif. De plus, dans de nombreux cas des abstractions sont nécessaires pour éviter l'explosion du nombre d'états à considérer, auquel cas la vérification exhaustive de modèle se réduit à l'analyse statique de programmes.

1.2.3 Analyse statique de programmes.

Un analyseur statique de programmes prouve des propriétés dynamiques des programmes en calculant effectivement une sémantique abstraite des programmes exprimée sous forme d'un calcul de point fixe ou de résolution de contraintes. L'information collectée peut alors être utilisée comme base pour la manipulation, le test ou la vérification partielle ou totale du programme. Selon la classe de propriétés considérées, divers types d'analyseurs statiques peuvent être conçus, tous fondés sur la théorie de l'interprétation abstraite [5].

1.3 Sur l'utilisation d'analyseurs statiques généraux de programmes

Les analyseurs statiques généraux de programmes ne requièrent aucune intervention humaine et donc, sont très peu coûteux à utiliser, en particulier en phase initiale de test, et ensuite pendant la phase de maintenance et de modifications successives du programme. Cependant, même pour des spécifications simples, ils ne sont guère utiles pour la vérification formelle à cause de leur temps d'exécution et des fausses alarmes. Pour obtenir une bonne précision le temps d'analyse doit être élevé ce qui, dans la phase de développement initial du logiciel, est une contrainte très forte pour une utilisation de routine comparable à celle d'un compilateur. Les fausses alarmes trop nombreuses excluent une vérification complète. Un taux de sélection de 95% — qui est très élevé pour un analyseur statique général — signifie qu'une partie significative

du code doit encore être inspectée et testée manuellement, ce qui reste un coût élevé ou, si les alarmes restantes sont ignorées, est incompatible avec des contraintes de sécurité sévères. De plus, quand les temps d'analyse sont de plusieurs heures, sinon de plusieurs jours, le raffinement des analyses, par exemple par insertion d'assertions de correction, est un processus très long qui, de toutes façons, ne permettra pas d'éliminer toutes les fausses alarmes à cause des approximations inhérentes qui sont incluses dans l'analyseur.

1.4 Analyseurs statiques spécialisés pour des familles de programmes.

À cause des résultats fondamentaux d'indécidabilité et des exigences d'efficacité et de précision de l'analyse, les fausses alarmes ne peuvent être complètement éliminées qu'en restreignant à la fois les classes de spécifications et de programmes considérées. Ceci conduit à l'idée nouvelle d'*analyseurs statiques spécialisés*. Leurs objectifs sont :

1. de traiter une famille restreinte de programmes (en général n'utilisant pas toute la complexité des langages de programmation modernes) ;
2. de traiter une famille restreinte de spécifications d'intérêt général (sans interventions de l'utilisateur, sauf peut-être des interventions légères, comme par exemple, pour fournir des hypothèses sur les variables d'entrée ou volatiles) ;
3. d'être assez précis pour éliminer toute fausse alarme (peut-être grâce à une nouvelle conception de l'analyse, ou mieux, par une paramétrisation adéquate de l'analyseur effectuée par un utilisateur entraîné sans qu'il soit nécessairement un spécialiste de l'analyse statique et de l'Interprétation Abstraite) ;
4. d'être assez efficace pour traiter de grands programmes (à des coûts de quelques mégaoctets de mémoires et minutes ou heures de calcul pour des centaines de milliers de lignes de code source).

En traitant une famille de programmes et non pas seulement un seul programme ou un seul modèle de ce programme, nous pouvons traiter le problème des modifications successives du programme au fil des ans à un coût économique raisonnable (pour construire l'analyseur spécialisé initial). En restreignant la classe de spécifications considérées et plus précisément en considérant des spécifications universelles (comme l'absence d'erreurs et d'interruptions inattendues à l'exécution), nous évitons le coût de développement de spécifications dédiées à un programme particulier et pouvons appliquer l'analyseur à des logiciels existants (par exemples des applications logicielles vieilles de plusieurs décades pour lesquelles les spécifications, si elles existent, n'ont pas été maintenue au fil des ans en même temps que les modifications successives du programme). De plus, l'équilibre entre la précision de l'analyse et son coût peut être ajusté avec soin par des choix appropriés d'abstractions réutilisables (par exemple sous forme de bibliothèques d'analyse statique).

1.5 Rapport sur une première expérience de conception d'un analyseur statique de programmes spécialisé

Dans ce papier, nous rapportons sur une première expérience de conception d'un analyseur statique de programmes spécialisé. La classe de logiciels considérée est celle des applications critiques temps réel embarquées synchrones écrites dans un sous-ensemble de C. La classe de spécifications considérée est celle de l'absence d'erreurs

à l'exécution. Ce rapport d'expérience explique les décisions cruciales de conception de l'analyseur et les quelques échecs qui ont mené d'un analyseur initialement lent et imprécis à un nouvel analyseur spécialisé, complètement réécrit, qui est rapide et précis. En fournissant des détails sur la conception et l'implantation de cet analyseur spécialisé aussi bien que sur sa précision (absence de fausses alarmes), ses temps d'exécution et les tailles de mémoire nécessaires, nous montrons que cette nouvelle approche est techniquement et économiquement viable.

2 Les spécificités de l'analyseur

À cause de leur aspect extrêmement critique, la classe de logiciels analysés dans cette première expérience, a été développé par un processus de conception très rigoureux. Dans ce processus de conception, le logiciel est d'abord conçu de façon schématique. Ces schémas sont ensuite traduits automatiquement en code C en utilisant des macros écrites à la main qui correspondent aux fonctionnalités de base dans les schémas. Ce code C, organisé en de multiples fichiers, constitue l'entrée de l'analyseur.

À cause des aspects temps réel synchrone de l'application, la structure globale du logiciel consiste en une phase d'initialisation suivie d'une boucle globale synchronisée. À cause de cette structure, presque toutes les variables du programme dépendent les unes des autres.

À cause du mode de génération automatique du code C, le programme contient un grand nombre de variables globales ou statiques, à peu près linéaire en la longueur du code (environ 1 300 pour 10 000 LOCs [3]). Par conséquent, l'espace mémoire nécessaire pour l'analyse ne peut pas être réduit par une analyse préliminaire de localité des données manipulées par le programme.

2.1 Sous-ensemble restrictif de C dans lequel la classe de logiciels est écrite

Les contraintes fortes imposées par la méthode de développement de ce logiciel critique ont l'heureuse conséquence que des traits du langage C difficiles à analyser ne sont pas à considérer pour cette classe de logiciels. Tout d'abord il n'y a pas de branchements inconditionnels (**goto**) ni d'appels récursifs de fonctions. Les structures de données sont très simples : le logiciel ne manipule pas de structures de données récursives et les seuls pointeurs sont les tableaux alloués statiquement (pas d'allocation dynamique de mémoire). Il n'y a pas d'arithmétique de pointeurs mis à part pour les opérations élémentaires sur les tableaux. Le code ne contenant pas non plus de chaînes de caractères, l'information d'alias est triviale.

2.2 Spécification à vérifier

L'analyseur doit prouver les propriétés suivantes :

– absence d'index de tableau en dehors des bornes ;

3. Le nombre de lignes de code (LOCs) est compté avec la commande Unix™ `wc -l` après élimination des commentaires et pré-expansion des macros. Ensuite, les abstractions que nous considérons vont essentiellement conserver toutes les variables et les LOCs, voir la Sec. 8.

– correction logique des opérations d'arithmétique entière et flottante (essentiellement, absence de débordements, de division par 0).

par conséquent l'analyse consiste à calculer statiquement une sur-approximation de l'ensemble des états accessibles en cours d'exécution.

3 Sémantique concrète du programme

La sémantique concrète du programme est une formalisation mathématique de l'exécution effective du programme. Une définition précise est nécessaire pour définir et prouver la correction d'un vérificateur ou d'un analyseur. par exemple en analyse statique, l'analyseur calcule effectivement une sémantique abstraite qui est une approximation sûre de cette sémantique concrète. Par conséquent, une définition rigoureuse de la sémantique concrète du programme est obligatoire pour toutes les méthodes formelles.

En pratique cette sémantique concrète est définie par :

– le standard ISO/IEC 9899 de définition du langage de programmation C [14] ainsi que par le standard ANSI/IEEE 754 pour l'arithmétique flottante [2] ;
– l'implantation de ces standards par le compilateur et la machine ;
– les hypothèses et attentes de l'utilisateur final.

Chaque sémantique est un raffinement de la sémantique précédente où un certain non-déterminisme est résolu.

