Renforcement du Noyau d'un Démonstrateur SMT

Conception et Implantation de Procédures de Décisions Efficaces

---

Strengthening the Heart of an SMT-Solver

Design and Implementation of Efficient Decision Procedures

Les systèmes informatiques sont maintenant utilisés dans divers domaines, comme la médecine, les transports et les centrales nucléaires. Une erreur dans ces systèmes peut avoir des conséquences allant de la gêne occasionnelle à des pertes financières ou humaines. Par exemple, un bogue de conception [45] dans le microprocesseur PENTIUM 4 a coûté à INTEL 500 millions de dollars. L'explosion d'ARIANE 5, en raison d'un bogue [35] dans son système de navigation, a coûté 370 millions de dollars. En outre, plusieurs patients sont décédés [34] après avoir reçu une dose anormale de radiations, en raison d'un bogue logiciel dans un appareil de radiothérapie.

Plusieurs méthodes formelles ont été conçues dans le but d'améliorer la fiabilité des systèmes informatiques. Ces approches comprennent le test [57], le model checking [3, 22] et la vérification déductive [12, 27, 5]. Les techniques de test sont très populaires dans l'industrie, car elles sont automatiques et passent bien à l'échelle. Les model checkers sont maintenant utilisés dans la conception de matériel [3, 43, 31], et la vérification déductive est utilisée dans le développement de logiciels [55, 4].

Les techniques mentionnées ci-dessus génèrent un grand nombre de formules mathématiques. Dans le domaine du test, ces formules sont utilisées pour caractériser les chemins faisables dans les graphes de flot de contrôle. Dans le model checking, elles encodent les états des systèmes et les relations de transition. Dans la vérification déductive, elles sont générées à partir de programmes annotés avec leurs spécifications formelles.

En général, les formules issues de la vérification de logiciels contiennent des dizaines d'hypothèses closes et des centaines d'axiomes quantifiés. Mais, seules quelques instances de certains axiomes sont nécessaires pour prouver leur validité. En outre, ces formules imbriquent — d'une manière non triviale — les connecteurs booléens avec certaines théories spécifiques, comme l'arithmétique linéaire, la théorie de l'égalité libre et de la théorie des tableaux.

Les démonstrateurs interactifs de théorèmes [56, 32] peuvent naturellement être utilisés pour prouver ces formules, notamment lorsque les preuves nécessitent un raisonnement inductif. Néanmoins, ces outils sont très fastidieux, car ils demandent beaucoup de temps et de détail à l'utilisateur. De ce fait, les démonstrateurs automatiques de théorèmes (ATPs) semblent être une bonne alternative, car ils permettent un haut degré d'automatisation et passent mieux à l'échelle sur les grands projets.

Les démonstrateurs SAT [24, 38, 53] désignent une famille d'outils automatiques qui prennent en entrée des formules de la logique propositionnelle. Ces outils ont fait un progrès spectaculaire et ont atteint un grand niveau de maturité au cours des deux dernières décennies. Cependant, ils ne sont pas vraiment adaptés pour prouver les formules provenant de la vérification de logiciels car ils ne gèrent pas les quantificateurs et le raisonnement modulo théories.

À l'opposé, les démonstrateurs de la famille TPTP [59, 37, 48] sont très génériques et prennent en entrée des formules de la logique du premier ordre. En général, ces ATPs sont basés sur le mécanisme de résolution, la méthode des tableaux ou le calcul de superposition. En outre, ils sont complets par réfutation et intègrent certaines théories équationnelles, comme la théorie de l'égalité libre et la théorie AC des symboles associatifs et commutatifs. En interne, ils utilisent la réécriture, l'unification et la completion comme techniques de calcul et d'inférence. Toutefois, ces démonstrateurs ne se sont pas vraiment imposés dans le cadre de la vérification de logiciels. La raison principale est due au manque de raisonnement efficace dans une combinaison de théories intéressantes, comme l'arithmétique linéaire.

A mi-chemin entre ces deux familles, une autre technologie, appelée Satisfiabilité Modulo Théories (SMT), a émergé durant la dernière décennie. Les formules logiques issues de la vérification de logiciels sont à la portée de ces ATPs. Cette thèse s'intéresse à cette famille de démonstrateurs automatiques.

