université PARIS-SACLAY

ENSTA ParisTech université PARIS-SACLAY

NNT : 2019SACLY007

Amélioration de performances du solveur SMT Alt-Ergo grâce à l'intégration d'un solveur SAT efficace

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Ecole nationale supérieure de techniques avancées

Ecole doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : informatique

Thèse présentée et soutenue à Orsay, le 16 décembre 2019, par

ALBIN COQUEREAU

Composition du Jury :

Catherine Dubois Professeure, ENSIIE	Présidente
Pascal Fontaine Professeur, Université de Liège (Institut Montefiore)	Rapporteur
Laurent Simon Professeur, INP Bordeaux (LaBRI)	Rapporteur
David Deharbe Ingénieur, ClearSy	Examinateur
Michel Mauny Directeur scientifique, Nomadic Labs	Directeur de thèse
Sylvain Conchon Professeur, Université Paris Sud (LRI)	Co-directeur de thèse
Fabrice Le Fessant Président, Origin Labs	Invité
Mohamed Iguernlala Ingénieur R&D, Origin Labs	Invité

Thèse de doctorat

Remerciements

Une sage personne (dont je tairai le nom) m'a jadis demandé "Qu'est-ce qui t'anime, qu'est ce qui te fait plaisir dans la vie Albin ?". J'ai eu beau chercher, je n'ai pas su quoi répondre et encore aujourd'hui je ne sais toujours pas quoi répondre à cette question. J'espère qu'un jour, je trouverais une réponse à ta question Muriel.

Le parcours fut fastidieux mais il est maintenant fini. Je tiens tout d'abord à remercier mes encadrants Michel, Sylvain, Fabrice et Mohamed, qui m'ont accompagné au long de ces années. Ils ont aussi bien su orienter mes recherches que me remotiver. Un merci particulier à Mohamed sans qui cette thèse n'aurait, sans aucun doute, pas abouti. Merci pour ton encadrement, ta motivation sans faille et ta bonne humeur.

Merci à mes rapporteurs, Pascal Fontaine et Laurent Simon, d'avoir accepté de relire une version préliminaire de ce document et pour leurs corrections et remarques pertinentes. Je remercie aussi David Deharbe et Catherine Dubois d'avoir accepté de faire partie de mon jury.

Au cours de ces années de doctorat, il m'a été permis de travailler dans différents lieux et dans différentes équipes. Tout d'abord, merci aux quelques collègues de l'équipe U2IS de l'ENSTA avec qui j'ai eu le plaisir de travailler. Merci de m'avoir accueilli pour ces premières années de thèse où j'ai aussi bien appris la complexité de l'administration que celle de la chasse aux Pokémons. Merci à l'équipe VALS du LRI pour votre accueil et pour toutes les discussions du coin café. Merci à mes co-bureaux, l'un spécialiste en compilation de la JVM à répétition, l'autre expert en géographie : "Vannes ça doit pas être loin de Cannes, ça sonne pareil" de m'avoir supporté.

Merci à Fabrice et Muriel de m'avoir accueilli au sein d'OCamlPro, m'ayant ainsi permis de rédiger ce document et me permettant maintenant d'apporter ma pierre à l'édifice, et quel édifice ! Je tiens à remercier mes collègues (la liste est maintenant trop longue pour être énumérée) pour toutes ces discussions scientifiques pointues qui c'est vrai ont pu nous occuper quelques bonnes heures (merci chef Chambart). Merci à vous pour toutes les discussions non scientifiques, les gamelles et autres pissettes, les fous rires, la bonne humeur et le régime alimentaire strict (surtout le vendredi). Un très grand merci à Laurène sans qui ce document serait beaucoup plus moins lisible. Enfin, je tiens à m'excuser auprès de mon co-bureau pour mes siestes parfois bruyantes.

Il est maintenant temps de remercier ceux qui ont su choisir les mots justes pour ralentir l'avancement de cette thèse. Merci Mattias pour être toi, sachant que c'est parfois un sacré handicap. Merci à Pierrick de m'avoir poussé à ne pas procrastiner lorsque j'avais des choses essentielles à faire. Merci Rémy pour tes talents de photographe ayant pu immortaliser ma pilosité.

Je remercie toustes celleux qui m'ont accompagné depuis mon arrivée à la capitale. Merci à l'habile camarade Tachon. Merci à Ornella et Julie de m'avoir permis de casser des choses chez vous. Merci à la famille Roux. Merci Bastien et Laurianne et à toustes les autres, peu importe leurs noms !

Ce voyage a débuté il y a maintenant plus de 10 ans sur les bancs de l'université de Nantes. Merci à Alwine et Mathilde d'avoir partagé ma vie, et pour tout le reste. Merci à Greg, le meilleur d'entre nous, tmtc. Merci à Aline (sympa ta playlist !), au petit Grau, à Alex, Manu, Zag, Léo, Noémie et bien d'autres encore.

Merci à mon amie d'enfance, Marion, la géographie n'aide pas mais le cœur est toujours là.

Merci à ma famille et en particulier mes parents, j'ai pas toujours été facile, vous savez ce que je vous dois et que je vous aime.

Table des matières

1 Introduction **7**
1 Satisfiabilité booléenne 8
1.1 Résolution et exploration 8
1.2 Méthode des tableaux analytiques 9
2 Satisfiabilité Modulo Théories 10
2.1 Le solveur SMT Alt-Ergo 13
2.2 Le standard SMT-LIB 2 13
3 Contributions de la thèse 14
3.1 Étude des performances d'un solveur CDCL en OCaml 14
3.2 Intégration efficace d'un solveur CDCL dans un solveur SMT 16
3.3 Extension du standard SMT-LIB 2 avec du polymorphisme 17
3.4 Participation à la compétition SMT-COMP 17
4 Plan de la thèse 18

2 Le solveur Alt-Ergo **19**
1 Architecture 19
1.1 Gestion paresseuse de la CNF 20
1.2 Solveur SAT historique d'Alt-Ergo 21
1.3 Gestion des théories et des termes quantifiés dans Alt-Ergo 25
2 Le langage natif d'Alt-Ergo 27

I SAT et SMT solveur **31**

3 Vers un solveur SAT efficace en OCaml **33**
1 Méthodologie de comparaison 33
1.1 Trace de décision 34
1.2 Consommation spatiale et temporelle 34
1.3 Paramètres 38
2 Résultats expérimentaux 39

2.1 Impact du langage OCaml sur les performances 43
2.2 Pistes d'optimisation 53
2.3 SatML vs solveur historique d'Alt-Ergo 58
3 Conclusion 59

4 Intégration efficace d'un solveur CDCL dans Alt-Ergo 63
1 Première approche : assister le solveur historique d'Alt-Ergo avec un solveur CDCL 64
2 CDCL(λ(T)) : CDCL modulo Tableau(T) 68
2.1 Réduction du modèle booléen par méthode des Tableaux 68
2.2 Formalisme 70
2.3 Terminaison, Correction, Complétude 74
3 Implémentation 75
3.1 Calcul paresseux de la pertinence 75
3.2 Calcul de pertinence par frontière 78
3.3 Gestion du modèle pour l'instanciation 79
4 Résultats expérimentaux 83
4.1 Comparaison des différents solveurs SAT 84
4.2 Impact du calcul de pertinence des littéraux 86
5 Conclusion 93

II SMT-LIB et SMT-COMP 97

5 Extension du standard SMT-LIB2 avec du polymorphisme 99
1 Extension polymorphe 100
1.1 Syntaxe 100
1.2 Typage 103
2 Implémentation 111
2.1 Création d'une bibliothèque "frontend" pour le langage 111
2.2 Intégration à Alt-Ergo 111
3 Résultats expérimentaux 113
3.1 Génération de bancs de tests avec l'outil Why3 113
3.2 Impact du polymorphisme 117
3.3 Alt-Ergo et les autres solveurs SMT 119
3.4 Conclusion 122

6 Participation à la compétition SMT-COMP 123
1 SMT-COMP 123
1.1 Les théories et logiques de la SMT-LIB 2 123
1.2 Bancs de tests de la communauté SMT 125
1.3 Règles de la compétition 125
2 Préparation pour la compétition 126
2.1 Amélioration des heuristiques 127

2.2 Implémentation d'une version distribuée d'Alt-Ergo . . 127
3 Résultats 129
4 Conclusion 132

7 Conclusion 137

Chapitre 1

Introduction

De l'invention de la roue à la révolution numérique, l'Humain a cherché à se faciliter la vie sur terre. De nos jours, rares sont les tâches qui ne sont pas liées à l'informatique et à l'automatisation. Que ce soit dans nos moyens de transport, pour notre santé ou simplement pour nous nourrir, les systèmes informatiques semblent incontournables. Qu'arrive-t-il quand ceux-ci nous abandonnent ou échouent à leur tâche ? Des avions s'écrasent car un capteur était défaillant, des patients meurent après avoir reçu une dose de radiation trop importante. Et si demain une erreur arrivait sur une chaîne de production agro-alimentaire rendant des produits impropres à la consommation ?

Pour éviter de tels désagréments, nous avons besoin d'outils et de méthodes permettant de nous assurer du bon fonctionnement de nos programmes ou circuits intégrés. Une technique connue et éprouvée consiste à faire des tests de fonctionnement. C'est une technique largement utilisée mais qui a comme principal défaut de ne pas être exhaustive. La vérification formelle permet de résoudre ce problème mais nécessite une spécification détaillée. La spécification peut générer un nombre considérable de formules à traiter. Ces formules peuvent être vérifiées par des assistants de preuve, mais ces derniers ne permettent pas de passer à l'échelle à cause du temps requis par l'utilisateur pour la vérification de chaque problème. Il existe cependant des solutions automatiques comme les solveurs de satisfiabilité modulo théories. Ils offrent une plus grande expressivité que des solveurs SAT, qui ne travaillent que sur des formules booléennes. La possibilité de raisonner dans la combinaison de nombreuses théories ainsi que sur des formules contenant des quantificateurs universels en font des outils de choix pour la preuve formelle.

Pour ces outils, il est à la fois important de répondre correctement au problème posé et de répondre en un temps acceptable. Nous souhaiterions avoir une réponse correcte en un temps réduit par rapport à l'utilisation d'assistants de preuve. Dans cette thèse, nous nous intéresserons à ce deuxième point : améliorer la vitesse de réponse de ces outils.

$$A \vee B \rightarrow C \Leftrightarrow (A \wedge C) \vee (B \wedge C) \tag{1.1}$$

FIGURE 1.1 – Formule représentant un problème de satisfiabilité booléenne.

1 Satisfiabilité booléenne

Le problème principal de nos travaux est celui de déterminer automatiquement la *validité* de formules logiques du premier ordre. On rappelle qu'en logique classique, ce problème est lié à celui de la *satisfiabilité* (existance d'un modèle) puisqu'une formule est valide si et seulement si sa négation est *insatisfiable*. Dans toute la suite de ce manuscrit, on s'intéresse donc uniquement au problème de la satisfiabilité.

Au niveau booléen, le problème de la *satisfiabilité* [11] est à la fois un problème difficile (NP-complet) et aussi un des problèmes en informatique théorique qui a reçu le plus d'attention depuis la fin des années 1950. La figure 1.1 représente un problème booléen dont les modèles A et C vrais ainsi que B et C vrais permettent d'en vérifier la satisfiabilité. Nous détaillons dans les sections suivantes deux méthodes qui ont été inventées pour résoudre ce problème de satisfiabilité.

1.1 Résolution et exploration

Il existe différentes manières et algorithmes non polynomiaux pour trouver des solutions à ces problèmes de satisfiabilité, si de telles solutions existent. Une des premières techniques fut présentée par Davis et Putman [31] en 1960. Le principal aspect de cette dernière est l'utilisation de la résolution pour supprimer des littéraux du problème. Cette technique fut améliorée deux ans plus tard en 1962, par Davis, Logemann et Loveland [30] dans l'algorithme appelé DPLL. Cet algorithme consiste à parcourir en profondeur un arbre de décision binaire où chaque noeud représente les deux valeurs de vérité assignables pour une variable booléenne.

Les problèmes sont mis en forme normale conjonctive (FNC ou CNF en anglais), conjonctions de disjonctions (clauses). Cette forme facilite la propagation des contraintes booléennes, appelées *BCP*, lors d'assignations. L'algorithme de Tseitin [67] de mise en CNF permet de ne pas avoir une transformation exponentielle en nombre de clauses. Cet algorithme est très largement utilisé par les solveurs SAT. Pour que le problème soit satisfiable, chaque clause doit ainsi être vraie : elle doit comporter au moins un littéral vrai.

La figure 1.2 rappelle les grandes lignes de l'algorithme DPLL sous la forme d'une fonction récursive. La première étape (ligne 2) est d'effectuer la propagation de contrainte booléenne sur les clauses du problème. Si une erreur survient durant cette phase cela signifie qu'un ensemble d'assignation

Input: Γ : SAT environnment
Output: Satisfiability status

```
1  Function solve(Γ)
2    (Γ, Error) ← propagate(Γ)        // Boolean constraint propagation
3    if Error then
4      return UNSAT
5    else
6      if all variable are assigned then
7        return SAT
8      else
9        A ← choose_lit_to_decide(Γ)
10       status ← solve(Γ ∪ {A})                       // decide on A
11
12       if status = UNSAT then
13         return solve(Γ ∪ {¬A})                     // propagate ¬A
14
15       else
16         return status
```

Figure 1.2 – Algorithme DPLL

rend le problème $UNSAT$. Sinon, si toutes les variables sont assignées (ligne 5), le problème est alors satisfiable. Pour le dernier cas, on choisit un littéral à assigner. Si l'assignation amène à un conflit booléen (ligne 10), on essaye la négation de ce littéral.

Bien que l'algorithme DPLL soit exponentiel, au cours des années, différentes optimisations ont permis de restreindre l'espace parcouru de l'arbre de décision grâce au retour en arrière non chronologique et à l'apprentissage des clauses conflits [65, 69](algorithme appelé CDCL) introduit en 1996. L'amélioration des *2-watched litterals* [55] a aussi permis de grandement améliorer les performances en réduisant le nombre d'accès aux clauses lors des phases de BCP.

1.2 Méthode des tableaux analytiques

Une autre technique pour la satisfiabilité booléenne provient des circuits booléens. La formule représentant le problème de base est définie par des portes logiques. Une des principales différences est que cette formule n'a pas besoin d'être mise en forme normale conjonctive. En 2000, Junttila et Niemel [51] présentent une résolution des circuits booléens par méthode des tableaux.

Les tableaux furent formalisés sous forme d'arbre de décision [1], où chaque branchement des arbres produits représente une disjonction. Contrai-

$$((x > 4) \wedge (f(y) \leq 2.0)) \vee (x = y) \ \wedge \ (\forall i : int.\ z : int\ list.\ cons(i, z) \neq nil) \tag{1.2}$$

FIGURE 1.3 – Formule représentant un problème de satisfiabilité modulo théories comportant différentes théories comme la théorie de l'égalité et de l'arithmétique entière et rationnelle. Cet exemple comporte aussi des quantificateurs universels sur des types de données algébriques.

rement à l'algorithme DPLL où chaque nœud de l'arbre de décision représente les deux valeurs de vérité d'une variable, les nœuds de branchement des tableaux représentent les deux variables à assigner pour satisfaire une disjonction binaire. La principale différence par rapport à l'algorithme DPLL de la figure 1.2 se situe après la ligne 8. Au lieu de choisir un littéral non assigné, on choisit une disjonction binaire $A \vee B$ où les deux littéraux sont non assignés. Comme précédemment, on commence par appeler récursivement $solve(\Gamma \cup A)$. Si cette décision mène à un conflit alors on appelle $solve(\Gamma \cup \neg A, B)$ pour tester l'autre branche de la disjonction.

Egly [41] présente un solveur SAT pour les circuits booléens basé sur la technique des tableaux. Ce solveur profite des quelques optimisations issues des solveurs SAT travaillant sur une CNF comme le retour en arrière non chronologique ou l'apprentissage de clauses conflits.

La satisfiabilité booléenne est un domaine qui est aujourd'hui largement traité. Elle fait partie intégrante de la satisfiabilité Modulo Théories qui permet une plus grande expressivité des termes issus de théories mathématiques.

2 Satisfiabilité Modulo Théories

Les formules logiques sur lesquelles nous portons notre attention sont issues de plateformes de preuve déductives de programmes comme Why3 [43]. Dans ces outils, un programme est tout d'abord spécifié formellement, puis des formules mathématiques sont extraites à l'aide d'un calcul de plus faibles préconditions pour s'assurer que, si elles sont valides, le programme respecte bien sa spécification.

Les formules engendrées par des outils comme Why3 sont des formules du premier ordre qui contiennent des connecteurs logiques et des symboles de fonctions de nombreuses théories comme l'égalité, l'arithmétique, les tableaux, etc. Le problème de la satisfiabilité dans cette logique est appelé *satisfiabilité modulo théories* (ou SMT).

La figure 1.3 représente une telle formule de logique du premier ordre. Plusieurs théories sont présentes dans cet exemple, comme l'arithmétique entière $(x > 4)$ et rationnelle $(f(y) \leq 2.0)$ ainsi que la théorie de l'égalité $(x = y)$. Cette formule contient aussi le terme universellement quantifié

$\forall i : int.\ z : int\ list.\ cons(i, z) \neq nil.$

Les solveurs de satisfiabilité modulo théories, solveurs SMT, sont généralement basés sur un solveur SAT qui en forme le noyau, combiné à un solveur de théories et un moteur d'instanciation. L'algorithme effectuant cette combinaison est généralement appelé CDCL(T) [59, 45]. Le solveur SAT effectue le raisonnement sur les variables booléennes. Les atomes mathématiques ($=, \leq, \neq$) sont abstraits sous forme de variables booléennes. Une fois ces variables assignées, elles sont transmises aux solveurs de théories.

Ce solveur combine différentes procédures de décision sur des théories. Par exemple, l'algorithme du simplexe [29] et la méthode de Fourier-Motzkin [52] sont utilisés pour décider des formules comportant des opérations d'arithmétique rationnelle linéaire sans quantificateurs. Deux méthodes sont historiquement utilisées pour combiner efficacement les différentes procédures de décision. La méthode dite de Shostak [64], et celle de Nelson-Oppen [57].

Instanciation des quantificateurs par filtrage Dans beaucoup d'applications du raisonnement modulo théories, la gestion et résolution des problèmes quantifiés est nécessaire. Nous pouvons l'observer par exemple pour la preuve de programme, dans laquelle les formules quantifiées sont utilisées pour encoder les modèles mémoires.

Les solveurs SMT supportant les problèmes quantifiés utilisent en général des systèmes d'instanciation qui sont guidés par des gardes de déclenchement, *triggers*. Par exemple nous reprenons la formule : $(\forall\ i : {}'a.\ z : {}'a\ list.\ cons(i, z) \neq nil)$, plus générique et polymorphe que celle présente dans la figure 1.3. En effet sur la figure 1.3 i est de type int et z de type $intlist$ contrairement à notre formule $(\forall\ i : {}'a.\ z : {}'a\ list.\ cons(i, z) \neq nil)$ qui quantifie aussi sur la variable de type $'a$. Une garde possible de cette formule serait $cons(i, z)$, car elle couvre toutes les variables de types et de termes de l'expression. Une garde peut parfois être constituée de plusieurs termes pour couvrir toutes les variables de types et termes de l'expression. Sur l'exemple suivant : $(\forall\ i, j : int.\ t : {}'a\ array.\ f(i) \neq f(j) \Rightarrow get(f(i), t) \neq get(f(j), t))$, nous pouvons prendre la garde $[get(f(i), t), f(j)]$ pour couvrir toute les variables de types et de termes de l'expression.

La génération d'instance guidée par ces gardes consiste à trouver une substitution σ pour la garde $[g]$ de la formule $\forall\ \vec{x}\ [g].\ F(\vec{x})$ tel que : $\epsilon \models (g\ \sigma) = t$, où t est un terme du modèle booléen courant et ϵ est la relation d'équivalence construite par les procédures de décision des théories. Nous pouvons alors produire l'instance $F(\vec{x})\ \sigma$. La substitution σ est calculée en utilisant le matching modulo égalités. La technique la plus répandue pour trouver des substitutions qui rendent des déclencheurs égaux à des termes du modèle booléen est basée sur le E-matching [56].

Nous reprenons sur la figure 1.4 l'algorithme DPLL (présenté sur la figure 1.2) étendu naïvement par l'interaction avec le solveur de théories et le

Input: Γ : SAT environment, T : Theory solver environment
Output: Satisfiability status

```
1  Function solve(Γ, T)
2      (Γ, Error) ← propagate(Γ)          // Boolean constraint propagation
3      if Error then
4          return UNSAT
5      else
                                          // Theory constraint propagation
6          (T, Error) ← theory_propagate(Γ, T)
7          if Error then
8              return UNSAT
9          else
10             if all variables are assigned then
11                 I ← instantiate(Γ, T)
12                 if I ≠ ∅ then
13                     Γ ← Γ ∪ to_CNF(I)       // add new instances to Γ
14
15                     return solve(Γ, T)
16                 else
17                     return SAT
18             else
19                 A ← choose_lit_to_decide(Γ)
20                 status ← solve(Γ ∪ {A}, T)          // decide on A
21
22                 if status = UNSAT then
23                     return solve(Γ ∪ {¬A}, T)       // propagate ¬A
24
25                 else
26                     return status
```

FIGURE 1.4 – Algorithme DPLL étendu naïvement par l'interaction avec un solveur de théories ainsi qu'un moteur d'instanciation

moteur d'instanciation. Une fois que la propagation de contrainte booléenne est effectuée sans conflit, nous communiquons au solveur de théories l'environnement du solveur SAT contenant les littéraux assignés. La cohérence de ces assignations est ensuite vérifiée modulo théories. Si ces assignations ne mènent à aucun conflit on retourne l'environnement du solveur de théories et on regarde si toutes les variables sont assignées. Si c'est le cas, on appelle la fonction $instanciate(\Gamma, T)$ (ligne 10) qui demande au moteur d'intanciations de créer des instances grâce au modèle booléen et à la valeur des termes modulo théories. Si aucune instance n'est créée (ligne 14) on retourne alors SAT. Sinon on met en CNF les formules des instances fraîchement créées. On appelle ensuite récursivement $solve(\Gamma, T)$ avec un Γ contenant ces formules en CNF.

2.1 Le solveur SMT Alt-Ergo

Alt-Ergo est un solveur SMT qui fut notre objet d'étude au cours de cette thèse. Il est développé au Laboratoire de Recherche en Informatique (LRI) de l'université Paris-Sud. Son développement débuta en 2006 et est maintenu depuis 2013 par l'entreprise OCamlPro en collaboration avec l'équipe VALS du LRI. Ce solveur fut développé dans le but de résoudre automatiquement des formules mathématiques issues de plates-formes de preuve de programmes. Son développement fut guidé par les besoins de la plate-forme Why/*Why3* qui utilise une spécification riche via un langage polymorphe à la ML [42].

Pour gérer directement les formules issues de cette plate-forme, Alt-Ergo possède un langage natif se rapprochant de cette syntaxe polymorphe. Il possède aussi des capacités de raisonnement sur des structures de données définies par l'utilisateur, sur les énumérations et tableaux, sur des formules quantifiées et sur des opérations arithmétiques entières et réelles. En 2017, le support pour l'arithmétique des flottants a été également ajouté.

2.2 Le standard SMT-LIB 2

De nombreux solveurs SMT ont vu le jour à partir des années 2000, accompagnés d'une tentative de standardisation. Ce standard, SMT-LIB, fut proposé par Silvio Ranise et Cesare Tinelli [61] en 2003. Les buts étaient multiples : SMT-LIB devait proposer un standard pour les théories utilisées par les solveurs SMT via des descriptions rigoureuses. Il devait permettre d'avoir une unification des entrées et sorties des solveurs grâce à un nouveau langage de description pour les problèmes. Un autre but était de connecter les différents acteurs du monde de la satisfiabilité modulo théories en créant une communauté de chercheurs, développeurs et utilisateurs. Pour cela l'initiative fut accompagnée par l'idée de créer une grande bibliothèque de problèmes accessible à tous.

```
(and (or (and (> x 4) (<= (f(y)) 2.0)) (= x y))
     (forall ((i Int)(z (List Int))) (not (= nil (cons i z)))))
```

FIGURE 1.5 – Terme représentant la formule 1.2 au format SMT-LIB 2.