3.1 La sémantique standard de C

La sémantique standard de C est très souvent non-déterministe pour prendre en compte diverses implantations possibles. Voici trois exemples :

Les comportements non spécifiés sont des comportements pour lesquels le standard prévoit deux ou plusieurs possibilités et n'impose aucune restriction sur celle qui doit être choisie. Un exemple de comportement non spécifié est l'ordre dans lequel sont évalués les arguments d'une fonction ;

Les comportements indéfinis correspondent à des constructions de programmes non portables ou erronées sur lesquels aucune restriction n'est imposée. Un exemple de comportement indéfini est le débordement entier ;

Les comportements définis par l'implémentation sont des comportements non spécifiés dans la norme tels que chaque implémentation doivent spécifier précisément quel comportement est choisi. Un exemple de comportement défini par l'implémentation est le nombre de bits dans un **int** ou la propagation du bit d'ordre supérieur quand un entier signé est décalé à droite.

Un analyseur statique basé sur la sémantique standard de C serait correct et sûr pour tous les compilateurs et machines conformes. Cette approche n'est pas très réaliste car les pires hypothèses doivent être faites par la sémantique concrète pour tous les comportements indéfinis et les comportements définis par l'implémentation. Ces derniers ne sont pas toujours faciles à imaginer quand aucune restriction n'est imposée et conduisent des toutes façons à de grosses pertes de précision qui produiraient de trop nombreuses fausses alarmes. Par exemple, le standard de C n'impose aucune restriction sur la taille et la précision des divers types arithmétiques, seulement

des tailles minimales, et donc l'analyse ne pourrait être que très imprécise en ce qui concerne les débordements arithmétiques.

3.2 La sémantique d'implémentation

Un compilateur correct pour un processeur donné implantera un raffinement de la sémantique standard en choisissant parmi tous les comportements possibles à l'exécution tels qu'ils sont défini par la norme. Pour obtenir des analyses précises, la conception de l'analyseur statique devra prendre en compte les comportements non spécifiés (ou même indéfinis) dans la norme mais qui sont parfaitement prévisibles pour un compilateur et un processeurs donnés (pourvu que le processeur soit prédictible). Par exemple :

comportements non spécifiés : les arguments d'une fonction sont évalués de gauche à droite ;

comportements indéfinis : le débordement arithmétique est impossible à cause de l'arithmétique modulaire (la division et le modulo étant les seules erreurs possibles à l'exécution en arithmétique entière) ;

comportements définis par l'implémentation : il y a 32 bits dans un **int** et le bit d'ordre supérieur est copié quand un entier signé est décalé à droite.

3.3 La sémantique de l'utilisateur final

L"utilisateur final peut avoir en tête un raffinement de la sémantique d'implémentation. Voici quelques exemples :

initialisation à zéro qui peut être effectuée par le système avant de lancer le programme (alors que le standard de C ne le prévoit seulement que pour les variable statiques) ;

les variables volatiles faisant l'interface avec le matériel peuvent avoir un domaine de variation imposé par le matériel, de sorte que toute lecture de cette variable volatile retourne toujours un valeur dans un intervalle spécifié ;

calculs arithmétiques entiers qui peuvent être sujets à des débordements (logiques puisqu'ils représentent des quantités entières bornées pour lesquelles l'arithmétique modulaire n'aurait aucun sens) ou non (comme des opérations de décalage pour extraire des champs de mots faisant l'interface avec le matériel pour lesquels un débordement n'a aucun sens logique).

Pour obtenir des résultats d'analyse qui aient un sens, on doit distinguer entre les cas ou l'exécution est supposée continuer ou non quand elle atteint un état où un comportement est indéfini ou défini par l'implémentation. Dans le cas d'un comportement indéfini dans la norme, il faut prendre en compte le comportement qui sera choisi dans une exécution réelle. Dans le cas d'un comportement définis par l'implémentation mais erroné, il faut considérer que l'exécution ne peut pas continuer. Prenons deux exemples :

- dans un contexte où $\mathtt{x} \in [0,\mathtt{maxint}]$ est une variable entière non signée, l'analyse de l'affectation $\mathtt{y} := 1/\mathtt{x}$ va signaler une erreur logique dans le cas $\mathtt{x} = 0$. Dans l'implémentation considérée, la division entière par zéro lève toujours une exception système qui termine l'exécution normale du programme. Par conséquent l'analyseur doit fournir une alerte dans le cas d'erreur logique $\mathtt{x} = 0$

et continuer l'analyse normalement dans les autres cas où il n'y a pas eu d'erreur à l'exécution auquel cas $\texttt{y} \in [1/\texttt{maxint},1]$. Dans ce cas, la sémantique concrète et l'implémentation coïncident.

– Dans un contexte où $\texttt{x} \in [0,\texttt{maxint}]$ est une variable entière, l'analyse de l'affectation $\texttt{y} := \texttt{x} + 1$ va lancer un avertissement dans le cas où $\texttt{x} = \texttt{maxint}$. Comme l'implantation utilise l'analyse modulaire, elle ne signalera aucune erreur à l'exécution. Dans ce cas, cet avertissement peut être ignoré par l'utilisateur final qui choisira de continuer selon diverses sémantiques concrètes possibles :

Sémantique concrète d'implémentation : d'un point de vue purement implémentatoire, l'exécution continue dans tous les cas $\texttt{x} \in [0,\texttt{maxint}]$, avec $\texttt{y} \in \{-\texttt{maxint} - 1\} \cup [1,\texttt{maxint}]$ (puisqu'une implantation modulaire de l'arithmétique entière ne signale pas l'erreur logique potentielle).

Ce choix peut conduire la suite de l'analyse à être polluée par les cas logiquement incorrects ($\texttt{y} = -\texttt{maxint} - 1$ dans notre exemple). Un tel comportement est en fait intentionnel dans certaines parties du programme (par exemple pour l'extraction de champs d'entiers non signés pour sélectionner des quantités volatiles fournies par le matériel, ce qui est logiquement correct malgré la circularité);

Sémantique concrète logique : d'un point de vue purement logique, l'exécution ne continue seulement que dans les cas non erronés où $\texttt{x} \in [0,\texttt{maxint} - 1]$, c'est-à-dire avec $\texttt{y} \in [1,\texttt{maxint}]$ (comme si l'implémentation avait signalé l'erreur logique).

On peut penser que ce point de vue purement logique serait correct du point de vue de l'implantation pour des programmes sans erreur (en supposant que les programmes ne seront pas exécutés tant qu'il n'aura pas été montré que tous les avertissements logiques fournis par l'analyseur sont des fausses alarmes donc impossibles à l'exécution). Mais ceci n'est pas le cas si le programmeur fait un usage explicite des caractéristiques du processeur (comme l'arithmétique modulaire). Par exemple la correction de certaines constructions du programme (comme l'extraction de champs) repose sur l'absence de débordement en arithmétique modulaire et donc, ignorer ce fait conduirait à la conclusion erronée que les points de programme qui suivent ne sont pas accessibles.

À cause du fait que certaines constructions (comme l'arithmétique d'entiers signés) requièrent d'utiliser une sémantique concrète logique et que d'autres constructions du programme (comme l'extraction de champs) requièrent d'utiliser une sémantique concrète d'implémentation, l'analyseur doit être paramétré de façon à laisser le choix définitif à l'utilisateur final (qui peut indiquer à l'analyseur quel type de sémantique doit être considéré, par exemple sur la base des types ou sur celle des opérations utilisées).

4 Manipulations préliminaires du programme

Pour réduire le coût ultérieur de l'analyse statique, l'analyseur commence par effectuer quelques manipulations préliminaires du programme. Comme le programme utilise des macros C et la sémantique des macros C n'est pas toujours très claire,

les macros sont expansées avant l'analyse de sorte que le programme analysé est le programme pré-processé. Ensuite tous les fichiers d'entrée sont regroupés en un seul fichier. Du fait que le programme est engendré automatiquement, il comporte de très nombreuses constantes symboliques de sorte d'une propagation classique des constantes est effectuée. Noter que les constantes flottantes doivent être évaluées dans le même mode d'arrondi qu'à l'exécution, en général au plus près, alors que pendant l'analyse, les opérations sur les intervalles doivent être sur-approximées de sorte qu'il faut considérer les hypothèses d'arrondi dans le pire des cas, ce qui assure que l'intervalle trouvé par l'analyse est au moins aussi grand que l'intervalle des variations possibles des valeurs flottantes en cours d'exécution. Cette propagation des constantes est étendue à une évaluation partielle [13] des expressions constantes y compris le cas particulier des accès à des tableaux constants avec un index constant. Ceci est particulièrement utile pour les tableaux contenant des indirections pour accéder à des adresses d'interfaces matérielles.

D'autres manipulations peuvent être spécifiées dans un fichier de configuration. On doit pouvoir spécifier les variables volatiles. Ces variables volatiles devraient être spécifiées dans le source, mais cette précision est parfois omise dans le source, ce qui est correct puisque le compilateur n'effectue aucune optimisation des accès mémoire, de sorte que la déclaration ou l'omission du caractère volatile des variables n'a aucun effet sur la compilation. Il est également possible de spécifier pour les variables volatiles un intervalle qui représente, par exemple, les valeurs possibles retournées par des capteurs. L'analyseur peut alors faire l'hypothèse que tous les accès à ces variables retournent une valeur dans l'intervalle indiqué. La première passe de l'analyseur insère cet intervalle comme un initialisateur spécial de ces variables. La syntaxe résultante est alors une extension de la syntaxe de C qui est prise en entrée des autres phases de l'analyseur.