1 Satisfiabilité Modulo Théories

Les démonstrateurs SMT prennent en entrée des formules exprimées dans des fragments de la logique du premier ordre où certains symboles ont des significations supplémentaires dans des théories spécifiques. Par exemple, la formule

$$(a \leq b + 0 \ \wedge \ a \geq b) \ \Rightarrow \ f(a) = f(b)$$

est composée de connecteurs booléens ($\wedge$, $\Rightarrow$), de symboles non interprétés (a, b, f), du prédicat de l'égalité ($=$) et de symboles arithmétiques ($\leq$, $\geq$, $+$, 0). Cette for-

mule est valide dans la combinaison de la théorie de l'égalité libre et de l'arithmétique linéaire sur les nombres entiers.

En interne, les démonstrateurs SMT combinent de puissants "petits moteurs de preuves spécialisés" pour permettre un raisonnement propositionnel efficace modulo des théories intéressantes. De plus, certains d'entre eux gèrent les formules quantifiées. La figure 1 illustre l'architecture basique d'un démonstrateur SMT. Le préprocesseur regroupe des opérations telles que l'analyse lexicale et syntaxique, le vérification de type, la simplification, l'apprentissage et la conversion des formules en forme normale conjonctive (CNF). Dans le reste de cette section, nous allons décrire les autres composants plus en détail.

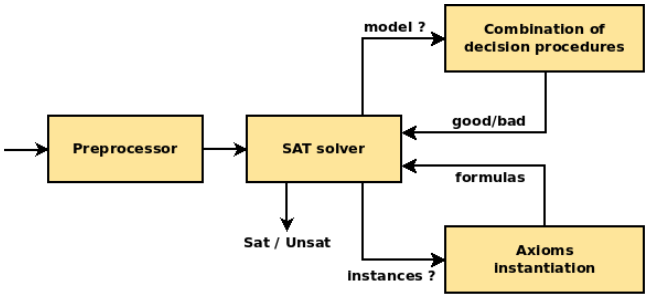

Figure 1: L'architecture basique d'un démonstrateur SMT

1.1 Les démonstrateurs SAT

Les démonstrateurs SAT gèrent le raisonnement modulo la logique propositionnelle. Ils sont utilisés pour déterminer si les variables propositionnelles apparaissant dans une formule donnée admettent une affectation booléenne rendant cette formule satisfiable. Les principales méthodes utilisées pour cela sont les BDDs [1], la méthode de résolution [49], l'algorithme DPLL [16] et la procédure CDCL [53]. Les deux dernières techniques sont les plus populaires en SMT. Elles travaillent sur des formules en forme normale conjonctive.

La procédure DPLL est une méthode qui tente de construire un modèle (solution) booléen(ne) pour une formule donnée en répétant les étapes suivantes: (1) assigner une variable non affectée à une valeur de vérité et (2) déduire et propager toutes les conséquences de l'étape (1). Quand un état conflictuel est rencontré, la procédure défait son raisonnement jusqu'à la dernière variable assignée, inverse la valeur de celle-ci, et continue la recherche d'une solution.

L'algorithme CDCL améliore énormément la procédure DPLL. Premièrement, il utilise de nouvelles structures de données pour la représentation des formules.

Cela accélère considérablement la déduction de conséquences. Deuxièmement, le choix de la prochaine variable à assigner est guidé par une heuristique dynamique sur l'activité des variables (appelée VSIDS) [38]. Aussi, une clause impliquée par la formule de départ est apprise après chaque conflit, et le processus de retour en arrière (décrit dans la procédure DPLL ci-dessus) est entièrement guidé par cette clause. La figure 2 montre une implantation naïve de la procédure CDCL.

```
1  procedure naive_cdcl() =
2    while true do
3      propagate();
4      if conflict() then
5        begin
6          if no_decisions() then return UNSAT;
7          analyze_conflict();
8          learn_a_clause();
9          backjump(); // non-chronological backtrack
10       end
11     else
12       begin
13         if full_model() then return SAT;
14         decide();
15       end
16   done
```

Figure 2: La procédure CDCL simplifiée

L'extension d'une procédure DPLL/CDCL avec un raisonnement modulo une théorie est décrit dans [42]. Dans les premières approches développées, les procédures de décision des théories étaient uniquement utilisées pour vérifier la satisfiabilité des modèles booléens complets construits par le SAT. Les schémas modernes d'intégration sont beaucoup plus raffinés. La figure 3 montre un exemple d'une telle intégration: à chaque fois que le modèle booléen partiel est augmenté avec certaines assignations, la procédure de décision vérifie la cohérence de cette solution partielle modulo la théorie (ligne 4). De plus, un mécanisme d'apprentissage — incomplet, mais rapide — est utilisé pour déduire des conséquences impliquées au niveau de la théorie par le modèle booléen actuel.