Le langage standardisé est un point clé de l'initiative SMT-LIB pour créer et entretenir les points présentés précédemment. Nous allons maintenant présenter la syntaxe et la sémantique de ce langage standardisé. Le langage a évolué au fil des années [62, 8, 5], la version la plus récente est la 2.6 [7]. Ce standard est un langage à ligne de commande avec une syntaxe parenthésée proche du langage de programmation LISP. Chaque terme doit être parenthésé comme le montre l'exemple de la figure 1.5 représentant la formule de la figure 1.2 au format SMT-LIB 2.

Un problème au format SMT-LIB 2 est avant tout composé de commandes ; une partie de ces commandes est présentée en figure 1.6. Les premières commandes concernent le cadre du problème. La commande `set-logic` permet de fournir au solveur des informations sur les théories dont il aura besoin pour résoudre le problème. Les commandes `set-info` sont présentes pour donner des informations sur l'origine du problème, sur son statut ou encore sur la version du standard utilisé. Les commandes `set-option` permettent par exemple de définir des options comme la production de modèle (`:produce-models`) ou d'unsat-core (`:produce-unsat-cores`).

Les commandes de déclaration et de définition permettent, quant à elles, l'ajout de types (`declare-sort`), de constantes (`declare-const`), et de symboles de fonctions (`declare-fun`, `define-fun`...) au problème. Ces ajouts peuvent ensuite être utilisés dans des termes et des assertions spécifiant le problème avec la commande `assert`. La commande `check-sat` permet de vérifier si le problème est satisfiable. Des commandes peuvent permettre de demander des informations au solveur concernant la résolution du problème comme par exemple la commande `get-unsat-core` qui demande au solveur de retourner un ensemble de formules rendant le problème insatisfiable si l'option `:produce-unsat-cores` est activée.

3 Contributions de la thèse

Cette thèse présente différents travaux permettant une amélioration significative des performances du solveur Alt-Ergo ainsi que diverses modifications de ce dernier nous ayant permis d'être compatibles avec les besoins de la communauté via le support d'un langage standardisé.

3.1 Étude des performances d'un solveur CDCL en OCaml

Des expérimentations préliminaires à cette thèse ont porté sur le remplacement du solveur SAT historique d'Alt-Ergo par méthode des tableaux, par

```
(set-info :smt-lib-version 2.6)
(set-logic AUFNIRA)
(set-info :category "crafted")
(set-info :status unknown)
(set-option :print-success false)

(declare-sort S 0)

(declare-const y Int)
(declare-fun f (Real Int) Real))

(define-fun g ((x Real)) Real (f x 3))
(define-fun-rec h ((c Bool)(d Int)) Bool
  (ite (= d 0) (h (not c) (- d 1)) c))

(define-funs-rec
  ((v ((a S)(b Int)) Int)
   (w ((c Int)(d S)) Int))
        ((w b a)
        (v d c)))

(assert (forall ((x Int)(y Int)) (= (+ x 1) y)))

(check-sat)
(exit)
```

Figure 1.6 – Exemple de commandes au format SMT-LIB 2.

un solveur SAT de référence (Minisat). Malheureusement cette expérimentation a montré de moins bons résultats qu'avec le solveur SAT historique de ce dernier. Une rapide comparaison entre deux implémentations de Minisat dans son langage de programmation C++ et une implémentation en OCaml ont permis de détecter des écarts de performances significatifs. Plusieurs questions ont été levées, mais sont restées sans réponse. Nous y répondons dans cette thèse. La première question portait sur les différences entre les deux implémentations C++ et OCaml. Bien qu'elles respectaient le même algorithme, de légères différences pouvaient faire varier les décisions et les résultats. Notre premier travail a donc porté sur l'implémentation rigoureusement exacte en terme fonctionnel du solveur de référence, les seules différences étant structurelles et non algorithmiques. Pour éviter au maximum des différences structurelles trop importantes, nous avons décidé d'utiliser les mêmes structures de données dans les deux implémentations. Les seules différences sont ainsi liées au langage de programmation utilisé. Nous avons ensuite étudié les différences de performances et essayé de trouver les raisons de celles-ci. Les travaux préliminaires soupçonnaient que les structures de données et la gestion automatique de la mémoire avaient un impact négatif sur les performances. Ce travail ainsi que nos conclusions sont très proches des travaux de Cruanes [28]. Enfin, nous comparerons le solveur SAT historique d'Alt-Ergo et l'implémentation du solveur SAT de référence sur des problèmes SAT, nous permettant ainsi d'étudier leur capacité de raisonnement booléen.

3.2 Intégration efficace d'un solveur CDCL dans un solveur SMT

Suite à ces travaux sur les performances du solveur SAT, nous avons décidé de nous pencher sur la combinaison de ce dernier au sein du solveur Alt-Ergo. Du fait que ses performances étaient semblables à celles du solveur de référence et bien meilleures que celles du solveur historique sur des problèmes purement booléens, nous avons validé le fait que les mauvaises performances de ce solveur dans Alt-Ergo sur des problèmes SMT n'étaient pas dues au solveur SAT mais à son intégration avec le combinateur de théories et le moteur d'instanciation.

Dans un premier temps nous avons assisté le solveur historique d'Alt-Ergo en le combinant au solveur CDCL performant pour le raisonnement booléen. Cela a permis d'obtenir une amélioration des résultats, nous motivant à approfondir cette problématique. Dans un second temps, nous avons formalisé et implémenté une solution pour améliorer l'intégration d'un solveur CDCL performant au sein d'Alt-Ergo. Cette solution fut guidée par le fonctionnement du solveur historique. Nous avons en effet utilisé le fonctionnement de la méthode des tableaux pour réduire le modèle booléen transmis aux théories et au moteur d'instanciation. Cette méthode se rapproche de la

technique de pertinence [33] utilisée par le solveur Z3 [32].

Plusieurs optimisations d'implémentation nous ont permis d'obtenir une solution efficace et performante que nous avons testée et comparée. Nous avons testé l'intérêt de notre optimisation sur les différentes parties d'un solveur SMT. Nous avons ainsi testé l'impact d'une telle méthode sur la combinaison de théories ainsi que sur le moteur d'instanciation.

3.3 Extension du standard SMT-LIB 2 avec du polymorphisme

Le manque de support du standard SMT-LIB 2 par Alt-Ergo nous empêche d'avoir accès à la communauté SMT et à sa base de fichiers de test. De plus, cela nous contraint pour comparer justement notre solveur aux solveurs de l'état de l'art sur des fichiers d'entrées identiques.

Nous avons décidé d'ajouter le support du langage de la SMT-LIB 2 à Alt-Ergo. Le polymorphisme faisant partie intégrante du langage natif de notre solveur, nous avons souhaité modifier le standard pour y ajouter une extension polymorphe. Des travaux [16] ont d'ailleurs déjà tenté d'ajouter un support pour le polymorphisme dans SMT-LIB 2. Malheureusement ces travaux ne se focalisent que sur l'analyse lexicale et syntaxique du polymorphisme. Contrairement à ces travaux et grâce au support natif du polymorphisme par Alt-Ergo dans son moteur de raisonnement, nous souhaitons montrer l'avantage de traiter nativement le polymorphisme dans un solveur SMT.

Durant cette thèse, nous avons développé une bibliothèque permettant d'effectuer l'analyse lexicale et syntaxique du standard de la SMT-LIB 2 avec une extension polymorphe conservative et non intrusive. Alt-Ergo utilise cette bibliothèque pour supporter le standard. Grâce à l'outil *Why3* nous avons pu générer un ensemble de bancs de test sous différents formats. Cela nous a permis de mettre en évidence l'importance du support natif du polymorphisme par rapport à la monomorphisation. Nous avons aussi pu comparer Alt-Ergo aux autres solveurs de la communauté sur des problèmes d'entrée strictement identiques.

3.4 Participation à la compétition SMT-COMP

L'ajout du support pour le standard SMT-LIB 2 nous a aussi permis de lancer Alt-Ergo sur les bancs de test de la communauté. Des résultats prometteurs sur certains de ces bancs de test nous ont motivés à nous lancer dans la compétition SMT de 2018.

En préparation pour la participation à la compétition, nous avons apporté quelques modifications à Alt-Ergo pour obtenir de meilleurs résultats. Des efforts au sujet de l'instanciation ont été effectués, en particulier concernant les calculs des déclencheurs. Nous avons aussi effectué des expérimentations

quant aux heuristiques d'Alt-Ergo pour obtenir un jeu d'option optimal. Le hardware de la compétition nous permettant d'utiliser jusqu'à quatre coeurs par problème, nous avons ajouté une option permettant de lancer plusieurs instances d'Alt-Ergo avec des options différentes. Cette parallélisation nous a permis d'optimiser les temps d'exécution par rapport à une exécution séquentielle du jeu d'option optimal.

Ces optimisations nous ont permis d'améliorer nos résultats préliminaires et ont permis à Alt-Ergo de sortir de cette participation à la SMT-COMP 2018 comme un solveur SMT performant et concurrentiel.

4 Plan de la thèse

Cette thèse est divisée en six chapitres qui seront organisés de la manière suivante : le chapitre 2 présente le solveur de satisfiabilité modulo théories Alt-Ergo. Dans un premier temps nous présentons le fonctionnement du solveur SAT d'Alt-Ergo, le différenciant des autres solveurs SMT et principal objet d'étude de cette thèse. Du fait que nos travaux ne concernent pas le solveur de théories et le moteur d'instanciation de ce dernier, nous nous focalisons principalement sur son solveur SAT. Puis nous présentons dans un second temps son langage d'entrée qui diffère de celui de la communauté.

Le chapitre 3 présente nos travaux de comparaison entre deux implémentations en OCaml et C++ du solveur SAT Minisat. Nos travaux concernant l'intégration efficace d'un solveur CDCL performant au sein d'un solveur SMT sont présentés dans le chapitre 4.

Les chapitres 5 et 6 se concentrent sur la communauté SMT. Le premier concerne nos travaux pour l'extension du langage standardisé SMT-LIB 2 avec du polymorphisme et son support par Alt-Ergo. Le second concerne notre participation à la compétition SMT-COMP 2018. Enfin, le chapitre 7 donne des pistes d'amélioration et d'extensions de nos travaux et conclut ce document.

Chapitre 2

Le solveur Alt-Ergo

Dans ce chapitre, nous présenterons l'architecture générale d'Alt-Ergo en nous focalisant sur le fonctionnement de son solveur SAT, principale différence par rapport au schéma et fonctionnement type d'un solveur SMT. Nous présenterons ensuite son langage d'entrée historique et ses particularités par rapport au langage SMT-LIB 2 de la communauté SMT.

1 Architecture

Alt-Ergo possède une architecture à trois composants, un solveur SAT, un solveur de théories et un moteur d'instanciation. La figure 2.1 représente l'architecture globale du solveur SMT Alt-Ergo. Premièrement la partie "Frontend" est composée de trois entrées différentes : `Alt-Ergo` l'entrée en ligne de commande, `AltGr-Ergo` l'interface graphique [26] et l'API. Une fois les entrées syntaxiquement analysées, nous vérifions leur typage : c'est durant cette phase que le calcul des déclencheurs pour les formules quantifiées est ef-

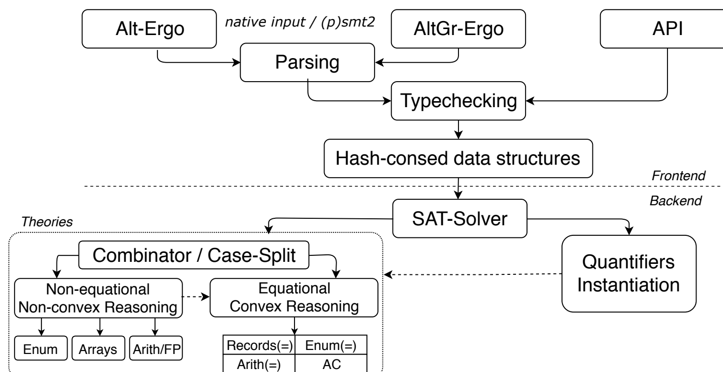

FIGURE 2.1 – Architecture du solveur SMT Alt-Ergo.

$$A \leftrightarrow x > 4 \quad (2.1a)$$
$$B \leftrightarrow f(y) \leq 2.0 \quad (2.1b)$$
$$C \leftrightarrow x = y \quad (2.1c)$$
$$D \leftrightarrow (\forall i : int.\ z : int\ list.\ cons(i, z) \neq nil) \quad (2.1d)$$

FIGURE 2.2 – Ajout de variable d'abstraction pour les atomes des théories.

$$X \leftrightarrow A \wedge B \quad (2.2a)$$
$$Y \leftrightarrow X \vee C \quad (2.2b)$$
$$Z \leftrightarrow Y \wedge D \quad (2.2c)$$

FIGURE 2.3 – Ajout de variable d'abstraction pour chaque opération non atomique.

fectué. Nous utilisons une technique de partage maximum (hashconsing [20]) pour nos structures de données évitant ainsi de devoir allouer plusieurs fois la représentation d'une même formule, avant d'être envoyées au solveur SAT.

Le solveur SAT historique d'Alt-Ergo fonctionne différemment d'un solveur DPLL ou CDCL, ce qui influe à la fois sur la gestion de la CNF et sur les interactions avec les combinaisons de théories et le moteur d'instanciation.

1.1 Gestion paresseuse de la CNF

Le développement d'Alt-Ergo commença peu après la sortie du solveur Simplify [36]. Ce solveur utilisait une gestion de la CNF dite paresseuse. Alt-Ergo utilise lui aussi cette technique. Elle consiste à ne pas mettre le problème en CNF avant de l'envoyer au solveur SAT mais à travailler avec la formule du problème en Forme Normale Négative (FNN). Cette forme a pour avantage d'avoir une transformation linéaire contrairement à la mise en CNF.

Nous reprenons l'exemple précédemment utilisé de la figure 1.3. Des variables abstraites sont introduites pour les atomes des théories comme sur la figure 2.2. Les variables d'abstraction des opérations non atomiques sont elles présentées dans la figure 2.3. Contrairement au fonctionnement traditionnel de mise en CNF, nous travaillons directement avec ces formules. Z représentant l'abstraction de la formule initiale, sa valeur de vérité doit être mise à vraie. Les éléments de sa conjonction le sont aussi par BCP. Nous

savons donc que Y et D sont vrais avant même de demander au solveur SAT de décider.

L'aspect paresseux apparaît pour le traitement de Y. Dans une disjonction, un terme est exploré uniquement s'il est assigné par décision ou propagation à vrai. Si par exemple C est décidé, nous n'avons pas besoin de considérer X et les sous termes et formules qu'il contient.

Nous allons maintenant présenter le fonctionnement d'un solveur SAT se basant sur une telle gestion des formules.

1.2 Solveur SAT historique d'Alt-Ergo

Le solveur SAT d'Alt-Ergo possède un comportement proche de celui du solveur Simplify concernant la gestion de la formule de base sous forme d'une CNF paresseuse. C'est la principale différence avec les solveurs SMT traditionnels. Pour rappel, les solveurs SAT DPLL et CDCL fonctionnent comme des arbres de décisions sur les littéraux non assignés des disjonctions. Contrairement à ces solveurs, lors d'un retour en arrière nous essayons d'assigner un autre littéral de la disjonction à vrai et non de tester la négation de ce dernier. Si l'on prend l'exemple $A \vee B$, si le choix d'assigner A à vrai amène à un conflit alors on va propager A à faux et propager B à vrai pour que la disjonction binaire soit vraie.

Comme le présente la figure 2.4, le fonctionnement synthétique du solveur ainsi que ses structures de données sont fonctionnels. Cela nous permet d'avoir des retours en arrière peu coûteux. Nous travaillons, comme présenté auparavant, sur un ensemble de formules représentant le problème de base en NNF et non sur sa CNF. En pratique, Φ ne contient que des disjonctions binaires. L'élimination des conjonctions est effectuée dans la fonction *assume*. Lors d'un conflit pendant la propagation des contraintes booléennes, nous retournons la réponse UNSAT ainsi qu'une raison. Cette raison contient l'ensemble des décisions ayant amené des éventuelles propagations rendant une clause fausse. Cette raison est ensuite utilisée pour le retour en arrière non chronologique. À la ligne 15, si le littéral A qui vient d'être décidé est dans la raison, alors cela signifie que cette décision a mené à un conflit. Si elle n'est pas dans la raison alors on revient au niveau de décision précédent. Lorsqu'un littéral A a mené à un conflit, on propage l'autre littéral de la disjonction, B en plus de $\neg A$, pour que la disjonction soit vraie (ligne 16). S'il n'existe plus de disjonctions non satisfaites cela signifie que le problème est SAT.

Les propagations de contraintes booléennes sont effectuées différemment comparées aux solveurs DPLL et CDCL. L'algorithme de la figure 2.5 présente le fonctionnement du BCP de ce solveur SAT. Pour toutes les disjonctions binaires que Φ contient (ligne 6), nous recherchons si elles sont vraies. Si les deux membres de la disjonction sont faux cela signifie que la clause n'est pas satisfiable et que nous avons donc un conflit. Si un des membres

```
Input: Φ : Set of formulas in Negatif Normal Form
Output: Satisfiability status, explanation
1  Function solve(Φ)
2      (Φ, Error) ← propagate(Φ)          // Boolean constraint propagation
3      if Error then
4          reason ← explain_conflict()
5          return (UNSAT,reason)
6      else
7          if ∃A ∨ B ∈ Φ then
8              (Φ, Error) ← (assume(Φ, [{A}])                // decide on A
9              if Error then
10                 reason ← explain_conflict()
11                 return (UNSAT,reason)
12             else
13                 (status, reason) ← solve(Φ)
14                 if status ≠ UNSAT then
15                     return (status,reason)
16                 else
17                     if A ∈ reason then
18                         (Φ, Error) ← (assume(Φ, [{¬A}; {B}])
                           // backtrack and propagate ¬A and B
19                         if Error then
20                             reason ← explain_conflict()
21                             return (UNSAT,reason)
22                         else
23                             return solve(Φ)
24                     else
25                         return (UNSAT,reason)         // backjump further
26         else
27             return (SAT,∅)                            // Φ is empty
28
```

FIGURE 2.4 – Algorithme fonctionnel décrivant le fonctionnement du solveur SAT historique d'Alt-Ergo basé sur la méthode des tableaux.

```
Input:  Φ : set of formula in NNF
Output: Φ, Error : boolean
 1 Function propagate(Φ)
 2   Error ← false
 3   Work ← true
 4   while Work do
 5     Work ← false
 6     for clauses C of the form A ∨ B in Φ do
 7       if A ∈ Φ or B ∈ Φ then
 8         Φ ← remove(Φ, C)
 9       else if ¬A ∈ Φ and ¬B ∈ Φ then
10         Error ← true
11         return (Φ,Error)        // clause A ∨ B is unsatisfiable
12       else if ¬B ∈ Φ then
13         Φ ← remove(Φ, C)
14         (Φ, Error) ← assume(Φ, [A])               // propagate A
15
16         if Error then
17           return (Φ,Error)
18         else
19           Work ← true
20       else if ¬A ∈ Φ then
21         Φ ← remove(Φ, C)
22         (Φ, Error) ← assume(Φ, [B])               // propagate B
23
24         if Error then
25           return (Φ,Error)
26         else
27           Work ← true
28   return (Φ,Error)
```

Figure 2.5 – Algorithme de propagations de contraintes booléennes du solveur SAT par méthode des tableaux

```
Input: Φ : set of formulas, L : list of formulas
Output: Φ, Error
1  Function assume(Φ, L)
2      Error ← false
3      for formula F in L do
4          if ¬F ∈ Φ then
5              Error ← true
6              return (Φ, Error)
7          else
8              if F ∉ Φ then
9                  if F is a conjonction A ∧ B then
10                     (Φ, Error) ← assume(Φ, [A; B])
11                     if Error then
12                         return (Φ, Error)
13                 else
14                     if F is a disjonction A ∨ B then
15                         Φ ← Φ ∪ (A ∨ B)
16     return (Φ, Error)
```

FIGURE 2.6 – Algorithme d'ouverture parresseuse de la CNF avec élimination de conjonctions

est vrai (ligne 7) (ou que l'autre est faux et donc la clause est unitaire) on supprime la clause de Φ car elle est satisfaite. Lorsqu'un des deux membres est faux, on supprime la clause dans Φ puis on appelle ensuite la fonction *assume* qui va traiter le membre vrai. La figure 2.6 présente son fonctionnement. Pour toute formule F à traiter, on vérifie que sa négation n'est pas déjà dans Φ ce qui causerait un conflit, car F est vraie. Si la formule est une conjonction binaire, on suppose récursivement ses deux sous-formules. Si la formule est une disjonction binaire on ajoute la disjonction à Φ. On s'assure ainsi que Φ ne contient que des disjonctions

Le schéma de la figure 2.7 représente une exécution de l'algorithme 2.4. Dans cet exemple X est la variable d'abstraction qui représente le problème initial. On choisit tout d'abord un littéral à décider dans la disjonction $A \vee B$ contenue dans X. A est choisi. On propage alors I et J, car comme A est vrai, les éléments de sa conjonction le sont aussi. On décide maintenant sur C. On remarque que la décision suivante sur E amène un conflit, on propage alors $\neg E$ et F pour que J soit vrai. Ces deux possibilités mènent à des conflits, on revient donc à la décision sur C et on propage D qui nous permet d'obtenir une réponse SAT. On note que dans cet exemple B n'est pas utilisé, ainsi que les valeurs contenues dans sa conjonction et celles contenues dans leurs

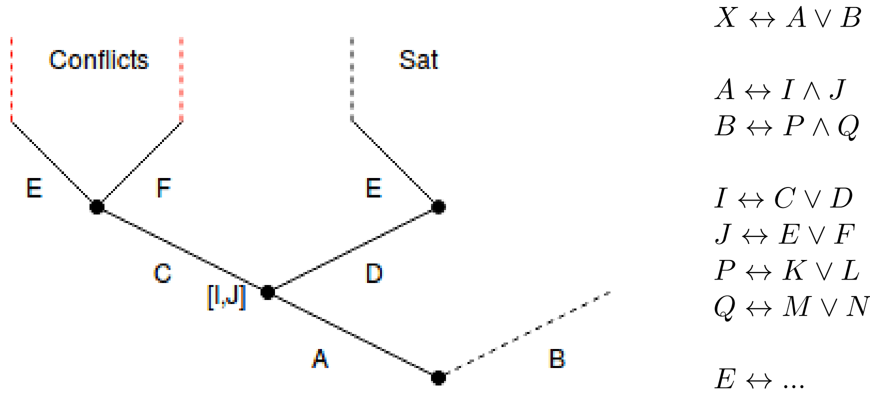

FIGURE 2.7 – Représentation sous forme d'arbre du parcours effectué par l'algorithme du solveur SAT historique d'Alt-Ergo.

disjonctions. Le solveur SAT n'a donc travaillé que sur un ensemble restreint de sous-formules.

1.3 Gestion des théories et des termes quantifiés dans Alt-Ergo

Alt-Ergo supporte de multiples théories telles que par exemple la théorie de l'arithmétique rationnelle et entière ou encore la théorie des tableaux. Pour pouvoir supporter plusieurs théories, Alt-Ergo utilise un combinateur de théories. La plupart des solveurs SMT utilise un combinateur de théories basé sur la technique de Nelson-Oppen [57] tandis qu'Alt-Ergo utilise un combinateur basé sur la technologie de Shostak [23, 49] pour l'égalité sur les théories convexes. Ce combinateur fut étendu pour supporter les symboles de fonctions associatives et commutatives [22]. Concernant le raisonnement sur l'arithmétique non linéaire, Alt-Ergo utilise un système de calcul d'intervalles. L'algorithme de fermeture de congruence [2, 58] ainsi que la procédure de ground completion [66] permettent de décider des conjonctions de littéraux modulo la théorie de l'égalité.

Comme présenté sur la figure 2.1, ces combinateurs permettent à Alt-Ergo de prendre en charge de multiples théories. Alt-Ergo fournit ainsi le raisonnement sur l'arithmétique linéaire et non linéaire sur les entiers et les rationnels grâce à l'algorithme du Simplex [29] et la méthode de Fourier Motzkin [52]. Alt-Ergo fournit aussi un raisonnement sur les types de données algébriques comme les énumérations et les records, ainsi que les tableaux. Un prélude d'axiomatisation de la théorie des flottants [25] permet à Alt-Ergo de raisonner sur cette théorie.