L'utilisateur peut également déclarer dans le fichier de configuration des fonctions à ignorer. Ces fonctions sont analysées comme si elle avaient un corps vide. Si elles n'étaient pas déjà définies, alors elle sont définies avec un code vide. Si elles sont déjà définies alors leur code est supprimé. Cette déclaration a deux fonctions. La première est de prendre en compte des appels système pré-définis qui n'influencent pas le reste du comportement du programme. La deuxième est d'aider à détecter l'origine des erreurs détectées par l'analyseur : en ignorant certaines déclarations, on peut simplifier le programme et voir si l'analyseur trouve toujours la même erreur dans le programme simplifié. Cet usage, non prévu à l'origine, s'est montré très utile en pratique.

5 Structure de l'analyseur

Pour surestimer l'ensemble des états accessibles d'un programme bien structuré, l'analyseur procède par induction sur la syntaxe du programme. Comme le nombre de variables globales est très grand (environ 1 300 et 1 800 après expansion des tableaux, voir la Sec. 6.3) et que le programme est grand (environ 10 000 LOCs), un environnement abstrait ne pas pas être maintenu en chaque point du programme comme c'est le plus souvent le cas dans des analyseurs jouets ou pédagogiques [4]. Au lieu de cela, l'analyse procède par induction sur la syntaxe avec un environnement courant seulement. Les boucles sont traitées par un calcul de point fixe local avec

élargissement et rétrécissement [9]. Pendant cette itération, un environnement abstrait est maintenu en tête de boucle simplement. Par conséquent le nombre d'environnements abstraits qui doivent être maintenus en mémoire est de l'ordre du niveau d'imbrication des boucles. Quand le point fixe est atteint, une itération supplémentaire est effectuée pour détecter toutes les erreurs à l'exécution possibles de sorte que toutes les alarmes possibles sont levées bien qu'un environnement abstrait ne soit pas maintenu en chaque point du programme. Cependant des précautions particulières doivent être prises pour implanter ces environnements efficacement, comme indiqué dans la Sec. 6.2. De plus, il y a seulement quelques procédures définies par l'utilisateur et elles ne sont pas récursives, de sortes qu'elles peuvent être traitées de manière polyvariante (équivalente à un appel par recopie).

6 Domaines abstraits spécialisés

6.1 Construction itérative de l'analyseur

Nous avons commencé par des analyses classiques (comme par exemple une analyse d'intervalles pour l'arithmétique entière et flottante, en traitant les tableaux de manière classique par abstraction en un seul élément, etc.), ce qui, comme on pouvait s'y attendre après usage d'un analyseur général commercial par l'utilisateur final, conduit à des temps d'analyse inacceptables et à beaucoup trop de fausses alarmes.

Le développement de l'analyseur a ensuite suivi des cycles de raffinements itératifs. Une version de l'analyseur est essayée, émettant une trace d'exécution et une liste d'alarmes. Chaque alarme (ou mieux groupe d'alarmes liées) est ensuite inspectée manuellement en s'aidant de la trace d'exécution. L'objectif est de différencier entre des alarmes légitimes provenant par exemple d'une spécification trop imprécise des données en entrée du programme, des fausses alarmes levées à cause de la précision insuffisante de l'analyse. Quand un manque de précision est détecté, ses causes doivent être déterminées. Quand la cause de la perte de précision est comprise, des raffinements de l'analyseur peuvent être proposés.

Divers raffinements de l'analyseur étaient liés à l'augmentation de l'efficacité de l'analyse en mémoire et en temps qui furent améliorés par une nouvelle conception des algorithmes et des structures de données ou en sélectionnant des domaines abstraits moins précis quand l'excès inutile de précision conduisait à des coûts d'analyse trop grands.

Un compte-rendu de ces raffinements est fourni ci-dessous. Certains sont spécifiques à la classes de programmes considérée, mais d'autres sont d'un intérêt général pour de nombreux analyseurs — comme l'utilisation d'applications fonctionnelles (*functional maps*) représentés par des arbres de recherche équilibrés comme rapporté dans la section suivante Sec. 6.2.

6.2 Implantation fonctionnelle efficace des environnements abstraits à l'aide d'arbres binaires de recherche équilibrés

Un des domaines abstraits les plus simples est le domaine des intervalles [7] : un environnement abstrait fait correspondre à chaque variable entière ou réelle $\mathtt{x} \in V$ un intervalle de variation $\mathtt{X} \in I$. La sémantique abstraite des opérations arithmétiques

est alors l'arithmétique d'intervalles ordinaire. La borne supérieure et l'opération d'élargissement opèrent point par point (c'est-à-dire pour chaque variable). Plus généralement, nous considérons le cas plus général d'un environnement abstrait qui associe à chaque variable V une valeur abstraite prise dans un domaine abstrait I.

Une implémentation naïve et classique de ce domaine abstrait consiste à représenter des domaines abstraits par des tableaux d'éléments de I. Si des mises à jour destructives sont autorisées dans les fonctions abstraites de transfert (par exemple quand l'environnement représentant l'état avant l'opération peut être supprimé), les fonctions abstraites correspondant à des affectations sont faciles à implémenter ; sinon, un nouveau tableau doit être alloué.

Malgré toute sa simplicité, cette approche souffre d'un certain nombre d'inconvénients :

- elle requiert de nombreuses allocations de tableaux ; ceci peut surcharger le système d'allocation de mémoire puisque la plupart des données allouées a une durée de vie très courte ;
- plus ennuyeux sur la classe de programmes considérée, la complexité des opérations est prohibitive : le coût de l'opération de borne supérieure, qui est effectuée en sortie de chaque test dans le programme, est linéaire en le nombre de variable ; sur le type de programmes considérés, le nombre de variables statiques est linéaire en la longueur du programme, ce qui conduit à un coût quadratique.

Une analyse plus fine du programme montre que les opérations de borne supérieure sont effectués entre des environnements très similaires puisque ces environnements ne différent que sur un petit nombre de variables correspondant à des mises à jour différentes dans les deux branches du test. Ceci suggère d'utiliser une structure de données qui permette de représenter les similitudes entre des environnements et qui optimise l'opération de borne supérieure entre des environnements similaires.

Nous avons décidé d'implanter les correspondances de V vers I comme des arbres binaires de recherche équilibrés qui contiennent, en chaque nœud, le nom d'une variable de V et sa valeur abstraite dans I. Cette implantation est fourni par le module d'applications fonctionnelles Map d'Objective Caml [15]. Le temps d'accès à l'environnement pour lire ou mettre à jour un élément est logarithmique en le nombre de variables (tandis qu'avec des tableaux par exemple le temps d'accès serait en temps constant).

Un point important est que toutes les opérations sont effectuées purement fonctionnellement (sans effet de bord) ce qui permet d'utiliser de grandes structures de données partagées pour décrire des environnements liés. La nature fonctionnelle de l'approche conduit à une programmation plus facile de l'analyseur – il n'y a aucune nécessité de garder trace des structures de données pouvant ou ne pouvant pas être écrasées — et ce partage conduit à un faible usage de la mémoire.

Les arbres de recherche binaires équilibrés fournissent également un moyen très efficace de calculer les opérations binaires entre des environnements similaires quand l'opération $o : I \times I \to I$ satisfait $\forall x \in I,\ o(x,x) = x$. Ceci est vrai en particulier pour l'opération de borne supérieure et celle d'élargissement. Plus précisément, nous utilisons une fonction map2 définie comme suit : si f_1 et $f_2 : V \to I$ et $o : I \times I \to I$ satisfait $\forall x \in I,\ o(x,x) = x$, alors $\mathsf{map2}(o,f_1,f_2) = x \mapsto o(f_1(x),f_2(x))$. Cette fonction s'implémente par un parcourt récursif des arbres représentant f_1 et f_2 ;

quand f_1 et f_2 partage un sous-arbre commun, le résultat est ce même sous-arbre, qui peut être retourné immédiatement. La fonction `map2` doit uniquement parcourir des nœuds qui différent entre f_1 et f_2 — ce qui correspond à des chemins de la racine aux variables modifiées. Cette stratégie conduit à une complexité en temps $\mathcal{O}(m \log n)$ où m est le nombre de variables *modifiées* entre f_1 et f_2, et n est le nombre total de variables dans l'environnement ($\log n$ est la longueur maximale d'un chemin de la racine à une variable). Quand peu de variables dans les arbres binaires ont des valeurs différentes (comme c'est le cas par exemple quand on fusionne deux environnement en sortie de test par une borne supérieure), une très grande partie du calcul peut être optimisée grâce à cette technique.

En conclusion les applications fonctionnelles implantées en utilisant des arbres binaires de recherche équilibrés décroît considérablement la complexité pratique de l'analyseur.