1.2 Les procédures de décision et leur combinaison

Les procédures de décision sont conçues pour raisonner efficacement modulo des théories spécifiques. Ces moteurs de preuves dédiés sont prévisibles par rapport aux mécanismes génériques de déduction. Par exemple, l'algorithme de fermeture par congruence [41, 2] et la procédure de completion close [54] sont utilisés

```
a 1  procedure propagate() =
2      while propagation_stack_not_empty() do
3        boolean_propagation();
4        theory_assume();
5        theory_learn();
6      done
```

Figure 3: L'integration d'un raisonnement modulo théorie dans DPLL/CDCL

pour décider des formules sans quantificateurs modulo la théorie de l'égalité libre. L'algorithme du simplexe [15] et la méthode de Fourier-Motzkin [33] sont utilisés pour décider des formules sans quantificateurs modulo l'arithmétique linéaire sur les rationnels. La procédure Omega-Test [46] et les diverses extensions du simplexe [51] sont utilisées pour résoudre des formules de l'arithmétique linéaire sur entiers.

Dans de nombreux domaines, les formules logiques mélangent des symboles de plusieurs théories. Par conséquent, le raisonnement dans l'union de ces théories est capital pour décider la validité de telles formules. Étant données deux théories $\mathcal{T}_1$ et $\mathcal{T}_2$ équipées de deux procédures de décision $\mathcal{D}_1$ et $\mathcal{D}_2$ respectivement, le problème de la combinaison de $\mathcal{T}_1$ et $\mathcal{T}_2$ consiste à construire — si possible — une procédure de décision $\mathcal{D}$ pour l'union $\mathcal{T}_1 \cup \mathcal{T}_2$ en utilisant les procédures individuelles $\mathcal{D}_1$ et $\mathcal{D}_2$.

Concevoir des mécanismes efficaces pour combiner des procédures de décision est un domaine de recherche très actif. Les deux algorithmes historiques pour combiner les théories ne partageant pas de symboles en commun sont la méthode de Nelson-Oppen [39] et celle de Shostak [52].

Le mécanisme de Nelson-Oppen est très générique. Il combine des procédures de décision pour les théories sur signatures disjointes qui sont *stable infinie*. Pour les théories convexes, (1) il purifie la formule mixte en utilisant l'abstraction par des variables; (2) il résout chaque formule pure en utilisant la procédure de décision correspondante; (3) il échange les égalités impliquées entre les variables partagées. Le dernier point est crucial pour la complétude, car les procédures individuelles doivent se mettre d'accord sur les égalités entre variables partagées pour pouvoir fusionner leurs solutions.

La version non-déterministe gère également les théories non convexes. Elle fonctionne en devinant les égalités impliquées entre les variables partagées au lieu de demander aux procédures de les déduire toutes. Toutefois, sa complexité est beaucoup plus élevée. Nous allons illustrer cette méthode sur la formule mixte

suivante:

$$0 \leq a \ \wedge \ a \leq 1 \ \wedge \ f(a) \neq f(0) \ \wedge \ g(a) \neq g(1)$$

L'étape de purification produit la conjonction équivalente ci-dessous:

$$\begin{array}{ll} 0 \leq x \ \wedge \ x \leq 1 \ \wedge \ y = 0 \ \wedge \ z = 1 & (\varphi_1) \\ \wedge & \\ x = a \ \wedge \ f(x) \neq f(y) \ \wedge \ g(x) \neq g(z) & (\varphi_2) \end{array}$$

où x, y et z sont des variables d'abstraction, φ_1 est pure dans l'arithmétique linéaire sur les entiers, et φ_2 est pure dans la théorie de l'égalité libre. Dans la deuxième étape, on déduit que $(0 \leq x \wedge x \leq 1)$ est équivalente à $(x = 0 \ \vee \ x = 1)$ modulo l'arithmétique linéaire sur les entiers. Mais, si $x = 0$ (*resp.* $x = 1$), l'égalité $x = y$ (*resp.* $x = z$) doit être propagée à la procédure de décision de la théorie de l'égalité libre. Cela implique que $f(x) = f(y)$ (*resp.* $g(x) = g(z)$), ce qui conduit à une incohérence.