Les principales différences qu'amène l'ajout des combinaisons de théories par rapport aux algorithmes précédents sont principalement situées dans la

```
Input: Φ : set of formulas, T : Theory environment, L : list of formulas
Output: Φ, T, Error
1  Function assume(Φ, T, L)
2      Error ← false
3      for formula F in L do
4          if ¬F ∈ Φ then
5              Error ← true
6              return (Φ, Error)
7          else if F ∉ Φ then
8              if F is a conjonction A ∧ B then
9                  (Φ, Error) ← assume(Φ, T, [A; B])
10                 if Error then
11                     return (Φ, T, Error)
12             else if F is a disjonction A ∨ B then
13                 Φ ← Φ ∪ (A ∨ B)
14             else if F is a litteral l then
15                 (T, Error) ← propagate_in_theory(T, l)
16                 if Error then
17                     return (Φ, T, Error)
18             else if F is an ∃ x. G then
19                 (Φ, T, Error) ← assume(Φ, T, [Skolemize(G(x))])
20                 if Error then
21                     return (Φ, T, Error)
22             else if F is an ∀ x. G then
23                 Φ ← Φ ∪ F                         // add for instantiation
24
25     return (Φ, T, Error)
```

FIGURE 2.8 – Algorithme d'ouverture parresseuse de la CNF avec élimination de conjonctions, intéraction avec les théories et skolémisation

fonction *assume* que nous présentons dans la figure 2.8. Lorsqu'une formule est un littéral, nous le transmettons au combinateur de théories (ligne 15). Si la formule est une formule existentiellement quantifiée nous la skolémisons (ligne 19). Enfin, si la formule est universellement quantifiée, nous l'ajoutons à Φ pour qu'elle puisse être instanciée (ligne 23). Φ ne contient plus seulement des disjonctions mais aussi des formules quantifiées.

Notre solveur SAT travaillant sur un ensemble potentiellement réduit de termes, le nombre de termes envoyés au combinateur de théories pour vérification du modèle diffère par sa taille mais aussi par les décision et propagations effectuées par rapport au solveur SMT basé sur un algorithme DPLL.

Concernant le fonctionnement du moteur d'instanciation d'Alt-Ergo, il est basé sur E-matching [56]. Du fait que nos travaux ne portent pas sur la gestion des instances, nous nous intéressons uniquement à l'interaction du moteur d'instanciation avec le solveur SAT.

La figure 2.9 nous présente les modifications apportées à l'algorithme de notre solveur SAT pour supporter l'interaction avec les théories ainsi qu'avec le moteur d'instanciation. La fonction de propagation prend maintenant l'environnement des théories en paramètre. Nous ne présenterons pas les modifications apportées à la fonction *propagate* car elles sont mineures et consistent en l'ajout de l'environnement T en paramètre ainsi que dans le type de retour de la fonction *assume*, comme c'est le cas pour les appels à la fonction *assume* des lignes 7, 16 et 27. La fonction *instantiate* permet de calculer les instances des formules quantifiées contenues dans Φ ajoutées par la fonction *assume*. Dû à l'aspect fonctionnel de la fonction *solve*, les instances sont oubliées lorsqu'un retour en arrière est effectué. Cela nous évite d'avoir à nettoyer une base de formules provenant d'instances.

Contrairement au solveur DPLL(T) qui retourne SAT si les procédures de décision sont complètes, Alt-Ergo ne peut retourner que "Unknown" (ligne 34). Cela est dû au fait qu'Alt-Ergo est un solveur conçu pour la preuve de programme, et que pour ce domaine d'application, nous souhaitons montrer l'insatisfiabilité du problème, et jamais sa satisfiabilité. Peu d'efforts ont donc été produits pour qu'Alt-Ergo puisse répondre SAT.

2 Le langage natif d'Alt-Ergo

Du fait qu'Alt-Ergo a été développé en partie pour les besoins de preuve de programme de la plate-forme Why, son langage d'entrée est très proche du langage logique de cet outil. La plate-forme Why ayant pour but principal la vérification de programmes, il y avait un besoin de pouvoir manipuler des types de données polymorphes pour pouvoir représenter des modèles mémoire, par exemple.

La figure 2.10 nous présente un exemple de syntaxe de ce langage. Cette

Input: Φ : Set of formulas in NNF, T : Theory environment
Output: Satisfiability status, explanation

```
1  Function solve(Φ, T)
2    (Φ, T, Error) ← propagate(Φ, T)                      // BCP modulo Theory
3    if Error then
4      reason ← explain_conflict()
5      return (UNSAT,reason)
6    else
7      if ∃A ∨ B ∈ Φ then
8        (Φ, T, Error) ← (assume(Φ, T, [{A}])               // decide on A
9        if Error then
10         reason ← explain_conflict()
11         return (UNSAT,reason)
12       else
13         (status, reason) ← solve(Φ, T)
14         if status ≠ UNSAT then
15           return (status,reason)
16         else
17           if A ∈ reason then
18             (Φ, T, Error) ← (assume(Φ, T, [{¬A}; {B}])
               // backtrack and propagate ¬A and B
19             if Error then
20               reason ← explain_conflict()
21               return (UNSAT,reason)
22             else
23               return solve(Φ, T)
24           else
25             return (UNSAT,reason)             // backjump further
26     else
27       I ← insantiate(Φ, T)
28       if I ≠ ∅ then
29         (Φ, T, Error) ← (assume(Φ, T, I))          // assume instances
30         if Error then
31           reason ← explain_conflict()
32           return (UNSAT,reason)
33         else
34           return solve(Φ, T)
35       else
36         return UNKNOWN
```

FIGURE 2.9 – Algorithme fonctionnel décrivant le fonctionnement du solveur historique d'Alt-Ergo basé sur la méthode des tableaux avec ses interactions avec le solveur de théories T et le moteur d'instanciation

```
type 'a set

logic empty : 'a set
logic add : 'a , 'a set -> 'a set
logic mem : 'a , 'a set -> prop

axiom mem_empty:
  forall x : 'a.
  not mem(x, empty : 'a set)

axiom mem_add:
  forall x, y : 'a. forall s : 'a set.
  mem(x, add(y, s)) <-> (x = y or mem(x, s))

logic is1, is2 : int set
logic iss : int set set

predicate not_mem(x: 'a, s: 'a set) = not mem(x, s)

goal g_4:
  is1 = is2 -> (not_mem(1, add(2+3, empty : int set)))
  and mem (is1, add (is2, iss))
```

Figure 2.10 – Exemple de problème au format natif d'Alt-Ergo.

syntaxe diffère du langage de la SMT-LIB 2, et se rapproche plus de la logique du premier ordre polymorphe. Ce langage fait en effet une différence entre des propositions et des variables booléennes. Cela rend alors impossible le fait d'avoir des propositions à l'intérieur de termes, ce qui n'est pas le cas dans le standard SMT-LIB 2.

Contrairement à SMT-LIB 2, ce langage comporte peu de commandes. La commande `type` permet de déclarer des symboles de type qui peuvent être polymorphes avec une notation à la ML. La commande `logic` permet de déclarer des symboles de fonction comme `add` et `mem` et des symboles de constantes comme `empty` ou `iss`. `predicate` permet de déclarer des fonctions dont le corps est connu et retourne un `prop`. `prop` est un type de données correspondant à une proposition booléenne. La commande `axiom` correspond à des assertions. Enfin, la commande `goal` représente, pour la preuve de programme, la propriété à démontrer. Contrairement au format SMT-LIB 2, nous dissocions les axiomes, les propriétés des fonctions à vérifier, et le but à prouver. Cette différence permet de guider la résolution.

Le polymorphisme représenté par la notation `'a` dans la figure 2.10 possède une syntaxe à la ML. Tout comme pour OCaml, la quantification des variables de type est faite implicitement, nous n'avons pas de `forall 'a. forall x: 'a. ...`. Ce polymorphisme s'exprime donc simplement et est géré nativement par le solveur Alt-Ergo [12] à l'intérieur de son moteur d'instanciation.

Dans ce chapitre, nous avons présenté le solveur SMT Alt-Ergo. Les principales différences avec un solveur SMT classique sont son solveur SAT proche de la méthode des tableaux ainsi que son langage d'entrée.

Première partie

SAT et SMT solveur

Chapitre 3

Vers un solveur SAT efficace en OCaml

Bien que le solveur SAT historique d'Alt-Ergo ait été perfectionné au fur et à mesure des années, grâce au retour en arrière non chronologique et à une forme d'apprentissage lors des conflits, il n'utilise toujours pas de technique performante concernant le BCP.

Des expérimentations préliminaires à cette thèse sur le solveur SAT d'Alt-Ergo, nous ont permis de capitaliser sur les conclusions existantes pour guider nos premiers travaux. Ces travaux ont consisté en l'implémentation d'un solveur SAT CDCL [69] basé sur le solveur de référence Minisat [40] au sein d'Alt-Ergo. Cette implémentation a montré de moins bons résultats que le solveur historique d'Alt-Ergo par méthode des tableaux sur des exemples issus de la preuve de programme.

La première question que nous nous sommes posée pour expliquer un tel manque de performance d'un solveur SAT performant vis-à-vis du solveur historique d'Alt-Ergo concernait les performances du langage OCaml. À quel point les performances d'une même implémentation d'un solveur de référence peuvent-elles différer en fonction du langage utilisé ? C'est la question à laquelle nous allons répondre dans ce chapitre. Nous présenterons la méthodologie que nous avons utilisée pour étudier et comparer les performances des implémentations dans deux langages différents dans la section 1. Nous présenterons ensuite les résultats expérimentaux de nos expériences dans la section 2. Nous présenterons nos conclusions dans la section 3.

1 Méthodologie de comparaison

Notre but a été de comparer deux implémentations du solveur Minisat dans deux langages différents, le langage C++ et le langage OCaml. Le but étant de comparer les performances des deux langages, nous utiliserons les structures de données fournies par OCaml et gérées par son GC pour implé-

-15 1	le négation de la variable 15 est assignée à vraie
66 0	la variable 66 est assignée à faux
2 1	...
43 1	
-17 1	

FIGURE 3.1 – Exemple de trace de décision, la première colonne représente le littéral décidé et la deuxième sa valeur de vérité.

menter la version en OCaml. Les deux implémentations sont identiques en termes d'algorithme, et ne diffèrent que par le langage utilisé. Nous présenterons les différents points de comparaison ainsi que les paramètres et outils utilisés.

1.1 Trace de décision

Le premier point de comparaison fut celui des résultats des deux implémentations. Tel qu'énoncé auparavant, nous avons utilisé le même algorithme pour les deux implémentations afin d'avoir la comparaison la plus viable possible. Minisat étant principalement implémenté avec des structures bas niveau tel que des tableaux, il nous a été possible d'utiliser les mêmes structures de données offertes par OCaml pour implémenter la version en OCaml sans devoir apporter de modification à l'algorithme original. Pour nous assurer que ces deux implémentations fournissaient les mêmes résultats, nous avons utilisé des traces de résolution. Ces traces enregistrent les variables assignées par décision lors de l'exécution. Elles se présentent sous forme de fichier : la figure 3.1 est un exemple de fichier représentant une résolution où il y a eu cinq décisions. Le 1 représente la valeur de vérité vrai et 0 représente faux. Ces traces nous permettent de garantir que les deux algorithmes se comportent et produisent les mêmes résultats par comparaison de leurs fichiers de traces de décisions.

1.2 Consommation spatiale et temporelle

Le second point de comparaison est la différence de consommation spatiale et temporelle des deux implémentations. Nous présenterons les outils que nous allons utiliser pour analyser ces consommations avant de rappeler l'importance d'une bonne gestion du cache processeur.

Consommation temporelle

Pour comparer nos deux implémentations en termes de consommation temporelle, nous allons utiliser la commande `time` qui nous permettra d'obtenir le temps mis par les différentes implémentations pour résoudre des

problèmes donnés. Cette commande time permet de donner un temps précis sans alourdir l'exécution de la commande à mesurer.

Consommation mémoire

Pour la consommation spatiale, nous allons analyser les poids en RAM de nos implémentations. Un plus gros poids mémoire peut être synonyme de moins bonnes performances. Pour mesurer ces poids nous utiliserons l'outil Massif de Valgrind [1], il nous permettra de connaître la taille en mémoire au fur et à mesure de la résolution. Le graphique 3.2 nous présente une telle consommation avec l'outil Massif-visualizer. Cet outil nous permet d'analyser de manière graphique la consommation mémoire enregistrée par Massif. Le graphique renvoyé correspond à la consommation en mémoire par rapport au temps mesuré ici en intervalle entre chaque enregistrement des données par Massif. Une légende nous permet de différencier les courbes correspondant aux parties allouant le plus de mémoire au cours de l'exécution. Le graphique nous permet ici aussi de connaître l'instant où le maximum de mémoire a été consommé, 45,6 Mb.

Cache mémoire

Le cache mémoire est un espace de stockage de données tampon situé entre les registres du processeur et la RAM. Il permet de stocker des données récemment chargées depuis la RAM pour accélérer leur réutilisation. La figure 3.3 [2] représente les différentes mémoires utilisées par un programme. Le haut de la pyramide représente les mémoires les plus rapides, les registres (Reg) et les caches mémoires. Les registres sont extrêmement rapides (de l'ordre du cycle machine) mais sont présents en nombre très réduit. Sur les architectures modernes il y a généralement trois niveaux de cache, le plus petit (L1) ne contient que 32 Ko de données et 32 Ko de code. C'est le seul stockage qui sépare code et données. Le deuxième niveau contient 256 Ko et le troisième peut contenir quant à lui un espace de l'ordre du méga-octet. Les temps d'accès de ces trois niveaux sont proches de 4 cycles pour le L1, 12 cycles pour le L2 et 40 cycles pour le L3 [50]. Les niveaux inférieurs de la pyramide présentent des mémoires de plus en plus volumineuses et de plus en plus lentes. On considère que le temps d'accès en RAM est de l'ordre d'une centaine de cycles.

Les chargements dans les caches s'effectuent par ligne (intervalles) de 64 octets (sur des architecture récentes) [50]. Dans le cas d'un chargement de la donnée à l'indice n d'un tableau d'entiers il y a de fortes chances que les cases adjacentes soient elles aussi présentes dans la ligne de cache. Le chargement d'une donnée en mémoire peut ainsi permettre de charger

1. http://valgrind.org/
2. https://en.wikipedia.org/wiki/File:ComputerMemoryHierarchy.png

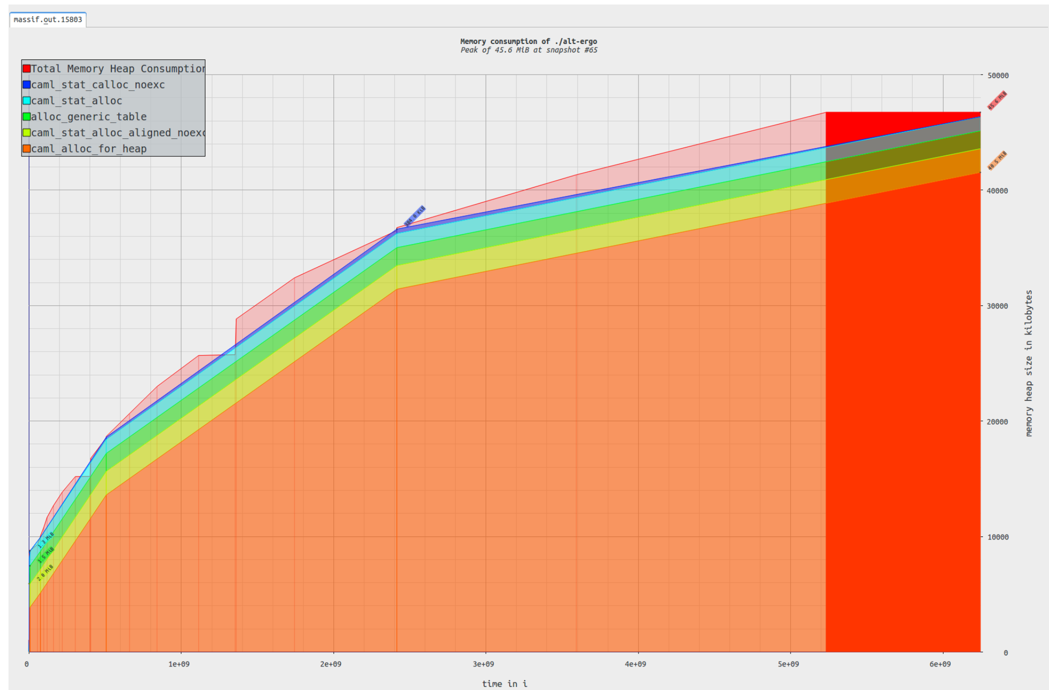

FIGURE 3.2 – Exemple de graphe de consommation mémoire en fonction du temps calculé par Massif-visualizer à partir de données récupérées par l'outil Massif de valgrind.

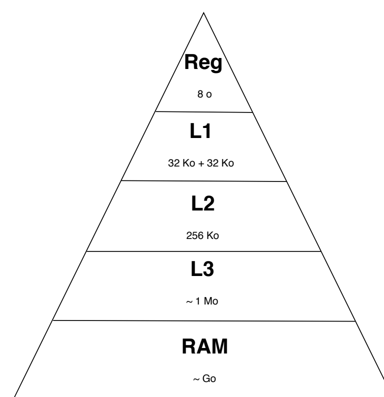

FIGURE 3.3 – représentation des mémoires d'un ordinateur de la plus rapide et petite à la plus volumineuse et lente.

```
not supported    L1-icache-loads
    4 023 467    L1-icache-load-misses                                          (+- 4,81%)
1 723 394 758    L1-dcache-loads                                                (+- 0,96%)
  244 615 813    L1-dcache-load-misses # 14,12% of all L1-dcache hits           (+- 1,10%)
  477 412 413    L1-dcache-stores                                               (+- 1,01%)
not supported    L1-dcache-store-misses
  111 531 044    LLC-loads                                                      (+- 1,01%)
   13 329 005    LLC-load-misses       # 23,41% of all LL-cache hits            (+- 2,13%)
    3 349 257    LLC-stores                                                     (+- 0,85%)
      566 249    LLC-store-misses                                               (+- 2,91%)

  2874,683263    task-clock (msec)     # 0,993 CPUs utilized                    (+- 2,20%)
5 218 241 950    cycles                # 1,815 GHz                              (+- 1,71%)
4 910 218 448    instructions          # 0,94  insns per cycle                  (+- 0,17%)
1 124 036 857    branches              # 391,012 M/sec                          (+- 0,15%)
   42 811 732    branch-misses         # 3,81% of all branches                  (+- 0,23%)

  2,895890092    seconds time elapsed                                           (+- 2,20%)
```

Figure 3.4 – Format de résultats de la commande perf stat -dr 10, les deux premières colonnes représentent le nombre et le type d'événements calculés par perf au cours de l'exécution, la troisième colonne donne des informations complémentaires, et la dernière nous montre la variation des résultats sur les 10 exécutions.

des données qui seront ultérieurement utiles (prefetch). Dans un souci de performances, il est donc intéressant que des données, qui seront utiles dans un laps de temps rapproché, soient proches au sein de la mémoire pour être chargées en même temps (localité).

Pour l'analyse des performances des caches mémoire nous utiliserons des outils comme Perf et l'outil Cachegrind de Valgrind[3]. Perf permet d'enregistrer les actions du processeur durant l'exécution d'un programme et d'obtenir un rapport de la forme présentée en figure 3.4. Ces résultats étant dépendants de la charge du CPU au moment de l'exécution, ils peuvent varier d'une exécution à l'autre (le cache de niveau L3 est partagé entre les applications). Nous utiliserons des moyennes sur plusieurs exécutions pour amortir les erreurs statistiques. Sur la figure 3.4 nous pouvons remarquer une liste d'événements ayant eu lieu au cours de l'exécution d'un programme. De haut en bas nous avons tout d'abord les lectures et les erreurs de lecture depuis le cache d'instruction, puis, depuis le cache de données. Pour le cache de données viennent s'ajouter les statistiques concernant l'écriture dans le cache. Ensuite nous avons ces mêmes statistiques sur le cache de plus haut niveau (LLC), ici un cache L3. On appelle erreur de cache tout chargement d'une donnée depuis un cache qui échoue car elle n'y est pas présente. Cela entraîne un chargement depuis le niveau de cache supérieur et ainsi de suite. Viennent ensuite les statistiques concernant le temps d'exécution, le nombre de cycles, le nombre d'instructions, les branchements et les erreurs de branchement sur les conditions.

Valgrind contrairement à Perf simule l'exécution du programme à analyser, permettant d'obtenir un résultat théorique précis au détriment du temps

3. http://valgrind.org/

```
I    refs:        6,246,067,320
I1   misses:         27,569,391
LLi  misses:            265,139
I1   miss rate:           0.44%
LLi  miss rate:           0.00%

D    refs:        2,686,395,060  (1,691,189,244 rd   + 995,205,816 wr)
D1   misses:         80,249,884  (   51,317,314 rd   +  28,932,570 wr)
LLd  misses:          7,696,199  (    5,907,814 rd   +   1,788,385 wr)
D1   miss rate:            3.0%  (          3.0%     +         2.9%  )
LLd  miss rate:            0.3%  (          0.3%     +         0.2%  )

LL refs:            107,819,275  (   78,886,705 rd   +  28,932,570 wr)
LL misses:            7,961,338  (    6,172,953 rd   +   1,788,385 wr)
LL miss rate:              0.1%  (          0.1%     +         0.2%  )
```

FIGURE 3.5 – Format de résultats de la commande valgrind –tool=cachegrind, les deux premiers blocs concernent les caches d'instructions et de données de la L1, le troisième concerne le dernier niveau de cache.

d'exécution (50 fois plus lent). À la fin de son exécution l'outil Cachegrind de Valgrind retourne à la manière de Perf un rapport concernant l'utilisation des caches mémoire (voir figure 3.5). On y trouve les informations sur les chargements du cache d'instructions, puis celui de donnée au niveau L1 et enfin ceux sur le niveau de cache maximum, LL. `rd` et `wr` représentent respectivement les accès en lecture et écriture aux caches. Il enregistre aussi ce rapport sous forme d'un fichier qui peut être relu de manière graphique grâce au programme KCachgrind, comme sur la figure 3.6. On peut ainsi connaître l'utilisation des caches par fonctions, comme sur la figure 3.6, où les statistiques sont triées par fichier d'origine puis par fonction contenue dans ces fichiers. Le fichier et la fonction dont les erreurs de cache du dernier niveau sont analysées sont major_gc.c et mark_slice.

1.3 Paramètres

Compilateurs

Nous avons aussi décidé de paramétrer nos comparaisons grâce à différentes options de compilation offertes par les compilateurs des langages. Nous utiliserons pour cela le compilateur GCC pour le langage C++ et le compilateur de la distribution standard pour OCaml. Ces deux compilateurs proposent différents niveaux d'optimisation pouvant faire varier grandement les résultats. Bien qu'il ne fut pas développé pour obtenir les meilleures performances possibles mais plutôt la portabilité (système d'exploitation et architecture), GCC a au fil des années gagné en performance pour offrir aujourd'hui un code optimisé offrant d'excellentes performances. Les efforts d'optimisation du compilateur d'OCaml sont plus récents et moins avancés (projet Flambda) et il reste encore beaucoup de travail d'optimisation.

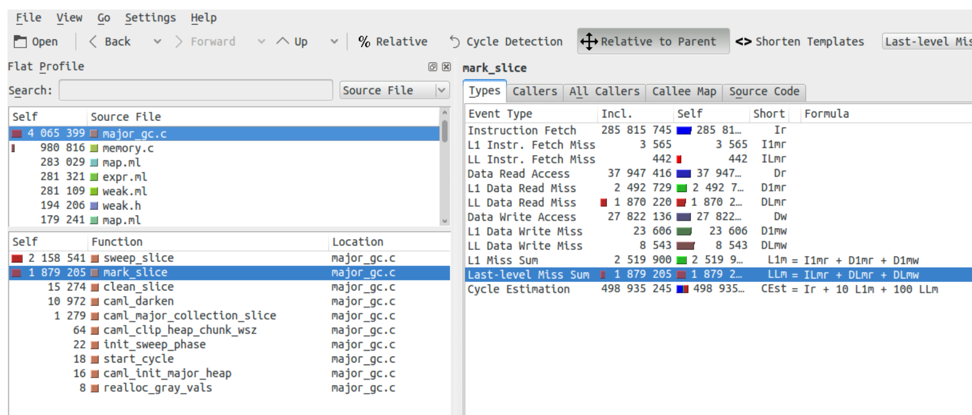

Figure 3.6 – Exemple de visualisation des données d'exécution grâce à l'outil Kcachgrind simulé par l'outil cachgrind de valgrind. Les données sont triées selon le nombre d'erreurs de cache de dernier niveau commises, le fichier major_gc.c en contient le plus, principalement présentes dans les fonctions sweep_slice et mark_slice.