6.3 Expansion des petits tableaux

Pour analyser une variable de type tableau, on peut simplement l'*expanser* dans l'environnement, c'est-à-dire, considérer une valeur abstraite dans I pour chaque index du tableau. On peut également le *compresser*, tous les éléments du tableau étant représentés par une seule valeur abstraite dans I qui représente toutes les valeurs possibles des éléments du tableau pour tous les index possibles.

Pour traiter les grands tableaux, la compression utilise moins de mémoire. Les fonctions de transfert sur des tableaux compressés sont également plus efficaces. Par exemple, l'affectation `tab[i] := exp` avec `i` $\in$ [0,99] conduit à 100 affectations abstraites si `tab` est expansé, et seulement une si `tab` est compressé.

L'expansion des tableaux conduit évidemment à une analyse beaucoup plus précise que la compression. Non seulement ils permettent de représenter des tableaux hétérogènes — comme par exemple des tableaux de flottants non nuls suivis d'un dernier élément nul pour marquer la fin de la partie utilisée dans le tableau — mais ils résultent également en mois de *mises à jour faibles* [4]. Par exemple, si le tableau à deux éléments `tab` est initialisé à zéro, puis est affecté par `tab[0] := 1; tab[1] := 1`, la compression du tableau résultera en des mises à jour faibles qui permettent de conclure que `tab[i]` $\in$ [0,1]. Le gain de précision pour les tableaux expansés est particulièrement intéressant quand il est combiné avec le déroulement sémantique de boucles (voir la Sec. 6.5).

Pour permettre à l'utilisateur d'équilibrer au mieux le rapport coût/précision de l'analyse de tableaux expansés ou compressés, l'analyseur est paramétré par la liste des tableaux devant être expansés soit en fournissant dans le fichier de configuration une taille de tableau (tous les tableaux de taille inférieure étant expansés) et/ou en les énumérant nominativement.

4. Une mise à jour faible dénote une affectation où une variable peut être ou ne pas être mise à jour, soit parce que la variable concernée n'est pas uniquement déterminée par l'analyseur (comme dans le cas d'une affectation à un élément de tableau expansé dont l'indice n'est pas exactement connu), soit parce que la variable affectée est compressée avec d'autres variables (comme dans le cas d'une affectation à un élément connu d'un tableau compressé).

6.4 Élargissements étagés par paliers

L'analyseur est paramétré par un fichier de configuration permettant à l'utilisateur de spécifier des raffinements locaux des domaines abstraits qui sont utilisés par l'analyseur. Un exemple est l'*élargissement étagé par paliers*.

L'élargissement classique sur les intervalles est $[a,b] \triangledown [c,d] = [(c < a ? -\infty : a),(d > b ? +\infty : b)]$ [5] [8]. Il est bien connu depuis longtemps que l'analyse d'intervalles avec cet élargissement peut être moins précise que l'analyse de signes, par exemple $[2,+\infty] \triangledown [1,+\infty] = [-\infty,+\infty]$ tandis que l'analyse de signes conduirait au résultat plus précis [0,+∞] (or [1,+∞] selon le choix du domaine abstrait [9]). Donc la plupart des élargissements sur les intervalles utilisent des paliers supplémentaires, tels que -1, 0 et +1. L'analyseur est paramétré par un fichier de configuration permettant à l'utilisateur de spécifier des paliers de son choix.

Ces paliers peuvent être choisis en comprenant l'origine de la perte de précision. Un exemple classique est le suivant (n est une constant entière donnée) :

```
int x;
x := 0;
while x <> n do
  x := x + 1;
end while
```

(tandis qu'en écrivant `x` $< n$, le rétressissement permettrait d'atteindre la borne n [7, 8]). Un exemple plus subtile est :

```
volatile boolean b;
int x;
x := 0;
while true do
  if b then
    x := x + 1;
    if x > n then
      x := 0;
    end if
  end if
end while
```

dans les deux cas, un élargissement en tête de boucle va extrapoler à $+\infty$ et le rétressissement qui suivra ne permettra de retrouver la borne constante n dans le corps de boucle. Ceci peut paraître surprenant puisque le rétressissement conduit à un résultat précis sur le morceau de code suivant :

```
int x;
x := 0;
while true do
  x := x + 1;
  if x > n then
    x := 0;
```

5. $(\text{true} ? a : b) = a$ tandis que $(\text{false} ? a : b) = b$.

end if

end while

Cependant, dans le premier cas, le test :

if `x` $> n$ **then**

`x` $:= 0$;

end if

peut ne pas être exécuté à chaque itération et donc dès que l'analyseur a surestimé l'intervalle de variation de `x`, il ne peut plus regagner d'information plus précise, même avec ce test. Une solution consisterai à demander à l'utilisateur de fournir une indication sous la forme d'une assertion `x` $\leq n$, dont l'analyse montre qu'elle est invariante et peut donc être éliminée. Une indication équivalente consiste à ajouter n comme palier d'élargissement. dans les deux cas l'élargissement ne conduit à aucune perte d'information. Si l'indication est erronée, l'assertion n'est pas vérifiée dans le premier cas et l'élargissement passe au palier suivant dans le deuxième cas. Une autre idée d'élargissements étagé consiste à utiliser des paliers en progressions arithmétiques, géométriques ou exponentielles susceptibles d'apparaître au cours des calculs du programme.

6.5 Déroulement sémantique des boucles

Bien que le logiciel analysé commence par un code d'initialisation avant la boucle principale, il reste des actions d'initialisation qui sont effectuées pendant la première itération de cette boucle principale. Le mécanisme utilisé pour ce faire est celui d'une variable booléenne globale qui est vraie au cours de la première itération de la boucle principale est fausse dans la suite.

Si l'analyseur cherche à calculer un invariant en tête de boucle avec des domaines abstraits non relationnels, alors ce booléen sera analysé comme pouvant avoir les valeurs *vrai* et *faux*, de sorte qu'il est impossible de distinguer entre la première itération d'initialisation et les suivantes. Pour pallier cette perte de précision, nous utilisons le déroulement sémantique de la boucle principale.

Le déroulement sémantique d'une boucle consiste, étant donné un facteur de déroulement n, à calculer les invariants I_0 qui est l'ensemble des valeurs possibles des variables avant de commencer la boucle, puis I_k, $1 \leq k < n$ qui est l'ensemble des valeurs possibles des variables après exactement k itérations, et finalement J_n qui est l'ensemble des valeurs possibles des variables après n itérations ou plus. Ensuite, la disjonction de $I_0, \ldots, I_{n-1}, J_n$ par l'opération de borne supérieure abstraite est utilisée pour calculer l'invariant de fin de boucle.

Un point de vue équivalent consiste à analyser la boucle **while** B **do** C comme **if** B **then** (C; **if** B **then** ($\ldots$ **if** B **then** (C; (**while** B **do** C))$\ldots$)). Cette technique est plus précise que l'analyse classique des boucles **while** quand les fonctions de transfert abstraites ne sont pas complètement distributives ou quand on utilise des élargissements.

Dans notre cas, la première itération de la boucle principale est une phase d'initialisation qui se comporte très différemment des itérations suivantes. Par conséquent en choisissant $n = 1$, l'invariant J_n est calculé en prenant en compte les valeurs des variables un fois initialisées, de sorte que le résultat est plus précis et permet d'éliminer certaines fausses alarmes.

6.6 Partitionnement de traces

La raison pour laquelle le déroulement sémantique de boucles est plus précis est que, pour chaque déroulement de boucle, un nouvel ensemble de valeurs est approché. Par conséquent au lieu d'avoir un seul ensemble de valeurs, nous avons une collection d'ensemble de valeurs, chacun d'entre eux étant approché séparément ce qui est plus précis que d'approcher leur union qui conduit à une approximation imprécise par l'opération de borne supérieure dans le domaine abstrait. L'analyse serait également plus précise si les diverses collections d'ensembles de valeurs n'étaient pas fusionnées à la fin de la boucle mais ultérieurement.

Considérons par exemple l'algorithme suivant, qui calcule une interpolation linéaire :

```
t = {-10, -10, 0, 10, 10};
c = {0, 2, 2, 0};
d = {-20, -20, 0, 20};
i := 0;
while i < 3 and x ≥ t[i+1] do
  i := i+1;
end while
r := (x - t[i]) × c[i] + d[i];
```

La variable résultat `r` prend ses valeurs dans $[-20, 20]$, mais si nous utilisons une analyse d'intervalles standard, le résulta sera l'intervalle de variation $[\min(-20, 2x^- - 40),\ \max(20, 2x^+ + 40)]$ (où `x` est dans $[x^-, x^+]$). Cette perte de précision vient du fait que l'analyse d'intervalles n'est pas distributive. C'est le cas même avec un déroulement sémantique de la boucle parce que lorsqu'on arrive à l'instruction où `r` est calculé, tous les déroulements sont fusionnés de sorte que la relation entre les valeurs de `i` et `x` sont perdues.