Un nombre important de travaux visent à améliorer ce mécanisme. Par exemple, la méthode appelée "*Delayed Theories Combination*" [13] délègue la tâche de combinaison au démonstrateur SAT. La méthode "*Model Based Theories Combination*" [21] tente, quant à elle, de réduire le nombre d'égalités entre variables partagées à propager tout en préservant la complétude. Ces méthodes sont utilisées dans plusieurs démonstrateurs SMT, comme CVC3 [10], CVC4 [6], MathSAT5 [29] et Z3 [18].

La combinaison à la Shostak est un mécanisme spécifique pour le raisonnement dans l'union (sur des symboles disjoints) de la théorie de l'égalité libre avec une théorie de Shostak: une théorie équationnelle qui admet un algorithme — appelé *canoniseur* — pour calculer les formes normales des termes modulo la théorie et un algorithme — appelé *solveur* — qui transforme les égalités en substitutions. Par exemple, les parties équationnelles de l'arithmétique linéaire et de la théorie des enregistrements (records) répondent à ces critères.

Cette approche résout des formules de la forme $s_1 = t_1 \wedge \cdots \wedge s_n = t_n \Rightarrow s = t$. Elle combine les opérations de canonisation, résolution d'égalités et réécriture, dans le but de transformer la conjonction $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$ en un système de réécriture convergent modulo la théorie de Shostak. Ensuite, elle réécrit et canonise s et t, pour vérifier si $s = t$ est une conséquence de la conjonction. Nous allons illustrer ce mécanisme sur l'exemple suivant:

$$(x = 2y \ \wedge \ f(y) - \frac{x}{2} = 0) \ \Rightarrow \ f(x - f(y)) = y$$

Résoudre la première égalité renvoie simplement $x \mapsto 2y$. La seconde égalité est ensuite réécrite en $f(y) - \frac{2y}{2} = 0$ et canonisée en $f(y) - y = 0$. Résoudre la dernière égalité retourne $f(y) \mapsto y$. Enfin, la réécriture et la canonisation de $f(x - f(y))$ retourne y.

La formulation initiale du mécanisme de Shostak [52] n'était pas complète et ne terminait pas. Après plusieurs tentatives, elle a été entièrement corrigée dans [50]. En outre, Shostak avait affirmé qu'il serait possible de combiner les canoniseurs et les solveurs de théories individuelles. Alors que la première affirmation a été prouvée correcte, il a été montré que les solveurs ne se combinent pas en général [14]. Ce mécanisme et ses variantes sont utilisés dans plusieurs outils, tels que PVS [44], SVC [8], ICS [25] et notre démonstrateur SMT, appelé ALT-ERGO [11].

1.3 Gestion des quantificateurs

Au sens strict, SMT désigne l'extension d'un démonstrateur SAT avec des procédures de décision pour raisonner modulo théories. Cependant, dans de nombreux domaines d'application, les formules contiennent à la fois des parties closes et des parties quantifiées. Dans la vérification déductive par exemple, les formules quantifiées sont utilisées pour encoder les modèles mémoire des langages de programmation et les propriétés données dans les spécifications des programmes.

En général, les démonstrateurs SMT qui gèrent les quantificateurs utilisent des techniques d'instanciation. Ces techniques sont basées sur le filtrage modulo les égalités closes (E-matching) et la notion de déclencheurs (triggers). En outre, certains démonstrateurs SMT intégrent le calcul de superposition [19] pour surmonter les limites des techniques de filtrage.

2 La révolution SMT

La Satisfiabilité Modulo Théories est un sujet de recherche relativement jeune. Les premiers développements remontent à la thèse de doctorat de Nelson [40] fin des années 1970 — début des années 1980. Au cours des dernières années, nous avons assisté à une amélioration importante de cette technologie. Actuellement les démonstrateurs SMT sont très populaires. Ils sont utilisés dans divers domaines, tels que la conception de microprocesseurs, la vérification de programmes, le model-checking, l'exécution symbolique, la génération de tests, etc. Cependant, il y a encore beaucoup de défis en SMT, comme la conception de procédures pour la théorie des nombres flottants et la théorie des ensembles, l'amélioration des procé-

dures existantes, l'amélioration des mécanismes de combinaison et une meilleure gestion des quantificateurs.