Taille des données

Un autre paramètre de notre comparaison est la taille des données qui diffère entre les deux implémentations. En effet, en C++ les données atomiques utilisées par Minisat sont principalement des entiers sur 32 bits ou des booléens sur 8 bits. En OCaml tous les types de données de base font la taille d'un mot mémoire, soit 64 bits sur une architecture 64 bits. Entre nos deux implémentations, le poids en mémoire d'un entier peut donc doubler et celui d'un booléen être multiplié par huit.

Il faut rajouter à cela un en-tête de la taille d'un mot mémoire pour chaque bloc en OCaml nécessaire au fonctionnement du GC. Cela accentue encore la différence de consommation mémoire entre nos deux implémentations, pouvant influer sur les performances des caches mémoire.

2 Résultats expérimentaux

Dans cette section nous allons traiter des différentes expérimentations que nous avons réalisées. Nous avons décidé de tester l'implémentation de Minisat en C++ et en OCaml sur un ensemble de problèmes issus de la SAT-LIB [4]. Cela nous permet de travailler sur des problèmes connus de la communauté (permettant la transparence et facilitant les comparaisons). Nous avons lancé nos deux implémentations avec trois niveaux d'optimisation du compilateur chacune sur cet ensemble de problèmes avec une limite de temps

4. `https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

d'une minute. Cette limite de temps est suffisante pour résoudre un grand nombre des problèmes étudiés.

Le compilateur standard de OCaml et GCC proposent une option d'agrégation d'option d'optimisation, -On où n peut varier entre 0 et 3. Plus n est élevé plus le nombre d'optimisations sont élevées. Le niveau 0 signifie qu'aucune optimisation ne sera effectué, c'est le niveau par défaut. Le niveau 3, quant à lui, active toutes les options des niveaux 1 et 2 plus certaines qui lui sont propres.

Les notations des niveaux d'optimisation que nous utiliserons sont les suivantes : pour Minisat le niveau 1 correspond à la compilation sans optimisation, soit -O0. Le niveau 2 correspond à l'option -O1 et le niveau 3 correspond à l'option -O3. Nous avons décidé de ne pas afficher les résultats de l'option -O2 proche des résultats de l'option -O1.

Concernant les niveaux d'optimisation de l'implémentation en OCaml, le niveau 1 correspond à une compilation avec un compilateur non optimisant. Le niveau 2 correspond à une compilation avec le compilateur optimisant (Flambda), et le niveau 3 ajoute l'option -O3 offerte par le compilateur optimisant.

Dans la suite l'implémentation en C++ sera appelée Minisat tandis que celle en OCaml sera appelée SatML. Sur les figures 3.7 et 3.8 on peut voir le temps de résolution des deux solveurs et leurs trois niveaux d'optimisation et le nombre de buts résolus. Ces résultats ont été obtenus via la plate-forme Starexec[5]. Cette plate-forme disponible à la communauté SMT, permet de tester rapidement via un nombre important de processeurs (192) et de manière reproduisible (matériel inchangé depuis 2013) nos tests. Pour la suite de ce documents, tous nos tests sur des ensemble des bancs de test seront effectués sur cette plate-forme avec une limite de temps de 60 secondes et une limite de consommation de mémoire de 2 Go.

Les résultats présentés sur la figure 3.7 correspondent au nombre de buts prouvé (et au temps d'exécution) par catégorie en fonction du solveur. Les résultats présentés sur la figure 3.8 correspondent au nombre de buts totaux des bancs de tests prouvés par rapport au temps d'exécution. Chaque courbe correspond à un solveur différent.

Ces résultats nous montrent que les implémentations en C++ optimisées (Minisat_2 et Minisat_3) sont meilleures en temps de résolution et en nombre de buts prouvés que celles en OCaml. Nous remarquons que SatML montre de meilleures performances que Minisat non optimisé. Nous expliquons cela par le fait que la compilation non optimisée en C++ est très naïve, en effet en examinant le code assembleur généré nous avons remarqué que les constantes sont stockées sur la pile et non dans des registres. Cette optimisation est effectuée lorsque l'option -O1 est activée, ce qui explique en partie les différences de résultats entre ces deux versions. Au-delà des

5. https://www.starexec.org/starexec/public/about.jsp

	sans Flambda	avec Flambda	Flambda -o3	gcc -o0	gcc -01	gcc -o3
	`satML_1`	`satML_2`	`satML_3`	`minisat_1`	`minisat_2`	`minisat_3`
aim	70 (0s)	70 (0s)	**72 (0s)**	**72 (0s)**	**72 (0s)**	**72 (0s)**
bf	**4 (0s)**	**4 (0s)**	**4 (0s)**	**4 (0s)**	**4 (0s)**	**4 (0s)**
dubois	**13 (0s)**	**13 (0s)**	**13 (0s)**	**13 (0s)**	**13 (0s)**	**13 (0s)**
hanoi	1 (0s)	1 (0s)	1 (0s)	1 (0s)	**2 (46s)**	2 (48s)
jnh	48 (0s)	48 (0s)	**50 (0s)**	**50 (0s)**	**50 (0s)**	**50 (0s)**
minisat	29 (23s)	29 (23s)	29 (20s)	29 (41s)	31 (65s)	**31 (62s)**
pigeon	4 (17s)	4 (16s)	4 (14s)	4 (41s)	5 (59s)	**5 (48s)**
pret	**8 (0s)**	**8 (0s)**	**8 (0s)**	**8 (0s)**	**8 (0s)**	**8 (0s)**
ssa	**8 (0s)**	**8 (0s)**	**8 (0s)**	**8 (0s)**	**8 (0s)**	**8 (0s)**
uuf100	**1000 (0s)**	**1000 (0s)**	**1000 (0s)**	**1000 (0s)**	**1000 (0s)**	**1000 (0s)**
uuf125	**100 (0s)**	**100 (0s)**	**100 (0s)**	**100 (0s)**	**100 (0s)**	**100 (0s)**
uuf150	**100 (0s)**	**100 (0s)**	**100 (0s)**	100 (1s)	**100 (0s)**	**100 (0s)**
uuf175	100 (29s)	100 (28s)	100 (22s)	100 (81s)	**100 (0s)**	**100 (0s)**
uuf250	34 (1149s)	35 (1176s)	39 (1251s)	22 (821s)	91 (1841s)	**91 (1802s)**
total	1519 (1221s)	1520 (1244s)	1528 (1309s)	1511 (990s)	1584 (2013s)	**1584 (1962s)**

Figure 3.7 – Nombre de buts résolus et temps d'exécution de SatML et Minisat sur des fichiers des la SATLIB avec différents niveaux d'optimisations et une limite de temps de 60 secondes. En gras sont présentés les meilleurs résultats pour chaque catégorie. Le solveur résolvant le plus de buts est considéré le plus performant. Si deux solveurs résolvent le même nombre de but, celui qui à répondu le plus rapidement est alors considéré le plus performant.

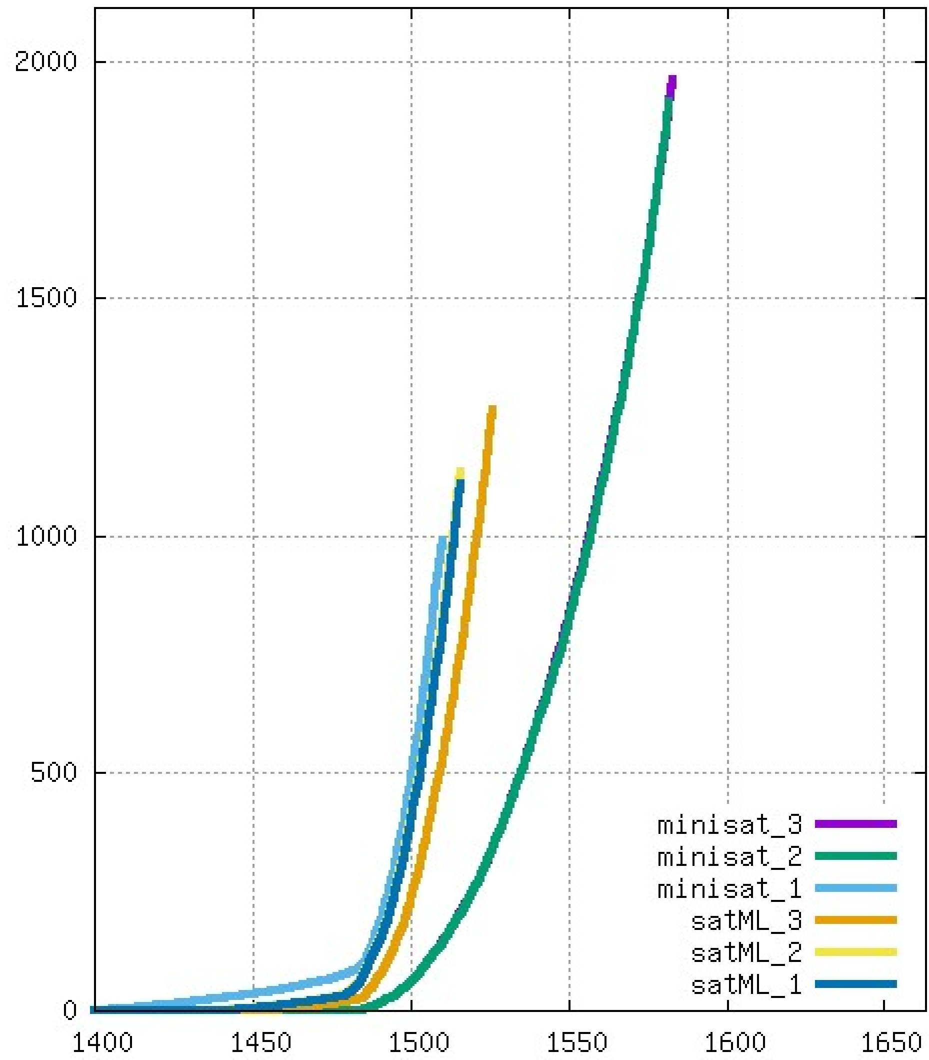

Figure 3.8 – Graphique du temps de résolution (en secondes) des solveurs et de leurs niveaux d'optimisations en fonction du nombre de buts résolus.

meilleures performances de la version en OCaml le résultat intéressant est surtout les gains obtenus grâce aux optimisations du compilateur. En effet lorsque l'on compare les versions optimisées (= 3) sur le banc de test uuf175 là où $SatML_3$ prend 23 s pour résoudre les 100 buts, $Minisat_3$ les résout instantanément. Nous pouvons aussi remarquer cette différence sur la figure 3.9. Sur le premier graphique, SatML_3 est plus performant sur tous les exemples que la version C++ non optimisée. Ces résultats s'inversent lorsque nous comparons SatML_3 à la version optimisée de Minisat où là, la version OCaml est bien moins performante.

Pour garantir la fiabilité de nos comparaisons comme expliqué en section 1.1, nous avons utilisé des traces de décision pour chacun des problèmes. Pour ce faire nous avons comparé ces traces deux à deux pour chaque problème de notre ensemble de problèmes. Une simple commande diff nous a permis de voir que les traces étaient identiques en tous points. Les écarts de performances montrés sur la figure 3.7 ne sont donc pas dus à une différence dans l'algorithme, mais bien dans les différences des langages d'implémentation et des options de compilation utilisées.

Dans la suite nous présenterons nos analyses sur la comparaison des performances des langages OCaml et C++, en utilisant les points de comparaison et les outils présentés en section 1.

2.1 Impact du langage OCaml sur les performances

Le tableau de la figure 3.7 précédemment présenté nous a montré que l'implémentation SatML était plus lente que l'implémentation Minisat. Nous allons présenter des résultats expérimentaux permettant de mettre en évidence l'importance et l'impact de la consommation spatiale sur la vitesse de résolution.

Tout d'abord les figures 3.10 et 3.11 nous présentent deux graphiques de consommation de la mémoire grâce à l'outil Massif-visualizer. Ces graphiques correspondent à l'évolution de la consommation de la mémoire des implémentations optimisées au cours du temps de résolution. Minisat consomme au maximum 6 Mo de mémoire tandis que SatML en consomme 19.4 Mo. Il semblerait que Massif soit un outil peu adapté au langage OCaml. En effet pour un exemple en C++ où il est capable de donner avec précision les points d'allocation de mémoire, les fonctions search et propagate du fichier Solver.c consomment le plus de mémoire. Pour OCaml il ne trace que les allocations très bas niveau c'est-à-dire au niveau du ramasseur de miettes. Les résultats pour OCaml correspondent non pas à la mémoire consommée mais à la mémoire allouée pour le ramasse-miettes qui peut être légèrement supérieure à la consommation réelle. L'outil ocp-memprof [18] [6] permet de tracer avec plus de précision la consommation mémoire d'un programme en

6. https://memprof.typerex.org/

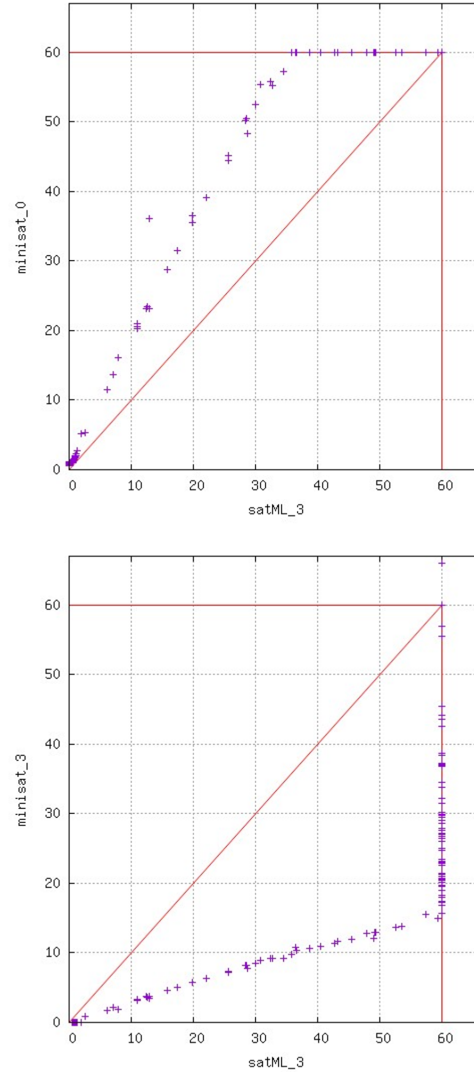

FIGURE 3.9 – Comparaison des temps de résolution entre SatML et deux niveaux d'optimisation de Minisat, les axes représentent le temps mis par le solveur correspondant pour résoudre le but. Les points sur les lignes rouges verticales et horizontales représentent des timeouts.

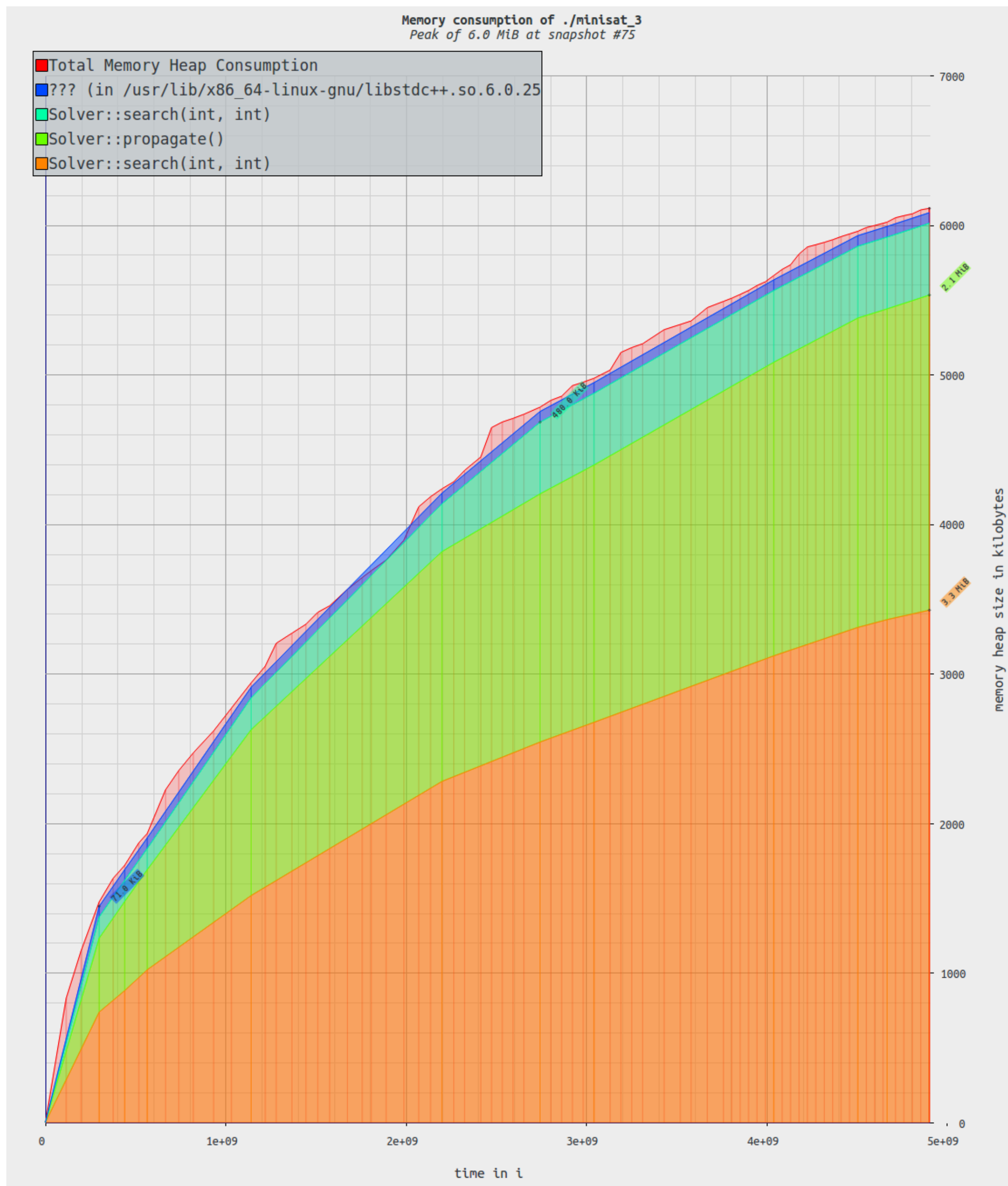

Figure 3.10 – Graphe issu de Massif-visualizer montrant la consommation de mémoire au cours du temps de l'exécution de Minisat optimisé avec l'option `-o3`.

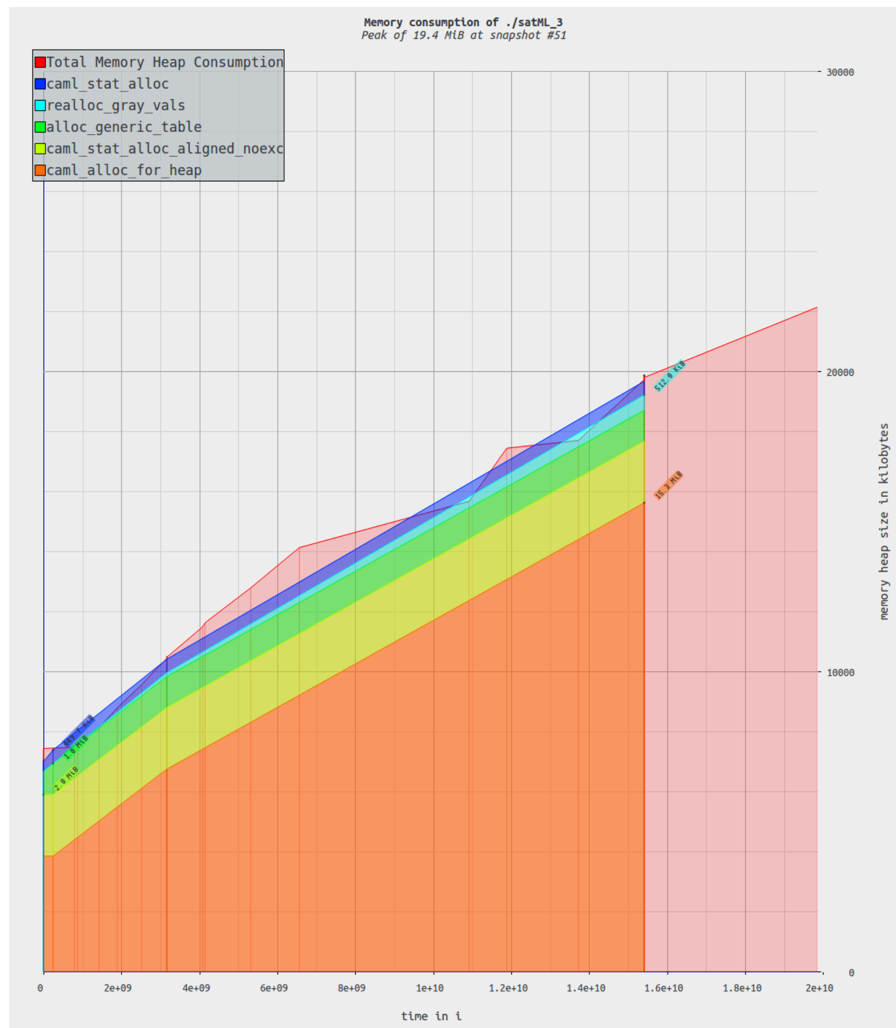

FIGURE 3.11 – vue de Massif-visualizer avec l'implémentation SatML_3.

```
===============================[ Problem Statistics ]===============================
|                                                                                  |
|  Number of variables:   44                                                       |
|  Number of clauses:     398                                                      |
===============================[ Search Statistics ]================================
| Conflicts |          ORIGINAL          |          LEARNT          | Progress |
|           |    Vars  Clauses  Literals |    Limit  Clauses Lit/Cl |          |
====================================================================================
|         0 |      44      398      2360 |       13        0   -nan |  0.000 % |
|       100 |      44      398      2360 |       14      100     24 |  0.053 % |
|       250 |      44      398      2360 |       16      250     20 |  0.053 % |
|       476 |      44      398      2360 |       17      476     18 |  0.053 % |
|       814 |      44      398      2360 |       19      814     17 |  0.053 % |
|      1322 |      44      398      2360 |       21     1322     15 |  0.053 % |
|      2081 |      44      398      2360 |       23     2081     14 |  0.053 % |
|      3220 |      44      398      2360 |       25     3220     14 |  0.053 % |
|      4928 |      44      398      2360 |       28     4928     13 |  0.053 % |
|      7491 |      44      398      2360 |       31     7491     13 |  0.053 % |
|     11335 |      44      398      2360 |       34    11335     13 |  0.053 % |
|     17102 |      44      398      2360 |       37    17102     13 |  0.053 % |
|     25752 |      44      398      2360 |       41    25752     13 |  0.105 % |
|     38727 |      43      398      2360 |       45    38726     13 |  2.326 % |
|     58188 |      40      398      2360 |       50    58184     13 |  9.302 % |
====================================================================================
restarts              : 15
conflicts             : 59663          (28963 /sec)
decisions             : 82504          (40050 /sec)
propagations          : 267213         (129715 /sec)
conflict literals     : 0              (-nan % deleted)
CPU time              : 2.06 s
unsat
```

Figure 3.12 – trace affiché au cours de l'exécution de Minisat montrant l'évolution de la résolution et affichant les statistiques à la fin de la résolution.

OCaml. Nous n'avons cependant pas pu utilisé cet outils du fait qu'il ne soit plus disponible ni mis à jour sur les versions récentes d'OCaml.