Le partitionnement de traces consiste à retarder les fusions qui peuvent avoir lieu dans les fonctions de transfert. De telles fusions, calculées dans l'abstrait par une borne supérieure, apparaît à la fin des deux branches d'un **if**, ou à la fin d'une boucle **while** quand il y a eu un déroulement sémantique. Le partitionnement de traces basé sur le contrôle d'exécution a été introduit par [11]. Le partitionnement de traces est plus précis pour des domaines abstraits non distributifs mais ils peuvent être très coûteux car ils multiplient le nombre d'environnements par 2 pour chaque **if** qui est partitionné et par k pour chaque boucle déroulée k fois. La situation est encore pire dans le cas d'un partitionnement de traces à l'intérieur d'une boucle partitionnée.

Nous avons donc amélioré la technique de [11] en autorisant le partitionnement temporaire : la fusion n'est pas repoussée à jamais mais jusqu'à un certain point paramétrable. Ceci fonctionne bien pour fusionner des partitions créent à l'intérieur d'une fonction juste avant le point de retour ou pour fusionner juste à la fin de la boucle des partitions créent *à l'intérieur* de cette boucle. En pratique cette technique semble une bonne alternative au produit cardinal réduit classique de [9] qui est plus complexe à mettre en œuvre.

6.7 Relation entre variables et compteurs de boucles

Comme expliqué en début de cette section, les domaines non relationnels, tels que le domaine des intervalles, peuvent être implantés efficacement. Cependant les domaines non relationnels peuvent être parfois insuffisants, même quand il s'agit de borner les valeurs possibles des variables. Considérons l'exemple de la boucle suivante :

```
volatile boolean b;
i := 0;
x := 0;
while i < 100 do
  x := x + 1;
  if b then
    x := 0;
  end if
  i := i + 1;
end while
```

Pour découvrir que $\mathtt{x} < 100$, il faut d'abord découvrir la relation invariante $\mathtt{x} \leq \mathtt{i}$. Des élargissements étagés par paliers sont inopérant ici car x n'est jamais explicitement comparé à 100. Le passage à un domaine complètement relationnel (comme les polyèdres ou même plus simplement les relations d'égalité linéaire) est clairement impossible à cause du très grand nombre de variables vivantes dans l'application (de fait, même les domaines abstraits relationnels seraient très coûteux sans la technique de représentation présentée à la Sec. 6.2).

Notre solution consiste à ne considérer que des relations entre les variables et un compteur de boucle δ (qui est explicite dans une boucle **for** et implicite dans une boucle **while**). Nous dénotons par Δ l'intervalle de variation de ce compteur δ (Δ est soit déterminé par l'analyse soit spécifié par l'utilisateur final dans le fichier de configuration, par exemple quand l'application est conçue pour fonctionner pendant un certain temps maximum). Au lieu d'associer un intervalle $\mathtt{X}$ à chaque variable x, nos invariants plus précis associent trois intervalles : $\mathtt{X}$, $\mathtt{X}^+$ et $\mathtt{X}^-$ qui sont, respectivement, les valeurs possibles de x, de $\mathtt{x} + \delta$, et de $\mathtt{x} - \delta$.

Quand trop d'information est perdue sur l'intervalle $\mathtt{X}$ (par exemple après un élargissement), $\mathtt{X}^+$, $\mathtt{X}^-$, et Δ sont utilisés pour récupérer de l'information à l'aide d'un *opérateur de réduction* (voir la Sec. 6.8 ci-dessous), qui remplace l'intervalle $\mathtt{X}$ par l'intervalle $\mathtt{X} \cap (\mathtt{X}^+ - \Delta) \cap (\mathtt{X}^- + \Delta)$. C'est une façon simple d'abstraire les variations de valeur de la variable au cours du temps (lui-même abstrait par le compteur de boucle).

En pratique, ce domaine relationnel est implanté comme un domaine non relationnel en utilisant les structures de données de la Sec. 6.2. Ceci augmente très grandement la précision de l'analyse pour un petit facteur de coût en temps et en mémoire.

6.8 La réduction et son interaction avec l'élargissement

Pour prendre en compte les relations entre les valeurs des variables du programme et les compteurs de boucles, l'analyseur utilise un opérateur de réduction ρ qui est

un endomorphisme conservatif (c'est-à-dire qu'au sens de l'interprétation abstraite tel que $\gamma(d) \subseteq \gamma(\rho(d))$). La façon d'utiliser cette réduction a un grand impact, non seulement sur la précision de l'analyse, mais également sur sa complexité : d'un côté il est crucial de faire la réduction avant de calculer une fonction de transfert (tester une garde par exemple) pour gagner en précision ; de l'autre côté, le coût de la réduction ne doit pas excéder le coût de calcul de la fonction de transfert elle-même.

Notre choix a été de faire les réductions au vol, à l'intérieur de chaque primitive abstraite. Ceci nous permet d'appliquer seulement la réduction aux variables du programme pour lesquelles ceci est strictement nécessaire. C'est très simple pour les opérateurs unaires. Quant aux opérateurs binaires, nous détectons quelle partie du résultat doit être réduite grâce à l'implantation du domaine abstrait par des applications fonctionnelle, ce qui conduit à une implantation sous-linéaire de la réduction — qui coïncide au coût amorti des fonctions de transfert abstraites.

Le seul véritable problème avec cette approche est que la réduction peut détruire l'extrapolation construite par élargissement [6]. Habituellement l'opérateur de réduction ne peut pas être appliqué aux résultats d'un élargissement. Quelques solutions existent déjà, mais elle ne sont pas compatibles avec notre exigence d'avoir une implémentation sous-linéaire de la réduction.

Pour résoudre ce problème, nous imposons des conditions non standards sur la réduction : nous remplaçons l'intervalle $\mathtt{X}$ par l'intervalle $\mathtt{X} \cap (\mathtt{X}^+ - \Delta) \cap (\mathtt{X}^- + \Delta)$, mais nous ne faisons pas la propagation dans l'autre sens (par exemple, nous ne remplaçons pas $\mathtt{X}^+$ par $\mathtt{X}^+ \cap (\mathtt{X} + \Delta)$, ni $\mathtt{X}^-$ par $\mathtt{X}^- \cap (\mathtt{X} - \Delta)$). Ceci permet à l'extrapolation d'être d'abord effectuées sur les intervalles Δ, $\mathtt{X}^+$, et $\mathtt{X}^-$. Une fois que les itérés sur les intervalles Δ, $\mathtt{X}^+$, et $\mathtt{X}^-$ se sont stabilisés, l'extrapolation de l'intervalle $\mathtt{X}$ n'est plus perturbée.

6.9 Sur l'analyse de l'arithmétique flottante

Une difficulté majeure de l'analyse de l'arithmétique flottante est qu'il faut tenir compte des erreurs d'arrondi, aussi bien dans la sémantique du programme analysé que dans l'analyseur lui-même. Il faut bien considérer que :

- les fonctions de transfert doivent modéliser l'arithmétique flottante, c'est-à-dire (selon le standard IEEE [2]), l'arithmétique réelle en précision infinie suivie par une phase d'arrondi ;
- les opérateurs abstraits doivent être eux-mêmes implantés en utilisant l'arithmétique flottante (pour des raisons d'efficacité le calcul flottant en précision infinie, les rationnels ou l'arithmétique algébrique sont exclus).

En particulier de grandes précautions doivent être prises pour tenir compte du fait la la plupart des égalités mathématiques classiques (associativité, distributivité, etc.) ne sont pas valides quand ces opérations sont traduites en arithmétique flottante ; il est nécessaire de connaître en chaque point du calcul si les quantités considérées sont des bornes supérieures ou inférieures.

L'arithmétique d'intervalles est relativement aisée : les opérations sur les bornes inférieures sont arrondies vers $-\infty$, tandis que les opérations sur les bornes supérieures sont arrondies vers $+\infty$. Une complication supplémentaire est due à l'utilisation

6. Le lecteur peut consulter la Fig. 3 de [16] pour avoir une illustration de ce problème dans le contexte d'un domaine relationnel.

des variables de type `float` — en simple précision IEEE — dans les programmes analysés : les opérations abstraites correspondantes doivent également être arrondies en arithmétique simple précision IEEE.

7 Voies sans issues et idées erronées

L'analyseur a évolué au travers de trois versions successives à causes de diverses voies qui se sont révélées être sans issue et pour permettre de faire des expérimentations pour déterminer quelles étaient les abstractions adéquates. Dans cette section nous discutons un certain nombre de décisions initiales qui se sont révélées erronées car trop coûteuses ou trop imprécises et qui ont été corrigées dans les versions suivantes.

Syntaxe. Une première idée qui n'était pas bonne était de considérer une syntaxe des programmes C correspondant exactement à la classe de programmes automatiquement générés que nous considérons. L'idée était de tester syntaxiquement des erreurs potentielles par exemple dans les macros. En plus d'une complexité plus grande de la grammaire, il s'est avéré impossible de tester l'analyseur avec des exemples simples. Finalement l'expansion des macros avant l'analyse et l'usage d'une grammaire de C standard avec des actions sémantiques qui testent que les restrictions locales sont satisfaites s'est avérée plus productive dans les versions suivantes.