Il existe une communauté très active autour de SMT. En 2003, l'initiative SMT-LIB [47] a été créée pour fournir des descriptions normalisées de quelques théories intéressantes, définir un langage d'entrée standard, et recueillir des tests. De plus, *"SMT-competition"* [7], *"SMT-workshop"* et *"the SAT/SMT summer school"* sont des événements annuels consacrés à ce sujet.

3 Résumé des contributions

Cette thèse porte sur l'amélioration de notre démonstrateur SMT, appelé Alt-Ergo, pour la rendre efficace et utilisable dans le cadre de la vérification de programmes. Nous résumons ci-après nos principales contributions.

3.1 La théorie de l'arithmétique linéaire sur les entiers

L'arithmétique linéaire sur les entiers et sans quantificateurs (quantifier-free linear integer arithmetic ou QF-LIA) est omniprésente dans de nombreux domaines, tels que la vérification, la programmation linéaire, l'optimisation des compilateurs, la planification et d'ordonnancement. En particulier, elle constitue — avec la théorie de l'égalité libre — une théorie incontournable dans la vérification déductive de programmes. La plupart des procédures de décisions implantées par les démonstrateurs SMT de l'état de l'art sont des extensions de l'algorithme du simplexe [17, 6, 29, 20] ou de la méthode de Fourier-Motzkin [10, 11]. Ces deux techniques résolvent d'abord la relaxation du problème initial dans les rationnels, qu'elles complètent par des techniques de branchement / projection.

Les extensions du simplexe, telles que le *"branch-and-bound"* et les *"cutting-planes"* [51], sont souvent noyées dans un espace de recherche immense quand elles sont amenées à effectuer une analyse par cas. De plus, elles ne sont pas complètent par rapport à la déduction de toutes les égalités impliquées. À l'inverse, les extensions de Fourier-Motzkin, telles que *"Omega-Test"* [46], ne passent pas à l'échelle en pratique, car elles introduisent potentiellement un nombre exponentiel d'inégalités intermédiaires, ce qui sature la mémoire.

La première contribution de cette thèse est une nouvelle procédure de décision pour la théorie QF-LIA. D'une manière simplifiée, étant donnée une conjonction d'inégalités $\bigwedge_i \sum_j a_{i,j}\, x_j + b_i \leq 0$ sur les entiers, notre algorithme tente de déduire des bornes inférieures précises pour les formes affines $a_{i,j}\, x_j + b_i$ afin de réduire

l'espace de recherche. Ces bornes sont calculées en utilisant un simplexe sur les rationnels qui résout des problèmes auxiliaires d'optimisation linéaire. Ces problèmes auxiliaires simulent des exécutions particulières de Fourier-Motzkin.

Avec notre méthode, nous pouvons directement déduire la satisfiabilité du problème original si aucune des formes affines n'admet une borne inférieure. Ceci est clairement un point positif par rapport aux extensions traditionnelles du simplexe. De plus, nous pouvons déduire toutes les (disjonctions d') égalités impliquées pour les formes affines $a_{i,j}\, x_j + b_i$ qui ont une borne inférieure. En outre, l'utilisation d'un simplexe rationnel pour effectuer les calculs nous permet de contourner le problème de passage à l'échelle des extensions de Fourier-Motzkin.

Vu que notre méthode est basée sur l'inférence et la maintenance de bornes, elle est facilement extensible avec des techniques de calcul d'intervalles. En pratique, cela nous permet d'intégrer un peu de raisonnement modulo la théorie de l'arithmétique non-linéaire. Notez que les contraintes d'égalité sont traitées dans notre méthode par résolution et substitution, comme dans *Omega-Test*.

3.2 Pré-traitement et raisonnement SAT

Afin de valider notre procédure pour l'arithmétique linéaire sur les entiers, nous avons mené quelques expériences avec ALT-ERGO en utilisant les tests QF-LIA [9]. Malheureusement, nous avons été rapidement confrontés à la complexité de certaines formules dans ces tests. En effet, ces formules ont une structure propositionnelle riche et utilisent abondamment certaines constructions de haut niveau, telles que LET-IN et IF-THEN-ELSE. Bien que suffisants dans le contexte de la vérification déductive de programmes, le préprocesseur et le démonstrateur SAT de ALT-ERGO ne sont pas adaptés pour ce genre de formules.