Minisat_3 prend 2 secondes pour résoudre l'exemple observé tandis que SatML_3 prend 14 secondes. Pour comprendre l'augmentation de la consommation mémoire nous analysons les statistiques que les implémentations affichent au cours de l'exécution. Les deux figures 3.12 et 3.13 nous montrent que mis à part le temps de résolution, les statistiques sont identiques. Les nombres de conflits de décision et de propagations sont identiques mais leurs nombres par seconde diffèrent grandement. On observe aussi un nombre important de clauses apprises (Learnt) au cours de la résolution, cela explique la montée en charge de la mémoire observée sur les graphiques issus de Massif. Ces clauses sont les seules données que le solveur ajoute au cours de l'exécution. Le solveur ne désalloue jamais de données du au fait que nous avons décidé de désactiver la suppression des clauses apprises.

Nous remarquons aussi que l'implémentation en OCaml consomme trois fois plus de mémoire que celle en C++. Cette différence de consommation est principalement due à deux facteurs. Le premier est la taille des données, en effet comme expliqué dans la section 1.3 les données en OCaml prennent plus de place en mémoire que des données en C++. Le second est la présence en OCaml d'un ramasse-miettes [38, 37]. Ce système de gestion automatique de la mémoire a besoin de méta-données pour fonctionner. En OCaml sur chaque bloc de données, tableau, ensemble, flottant, etc., qui ne sont ni

```
============================[ Problem Statistics ]============================
|                                                                            |
|  Number of variables:  44                                                  |
|  Number of clauses:    398                                                 |
============================[ Search Statistics ]=============================
| Conflicts |          ORIGINAL          |          LEARNT          | Progress |
|           |    Vars  Clauses Literals  |    Limit  Clauses Lit/Cl |          |
==============================================================================
|         0 |      44      398     2360  |       13        0       0 |  0.000 % |
|       100 |      44      398     2360  |       14      100      23 |  0.053 % |
|       250 |      44      398     2360  |       16      250      20 |  0.053 % |
|       476 |      44      398     2360  |       17      476      17 |  0.053 % |
|       814 |      44      398     2360  |       19      814      16 |  0.053 % |
|      1322 |      44      398     2360  |       21     1322      14 |  0.053 % |
|      2081 |      44      398     2360  |       23     2081      14 |  0.053 % |
|      3220 |      44      398     2360  |       25     3220      13 |  0.053 % |
|      4928 |      44      398     2360  |       28     4928      13 |  0.053 % |
|      7491 |      44      398     2360  |       31     7491      13 |  0.053 % |
|     11335 |      44      398     2360  |       34    11335      12 |  0.053 % |
|     17102 |      44      398     2360  |       37    17102      13 |  0.053 % |
|     25752 |      44      398     2360  |       41    25752      13 |  0.105 % |
|     38727 |      43      398     2360  |       45    38726      13 |  2.326 % |
|     58188 |      40      398     2360  |       50    58184      12 |  9.302 % |
==============================================================================
restarts              : 15
conflicts             : 59663          (4172 /sec)
decisions             : 82504          (5770 /sec)
propagations          : 267213         (18686 /sec)
conflict literals     : 0              (-nan /sec)
CPU time              : 14.299827 s
unsat
```

Figure 3.13 – trace affiché au cours de l'exécution de SatML montrant l'évolution de la résolution et affichant les statistiques à la fin de la résolution.

des entiers ni des pointeurs il y a un en-tête. Cet en-tête est invisible pour l'utilisateur mais est présent en mémoire, il contient des informations comme la taille du bloc.

Pour comprendre ces deux points et leur incidence, prenons l'exemple des clauses et de leur implémentation. Les figures 3.14 et 3.15 représentent l'implémentation d'une clause en C++ et en OCaml. En C++ une clause prendra 4 octets pour l'entier size_etc, 4 octets pour l'union extra et 4 octets par littéral contenu dans le tableau de littéraux. Le tableau de littéraux est directement alloué dans la définition de la clause, évitant ainsi une indirection d'un pointeur vers le tableau.

En OCaml une clause aura un en-tête de 8 octets puis chacun de ses 4 autres éléments prendra 64 bits. Il faut rajouter à cela les 8 octets d'en-tête du tableau qui sera alloué en dehors de la clause, seul un pointeur vers ce tableaux sera dans la clause. Le type union n'existant pas en OCaml, nous avons utilisé le type extra. De plus dans l'implémentation C++ le booléen learnt est définit dans le bit de poids faible de la taille. Nous avons dans un premier temps choisi d'utiliser un champ supplémentaire. Par la suite dans la section 2.2 nous essayerons de nous rapprocher de l'implémentation en C++.

Prenons une clause contenant 10 littéraux : nous avons donc une clause qui pèse 48 octets en C++ alors que pour l'implémentation en OCaml le tableau pèse 88 octets, la valeur pointée par le champ extra pèse 16 octets

```
class Clause {
  uint32_t size_etc;
  union { float act; uint32_t abst; } extra;
  Lit     data[0];
...
}
```

Figure 3.14 – implémentation d'une clause en C++.

```
module Clause = struct
  type extra =
    | Abst of int
    | Act of float

  type t = {
    mutable size : int;
    mutable extra : extra;
    learnt : bool;
    data : Lit.Array.t;
  }
...
end
```

Figure 3.15 – implémentation d'une clause en OCaml.

```
Performance counter stats for './minisat_3 examples/minisat/19-sec.cnf' (10 runs)

not supported   L1-icache-loads
    4 023 467   L1-icache-load-misses                                          (+- 4,81%)
1 723 394 758   L1-dcache-loads                                                (+- 0,96%)
  244 615 813   L1-dcache-load-misses # 14,12% of all L1-dcache hits           (+- 1,10%)
  477 412 413   L1-dcache-stores                                               (+- 1,01%)
not supported   L1-dcache-store-misses
  111 531 044   LLC-loads                                                      (+- 1,01%)
   13 329 005   LLC-load-misses       # 23,41% of all LL-cache hits            (+- 2,13%)
    3 349 257   LLC-stores                                                     (+- 0,85%)
      566 249   LLC-store-misses                                               (+- 2,91%)

  2874,683263   task-clock (msec)     # 0,993 CPUs utilized                    (+- 2,20%)
5 218 241 950   cycles                # 1,815 GHz                              (+- 1,71%)
4 910 218 448   instructions          # 0,94  insns per cycle                  (+- 0,17%)
1 124 036 857   branches              # 391,012 M/sec                          (+- 0,15%)
   42 811 732   branch-misses         # 3,81% of all branches                  (+- 0,23%)

  2,895890092   seconds time elapsed                                           (+- 2,20%)
```

FIGURE 3.16 – résultats de la commande perf -dr 10 minisat_3 sur l'exemple 19-sec.cnf.

et la clause pèse 40 octets soit un total de 144 octets. On comprend ainsi comment l'implémentation SatML peut consommer près de 3 fois plus de mémoire que Minisat.

Cet exemple nous permet aussi de mettre en évidence les différences dans les indirections. En plus de l'indirection causée par le tableau de littéraux, le champ extra en créera une lui aussi. `extra` prendra 4 octets en C++, la construction `union` prendra en mémoire le plus grand des arguments donc 4 octets. Or en OCaml nous devons utiliser un type énuméré pour créer le champ extra, ce type est ici accessible via un pointeur vers un bloc (avec en-tête) contenant l'information.

Non seulement cela alourdit l'empreinte mémoire mais aussi crée une indirection supplémentaire pouvant dégrader la localité des données en mémoire. En effet ces indirections sont d'autant plus gênantes que le GC d'OCaml peut déplacer des blocs de données. Il est donc possible que la donnée contenue dans `extra` ne soit pas contenue dans la ligne de cache lors du chargement en mémoire de la clause, cela entraîne un défaut de cache et requiert un chargement supplémentaire, augmentant le nombre de cycles nécessaires pour charger les données nécessitant une indirection depuis une clause.

Les figures 3.16 et 3.17 nous montrent les statistiques concernant l'utilisation des caches mémoires pour nos deux implémentations. SatML effectue bien plus d'instructions, de branchements et de chargements de données que son homologue en C++. Si l'on prend les erreurs de chargement sur la couche L3 (ici LLC), là ou Minisat effectue 13 millions d'erreurs, SatML en effectue plus de 111 millions, ces erreurs entraînant des chargements depuis la RAM et donc un nombre de cycles bien supérieur. Ces chiffres sont semblables à ceux que nous obtenons grâce à Cachegrind (voir figure 3.18 et 3.19). Nous expliquons ces différences en grande partie par la taille des données et les en-tête de chaque bloc de données. En effet le système sur lequel ont été

```
Performance counter stats for './satML_3 examples/minisat/19-sec.cnf' (10 runs)

not supported   L1-icache-loads
   13 736 460   L1-icache-load-misses                                        (+- 9,52%)
8 060 592 846   L1-dcache-loads                                              (+- 0,36%)
  577 564 330   L1-dcache-load-misses # 7,15% of all L1-dcache hits          (+- 0,30%)
2 470 561 222   L1-dcache-stores                                             (+- 0,28%)
not supported   L1-dcache-store-misses
  310 625 318   LLC-loads                                                    (+- 0,45%)
  111 507 890   LLC-load-misses       # 71,78% of all LL-cache hits          (+- 7,41%)
    2 228 750   LLC-stores                                                   (+- 3,70%)
      904 652   LLC-store-misses                                             (+- 3,73%)

 16828,190432   task-clock (msec)     # 0,997 CPUs utilized                  (+- 6,33%)
33 015 619 906  cycles                # 1,801 GHz                            (+- 6,26%)
27 937 774 197  instructions          # 0,85 insns per cycle                 (+- 0,06%)
 7 074 404 400  branches              # 385,985 M/sec                        (+- 0,09%)
    58 151 332  branch-misses         # 0,82% of all branches                (+- 0,49%)

 16,949259713   seconds time elapsed                                         (+- 5,22%)
```

FIGURE 3.17 – résultats de la commande perf -dr 10 SatML_3 sur l'exemple 19-sec.cnf.

```
I    refs:       4,913,282,637
I1   misses:             2,288
LLi  misses:             2,141
I1   miss rate:          0.00%
LLi  miss rate:          0.00%

D    refs:       2,230,801,614  (1,747,224,808 rd   + 483,576,806 wr)
D1   misses:       165,277,976  (  161,472,131 rd   +   3,805,845 wr)
LLd  misses:           301,479  (      171,741 rd   +     129,738 wr)
D1   miss rate:            7.4% (          9.2%     +         0.8%  )
LLd  miss rate:            0.0% (          0.0%     +         0.0%  )

LL refs:           165,280,264  (  161,474,419 rd   +   3,805,845 wr)
LL misses:             303,620  (      173,882 rd   +     129,738 wr)
LL miss rate:              0.0% (          0.0%     +         0.0%  )
```

FIGURE 3.18 – résultats obtenues avec l'outil cachegrind pour l'exécution de Minisat_3 sur l'exemple 19-sec.cnf.

réalisées ces expérimentations possède 4 Mo de cache de niveau L3. On sait donc grâce à Massif que notre cache peut contenir une grande partie des données de l'exécution en C++. Pour SatML par contre, le cache ne peut contenir qu'un fragment des 19 Mo de données (20%). Cela peut expliquer le si grand nombre d'erreurs pour ce niveau de cache.

Grâce à l'outil de visualisation des données calculées par Cachegrind, nous avons remarqué un nombre de défaut de cache important dû au déclenchement du ramasse-miettes (10%). La figure 3.20 nous montre le détail des erreurs de cache au niveau L1 et au niveau LL soit le niveau 3 du cache. Bien que la majorité des erreurs se produisent dans la fonction get de Types.ml et propagate du solveur, ces défauts sont attendus car ils correspondent au chargement de clauses et des observateurs (watchers) lors de la propagation. Ce n'est pas le cas pour les défauts dus au GC, pour les deux niveaux de cache, le GC représente 10% des défauts de cache de l'exécution.

Le GC provoque des défauts de cache pour deux raisons. Avant de présenter ces deux raisons, nous allons rappeler le fonctionnement du GC d'OCaml.

```
I    refs:        27,098,622,198
I1   misses:               7,280
LLi  misses:               6,118
I1   miss rate:             0.00%
LLi  miss rate:             0.00%

D    refs:        10,141,601,516  (7,676,787,419 rd   + 2,464,814,097 wr)
D1   misses:         389,604,992  (  385,992,178 rd   +     3,612,814 wr)
LLd  misses:          17,807,216  (   15,992,219 rd   +     1,814,997 wr)
D1   miss rate:             3.8%  (          5.0%     +           0.1%  )
LLd  miss rate:             0.2%  (          0.2%     +           0.1%  )

LL  refs:            389,612,272  (  385,999,458 rd   +     3,612,814 wr)
LL  misses:           17,813,334  (   15,998,337 rd   +     1,814,997 wr)
LL  miss rate:              0.0%  (          0.0%     +           0.1%  )
```

FIGURE 3.19 – résultats obtenues avec l'outil cachegrind pour l'exécution de SatML_3 sur l'exemple 19-sec.cnf.

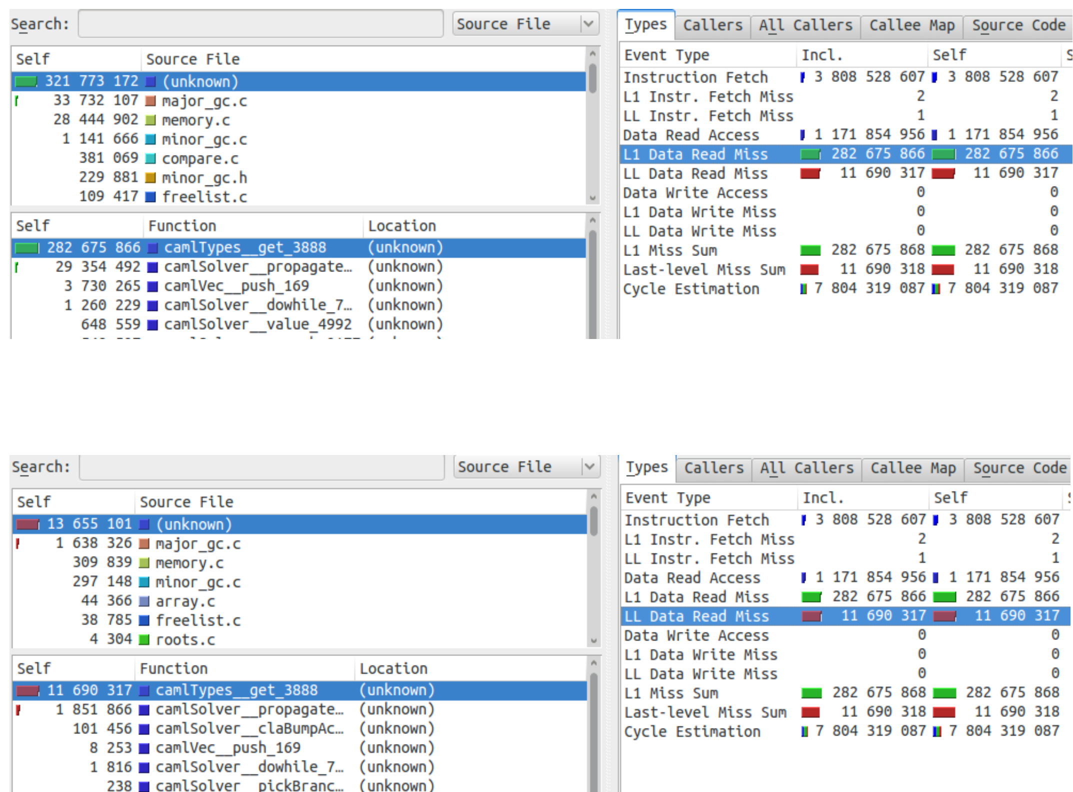

FIGURE 3.20 – Visualisation des défauts de cache au niveau L1 (haut) et LL (bas) lors de l'exécution de SatML grâce à l'outil Kcachgrind. La fonction effectuant le plus de défaut de cache au niveau L1 et LL est la fonction get de Types.ml. On remarque aussi que ˜10% des défauts de cache sont issus du fichier major_gc.c.

C'est un GC à générations constitué de deux tas, le tas mineur, et le tas majeur [17]. Le tas mineur permet d'allouer linéairement une petite quantité de données (256 Ko). Lors d'une allocation, s'il n'y a plus assez de mémoire dans le tas mineur son contenu encore "vivant" (toujours pointé dans l'environnement courant) est copié dans le tas majeur en utilisant l'algorithme de stop and copy [19]. Le tas majeur quant à lui fonctionne grâce à l'algorithme de "mark and sweep", permettant de supprimer les données qui ne sont plus vivantes. Des phases de compaction permettent de déplacer des données pour réduire les espaces entre chacune d'elles.

La première raison pour laquelle le GC provoque des défauts de cache est due au fait que lorsqu'il se déclenche, il déplace des blocs de données. Il déplace des blocs du tas mineur vers le tas majeur et peut aussi déplacer des blocs au sein du tas majeur dans des cas de compression. Ces modifications d'adresses des blocs entraînent des rechargements dans le cache et donc des défauts lorsque ces blocs sont réutilisés par le solveur. La deuxième raison pour laquelle le GC provoque des défauts de cache concerne le parcours des données non utilisées. Lors de son déclenchement, il parcourt une partie de la mémoire. Cette partie peut ne pas être en cache, car non utilisée à ce moment là. Cela provoque alors des défauts de cache. Mais ce n'est pas tout : en écrasant les valeurs dans le cache qui étaient utiles pour le solveur, le GC va provoquer d'autres défauts mémoire lorsque le solveur reprendra son exécution et devra recharger en mémoire ses données utiles. Une partie des défauts de cache dans la propagation booléenne est donc due au rechargement en mémoire après un déclenchement de GC.

2.2 Pistes d'optimisation

Nous allons maintenant aborder les pistes possibles pour optimiser notre implémentation en OCaml. Concernant la taille des données, une rapide instrumentation de notre code nous a permis d'obtenir le nombre maximum de blocs de données vivant au cours de nos exécutions, sur l'exemple traité précédemment nous obtenons un résultat de 2292224 blocs. Sachant que chaque bloc mémoire en OCaml prend 8 octets on obtient 18337792 octets soit 18.3 Mo un résultat proche de celui de Massif. Pour nous approcher de l'implémentation en C++ où la grande majorité des données utilisées tient sur 4 octets, nous aurions pu compiler et lancer SatML sur un système 4 octets, cela nous aurait permis d'obtenir une consommation en mémoire divisée par 2 et de nous approcher de Minisat. Il existe une version du compilateur d'OCaml permettant de compiler et exécuter du code 4 octets sur une machine 8 octets nous n'avons malheureusement pas pu tester cette solution sur notre système par manque de temps. Ceci serait une piste d'évaluation intéressante pour des travaux futurs.

```
module Clause = struct
  type t = {
    mutable size : int;
    mutable extra : int;
    data : Lit.Array.t;
  }
...
end
```

FIGURE 3.21 – implémentation d'une clause en OCaml proche de celle en C++.

Taille des données

Une autre optimisation concernant la taille des données qui aurait demandé plus de travail aurait été d'utiliser des bibliothèques permettant la compaction des données, par exemple représenter des booléens sur un octet et non huit. Nous avons tout de même essayé d'étudier les gains potentiels de telles optimisations.

Nous avons modifié la représentation des clauses en OCaml pour nous approcher de celles en C++ présentées sur la figure 3.14. La figure 3.21 présente cette nouvelle représentation. Comme dans l'implémentation en C++, le booléen learnt est inliné dans le bit de poids fort de la taille. Un simple test sur le signe de l'entier size permet de savoir si la clause est apprise ou non. Nous avons aussi remplacé le type extra par un entier. Nous n'avons pas besoin de la possibilité de stocker un flottant dans le champ extra car ce dernier servait pour l'heuristique de suppression des clauses apprises, heuristique que nous avons désactivée. Nous avons réduit le nombre de champs dans une clause et supprimé une de ses indirections. Malgré le fait que cette solution ne supprime pas l'indirection vers le tableau de littéraux nous l'avons testée pour observer d'éventuels gains de performance.

La figure 3.22 nous montre les résultats obtenus avec cette nouvelle représentation des clauses et l'option -O3 activée, comparés à l'implémentation SatML_3 et Minisat_3. Il apparaît que les gains sont faibles voire négligeables, +0.13% de buts supplémentaires résolus pour 1.5% de temps en moins. Nous avons cependant étudié plus en détail notre exemple précédemment analysé en lançant les outils cachegrind et valgrind. La figure 3.23 nous montre les résultats obtenus par l'outil cachegrind. On remarque une baisse significative des défauts de cache par rapport à la figure 3.19 en particulier pour le niveau LL. Pour ce qui est des résultats obtenus par l'outil massif ils sont identiques à ceux de la figure 3.11. Cela peut être dû au manque de précision de l'outil qui ne voit que la mémoire allouée par le GC de OCaml et non celle consommée.

	satML_3	satML_3_flat	minisat_3
aim	72 (0s)	72 (0s)	72 (0s)
bf	4 (0s)	4 (0s)	4 (0s)
dubois	13 (0s)	13 (0s)	13 (0s)
hanoi	1 (0s)	1 (0s)	2 (48s)
jnh	50 (0s)	50 (0s)	50 (0s)
minisat	29 (20s)	29 (19s)	31 (62s)
pigeon	4 (14s)	4 (14s)	5 (48s)
pret	8 (0s)	8 (0s)	8 (0s)
ssa	8 (0s)	8 (0s)	8 (0s)
uuf100	1000 (0s)	1000 (0s)	1000 (0s)
uuf125	100 (0s)	100 (0s)	100 (0s)
uuf150	100 (0s)	100 (0s)	100 (0s)
uuf175	100 (22s)	100 (21s)	100 (0s)
uuf250	39 (1251s)	41 (1272s)	91 (1802s)
Total	1528 (1309s)	1530 (1328s)	1584 (1962s)

Figure 3.22 – Résultats de SatML_3 avec une représentation des clauses optimisé comparé à SatML_3 et Minisat_3 sur des fichiers des la SATLIB avec une limite de temps de 60 secondes.

```
I    refs:       26,759,971,836
I1   misses:              5,608
LLi  misses:              4,802
I1   miss rate:           0.00%
LLi  miss rate:           0.00%

D    refs:       10,038,072,727  (7,610,301,860 rd   + 2,427,770,867 wr)
D1   misses:        363,050,110  (  359,751,522 rd   +     3,298,588 wr)
LLd  misses:         10,541,772  (    9,005,641 rd   +     1,536,131 wr)
D1   miss rate:            3.6%  (          4.7%     +           0.1%  )
LLd  miss rate:            0.1%  (          0.1%     +           0.1%  )

LL refs:            363,055,718  (  359,757,130 rd   +     3,298,588 wr)
LL misses:           10,546,574  (    9,010,443 rd   +     1,536,131 wr)
LL miss rate:              0.0%  (          0.0%     +           0.1%  )
```

Figure 3.23 – résultats obtenues avec l'outil cachegrind pour l'exécution de SatML_3 sur l'exemple 19-sec.cnf avec une représentation des clauses optimisées.

```
I    refs:        23,917,943,782
I1   misses:               2,899
LLi  misses:               2,565
I1   miss rate:             0.00%
LLi  miss rate:             0.00%

D    refs:         9,238,357,511  (7,073,405,466 rd   + 2,164,952,045 wr)
D1   misses:         217,859,645  (  213,818,570 rd   +     4,041,075 wr)
LLd  misses:           5,972,695  (    4,116,921 rd   +     1,855,774 wr)
D1   miss rate:              2.4% (          3.0%     +           0.2%  )
LLd  miss rate:              0.1% (          0.1%     +           0.1%  )

LL refs:             217,862,544  (  213,821,469 rd   +     4,041,075 wr)
LL misses:             5,975,260  (    4,119,486 rd   +     1,855,774 wr)
LL miss rate:                0.0% (          0.0%     +           0.1%  )
```

FIGURE 3.24 – résultats obtenues avec l'outil cachegrind pour l'exécution de SatML_3 sur l'exemple 19-sec.cnf sans lancement du GC d'OCaml.

Cette expérimentation tend à montrer que des optimisations concernant la taille des données et la réduction des indirections pourraient améliorer les performances de notre implémentation en OCaml.