Représentation des environnements par des tableaux fonctionnels. Les premières versions de l'analyseur utilisaient les tableaux fonctionnels de CAML pour représenter les environnements abstraits. Comme nous l'avons montré dans la Sec. 6.2, l'idée qu'un temps d'accès aux valeurs abstraites des variables en $\mathcal{O}(1)$ rend les analyses non relationnelles efficaces est erronée.

Analyse de vivacité. Le nombre de variables globales étant très grand, on peut penser qu'une analyse préliminaire de vivacité serait utile pour éliminer les mises à jours inutiles des valeurs abstraites dans les environnements représentés comme des tableaux. Le gain était en fait négligeable.

Des idées similaires basées sur des représentations intermédiaires utilisées dans les analyses de flots de données (comme les chaînes utilisation-définition (*use-def chains*), l'affectation statique unique (*SSA*, single static assignment), etc auraient probablement été également ineffectives. L'idée centrale a été d'utiliser des applications fonctionnelles représentées par des arbres binaires de recherche équilibrés, comme expliqué à la Sec. 6.2.

Intervalles flottants ouverts. La première version de l'analyseur utilisait des intervalles flottants ouverts et fermés. Pour être correct, les intervalles devaient prendre en compte les erreurs d'arrondi (comme expliqué à la Sec. 6.9), ce qui rend l'analyse très complexe sans amélioration de la précision et par conséquent l'idée d'utiliser des intervalles flottants ouverts a été abandonnée.

Domaines relationnels flottants. La plus grande part de la littérature considère seulement des domaines relationnels sur des anneaux, tels que les rationnels ou les réels et ne prend pas en compte les problèmes posés par l'usage de l'arithmétique flottante. En programmant avec soin, il serait possible de concevoir une approximation sûre de l'arithmétique réelle utilisant l'arithmétique flottante : chaque calcul doit être arrondi de sorte que le résultat soit toujours plus large, de façon à préserver la correction de l'approximation supérieure. Ensuite chaque opérateur flottant abstrait peut être implanté comme un opérateur abstrait réel suivi d'un arrondi abstrait qui ajoute simplement au résultat un intervalle représentant l'erreur absolue — ou, plus précisément l'*ulp* (*Unit in the Last Place*) [10]. Cependant cette approche grossière de l'arrondi peut conduire un élément abstrait à dériver à chaque itération, ce qui interdit sa stabilisation par élargissement. Aucune solution vraiment satisfaisant n'a encore été trouvée pour résoudre ce problème, ni d'ailleurs celui de de la complexité en temps et en espace inhérente aux domaines abstraits relationnels, qui ne sont donc pas utilisés dans le prototype.

Analyse de cas. L'analyse de cas est un raffinement classique en analyse statique. Par exemple [9, Sec. 10.2] illustre le produit cardinal réduit de domaine abstraits par une analyse de cas sur une variable booléenne, l'analyse étant décomposée selon les cas *vrai* et *faux*. Des implantations sur plusieurs variables booléennes peuvent être basées sur des BDDs (Boolean Decision Diagrams). De la même façon les valeurs abstraites peuvent être décomposées en fonction de diverses valeurs concrètes des variables (comme des intervalles en sous-intervalles). Ces raffinements se sont avérés ineffectifs car les coûts explosent exponentiellement au fur et à mesure de l'introduction de nouvelles décompositions par cas pour obtenir des gains de précision. Par conséquent, nous avons finalement remplacé l'analyse de cas par le partitionnement de trace, comme présenté à la Sec. 6.6.

Du prototypage. Les premières versions de l'analyseur peuvent être comprises comme des prototypes initiaux pour aider à découvrir les abstractions devant être utilisées. La réécriture complète des versions successives par des personnes différentes a permis d'éviter l'accumulation des niveaux, corrections, verrues, traductions successives qui au fil du temps rendent les grands programmes incompréhensibles et inefficaces.

8 Performances

Tout le problème de l'analyse statique est de trouver le bon équilibre entre le coût et la précision de l'analyse. Dans le cas de la vérification de programme, la précision doit être suffisante pour obtenir aucune fausse alarme. Une fois que la précision est suffisante pour n'avoir aucune fausse alarme, il reste à estimer (voire à améliorer) les performances de l'analyse. Pour estimer les performances en temps et en mémoire, diverses expériences ont été conduites soit en utilisant des parties de programme soit en utilisant des produits synchrones du programme plusieurs fois avec lui-même.

La mémoire utilisée pour ranger les environnements abstraits croît linéairement avec le nombre de variables, qui est lui-même proportionnel, pour l'application considérée, à la taille ℓ du programme lui-même. À cause des boucles imbriquées, du

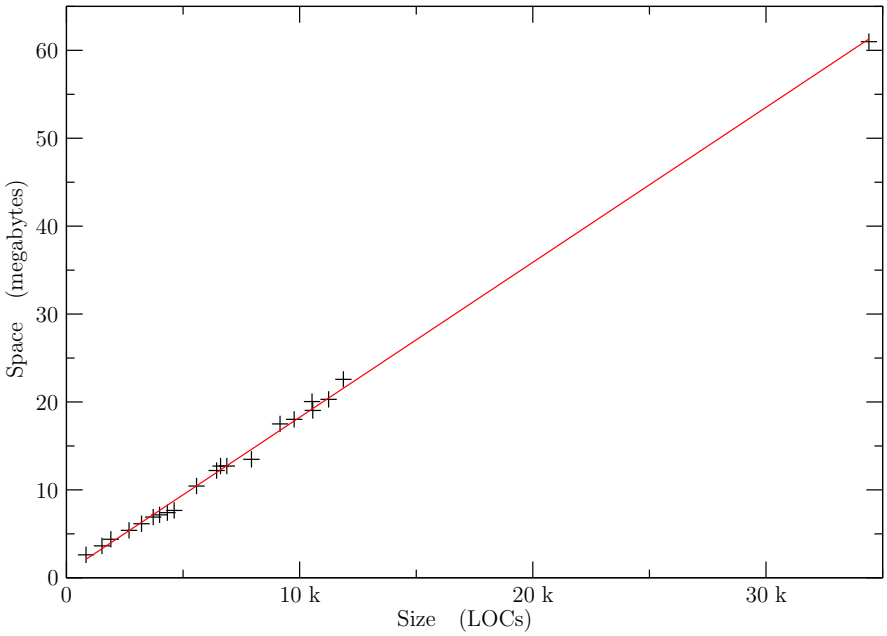

FIG. 1 – *Usage mémoire en fonction de ma taille du programme analysé.*

dépliage de boucles et du partitionnement de traces, l'analyseur peut avoir besoin de mémoriser plusieurs environnements abstraits pendant l'analyse. Cependant, grâce à l'usage d'applications fonctionnelles représentées par des arbres binaires de recherche équilibrés et partagés, une grande partie de ces environnements est partagée, ce qui réduit considérablement l'encombrement mémoire. Nous avons trouvé expérimentalement (voir la Fig. 1[7]) que la consommation mémoire maximale de l'analyseur est de fait de l'ordre de $\mathcal{O}(\ell)$. Pour notre application, il s'agit de quelques mégaoctets, ce qui ne pose aucun problème avec les calculateurs modernes.

Grâce à l'usage d'applications fonctionnelles (Sec. 6.2), le coût amorti des opérations abstraites élémentaires peut être estimé comme étant au plus de l'ordre du temps d'accès aux valeurs abstraites des variables, qui est en $\mathcal{O}(\ln v)$, où v est le nombre de variables du programme. Dans le programme considéré pour notre expérimentation, le nombre de variables du programme est lui-même proportionnel au nombre ℓ de lignes de code (LOCs). Par conséquent, le coût des opérations abstraites élémentaires est en $\mathcal{O}(\ln \ell)$. Une itération de point fixe balaye tout le programme. Puisque l'analyse abstraite des procédures est sémantiquement équivalente à une expansion (Sec. 5), chaque pas d'itération du point fixe prend $\mathcal{O}(\ell' \times \ln \ell \times p \times i')$ où ℓ' est le nombre de lignes de code (LOCs) après expansion des procédures, p est une borne sur le nombre d'environnements abstraits qui doivent être traités en chaque point donné du programme[8], et i' est une borne sur le nombre d'itérations des points fixes intérieurs. Le calcul de points fixes est alors de l'ordre de $\mathcal{O}(i \times p \times i' \times \ell' \times \ln \ell)$

7. Le meilleur ajustement de la courbe (*best curve fitting*) [3] avec la formule $y = a + bx$ et une tolérance de 10^{-6} donne $a = 0{,}623994$ et $b = 0{,}00176291$ pour une adéquation estimée par une distance du chi-deux de 6,82461, un coefficient de corrélation de 0,951324, une régression des moindres carrés (RMC) avec un taux d'erreur de 0,0721343 pour cent et un coefficient d'inégalité de Theil (U) de 0,0309627.