Pour contourner ce problème, nous avons implanté un petit démonstrateur SMT dédié à la théorie QF-LIA. Ce prototype, appelé CTRL-ERGO, est basé sur une reimplantation de MINISAT [23] en OCAML. Son préprocesseur intègre une nouvelle étape de simplification contextuelle ayant pour but de minimiser les formules contenant des LET-IN et des IF-THEN-ELSE. De plus, la procédure QF-LIA implante certaines fonctionnalités, telles que la propagation des faits impliqués modulo la théorie et le partitionnement (*clustering*) dynamique des problèmes donnés.

3.3 Raisonnement AC pré-défini modulo des théories de Shostak

De nombreux opérateurs mathématiques utilisés dans le raisonnement automatisé possèdent les propriétés d'associativité et de commutativité (AC). Des exem-

ples de ces opérateurs sont l'union et l'intersection d'ensembles, les connecteurs booléens (et, ou, ou exclusif), et les opérateurs arithmétiques (addition, multiplication linéaire et non-linéaire). La manière la plus "légère" de gérer ces propriétés dans un démonstrateur SMT est d'ajouter les axiomes ci-dessous dans son moteur d'instanciation pour chaque symbole de fonction u qui est AC.

$$\begin{array}{rrcll} \forall x.\forall y.\forall z. & u(x,u(y,z)) & = & u(u(x,y),z) & \text{(A)} \\ \forall x.\forall y. & u(x,y) & = & u(y,x) & \text{(C)} \end{array}$$

Malheureusement, les résultats de cette méthode ne sont pas concluent en pratique. En effet, le simple ajout de ces axiomes à un démonstrateur font que son monteur d'instantiation génère plein de termes et d'égalités intermédiaires, ce qui impacte négativement ses performances.

Alors que les démonstrateurs SMT fournissent un support intégré pour quelques symboles AC spécifiques, tels que les fonctions de l'arithmétique linéaire et les opérateurs booléens, ils reposent plutôt sur les axiomes pour traiter les symboles AC génériques définis par l'utilisateur. À l'opposé, le raisonnement modulo la théorie AC est bien étudié dans la communauté de réécriture. Par exemple, la procédure de completion AC, implantée au cœur de certains démonstrateurs TPTP, permet un traitement générique efficace des symboles AC. De plus, lorsque le problème en entrée n'a pas de variables, la procédure de completion AC fournit un algorithme de décision [36] pour l'union de la théorie de l'égalité et AC.

Dans de nombreux domaines d'application, la théorie AC n'est qu'une partie du problème de déduction automatique. Ce dont on a besoin, c'est de décider la validité de formules mélangeant des symboles AC et des symboles d'autres théories. Par exemple, une combinaison d'un raisonnement AC avec l'arithmétique linéaire et la théorie de l'égalité libre est nécessaire pour prouver la formule

$$\left(\begin{array}{l} u(a, c_2 - c_1) = a \ \wedge \ d = c_1 + 1 \ \wedge \\ u(e_1, e_2) - f(b) = u(d, d) \ \wedge \ e_2 = b \ \wedge \\ u(b, e_1) = f(e_2) \ \wedge \ c_2 = 2\, c_1 + 1 \end{array}\right) \ \Rightarrow \ a = u(a, 0)$$

où u est un symbole AC, les symboles $+, -, 0, 1, 2$ appartiennent à l'arithmétique linéaire, et le reste des symboles appartiennent à la théorie de l'égalité libre.

Dans cette thèse, nous avons étudié l'incorporation d'un raisonnement prédéfini générique de la théorie AC dans un démonstrateur SMT. Pour cela, nous avons conçu un mécanisme de combinaison, appelé AC(X), pour raisonner dans

l'union de la théorie de l'égalité libre, la théorie AC et une théorie de Shostak arbitraire ne partageant pas de symboles.

La théorie AC ne peut pas être directement combinées en utilisant le mécanisme de Shostak, car elle n'a pas d'algorithme pour résoudre les équalités, comme c'est le cas pour l'arithmétique par exemple. Pour contourner cette limitation, nous avons adopté une approche légèrement différente: nous nous sommes inspirés de la technique de combinaison de Shostak et avons étendu la procédure de completion AC close avec une théorie Shostak. Cette extension modulaire et non-intrusive repose sur l'intégration du *canoniseur* et du *solveur*, fournis par la théorie de Shostak, dans la procédure de completion AC close.

À la même période que nos investigations, Tiwari a étudié [58] la combinaison du raisonnement AC avec la théorie de l'égalité libre et la théorie des anneaux de polynômes dans le cadre du mécanisme de Nelson-Oppen. Cependant, aucune implantation ou expérimentation n'ont été données. De plus, notre philosophie dans ALT-ERGO est de favoriser plutôt les méthodes inspirées de celles de Shostak pour combiner les théories équationnelles convexes.