Impact du GC

Pour ce qui est du GC, nous avons essayé de simuler au mieux son absence pour analyser les conséquences qu'il peut avoir sur les défauts de cache. Pour cela, sur notre problème précédemment étudié, nous avons défini la taille du tas mineur du GC avec une valeur supérieure à la consommation totale de son exécution, soit plus de 19 Mo. Cela permet que le GC ne se déclenche jamais. Cela nous permet d'améliorer le temps de résolution passant de 14 secondes à 8.5 secondes. La figure 3.24 nous montre les statistiques de cette nouvelle exécution. Nous passons de près de 400 millions de défauts de cache en L1 à 218 millions. De même pour les défauts de cache en L3 où l'on observe un gain de près de 75%. Si nous regardons les détails dans la figure 3.25 nous pouvons remarquer qu'il n'y a plus de défaut de cache produit directement par le déclenchement du GC qui correspondait au 10% des défauts totaux précédemment calculés. Au vu des chiffres nous remarquons aussi que le lancement du GC a bien une incidence néfaste sur nos fonctions de propagation où l'on peut observer un gain bien plus important en termes de défaut de cache que pour le lancement de ce dernier.

Des recherches ont été effectuées pour éviter ce type de conséquence lors des lancements des GC. Boehm [15] utilise ainsi des opérations permettant de ne pas charger les données depuis la mémoire en passant par les caches permettant un gain d'environ 10% de la vitesse d'exécution du GC.

L'opération assembleur `prefetch_NTA` permet ainsi de ne pas écraser les données utiles dans les caches. Cette opération est normalement utilisée pour les flux de données où chacune des données n'est utilisée qu'une seule fois. Cela convient parfaitement au fonctionnement d'un GC qui lit et parcourt chaque donnée une seule fois lors d'une phase de marquage. Une telle opti-

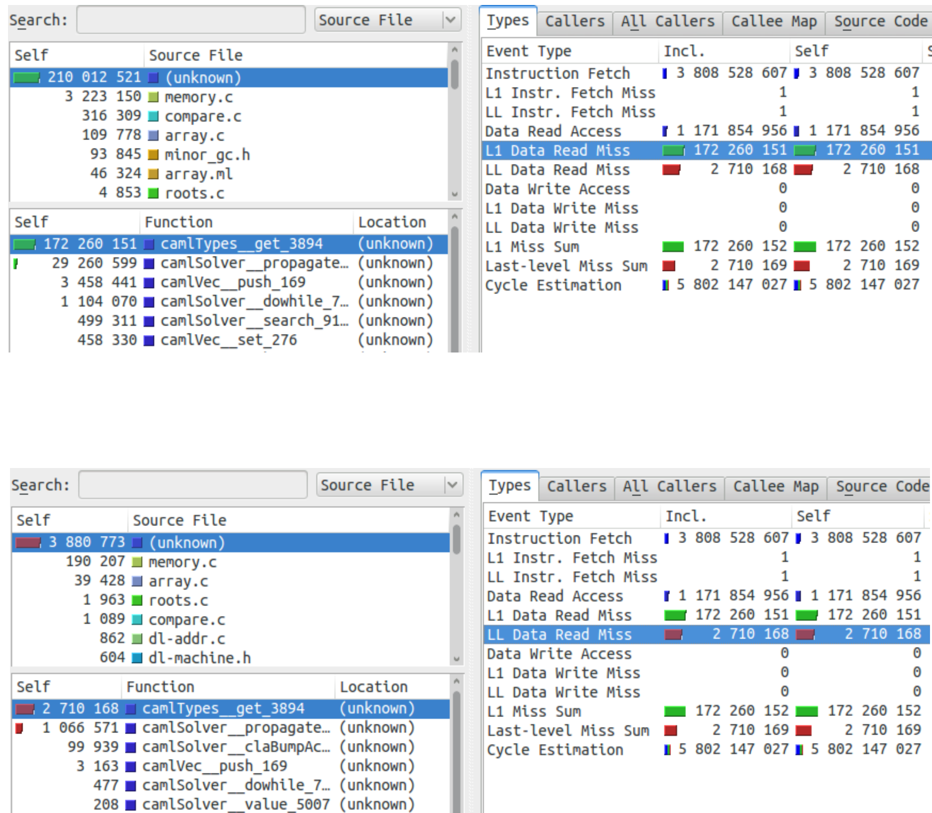

Figure 3.25 – Visualisation des défauts de cache au niveau L1 (haut) et LL (bas) lors de l'exécution de SatML grâce à l'outil Kcachgrind. La fonction effectuant le plus de défaut de cache au niveau L1 et LL est la fonction get de Types.ml. Elle provoque cependant moins de défaut de cache que lorsque le GC est lancé au cours de l'exécution. On remarque d'ailleurs la disparition du fichier major_gc.c dans les fichiers provoquant des défaut de cache.

misation sur le compilateur d'OCaml permettrait non seulement d'améliorer les performances de notre solveur mais aussi des performances de OCaml.

Concernant l'impact des déplacements de blocs par le GC, la majeure partie des données semblent être des clauses. Notre algorithme ne supprimant pas ces clauses, la gestion automatique de la mémoire est ici utile uniquement pour la libération de la mémoire à la fin de l'exécution. Il n'existe malheureusement aucun moyen de déclarer nos clauses inutiles à parcourir sans risquer de faire planter le programme. En effet, il est possible de modifier leur `tag` utilisé par le GC pour que celui-ci ne les parcourt et ne les déplace pas. Mais ces clauses comprenant des pointeurs, le GC n'aura plus connaissance de ces objets pointés, cela peut donc entraîner un écrasement de ces objets par d'autres allocations ou par une compaction. Une autre solution traitée par Bozman [17] concerne la gestion de la mémoire par région, cela nous permettrait ici de ne pas avoir à parcourir nos clauses et ainsi de garder nos localités et d'éviter des erreurs de cache inutiles.

2.3 SatML vs solveur historique d'Alt-Ergo

Les résultats de SatML et de Minisat étant proches nous avons voulu les comparer au solveur historique d'Alt-Ergo. Ce solveur étant basé sur la méthode des tableaux pour parcourir la formule du problème, nous l'appellerons `AE_tableaux`. Alt-Ergo comporte depuis quelques années un solveur SAT CDCL basé sur Minisat, nous l'appellerons `AE_satml`. Contrairement à notre implémentation SatML, `AE_satml` diffère en termes de décision par rapport à Minisat et SatML et n'effectue en effet pas exactement les mêmes décisions et donc pas les mêmes propagations.

Nous avons donc décidé de comparer ces deux solveurs SAT au sein d'Alt-Ergo sur les mêmes problèmes provenant de la SATLIB utilisés précédemment. Nous utilisons l'option `-no-theory` d'Alt-Ergo nous assurant qu'aucun raisonnement modulo théories ne sera effectué.

Nous avons lancé nos deux configurations de solveur SAT dans Alt-Ergo, et nous n'avons aussi ajouté les résultats de SatML_3 comme référence. Nous avons pris une limite de temps d'une minute, au vu du mauvais score obtenu par le solveur `AE_tableaux` nous avons décidé de le relancer avec une limite de temps de 10 minutes qui correspond à la quatrième colonne. Les résultats présentés sur le tableau 3.26 et sur la figure 3.27 ne représentent que les réponses UNSAT, Alt-Ergo ne pouvant répondre SAT. Pour les résultats de SatML_3 nous avons aussi sélectionné que les UNSAT. Ces résultats confirment que le solveur `AE_tableaux`, n'est pas performant pour le travail booléen pur n'impliquant ni théories ni quantificateurs. Les différences de temps et de résultats entre SatML_3 et `AE_satml` sont expliquées par la différence dans les décisions, ainsi que le fait qu'Alt-Ergo n'utilise pas des structures de données aussi optimisées que SatML.

	satML_3	AE_satml	AE_tableaux(60s)	AE_tableaux_long(600s)
aim	**24(0s)**	0(0s)	0(0s)	0(0s)
bf	**4(0s)**	**4(0s)**	1(11s)	2(366s)
dubois	**13(0s)**	**13(0s)**	0(0s)	0(0s)
hanoi	0(0s)	0(0s)	0(0s)	0(0s)
jnh	**34(0s)**	**34(0s)**	8(162s)	15(1863s)
minisat	**18(21s)**	18(44s)	11(0s)	11(0s)
pigeon	4(14s)	**4(13s)**	2(7s)	3(89s)
pret	**8(0s)**	**8(0s)**	0(0s)	1(307s)
ssa	**4(0s)**	4(3s)	1(18s)	2(408s)
uuf100	**1000(0s)**	**1000(0s)**	6(336s)	840(263914s)
uuf125	**100(0s)**	**100(0s)**	0(0s)	0(0s)
uuf150	**100(0s)**	100(34s)	0(0s)	0(0s)
uuf175	**100(23s)**	100(241s)	0(0s)	0(0s)
uuf250	**38(1204s)**	0(0s)	0(0s)	0(0s)
Total	**1447(1263s)**	1385(337s)	29(535s)	874(266950s)

Figure 3.26 – Résultats de satml optimisé comparé aux solveurs SAT d'Alt-Ergo sur des fichiers de la SAT-LIB.

3 Conclusion

Nous avons présenté une comparaison de deux implémentations d'un algorithme de résolution SAT. Ces deux implémentations fonctionnellement identiques sont écrites dans deux langages différents. Après nous être assurés que les deux implémentations effectuaient les mêmes décisions et propagations, nous avons étudié et constaté des différences de performance. Nous avons expliqué ces différences par différentes raisons. La première est que le langage OCaml ne permet nativement pas au développeur d'optimiser la taille des structures de données utilisées. La seconde raison concerne le système de gestion automatique de la mémoire de OCaml. Il impacte la consommation mémoire du programme à cause des en-têtes alourdissant l'empreinte mémoire de chaque bloc et il impacte la gestion du cache dégradant fortement les performances. Ce système de gestion automatique de mémoire peut avoir un impact important sur les performances, allant jusqu'à doubler le temps de résolution pour certains exemples. Enfin, la dernière raison concerne le manque d'optimisation offerte par le compilateur d'OCaml comparé à un compilateur de référence pour le langage C++.

Nos résultats expérimentaux ont permis de montrer que l'implémentation optimisée en C++ était plus performante que celle en OCaml. Bien que l'implémentation SatML en OCaml soit moins performante, elle montre de très bons résultats concernant la résolution de problèmes SAT. Ces résultats ont été confirmés lorsque nous avons comparé une implémentation proche de celle de SatML à un solveur SAT par méthode des tableaux au sein du

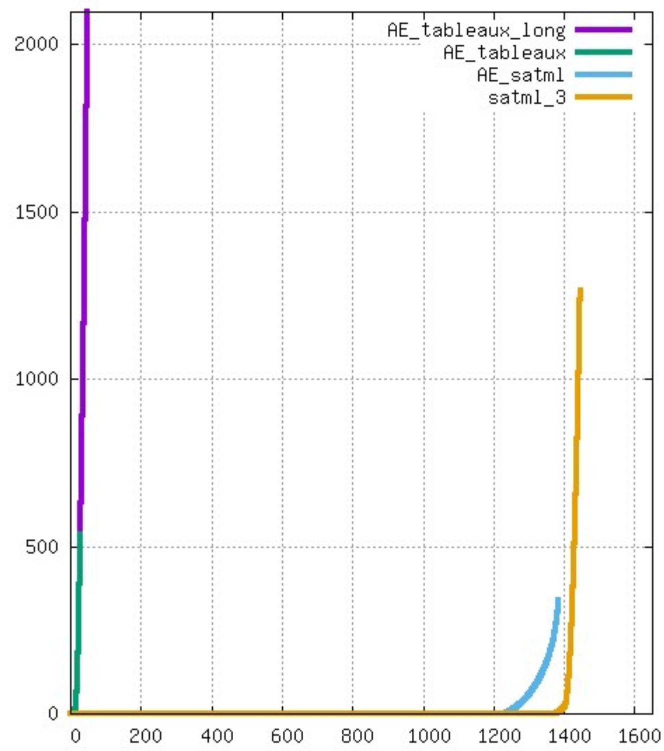

Figure 3.27 – Graphique du temps de résolution des solveurs SAT d'Alt-Ergo et de SatML optimisé en fonction du nombre de buts résolus sur des fichiers issus de la SAT-LIB.

solveur SMT Alt-Ergo.

Chapitre 4

Intégration efficace d'un solveur CDCL dans Alt-Ergo

Comme nous l'avons présenté dans le chapitre 2 section 1.2, le solveur SAT historique d'Alt-Ergo est basé sur la méthode de résolution par tableaux. Ce solveur n'utilise pas une mise en CNF classique et n'utilise pas des techniques performantes pour le BCP.

Dans le chapitre précédent, nous avons montré qu'une implémentation en OCaml d'un solveur SAT performant offrait de bons résultats. Nous avons aussi montré que cette dernière offrait de bien meilleures performances sur des problèmes SAT que le solveur historique d'Alt-Ergo. Nous allons dans un premier temps montrer que le branchement d'un solveur SAT performant au sein d'un solveur SMT n'est ni direct, ni trivial. Nous allons montrer cela à travers un ensemble de problèmes comportant des quantificateurs universels en section 1. Puis nous présenterons une piste d'optimisation du solveur historique en assistant ses décisions à l'aide d'un solveur CDCL servant d'oracle.

Nous allons ensuite présenter (section 2) un algorithme permettant d'obtenir un solveur SMT utilisant un solveur SAT CDCL performant. Nous présenterons tout d'abord le cheminement nous ayant amené à une optimisation performante de cette combinaison, que nous formaliserons. Puis nous expliquerons quelques détails d'implémentation permettant d'obtenir de meilleurs résultats. Nous finirons par présenter les performances de cette optimisation grâce à des résultats expérimentaux dans la section 4. Nous y étudierons l'impact de cette optimisation sur les différentes parties d'un solveur SMT ainsi que sur des heuristiques, et enfin nous comparerons Alt-Ergo à d'autres solveurs SMT.

Nous rappelons tout d'abord quelques terminologies : SatML : implémentation de Minisat en OCaml. CDCL : algorithme de décision du problème SAT. AE_Tab-like : nom donné au solveur historique d'Alt-Ergo utilisant la méthode des tableaux.

	AE_cdcl	AE_Tab-like
BWARE-VCs–DAB	849(98.72%)(261s)	**860(100.00%)(425s)**
BWARE-VCs–RCS3	2228(98.75%)(761s)	**2233(98.98%)(696s)**
BWARE-VCs–p4	9194(98.42%)(2056s)	**9272(99.26%)(2312s)**
BWARE-VCs–p9	242(65.22%)(1211s)	**252(67.92%)(345s)**
EACSL-BY-EXAMPLE-VCs	725(75.59%)(64s)	**895(93.32%)(261s)**
SPARK-VCs	13498(81.19%)(1804s)	**14026(84.36%)(2723s)**
WHY3-VCs	779(38.89%)(601s)	**1442(71.99%)(1954s)**
Total	27515(84.88%)(6761s)	**28980(89.40%)(8719s)**

Figure 4.1 – Résultats du solveur CDCL (AE_cdcl) et du solveur historique (AE_Tab-like) d'Alt-Ergo sur des fichiers SMT issus de la preuve de programme.

1 Première approche : assister le solveur historique d'Alt-Ergo avec un solveur CDCL

Les travaux présentés dans le chapitre 3 section 2.3 nous ont permis de mesurer la faiblesse du solveur historique d'Alt-Ergo concernant le raisonnement booléen. Suite aux bons résultats du solveur AE_cdcl, dont l'implémentation est proche de Minisat, nous avons souhaité comparer ses performances face au solveur historique d'Alt-Ergo, AE_Tab-like. Nous avons pour cela lancé ces deux solveurs sur un ensemble de problèmes SMT issus de la preuve de programme fournis par nos partenaires. Comme pour les expérimentations précédentes, nous utiliserons la plate-forme Starexec avec une limite de temps d'une minute par but. Nous présentons dans la figure 4.1 les résultats obtenus. AE_Tab-like montre de bien meilleurs résultats (+4.52% de buts résolus en plus) sur ce type d'exemple que son homologue AE_cdcl, en particulier sur la section WHY3 (+33% de buts résolus supplémentaire).

Au vu de ces résultats, il apparaît qu'il ne suffit pas d'un solveur SAT performant pour obtenir de bonnes performances sur des problèmes SMT avec quantificateurs universels. Partant du constat que le solveur SAT utilisant la méthode des tableaux était plus performant sur les problèmes SMT mais n'était pas efficace pour le raisonnement booléen, nous avons voulu utiliser la puissance du calcul BCP d'un SAT CDCL pour assister celui de AE_Tab-like.

Le but était de garder le fonctionnement normal du solveur historique concernant les théories et les gestions des quantificateurs. Les décisions restent effectuées par le solveur historique, ses heuristiques étant liées aux théories et aux quantificateurs. Nous nous servons du solveur SatML comme un oracle, tel que présenté sur la figure 4.2. Le solveur CDCL est vu comme un élément extérieur au solveur SAT, qui ne fait que l'interroger concernant le raisonnement booléen.

Nous présentons ces modifications de l'algorithme du solveur historique

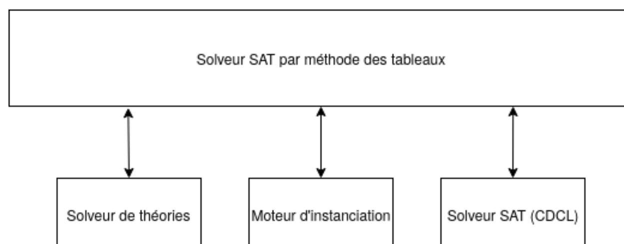

Figure 4.2 – Diagramme représentant la combinaison d'un solveur SAT basé sur la méthode des tableaux au sein d'un solveur SMT, avec un solveur SAT CDCL pour son raisonnement booléen.

d'Alt-Ergo (présenté dans le chapitre 2) dans la figure 4.3. La fonction *solve* prend un argument supplémentaire correspondant à l'environnement d'un solveur SatML modifié. Ce solveur est modifié pour ne plus faire de décisions et uniquement effectuer du BCP sur les assignations que nous lui fournissons. Il est initialisé avec une mise en CNF du problème à résoudre. Les propagations au sein du solveur AE_Tab-like restent inchangées. Nous avons factorisé le code permettant de vérifier qu'il n'y a pas eu de conflit suite à une propagation ou une décision dans la fonction *check_error* présenté sur la figure 4.4. Lors de chaque décision, si un des littéraux de la disjonction à décider est vrai dans SatML alors on propage l'information dans AE_Tab-like, sinon on appelle la fonction *cdcl_solve*.

La fonction *cdcl_solve* présentée dans la figure 4.5 prend les mêmes arguments que *solve* ainsi qu'un ensemble de variables S à assigner. La première étape (ligne 2) est d'assigner ces variables dans SatML. La fonction *assign* force l'assignement des variables dans S et propage l'information avec un BCP performant. Si aucun conflit n'est détecté, l'environnement modifié est retourné. Sinon l'algorithme CDCL résout le conflit et apprend une clause à partir du conflit booléen. Le statut UNSAT est ensuite retourné, ainsi qu'une explication permettant au solveur historique de retourner à un niveau de décision non conflictuel. Si la fonction *assign* ne retourne pas UNSAT nous pouvons alors appeler récursivement *solve* avec nos décisions et propagations contenues dans S. Contrairement à Φ et T, Π n'est pas développé avec des structures de données fonctionnelles et nous devons donc forcer le backtrack des assignations. La fonction $forget$ permet de désassigner les variables contenues dans S pour permettre à Π de revenir à son état initial.

La modification apportée ne modifie pas l'interaction du solveur SAT avec les théories. Pour ce qui est des instances, nous les convertissons en CNF avant de les ajouter à l'environnement Π (ligne 28). Comme pour les clauses apprises, les instances ne sont pas supprimées dans l'environnement Π.

Cette expérimentation a été implémentée et testée. Nous présentons les

```
Input: Φ : Set of formula in NNF, Π : CDCL BCP environment, T : Theory
       environment
Output: Satisfiability status, explanation, CDCL BCP environment
1  Function solve(Φ, Π, T)
2      (Φ, T, Error) ← propagate(Φ, T)                       // BCP modulo Theory
3      check_error(Φ, Π, T, Error)
4      if ∃A ∨ B ∈ Φ then
5          if A is assigned in Π then
6              (Φ, T, Error) ← (assume(Φ, T, [{A}])                 // propagate A
7              check_error(Φ, Π, T, Error)
8              return solve(Φ, Π, T)
9          else
10             if B is assigned in Π then
11                 (Φ, T, Error) ← (assume(Φ, T, [{B}])             // propagate B
12                 check_error(Φ, Π, T, Error)
13                 return solve(Φ, Π, T)
14             else
15                 (Φ, T, Error) ← (assume(Φ, T, [{A}])             // decide on A
16                 check_error(Φ, Π, T, Error)
17                 (status, reason) ← cdcl_solve(Φ, Π, T, {A})
18                 if status ≠ UNSAT then
19                     return (status,reason)
20                 else
21                     if A ∈ reason then
22                         (Φ, T, Error) ← (assume(Φ, T, [{¬A}; {B}])
                           // backtrack and propagate ¬A and B
23                         check_error(Φ, Π, T, Error)
24                         return cdcl_solve(Φ, Π, T, {¬A, B})

25     else
26         I ← insantiate(Φ, T)
27         if I ≠ ∅ then
28             (Φ, T, Error) ← (assume(Φ, T, I))              // assume instances
29             check_error(Φ, Π, T, Error)
30             Π ← add_to_cnf(Π, I)
31             return solve(Φ, Π, T)
32         else
33             return UNKNOWN
```

Figure 4.3 – Modification de l'algorithme 2.9 avec ajout d'un module pour assister la décision booléenne

Input: Φ : Set of formula in NNF, Π : CDCL BCP environment, T : Theory environment

1 **Function** $check_error(\Phi, \Pi, T, Error)$
2   **if** $Error$ **then**
3         $reason \leftarrow explain_conflict(\Phi, T, \Pi)$
4   **return** (UNSAT,$reason$)

Figure 4.4 – vérifie qu'une propagation ou décision n'a pas amené à un conflit.

Input: Φ : Set of formulas, Π : CDCL BCP environnment, T : Theory environment, S : set of assignments
Output: Satisfiability status, explanation, Π

1 **Function** $cdcl_solve(\Phi, \Pi, T, S)$
2     $(status, reason, \Pi) \leftarrow assign(\Pi, S)$
3   **if** $status \neq UNSAT$ **then**
4         $(status, reason, \Pi) \leftarrow solve(\Phi \cup S, \Pi, T)$
5         $\Pi \leftarrow forget(\Pi, S)$
6   **return** $(status, reason, \Pi)$

Figure 4.5 – Synchronisation des deux environnement SAT. Si le littéral à propager ne déclenche pas un conflit dans le SAT CDCL, on le propage aussi dans le SAT historique. On oublie cette assignation lorsque l'on fait un backjump.

	`AE_Tab-like`	`AE_Tab-like+cdcl`
BWARE–DAB	860 (417s)	**860 (277s)**
BWARE–RCS3	**2233 (685s)**	2233 (702s)
BWARE–p4	**9272 (2279s)**	9271 (907s)
BWARE–p9	252 (342s)	**264 (769s)**
EACSL-BY-EXAMPLE	895 (258s)	**896 (244s)**
SPARK	14026 (2757s)	**14028 (2710s)**
WHY3	1442 (1876s)	**1443 (1936s)**
Total	28980 (8617s)	**28995 (7549s)**

Figure 4.6 – Résultats du solveur historique d'Alt-Ergo avec (AE_Tab-like+cdcl) et sans (AE_Tab-like) l'assistance d'un solveur CDCL basé sur SatML pour le BCP performant.

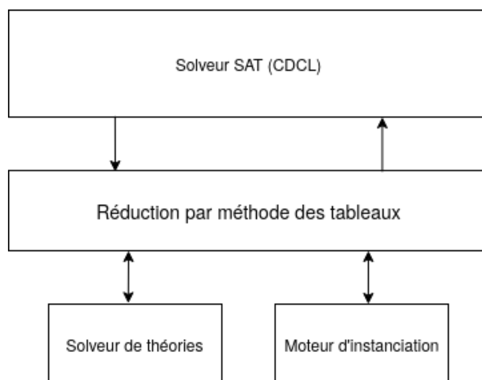

Figure 4.7 – Diagramme représentant la combinaison d'un solveur SAT avec le solveur de théories et le moteur d'instanciation dans un solveur SMT.

résultats obtenus dans la figure 4.6. Cette expérimentation notée AE_Tab-like+cdcl offre des gains vis-à-vis du solveur historique AE_Tab-like, en particulier concernant les temps d'exécution. Cela nous a permis de valider notre hypothèse concernant la combinaison des éléments d'un solveur SMT.