8. Ce nombre ne dépend que du dépliage des boucles et du partitionnement des traces.

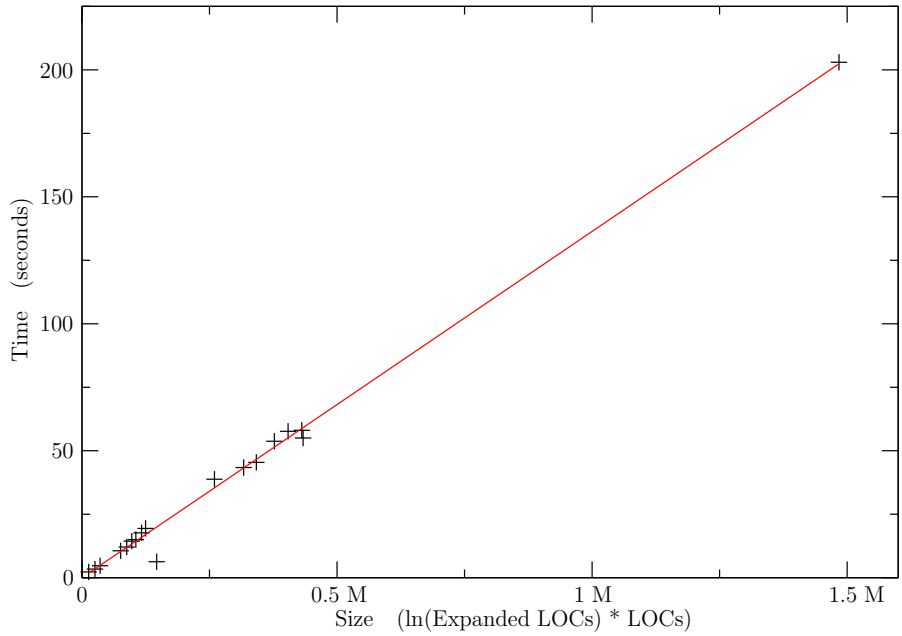

FIG. 2 – *temps d'exécution en fonction de $\ell' \times \ln \ell$, où ℓ' est la taille du programme analysé après expansion et ℓ est sa taille.*

où i est une borne sur le nombre d'itérations.

Nous devons maintenant estimer les bornes p, i, and i'. Le nombre p ne dépend que de paramètres fournis par l'utilisateur final. Les nombres i et i' sont au pire en $\mathcal{O}(\ell \times t)$ où t dénote le nombre de paliers, mais sont constants en pratique. Par conséquent, le temps d'exécution devrait être de l'ordre de $\mathcal{O}(\ell' \times \ln \ell)$. Ces hypothèses se confirment expérimentalement par meilleur ajustement de courbe [3] sur diverses expérimentation du temps d'exécution de l'analyseur. La formule d'ajustement [9] $y = ax$ donne $a = 0{,}000136364$, comme le montre la Fig. 2.

Le facteur d'expansion des procédures qui donne ℓ' en fonction de la taille ℓ du programme a été également déterminée expérimentalement, voir la Fig. 3. Le meilleur ajustement de courbe avec la formule [10] $y = a \times x \times (\ln x)^b$ donne $a = 0{,}927555$, $b = 0{,}638504$. Ceci montre que pour la famille de programmes considérés, l'analyse polyvariante de procédures (équivalente à une sémantique d'appel par copie), dont il est bien connu qu'elle est plus précise et beaucoup plus coûteuse qu'une analyse monovariante (où tous les appels possibles sont fusionnés), a un coût raisonnable.

Par composition nous obtenons un temps d'exécution de l'analyseur de l'ordre de $\mathcal{O}(\ell(\ln \ell)^a)$ où ℓ est la taille du programme. Cette hypothèse se confirme expérimentalement par meilleur ajustement de la courbe des temps d'exécution de l'analyseur statique obtenue par expérimentation avec divers programmes de taille différente. La

9. avec une qualité de l'ajustement estimée par une distance du chi-deux de 239,67, un coefficient de corrélation de 0,941353, RMS avec taux d'erreur de 0,515156 pour cent et un coefficient d'inégalité de Theil (U) de 0,0628226 pour une tolérance de 10^{-6}.

10. avec une qualité de l'ajustement estimée par une distance du chi-deux de $7{,}00541 \times 10^{07}$, un coefficient de corrélation de 0,942645, RMS avec taux d'erreur de 0,113283 pour cent et un coefficient d'inégalité de Theil (U) de 0,0464639 pour une tolérance de 10^{-6}.

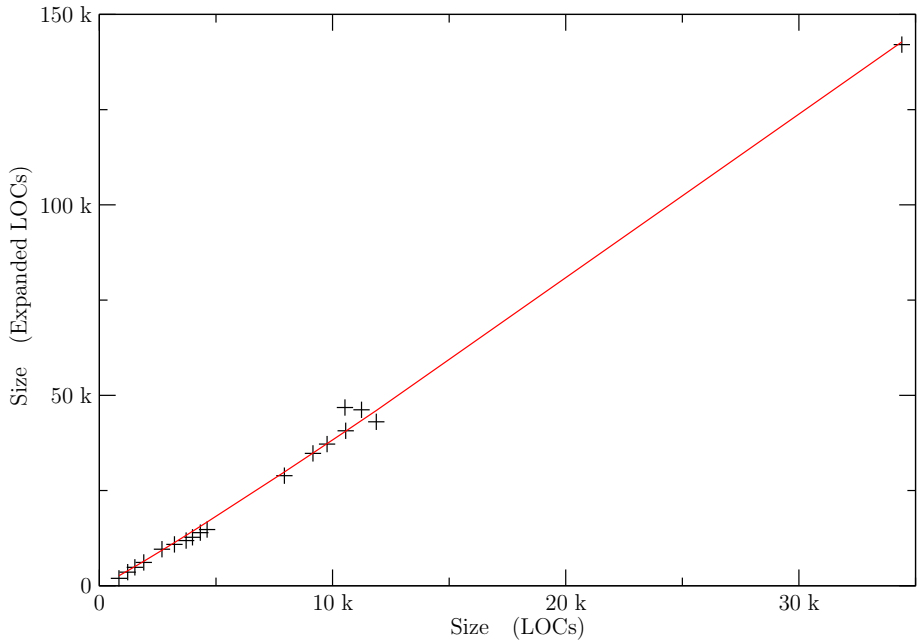

FIG. 3 – *Taille ℓ' du programme après expansion des procédures en fonction de la taille ℓ du programme original.*

formule d'ajustement non-linéaire[11] $y = a+bx+cx(\ln x)^d$ donne $a = 2{,}2134\times10^{-11}$, $b = 5{,}16024\times10^{-08}$, $c = 0{,}00015309$ et $d = 1{,}55729$, voir la Fig. 4.

Les performances de l'analyseur statique mesurées en occupation mémoire et en temps d'exécution montrent, selon l'extrapolation donnée dans la Fig. 5, qu'une précision extrême n'est pas incompatible avec une analyse efficace. Par conséquent on peut s'attendre à ce que de tels analyseurs statiques spécifiques soient utilisables de manière courante pour vérifier l'absence d'erreurs à l'exécution pendant le développement, le test et la maintenance de programmes. Grâce à la paramétrisation, l'utilisateur final pourra facilement ajuster l'analyse pour tenir compte de petites modifications du programmes comme par exemples celles correspondant à des versions successives.

9 Conclusion

À notre première lecture du programme, nous étions assez pessimistes sur les chances de succès avec un objectif ambitieux d'absence de fausses alarmes car les calculs numériques qui représentent 80 % du programme, ce qui n'est pas surprenant pour un programme de contrôle non linéaire, semblaient à la fois complexes et presque incompréhensibles pour le néophyte. Le fait que le code soit engendré automatiquement n'était pas fait pour aider. L'utilisation de domaines abstraits numériques complexes aurait été terriblement coûteuse ; Par conséquent, le critère de conception essentiel a toujours été le plus simple, c'est-à-dire le plus abstrait, sera le mieux,

11. avec une qualité de l'ajustement estimée par une distance du chi-deux de 40,1064, un coefficient de corrélation de 0,956011, RMS avec taux d'erreur de 0,0595795 pour cent et un coefficient d'inégalité de Theil (U) de 0,0248341 pour une tolérance de 10^{-6}.