3.4 Implantation dans ALT-ERGO

ALT-ERGO [11] est un démonstrateur SMT utilisé pour prouver la validité de formules logiques issues de la vérification de programmes. Il est actuellement utilisé dans des outils tels que Why3 [26], Caveat [55], Frama-C [27], Spark [4] et GNATprove [30].

Au cours de cette thèse, nous avons amélioré le noyau d'ALT-ERGO de plusieurs façons. Premièrement, nous avons étendu notre démonstrateur avec le mécanisme de combinaison AC(X). Actuellement, AC(X) est utilisé pour gérer les propriétés d'associativité et de commutativité des symboles AC définis par l'utilisateur et de la multiplication non-linéaire. De plus, une interaction subtile entre l'arithmétique linéaire et AC nous permet de gérer partiellement la propriété de distributivité de la multiplication non-linéaire sur l'addition. Deuxièmement, nous avons implanté une procédure de décision de l'état de l'art pour la théorie des tableaux fonctionnels [28] et une procédure pour la théorie des types de données énumérés.

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978. (Cited on page 3.)

[2] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003. (Cited on page 4.)

[3] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. Slam2: Static driver verification with under 4% false alarms. In Roderick Bloem and Natasha Sharygina, editors, *FMCAD*, pages 35–42. IEEE, 2010. (Cited on page 1.)

[4] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. (Cited on pages 1 and 11.)

[5] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*, volume 4111 of *LNCS*, 2005. (Cited on page 1.)

[6] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on pages 6 and 8.)

[7] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, 2012. 10.1007/s10817-012-9246-5. (Cited on page 8.)

[8] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. pages 187–201. Springer-Verlag, 1996. (Cited on page 7.)

[9] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010. (Cited on page 9.)

[10] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, Berlin, Germany, July 2007. Springer. (Cited on pages 6 and 8.)

[11] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. http://alt-ergo.lri.fr/. (Cited on pages 7, 8 and 11.)

[12] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, February 2011. http://why3.lri.fr/. (Cited on page 1.)

[13] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzen, Alberto Griggio, and Roberto Sebastiani. Delayed theory combination vs. nelson-oppen for satisfiability modulo theories: a comparative analysis. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):63–99, February 2009. (Cited on page 6.)

[14] Sylvain Conchon and Sava Krstić. Strategies for combining decision procedures. *Theoretical Computer Science*, 354(2):187–210, 2006. Special Issue of TCS dedicated to a refereed selection of papers presented at TACAS'03. (Cited on page 7.)

[15] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963. (Cited on page 5.)

[16] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. (Cited on page 3.)

[17] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. http://research.microsoft.com/projects/z3/. (Cited on page 8.)

[18] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. (Cited on page 6.)

[19] Leonardo de Moura and Nikolaj Bjørner. Engineering dpll(t) + saturation. In *PROC. 4TH IJCAR*, 2008. (Cited on page 7.)

[20] Leonardo de Moura and Bruno Dutertre. Yices: An SMT Solver. http://yices.csl.sri.com/. (Cited on page 8.)

[21] Leonardo Mendonça de Moura and Nikolaj Bjørner. Model-based theory combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, 2008. (Cited on page 6.)

[22] David L. Dill. A retrospective on mur*phi*. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2008. (Cited on page 1.)

[23] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. (Cited on page 9.)

[24] Niklas Een and Niklas Sörensson. An extensible sat-solver [ver 1.2], 2003. (Cited on page 2.)

[25] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. ICS: Integrated Canonization and Solving (Tool presentation). In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer, 2001. (Cited on page 7.)

[26] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013. (Cited on page 11.)

[27] The Frama-C platform for static analysis of C programs, 2008. http://www.frama-c.cea.fr/. (Cited on pages 1 and 11.)

[28] Amit Goel, Sava Krstić, and Alexander Fuchs. Deciding array formulas with frugal axiom instantiation. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, pages 12–17, New York, NY, USA, 2008. ACM. (Cited on page 11.)

[29] A. Griggio, B. Schaafsma, A. Cimatti, and R. Sebastiani. MathSAT 5: An SMT Solver for Formal Verification. http://mathsat.fbk.eu//. (Cited on pages 6 and 8.)