2 CDCL(λ(T)) : CDCL modulo Tableau(T)

2.1 Réduction du modèle booléen par méthode des Tableaux

Après cette première expérimentation sur le solveur AE_Tab-like, bien que les résultats soient encourageants, nous ne tirons pas pleinement profit du solveur SatML et en particulier de son heuristique de décision. Nous avons toujours les points négatifs de AE_Tab-like tels que son manque d'apprentissage ou son BCP peu performant. Nous avons recherché une méthode pour optimiser cette combinaison et rendre notre solveur SMT efficace avec un solveur SAT performant.

Nous avons supposé que les bonnes performances du solveur AE_Tab-like étaient en partie dues au fait que le nombre de littéraux envoyés aux théories et au moteur d'instanciation est moins important que pour notre implémentation SatML seule.

Étudiant cette hypothèse, nous avons implémenté une solution permettant de réduire l'ensemble des littéraux envoyés aux autres composants du solveur SMT. Cette solution consiste à garder la "forme" de la formule originale et de la parcourir par méthode des tableaux. L'interaction entre les différentes parties du solveur SMT est résumée dans la figure 4.7. Le solveur CDCL est maître des décisions et plus globalement du raisonnement booléen. Lorsqu'une interaction avec le solveur de théories ou le moteur d'instanciation est nécessaire, il envoie ses affectations booléennes au raisonnement

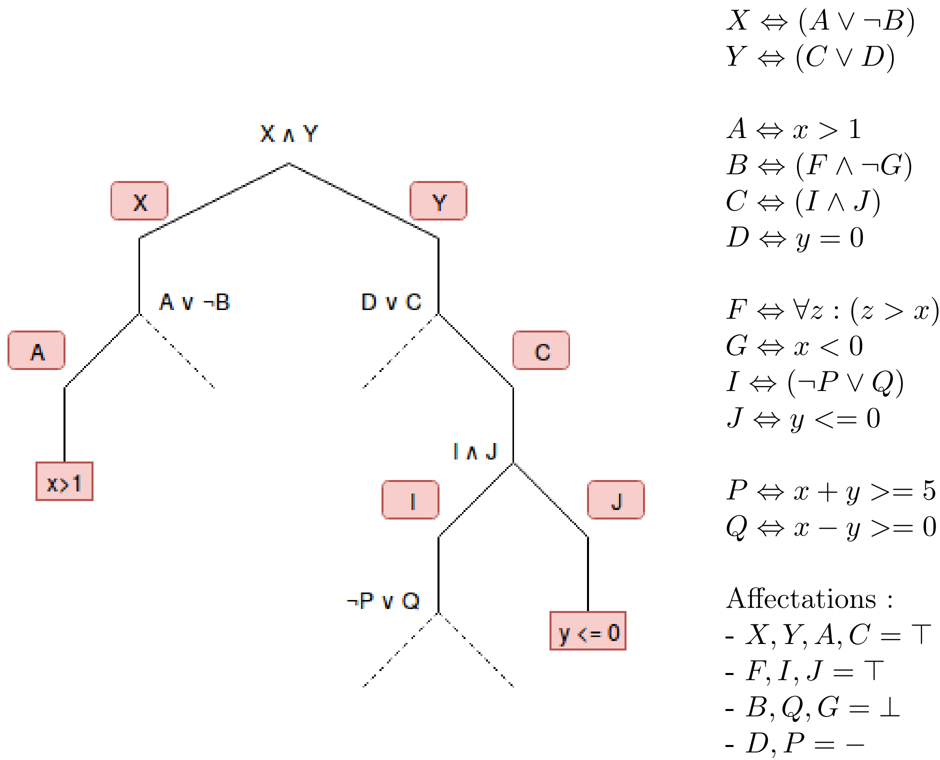

FIGURE 4.8 – Exemple de calcul de pertinence via la méthode des tableaux. Les formules en rouge sont dites pertinentes.

Tableau qui se comporte comme un intermédiaire.

Nous appelons formule pertinente, toute formule assignée à vraie rencontrée au cours du parcours de la formule originale. La figure 4.8 nous montre le parcours de la formule $X \wedge Y$. La formule étant une conjonction et devant être vraie, X et Y sont vraies par BCP. Pour connaître la valeur de vérité correspondant à une formule, on utilise un proxy. Comme expliqué auparavant (chapitre 2) un proxy est un littéral correspondant à une formule. Le SAT ne travaillant pas sur les formules nous utilisons leurs proxys pour connaître leur valeur de vérité. X et Y sont des variables intermédiaires représentant respectivement $A \vee \neg B$ et $C \vee D$. Une fois X et Y marquées comme pertinentes, nous les ouvrons et étudions $A \vee \neg B$ et $C \vee D$. Dans notre parcours, lorsque l'on rencontre une disjonction, nous parcourons ses membres à la recherche d'une formule vraie. Ici A est vraie, nous la marquons comme pertinente et l'ouvrons. A représente le terme $X > 1$ qui est à son tour marqué comme pertinent. La branche $\neg B$ de X n'est pas étudiée car nous n'étudions qu'une formule vraie par disjonction. Nous faisons ensuite de même pour Y où D n'est pas assignée mais C l'est. C est marquée comme pertinente ainsi que les formules I et J incluses dans sa conjonction.

Q étant faux et P n'étant pas assigné, nous ne pouvons parcourir davantage cette branche issue de I. Une branche du parcours n'étant pas terminée, il n'existe pas encore de modèle booléen satisfaisant $X \wedge Y$.

Bien que B soit affecté, nous ne parcourons pas, il en est ainsi pour les formules qu'il renferme. F et $\neg G$ bien qu'elles soient affectées par le SAT, ne seront pas ajoutées à notre ensemble de formules pertinentes. Au final, le nombre de formules pertinentes(X, Y, A, C, I, J) est moindre que les affectations du SAT $(X, Y, A, B, C, F, G, I, J, Q)$

C'est en quelque sorte une simulation de la CNF paresseuse utilisée par le solveur historique. Cette solution permet de ne pas interférer avec le solveur SAT et ses heuristiques fines et peut être implémentée autour de n'importe quel solveur SAT.

Le solveur Z3 [32] de Microsoft utilise lui aussi une technique semblable basée sur la technique des tableaux. Leurs travaux sont rapidement présentés dans le papier [56] et dans le rapport de recherche [33] où ils expliquent s'être basés sur la CNF paresseuse de Simplify [36]. Seule l'idée est présentée dans ces documents, et l'implémentation est elle présentée comme un hack, aucun formalisme ou résultats expérimentaux concrets ne sont présentés.

Le solveur CVC4 [10] implémente une idée similaire de réduction de modèle grâce au circuit électronique [46] permettant de maintenir des frontières de justification [4].

Le solveur VeriT [39] utilise quant à lui en partant d'un modèle booléen complet et combiné à la technique des deux littéraux observés, des implicants premiers [34] permettant de réduire le modèle booléen.

Il s'avère après discussion avec des membres de la communauté que cette optimisation est connue, mais est considérée comme du folklore. Nous avons jugé qu'une étude plus approfondie était nécessaire. C'est ce que nous présenterons dans la suite de ce chapitre grâce à une formalisation de cette optimisation via la présentation d'un algorithme montrant les modifications à apporter par rapport à celui d'un solveur SMT classique basé sur un SAT CDCL. Nous présenterons ensuite des preuves de complétude, correction et terminaison pour cet algorithme ainsi que des détails d'implémentation le rendant performant. Pour finir nous présenterons des résultats expérimentaux montrant l'intérêt d'une telle optimisation.

2.2 Formalisme

Nous allons maintenant formaliser notre solution en partant d'un algorithme CDCL(T) avec instanciations. La figure 4.9 présente les modifications que nous devons apporter pour notre optimisation. Le premier changement notable par rapport à un algorithme CDCL standard est l'ajout de Φ dans la fonction *propagate*, et Δ dans son type de retour. On ajoute Φ dans l'appel de la fonction car elle contient désormais le calcul de pertinence. Nous la détaillerons par la suite. Δ correspond aux formules pertinentes qui ont

```
Input: Φ: formula in Negative Normal Form
Output: Satisfiability status
1  Function solve (Φ, T)
2      Γ ← to_CNF(Φ)                      // initialize SAT environment
3
4      T ← ∅                              // initialize theories environment
5
6      while true do
7          (Γ, Δ, T, Error) ← propagate(Γ, Φ, T)
           // BCP modulo Theory with relevancy
8          if Error then
9              C ← explain_and_resolve_conflict(Γ, T)
10             if clause C is empty then
11                 return UNSAT
12             else
13                 Γ ← Γ ∪ C              // Conflict driven clause learning
14                 (Γ, Φ, T) ← backjump(Γ, Φ, T, C)
                   // non chronological backtracking
15         else
16             if all Tableaux branches are closed then
17                 I ← instantiate(Γ, Δ, T)
18                 if I ≠ ∅ then
19                     Γ ← Γ ∪ to_CNF(I)   // add new instances to Γ
20
21                     Φ ← Φ ∧ I           // add new instances to Φ
22
23                 else
24                     if check_theories_model(Γ, T) then
                           // complete theories procedure and
                              combination
25                         return SAT
26                     else
27                         return UNKNOWN
28             else
29                 Γ ← choose_and_assign(Γ)   // choose a literal to
                    assign
```

FIGURE 4.9 – Prise en compte des littéraux pertinents dans le calcul d'instance

Input: Γ: SAT environment, Φ: set of formula in negative normal form, T: Theory environment
Output: Γ, Δ: set of relevant formula, T, *Error*: Boolean

```
1  Function propagate(Γ, Φ, T)
2      Γ, Error ← boolean_propagation(Γ)
3      if Error then
4          return (Γ, ∅, T, Error)                    // Boolean Conflict
5
6      else
7          Δ ← compute_relevancy(Γ, Φ)                 // Tableaux method
8
9          Γ, T, Error ← theory_propagation(Γ, T, Δ)
10         return (Γ, Δ, T, Error)
```

Figure 4.10 – Ajout du calcul de pertinence dans la propagation modulo théories.

été calculées. Ces formules seront utilisées dans l'instanciation (ligne 14). Le deuxième changement notable est la modification de la condition pour l'instanciation. Là où un solveur CDCL(T) traditionnel vérifie que toutes les variables sont assignées, nous vérifions que toutes les branches que nous avons explorées par méthode des tableaux, lors du calcul de pertinence, sont fermées. Une branche est dite fermée quand ses feuilles ne sont pas des disjonctions ou conjonctions que nous pourrions traiter. Par exemple sur l'exemple 4.8 précédemment présenté, les branches X, A et Y, C, J sont fermées. Ce qui n'est pas le cas pour la branche Y, C, I. Lorsque toutes les branches explorées par méthode des tableaux sont fermées, nous instancions en prenant Δ en paramètre. Nous prenons aussi Γ et T en paramètre car l'instanciation est liée aux heuristiques. Lorsqu'aucune instance n'est trouvée avec Δ nous utilisons un modèle contenant plus de littéraux comme cela était fait avant lorsque nous utilisions un solveur SAT sans réduction du modèle.

Φ est maintenant ajouté à la fonction de *backjump* car nous devons supprimer les éventuelles instances apprises à un niveau supérieur de décision. Bien que ces instances soient ajoutées au solveur SAT mais jamais supprimées, le retour en arrière de Φ nous permet de ne pas tenir compte des instances qui ne correspondent plus aux décisions de l'état actuel du solveur SAT, pour le calcul de pertinence.

La figure 4.10 présente la fonction *propagate*. À la ligne 6, nous remarquons la fonction compute_relevancy. Cette fonction prend en paramètre l'environnement du solveur SAT ainsi que Φ l'ensemble des formules représentant le problème. *compute_relevancy* parcourt Φ avec la méthode des tableaux et marque la formule comme pertinente. Elle retourne cet ensemble, Δ, qui est ensuite utilisé pour les propagations modulo théories. La fonction

```
Input: Γ: SAT environment, Φ: set of formula in negative normal form
Output: Relevant: set of relevant formula
1  Function compute_relevancy(Γ, Φ)
2      Work ← ∅                                  // initialize working queue
3
4      forall formula f in Φ do
5          if proxy(f) ∈ Γ then
6              push(f, Work)
7      while Work is not empty do
8          f ← pop(Work)
9          add proxy(f) in Relevants
10         if f is a conjunction of f_i then
11             forall f_i do
12                 push(f_i, Work)
13         else
14             if f is a disjunction of f_i then
15                 if ∃ proxy(f_j) ∈ Γ then
16                     push(f_j, Work)
17     return Relevants
```

FIGURE 4.11 – Calcul des littéraux pertinants par méthode des tableaux.

theory_propagation prend en effet Δ en argument, et ne travaillera donc plus sur l'ensemble des assignations booléennes de Γ mais uniquement sur les littéraux pertinents de Δ.

La figure 4.11 détaille une version naïve du fonctionnement du calcul de pertinence. Nous prenons en entrée Γ l'environnement du solveur SAT, il contient en particulier les littéraux assignés. Φ correspond à la formule du problème initial et représente donc une conjonction de formules. Elles peuvent être des littéraux, des conjonctions ou des disjonctions. Nous ajoutons dans un premier temps toutes les formules dont le proxy est vrai de Γ à notre queue de travail, $Work$. Ensuite nous ajoutons tous les proxys des formules contenues dans $Work$ dans notre ensemble de valeurs pertinentes. Nous utilisons les proxys des formules car le solveur de théories ainsi que le moteur d'instanciation travaillent sur des littéraux. Si les formules sont des conjonctions de f_i nous ajoutons ces f_i à notre queue de formules à traiter. Par contrainte booléenne, si une conjonction est vraie, alors toutes ses formules le sont aussi. Dans le cas d'une disjonction, si au moins une formule contenue dans cette disjonction a son proxy vrai dans Γ, alors on l'ajoute à notre queue $Work$. Nous parcourons ainsi les formules de Φ par méthode des tableaux en ajoutant tous leurs proxys à notre ensemble de valeurs perti-

nentes. Les éventuels littéraux obtenus sont ensuite envoyés à la théorie pour vérifier la cohérence des affectations. Ils sont aussi utilisés lors de la phase d'instanciation.

Notre optimisation ne modifie donc pas le fonctionnement SAT du solveur SMT. Le filtrage par pertinence n'impacte que l'interaction entre le solveur SAT et le solveur de théories ainsi que le moteur d'instanciation.

2.3 Terminaison, Correction, Complétude

Dans cette section, nous allons démontrer que notre optimisation n'influe pas sur la correction, la complétude ou la terminaison d'un solveur SMT. Nous allons pour cela proposer trois preuves pour garantir ces trois caractéristiques d'un algorithme et prouver que notre optimisation est correcte.

Axiome 1. *Le solveur CDCL(T) est correct, complet et il termine si la théorie T est décidable, voir [59] pour plus de détails.*

Théorème 1. *Soit P un problème et C sa CNF. Notons $CDCL(Tableaux(T))$ un solveur CDCL avec calcul de pertinence par méthode des tableaux pour la combinaison de son modèle booléen avec les théories. Notons M le modèle booléen du solveur CDCL, et m le filtre par méthode des Tableaux sur M. Soit $Z = CDCL(Tableaux(T))$, Z est correct, complet et termine.*

Terminaison

Z termine sur $P + C$.

Démonstration. Indépendamment des heuristiques du solveur SAT (redémarrage, suppression des clauses apprises...) ; si il y a n variables booléennes dans l'abstraction de C, alors le CDCL de Z va explorer et choisir librement au maximum 2^n modèles. Chacun de ces modèles sera exploré une seule fois au maximum. En effet l'apprentissage des clauses de conflit nous permet de garantir qu'aucune chaîne de décision ne peut amener à un conflit ayant déjà eu lieu. Cela nous permet de garantir la terminaison du CDCL de Z qui ne peut pas parcourir plus de 2^n modèles. Le calcul de pertinence n'influant pas sur le fonctionnement du CDCL de Z, Z termine. □

Correction

Si Z répond SAT sur $P + C$ alors P(et C) sont SAT.

Démonstration. Si Z répond SAT, c'est qu'il s'est arrêté sur un modèle m tel que P s'évalue à vrai sous ce modèle. m est donc un modèle de P. Comme P et C sont équi-satisfiables, C est aussi SAT. □

Complétude

Si Z répond UNSAT sur $P + C$ alors P (et C) sont UNSAT.

Démonstration. Si Z répond UNSAT c'est qu'il a déduit un ensemble de clauses rendant le problème C UNSAT. Toutes les clauses déduites par Z auraient été déduites par CDCL(T). En effet :

— Chaque conflit booléen déduit par Z aurait été déduit de la même façon par CDCL(T) car Z n'influe pas sur le raisonnement booléen.

— Si Z déduit un conflit modulo théories à partir de m, T aurait pu le déduire à partir de M (car $m \subseteq M$).

Donc CDCL(T) aurait répondu UNSAT sur C. Par équi-satisfiabilité P est aussi UNSAT. □

3 Implémentation

Dans cette section, nous allons donner quelques détails sur l'implémentation de notre méthode pour l'intégration efficace d'un solveur SAT dans un solveur SMT. Le calcul de pertinence que nous avons présenté dans la figure 4.11 est une version très naïve qui nécessite le parcours de la formule Φ du problème à chaque appel de la fonction dudit calcul. Nous effectuons donc à chaque calcul de pertinence beaucoup de travail non nécessaire et redondant.

3.1 Calcul paresseux de la pertinence

Nous avons donc implémenté une version incrémentale qui permet de ne traiter que les formules pas encore parcourues. Comme dans la figure 4.11, notre nouvelle implémentation présentée sur la figure 4.12, nous ajoutons toutes les formules dont le proxy est vrai dans Γ à notre queue de travail. La première différence (ligne 6) consiste à supprimer ces formules de Φ. Cela permet de ne pas les traiter de nouveau lors du prochain appel au calcul de pertinence.

Comme précédemment, notre fonction *compute_relevancy* différencie trois types de nœuds dans le parcours de la formule. Les nœuds terminaux et les nœuds représentant des conjonctions et disjonctions n-aires. Pour chaque niveau de décision, on stocke les disjonctions, qui ne possèdent pas encore de formule assignée à vraie (ligne 17) rencontrée lors du parcours. Cela permet lors du prochain appel au calcul de pertinence de travailler sur ces disjonctions plutôt que de repartir du début et de la formule de base. Pour cela la fonction retourne maintenant en plus de l'ensemble des formules pertinentes, Φ qui sera sauvegardé comme l'ensemble des formules restant à traiter pour ce niveau de décision.

Lors d'un retour en arrière, on réutilise les disjonctions stockées dans Φ au niveau correspondant. Contrairement à la version précédente, la fonction

Input: Γ: SAT environment, Φ: set of formula in negative normal form
Output: *Relevants*: set of relevant formula, Φ

```
1  Function compute_relevancy(Γ, Φ)
2      Work ← ∅                                   // initialize working queue
3
4      forall formula f in Φ do
5          if proxy(f) ∈ Γ then
6              push(f, Work)
7              Φ ← Φ \ f
8      while Work is not empty do
9          f ← pop(Work)
10         add proxy(f) in Relevants
11         if f is a conjunction of f_i then
12             forall f_i do
13                 push(f_i, Work)
14         else
15             if f is a disjunction of f_i then
16                 if ∃ proxy(f_j) ∈ Γ then
17                     push(f_j, Work)
18                 else
19                     Φ ← Φ ∪ f
20     return Relevants,Φ
```

Figure 4.12 – Calcul incrémental des littéraux pertinants

```
Input: Φ: set of formulas in negative normal form, T: Theory environment
Output: Satisfiability status
1  Function solve (Φ, T)
2      Γ ← to_CNF(Φ)                          // initialize SAT environment
3
4      T ← ∅                                   // initialize theories environment
5
6      Δ ← ∅
7      while true do
8          (Γ, Φ, Δ, T, Error) ← propagate(Γ, Φ, Δ, T) // BCP modulo Theory
            with relevancy
9
10         if Error then
11             C ← explain_and_resolve_conflict(Γ, T)
12             if clause C is empty then
13                 return UNSAT
14             else
15                 Γ ← Γ ∪ C             // Conflict driven clause learning
16                 (Γ, Φ, Δ, T) ← backjump(Γ, Φ, T, C) // non chronological
                    backtracking
17         else if all Tableaux branches are closed then
18             ...
```

Figure 4.13 – Ajout de l'incrémentalité pour le calcul de pertinence. Φ est ici propagé et synchronisé à chaque niveau de décision

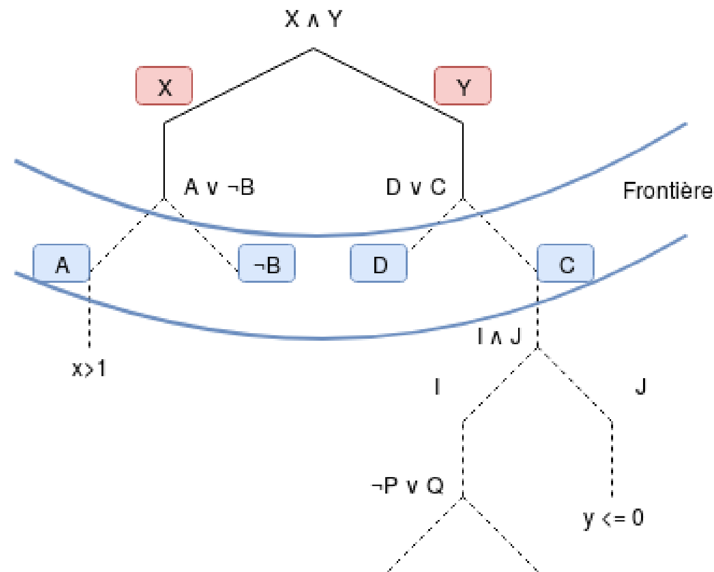

FIGURE 4.14 – Exemple de frontière lors du calcul de pertinence via la méthode des tableaux.

backjump de la figure 4.13 ne retourne plus l'ensemble des formules de base avec les instances du niveau de décision mais retourne l'ensemble des formules non résolues du niveau de décision. Elle retourne aussi l'ensemble Δ des littéraux pertinents de ce niveau de décision.

3.2 Calcul de pertinence par frontière

Bien que l'implémentation présentée précédemment offre de meilleurs résultats que la solution naïve, elle reste gourmande sur certains exemples. Une analyse du comportement du solveur nous a permis de mettre en évidence qu'une trop grande consommation du temps de calcul était due au calcul de pertinence. Plus particulièrement cette surconsommation était due au parcours des nœuds internes des disjonctions. En effet pour chaque niveau de décision on parcourt ces nœuds dans l'espoir qu'un d'entre eux soit vrai pour continuer le parcours de la formule. Il est possible qu'aucune décision et propagation n'ait influencée ces nœuds que nous parcourons tout de même.

Sur la figure 4.14 nous reprenons l'exemple de la figure 4.8 où plus aucune affectation n'est effectuée. Les deux disjonctions à surveiller sont $A \vee \neg B$ et $D \vee C$. Si une décision porte sur I cela n'aura pas d'incidence sur nos disjonctions ; nous allons pourtant les parcourir à la recherche d'un candidat (formule vraie). Seule une affectation sur $A, \neg B, C$ ou D pourra faire progresser le calcul de pertinence. Nous créons donc un ensemble que nous appelons Frontière contenant ces formules. C'est cet ensemble que nous devons

étudier pour éviter de parcourir l'ensemble des disjonctions.

Nous présentons dans la figure 4.15 une nouvelle implémentation permettant de supprimer ce problème de surconsommation en évitant de parcourir les nœuds non modifiés. Seuls les littéraux récemment ajoutés (au niveau de décision courant) dans l'environnement des littéraux assignés, la trail, et présents dans au moins une disjonction doivent être étudiés.

On stocke dans un dictionnaire l'ensemble des formules représentant des disjonctions n'ayant pas encore de sous-formule vraie. On lie chacune de ces formules à l'ensemble des littéraux qu'elles contiennent. Nous appelons ce dictionnaire $Frontier$. Nous stockons aussi, dans un dictionnaire, un ensemble de littéraux et les disjonctions dans lesquelles ils apparaissent, noté $Inv_Frontier$.