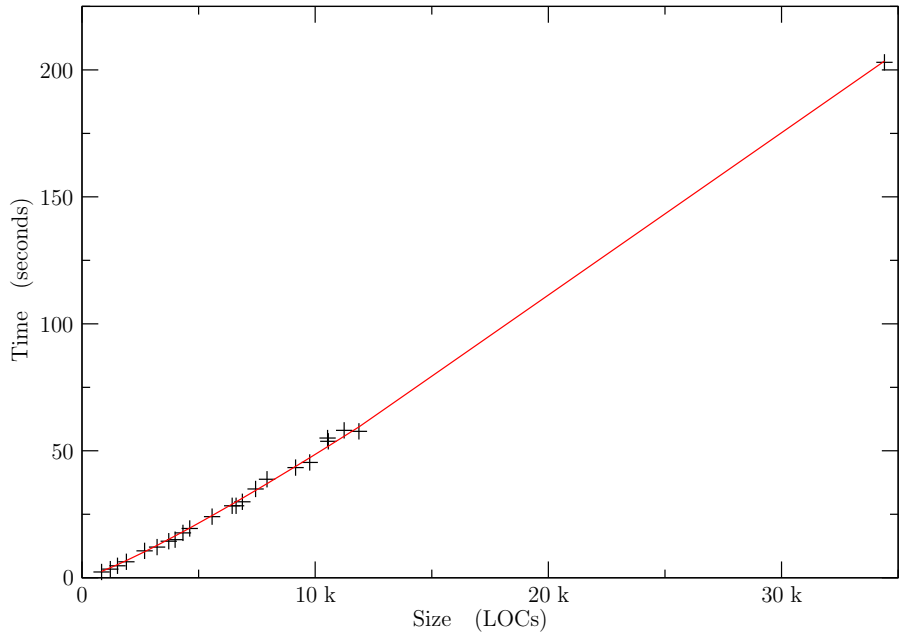

FIG. 4 – *temps d'exécution en fonction de la taille du programme analysé.*

c'est-à-dire le plus efficace.

À cause de l'indécidabilité, des indications humaines sont nécessaires pour être capable d'analyser un programme complètement automatiquement :

- dans les méthodes déductives ceci se fait en fournissant des hypothèses inductives (comme par exemple des invariants) aussi bien que des indications pour la stratégie ou le choix des tactiques de preuve ;
- en vérification exhaustive (*model-checking*), ceci se fait en fournissant un modèle fini du programme ;
- pour l'analyse statique, nous avons montré sur un exemple non trivial que ceci peut être fait en fournissant des indications sur le choix local des domaines abstraits et des élargissements.

Dans tous les cas, une bonne compréhension des méthodes de vérification est nécessaire. Nous avons le sentiment que des indications fournies à un analyseur paramétré sont beaucoup plus simples à fournir que des invariants correctes ou des modèles de programmes. Une fois que des spécialistes ont conçu un analyseur statique pour une classe de programmes, le processus de raffinement local par paramétrisation est relativement aisé pour les utilisateurs finaux qui ne sont pas spécialistes en analyse statique.

Nous avons de sérieux doutes quant à la possibilité d'automatiser complètement le processus de raffinement. Un raffinement basé sur des contre-exemples pour traiter les fausses alarmes ne pourrait raffiner des domaines abstraits qu'en procédant élément abstrait par élément abstrait, où ces éléments abstraits se réfèrent directement à des éléments concrets. Dans une telle approche, la taille de l'analyse raffinée ne peut que croître exponentiellement. Clairement, une inférence non triviale et une réécriture significative de l'analyseur est nécessaire pour passer de contre-exemples à des techniques d'abstraction comme le partitionnement ou au domaine relationnel

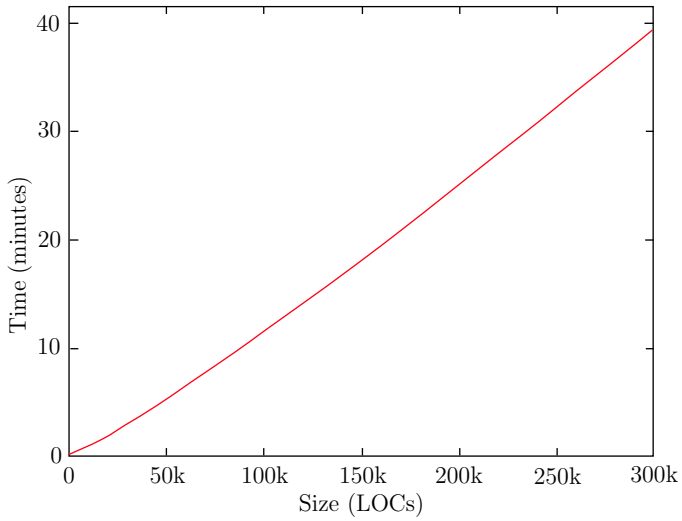

FIG. 5 – *Extrapolation du temps d'exécution en fonction de la taille du programme analysé* ($y = 2{,}2134 \times 10^{-11} + 5{,}16024 \times 10^{-8} \times x + 0{,}00015309 \times x \times (\ln x)^{1{,}55729}$).

prenant en compte les compteurs de boucles (ceci étant d'autant difficile qu'un raffinement de la sémantique concrète est nécessaire où le temps est abstrait par un compteur de boucle).

Par conséquent, notre approche pour obtenir aucune fausse alarme a été de concevoir un analyseur statique spécialisé qui est paramétré pour permettre à un utilisateur final non spécialiste de choisir les raffinements spécifiques qui doivent être appliqués pour traiter un programme quelconque appartenant à la famille considérée.

Le projet continue maintenant avec des programmes réels beaucoup plus grands (environ 250 000 LOCs) . L'estimation des ressources nécessaires donnée dans les Figures 1 et 5 a été confirmée. De façon non surprenante, de nouvelles fausses alarmes sont apparues car les calculs flottants dans ces programmes sont beaucoup plus complexes que ceux mis en œuvre dans la première expérimentation dont nous faisons état. Un nouveau cycle de raffinement doit donc être redémarré pour concevoir de nouveau domaines abstraits qui sont définitivement nécessaires pour atteindre l'objectif d'aucune fausse alarme avec un coût d'analyse assez bas.

Références

[1] J.-R. Abrial. On B. *In : Proc. 2nd Int. B Conf., B'98: Recent Advances in the Development and Use of the B Method*, éd. par D. Bert, pp. 1–8. Springer-Verlag, 1998.

[2] American National Standards Institute, Inc. *IEEE Standard for Binary Floating-Point Arithmetic*. Rapport technique n 754-1985, ANSI/IEEE, 1985. `http://grouper.ieee.org/groups/754/`.

[3] P. R. Bevington et D. K. Robinson. *Data Reduction and Error Analysis for the Physical Sciences*. McGraw-Hill, 1992.

[4] P. Cousot. The Marktoberdorf'98 generic abstract interpreter. Nov. 1998. `http://www.di.ens.fr/~cousot/Marktoberdorf98.shtml`.

[5] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. *« Informatics — 10 Years Back, 10 Years Ahead »*, éd. par R. Wilhelm, pp. 138–156. Springer-Verlag, 2000.

[6] P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. *Proc. 4^{th} Int. Symp. SARA '2000*, éd. par B.Y. Choueiry et T. Walsh, pp. 1–25. Springer-Verlag, 26–29 jul. 2000.

[7] P. Cousot et R. Cousot. Static determination of dynamic properties of programs. *Proc. 2^{nd} Int. Symp. on Programming*, pp. 106–130. Dunod.

[8] P. Cousot et R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *4^{th} POPL*, pp. 238–252. Los Angeles, CA, 1977.

[9] P. Cousot et R. Cousot. Systematic design of program analysis frameworks. *6^{th} POPL*, pp. 269–282. San Antonio, TX, 1979.

[10] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, vol. 23, n 1, mars 1991, pp. 5–48.

[11] M. Handjieva et S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. *Proc. 5^{th} Int. Symp. SAS '98*, éd. par G. Levi, pp. 200–214. Springer-Verlag, 1998.

[12] G.J. Holzmann. Software analysis and model checking. *Proc. 14^{th} Int. Conf. CAV '2002*, éd. par E. Brinksma et K.G. Larsen, pp. 1–16. Springer-Verlag, 2002.

[13] N. Jones, C.K. Gomard et P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, juin 1993, *Int. Series in Computer Science.*

[14] JTC 1/SC 22. *Programming languages — C.* Rapport technique, ISO/IEC 9899:1999, 16 déc. 1999.

[15] X. Leroy, D. Doligez, J. Garrigue, D. Rémy et J. Vouillon. *The Objective Caml system, Documentation and user's manual (release 3.06).* Rapport technique, INRIA, Rocquencourt, 19 août 2002. `http://caml.inria.fr/ocaml/`.

[16] A. Miné. A new numerical abstract domain based on difference-bound matrices. *Proc. 2^{nd} Symp. PADO '2001*, éd. par 0. Danvy et Filinski (A.), pp. 155–172. Springer-Verlag, 2001. `http://www.di.ens.fr/~mine/publi/article-mine-padoII.pdf`.

[17] S. Owre, N. Shankar et D.W.J. Stringer-Calvert. PVS: An experience report. *PROC Applied Formal Methods - FM-Trends'98, International Workshop on Current Trends in Applied Formal Method*, éd. par D. Hutter, W. Stephan, P. Traverso et M. Ullmann, pp. 338–345. Springer-Verlag, 1998.

Cet article est la traduction française de :

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer-Verlag, 2002.