[30] Jérôme Guitton, Johannes Kanig, and Yannick Moy. Why Hi-Lite Ada? In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 27–39, Wrocław, Poland, August 2011. (Cited on page 11.)

[31] Aarti Gupta, Malay K. Ganai, and Chao Wang. Sat-based verification methods and applications in hardware verification. In Marco Bernardo and Alessandro Cimatti, editors, *SFM*, volume 3965 of *Lecture Notes in Computer Science*, pages 108–143. Springer, 2006. (Cited on page 1.)

[32] The ISABELLE system. http://isabelle.in.tum.de/. (Cited on page 2.)

[33] Leonid Khachiyan. Fourier-motzkin elimination method. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization*, pages 1074–1077. Springer, 2009. (Cited on page 5.)

[34] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993. (Cited on page 1.)

[35] J. L. LIONS. Ariane 5 flight 501 failure. http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html. (Cited on page 1.)

[36] Claude Marché. On ground AC-completion. In Ronald. V. Book, editor, *4th International Conference on Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, Como, Italy, April 1991. Springer. (Cited on page 10.)

[37] William McCune. Otter 3.3 reference manual. *CoRR*, cs.SC/0310056, 2003. (Cited on page 2.)

[38] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *ANNUAL ACM IEEE DESIGN AUTOMATION CONFERENCE*, pages 530–535. ACM, 2001. (Cited on pages 2 and 4.)

[39] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27:356–364, 1980. (Cited on page 5.)

[40] Greg Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1981. http://www.cs.washington.edu/education/courses/cse599f/06sp/papers/NelsonThesis.pdf. (Cited on page 7.)

[41] Robert Nieuwenhuis and Albert Oliveras. Fast Congruence Closure and Extensions. *Inf. Comput.*, 2005(4):557–580, 2007. (Cited on page 4.)

[42] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract dpll and abstract dpll modulo theories. In *In LPAR'04, LNAI 3452*, pages 36–50. Springer, 2005. (Cited on page 4.)

[43] John O'Leary. Theorem proving in intel hardware design. In Ewen Denney, Dimitra Giannakopoulou, and Corina S. Pasareanu, editors, *NASA Formal Methods*, volume NASA/CP-2009-215407 of *NASA Conference Proceedings*, page 5, 2009. (Cited on page 1.)

[44] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga Springs, NY, June 1992. Springer. (Cited on page 7.)

[45] Vaughan R. Pratt. Anatomy of the pentium bug. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107. Springer, 1995. (Cited on page 1.)

[46] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM. (Cited on pages 5 and 8.)

[47] Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://smtcomp.sourceforge.net/`, 2006. (Cited on page 8.)

[48] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2-3):91–110, 2002. (Cited on page 2.)

[49] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965. (Cited on page 3.)

[50] Harald Rueß and Natarajan Shankar. Deconstructing shostak. In *LICS*, pages 19–28. IEEE Computer Society, 2001. (Cited on page 7.)

[51] A. Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. John Wiley & sons, 1998. (Cited on pages 5 and 8.)

[52] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31:1–12, 1984. (Cited on pages 5 and 7.)

[53] Joso L Marques Silva. Grasp - a new search algorithm for satisfiability. pages 220–227, 1996. (Cited on pages 2 and 3.)

[54] W. Snyder. Efficient completion: a $o(n.log(n))$ algorithm for generating reduced sets of ground rewrite rules equivalent to a set of ground equations e. In N. Dershowitz, editor, *Proc. 3rd Conf. on Rewriting Techniques and Applications*. Springer-Verlag, 1989. Lecture Notes in Computer Science. (Cited on page 4.)

[55] Jean Souyris and Denis Favre-Félix. Proof of properties in avionics. In Renè Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 527–535. Springer US, 2004. (Cited on pages 1 and 11.)

[56] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, 2010. `http://coq.inria.fr`. (Cited on page 2.)

[57] Nikolai Tillmann and Jonathan De Halleux. Parameterized unit testing with microsoft pex (long tutorial), 2010. (Cited on page 1.)

[58] Ashish Tiwari. Combining equational reasoning. In Silvio Ghilardi and Roberto Sebastiani, editors, *FroCos*, volume 5749 of *Lecture Notes in Computer Science*, pages 68–83, Trento, Italy, September 2009. Springer. (Cited on page 11.)

[59] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *Proceedings of the 22nd International Conference on Automated Deduction*, CADE-22, pages 140–145, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 2.)