Lorsqu'un des littéraux(l) présent dans la trail des littéraux récemment assignés (au niveau de décision courant) Δ est aussi présent dans cet ensemble $Inv_frontier$, nous ajoutons la formule qu'il représente, fonction $formula$(ligne 5), à la queue de formules à traiter. Ensuite, pour toutes formules f contenant l, ces formules ayant un candidat, nous les supprimons du dictionnaire des autres candidats. Puis nous supprimons f de la frontière $Frontier$. Cela nous garantit que les formules contenant l ne seront pas traitées plusieurs fois, car la branche de l l'est déjà.

Les dernières modifications de l'algorithme concernent le cas où une disjonction, f, n'a pas de candidat (ligne 20). Dans ce cas on ajoute à $Frontier$ tous les littéraux contenus dans la disjonction f, et pour chacun de ces littéraux nous ajoutons f comme une disjonction à satisfaire. Enfin nous retournons ces deux dictionnaires qui comme Φ précédemment, seront synchronisés avec le niveau de décision courant. Cela permet, lors d'un retour en arrière, de travailler sur une frontière correspondant à l'état du parcours du calcul de pertinence du niveau de retour.

Les modifications apportées à la fonction *solve* sont présentées dans la figure 4.16. Fr et Inv_Fr représentent les dictionnaires $Frontier$ et $Inv_Frontier$. À l'initialisation nous devons calculer la frontière du problème de base. La fonction $compute_frontier$ (ligne 3) initialise nos deux structures. Les formules de Φ sont parcourues par méthode des tableaux. Les conjonctions et formules terminales sont ajoutées à Δ. Les disjonctions rencontrées sont quant à elles ajoutées dans la frontière. Cela nous permet de garantir que la frontière ne contient que des disjonctions. Cette fonction est aussi utilisée lors de l'ajout des instances (ligne 18).

Cette implémentation permet de n'effectuer que le nécessaire lors de chaque calcul de pertinence et donc de réduire son coût.

3.3 Gestion du modèle pour l'instanciation

Nous allons maintenant expliquer plus en détail le fonctionnement des heuristiques d'instanciation d'Alt-Ergo. Dans les algorithmes précédents,

```
Input: Γ: SAT environment, Frontier: map of formula to literals,
       Inv_Frontier: map of literals to disjunctions, Ω: set of literals assigned by the last BCP
Output: Relevants: set of relevant formula, Frontier, Inv_Frontier
1  Function compute_relevancy(Γ, Frontier, Inv_Frontier, Ω)
2    Work ← ∅                                   // initialize working queue
3
4    forall literals l in Ω do
                                   // For all literals recently assigned
5      if l ∈ Γ and l ∈ Inv_Frontier then
6        push (formula(l),Work)
7        forall formula f in Inv_Frontier(l) do
8          forall literals l_i in Frontier(f) do
9            remove f from Inv_frontier(l_i)
10         remove f from Frontier
11       remove l from Inv_Frontier
12   while Work is not empty do
13     f ← pop(Work)
14     add proxy(f) in Relevants
15     if f is a conjunction of f_i then
16       forall f_i do
17         push(f_i, Work)
18     else
19       if f is a disjunction of f_i then
20         if ∃ proxy(f_j) ∈ Γ then
21           push(f_j, Work)
22         else
23           forall f_i in f do
24             add (f, f_i) in Frontier
25             add (f_i, f) in Inv_Frontier
26   return Relevants,Frontier,Inv_Frontier
```

Figure 4.15 – Calcul des littéraux pertinents par gestion d'une frontière. Seule les littéraux récemment assignés par le solveur SAT sont étudié. Si ils sont dans la frontière nous les ajoutons comme pertinent et les explorons.

```
Input: Φ: set of formulas in negative normal form, T: Theory environment
Output: Satisfiability status
1  Function solve (Φ, T)
2      Γ ← to_CNF(Φ)                              // initialize SAT environment
3
4      T ← ∅                                      // initialize theories environment
5
6      (Δ, Fr, Inv_Fr) ← compute_frontier(Φ)
7      while true do
8          (Γ, Fr, Inv_Fr, Δ, T, Error) ← propagate(Γ, Fr, Inv_Fr, Δ, T)
           // BCP modulo Theory with relevancy
9          if Error then
10             C ← explain_and_resolve_conflict(Γ, T)
11             if clause C is empty then
12                 return UNSAT
13             else
14                 Γ ← Γ ∪ C                      // Conflict driven clause learning
15                 (Γ, Fr, Inv_Fr, Δ, T) ← backjump(Γ, Fr, Inv_Fr, T, C)
                   // non chronological backtracking
16         else
17             if all Tableaux branches are closed then
18                 I ← instantiate(Γ, Δ, T)
19                 if I ≠ ∅ then
20                     Γ ← Γ ∪ to_CNF(I)          // add new instances to Γ
21
22                     (I_Δ, I_Fr, I_Inv_Fr) ← compute_frontier(I)
23                     Δ ← Δ ∪ I_Δ
24                     Fr ← Fr ∪ I_Fr
25                     Inv_Fr ← Inv_Fr ∪ I_Inv_Fr
26                 else
27                     ...
```

FIGURE 4.16 – Ajout de l'initialisation de la frontière ainsi que l'ajout des nouvelles instances dans la frontière

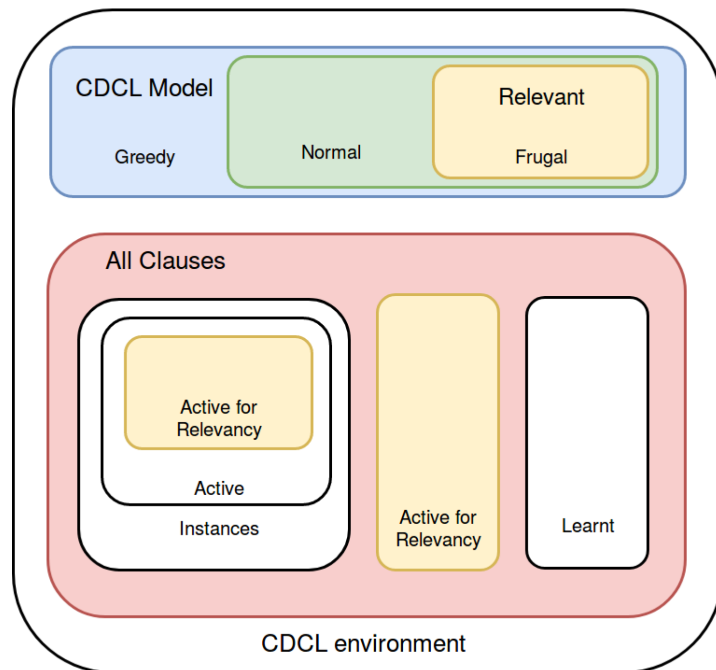

Figure 4.17 – Graphique représentant les différents ensembles de littéraux utilisés selon l'heuristique d'instanciation voulue, ainsi que les clauses utilisées pour le calcul de la pertinence.

nous avons expliqué que nous utilisions l'ensemble des littéraux pertinents pour l'instanciation. En réalité ceci n'est qu'une des trois heuristiques d'instanciation d'Alt-Ergo que nous allons maintenant présenter. La première, que nous appelons $frugal$, n'utilise que les littéraux pertinents pour le calcul d'instance. La seconde, *normal*, consiste à ajouter pour chaque littéral pertinent l, son graphe de dépendances à l'ensemble des littéraux utilisés pour l'instanciation M. Un littéral n'a pas de dépendance si il a été assigné par décision. Pour calculer le graphe de dépendances d'un littéral, nous ajoutons à M, les littéraux l_i de la clause ayant rendu le littéral l vrai. Puis nous effectuons la même opération pour chacune des clauses ayant rendu vrais ces littéraux l_i. Nous effectuons l'opération jusqu'à n'avoir plus que des décisions. Cette seconde heuristique permet donc d'ajouter des littéraux liés aux littéraux pertinents pour le calcul d'instance. La troisième heuristique, plus gourmande que nous appelons *greedy*, prend toutes les assignations de l'environnement du SAT CDCL. La figure 4.17 représente de manière graphique les différents modèles utilisés pour l'instanciation en fonction des heuristiques choisies.

Elle montre aussi comment nous gérons les clauses dans le SAT CDCL. Comme nous l'avons rappelé plus tôt, notre solveur SAT ne supprime ni les clauses apprises, ni les clauses correspondant aux instances. Dans l'environnement CDCL nous avons donc des clauses représentant le problème de base, des clauses apprises et des clauses représentant des instances. Seules les clauses représentant des formules présentes dans Φ ou $Frontier$ sont utilisées pour le calcul de pertinence. Le calcul de pertinence utilisant un système d'exploration de branche, certaines branches n'étant pas explorées, les clauses les représentant ne le sont pas non plus. C'est aussi le cas pour les clauses apprises qui ne sont pas dans le problème de base ainsi que les instances qui ne correspondent plus au niveau de décision courant.

4 Résultats expérimentaux

Après avoir présenté et formalisé notre combinaison optimisée d'un solveur SAT CDCL efficace avec un solveur de théories et un moteur d'instanciation, nous allons présenter les résultats de performances de cette optimisation. Nous utiliserons pour cela des bancs de tests issus de la preuve de programme et générés par l'outil Why3 [43]. Cette section sera une combinaison de différentes études. La première concernera la comparaison des différents solveurs SAT utilisés. La seconde concernera l'impact du calcul de pertinence des littéraux sur les différentes parties d'un solveur SMT. La troisième concernera des travaux sur l'heuristique de décision booléenne du solveur SAT. Enfin, la dernière étudiera l'impact des différents modèles pour l'instanciation présentés ci-avant en section 3.3.

	satml	satml + pertinence	historique	historique + satml
	`cdcl`	`cdcl_tableaux`	`tableaux`	`tableaux_cdcl`
BWARE–DAB	849(258s)	**860(47s)**	860(417s)	860(277s)
BWARE–RCS3	2228(742s)	**2234(725s)**	2233(685s)	2233(702s)
BWARE–p4	9195(2097s)	**9287(790s)**	9272(2279s)	9271(907s)
BWARE–p9	240(1104s)	**268(492s)**	252(342s)	264(769s)
EACSL	725(64s)	886(293s)	895(258s)	**896(244s)**
SPARK	13498(1769s)	**14089(2298s)**	14026(2757s)	14028(2710s)
WHY3	780(616s)	1399(1471s)	1442(1876s)	**1443(1936s)**
Total	27515(6652s)	**29023(6119s)**	28980(8617s)	28995(7549s)

Figure 4.18 – Résultats sur des fichiers issus de la preuve de programmes des différents solveurs d'Alt-Ergo : CDCL, cdcl avec pertinence(cdcl_tableaux), historique(tableaux) et historique assisté par un CDCL(tableaux_cdcl) pour son raisonnement booléen.

4.1 Comparaison des différents solveurs SAT

Le but de ces résultats expérimentaux étant de montrer l'impact qu'a le calcul de pertinence sur les performances globales d'un solveur SMT, nous avons testé les différents solveurs SAT d'Alt-Ergo :

- `tableaux` : solveur historique utilisant la méthode des tableaux
- `cdcl` : solveur CDCL proche de l'implémentation SatML
- `tableaux_cdcl` : solveur `tableaux` assisté d'un CDCL
- `cdcl_tableaux` : solveur CDCL utilisant le calcul de pertinence

Concernant la version du solveur CDCL avec calcul de pertinence, `cdcl_tableaux`, nous ne testerons que la version présentée en section 3.2 (ie. avec calcul de pertinence par frontière).

Les figures 4.18 et 4.19 montrent les résultats obtenus sur notre ensemble de benchmarks avec nos différents solveurs SAT. On peut voir que le SAT historique (tableaux) est bien meilleur que le CDCL non optimisé comme montré dans la section 2.3. On peut aussi remarquer que le solveur historique montre parfois de meilleures performances quand il est assisté d'un solveur CDCL. En particulier en termes de temps d'exécution avec un gain de l'ordre de 12%. Toutefois, le solveur montrant les meilleures performances est le solveur CDCL avec le calcul de pertinence. Il est en effet le plus performant en temps et en nombre de buts résolus. Nous remarquons cependant que sur l'ensemble de buts de `WHY3-VCs` il est moins efficace que les solveurs historique et historique assisté par un solveur CDCL.

En ce qui concerne les résultats uniques par solveur, le solveur CDCL prouve 30 buts qu'aucun autre solveur ne prouve, 111 buts pour le solveur CDCL assisté par méthode des tableaux. Les solveurs historiques avec et sans assistance résolvent 105 et 102 buts qui ne sont pas résolus par les solveurs `cdcl` et `cdcl_tableaux`. Les solveurs `cdcl_tableaux` et `tableaux` utilisant le même solveur de théories et moteur d'instanciation nous pensons

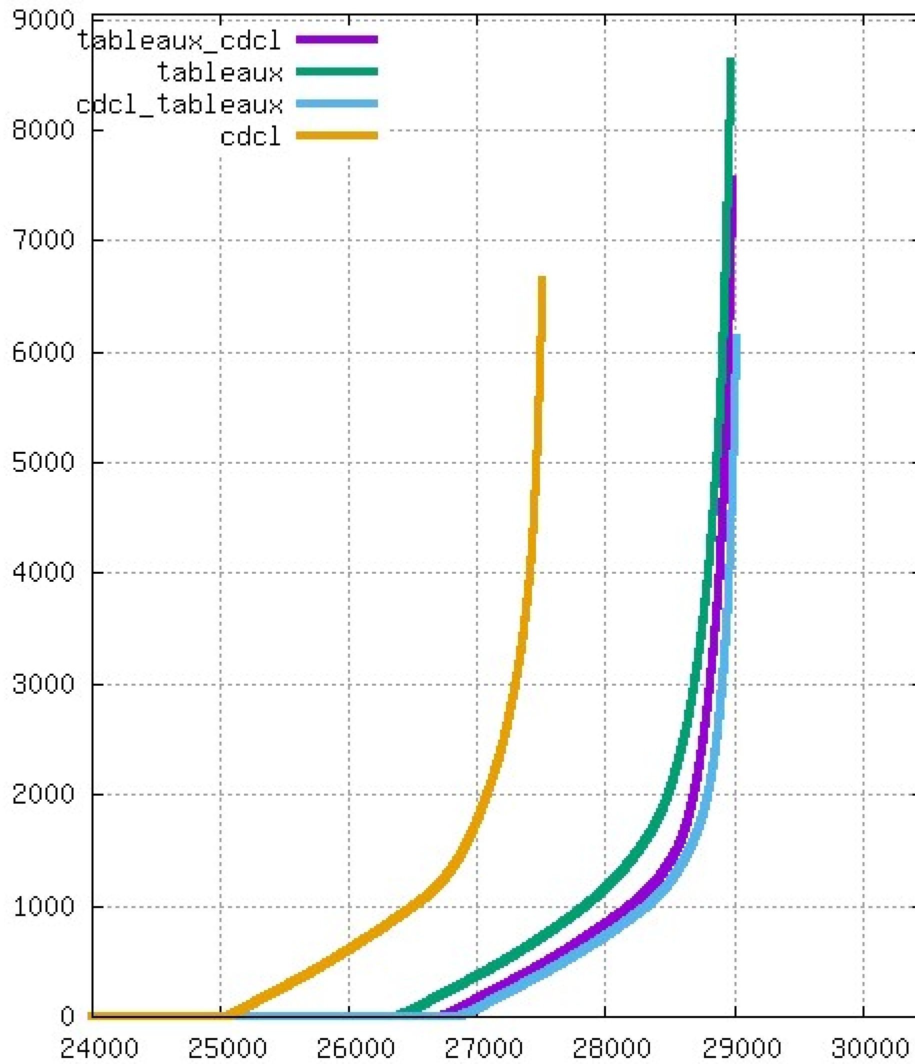

Figure 4.19 – Graphique du temps de résolution des solveurs SAT d'Alt-Ergo en fonction du nombre de buts résolus sur des fichiers issus de la preuve de programme.

que cette différence, dans les résultats uniques, est due au solveur SAT. La manière de traiter la formule d'entrée à prouver étant très différente pour nos deux solveurs, cela peut influer sur les performances. Un problème peut être très rapidement évalué à UNSAT pour le solveur historique tandis qu'il faut plusieurs décisions et propagation du solveur CDCL pour arriver à la même conclusion, et vice versa.

Les figures 4.20 et 4.21 montrent ces différences sous forme de nuage de points. Nous utilisons une échelle logarithmique nous permettant d'afficher plus clairement les différences de temps d'exécution sur les exemples UNSAT. Une ligne verticale et une ligne horizontale à 60 secondes représentent la limite de temps. Les points sur ces lignes correspondent donc à des dépassements de la limite de temps imposée (Timeout) et ceux la dépassant représentent une erreur ou réponse différente de UNSAT (Unknown).

Dans le premier graphique, de la figure 4.20, nous observons que les solveurs `cdcl` et `tableaux` ont des résultats qui diffèrent beaucoup en temps de résolution. Le nombre de points sur la barre verticale à 60s et à coté (Unknown) nous indique non seulement que le solveur `cdcl` est moins efficace en temps que `tableaux` mais aussi en termes de nombre de buts UNSAT résolus.

Le deuxième graphique présente les différences entre le solveur historique avec et sans assistance d'un solveur CDCL. Les résultats de ces deux versions sont très similaires à l'exception de quelques buts qui profitent de l'assistance du solveur CDCL.

Sur le premier graphique de la figure 4.21, nous observons que notre solveur CDCL optimisé nous permet de réduire le temps d'exécution et de résoudre plus de buts au vu des deux lignes verticales formées par les fichiers faisant des timeouts ou Unknown avec le solveur CDCL non optimisé.

Dans le second graphique de cette même figure, nous comparons le solveur historique au solveur CDCL optimisé. Là encore, nous pouvons remarquer un nombre supérieur de points au-dessus de la ligne médiane montrant les gains de temps du solveur CDCL optimisé.

4.2 Impact du calcul de pertinence des littéraux

Dans cette section, nous allons étudier l'impact de différents paramètres liés au calcul de pertinence des littéraux. Premièrement, nous allons utiliser ce calcul uniquement sur les instanciations, ou uniquement sur les théories. Cela nous permettra de connaître l'impact individuel de notre optimisation sur ces deux parties d'un solveur SMT. Nous allons ensuite présenter les résultats que nous avons obtenus lorsque l'on a essayé de guider l'heuristique VSIDS [55] en choisissant en priorité des littéraux qui sont dans notre frontière. Cela nous permet de décider sur plus de littéraux pertinents et de nous rapprocher de l'heuristique de décision du solveur SAT historique. Enfin nous testerons les heuristiques d'Alt-Ergo concernant le modèle envoyé au moteur

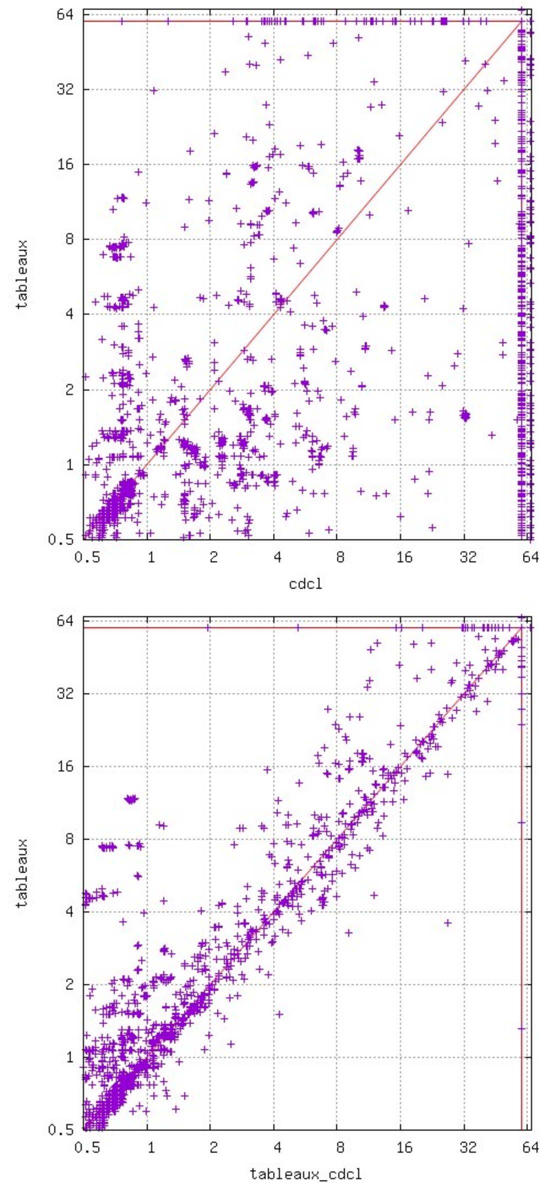

Figure 4.20 – Comparaison des différents solveurs SAT d'Alt-Ergo. La première figure compare les temps d'exécution du solveur CDCL avec le solveur historique d'Alt-Ergo. La comparaison présenté sur le deuxième graphique concernent le solveur historique avec et sans assistance d'un CDCL.

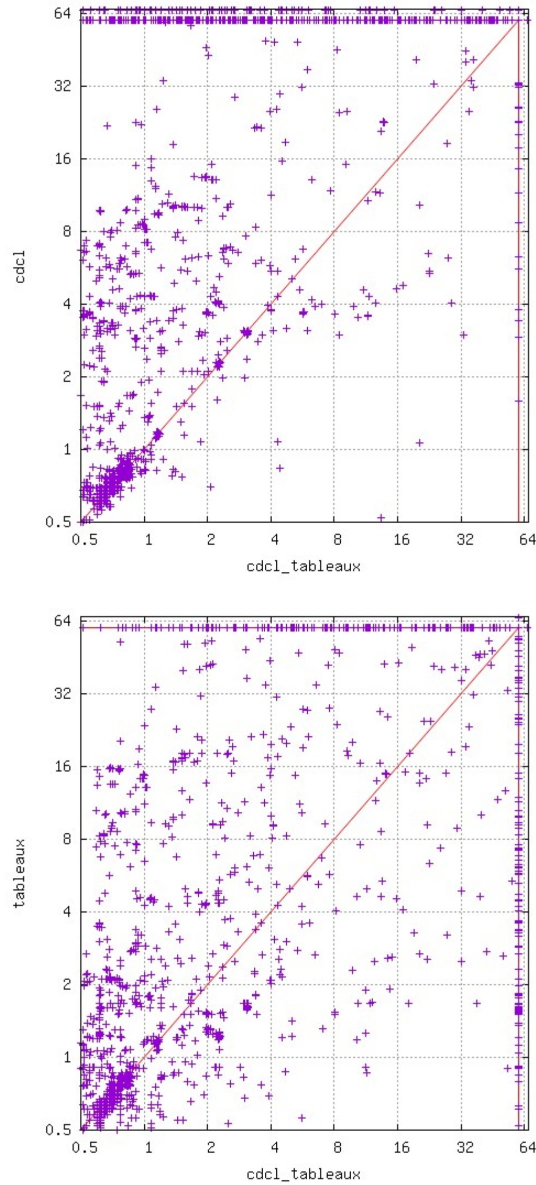

Figure 4.21 – Comparaison des différents solveurs SAT d'Alt-Ergo. Le premier graphique compare les deux solveurs SAT avec et sans le calcul de pertinence. Le deuxième graphique compare le solveur historique avec satml équipé du calcul de pertinence.

	`cdcl`	`cdcl_tableaux`	`tab_th`	`tab_inst`
BWARE–DAB	849(258s)	**860(47s)**	850(19s)	858(407s)
BWARE–RCS3	2228(742s)	**2234(725s)**	2226(721s)	2227(682s)
BWARE–p4	9195(2097s)	**9287(790s)**	9195(555s)	9267(1348s)
BWARE–p9	240(1104s)	**268(492s)**	251(565s)	260(520s)
EACSL	725(64s)	886(293s)	705(288s)	**891(238s)**
SPARK	13498(1769s)	**14089(2298s)**	13389(1339s)	14051(3580s)
WHY3	780(616s)	**1399(1471s)**	747(471s)	1329(1367s)
Total	27515(6652s)	**29023(6119s)**	27363(3960s)	28883(8145s)

Figure 4.22 – Résultats sur des fichiers issus de la preuve de programmes des solveurs SAT d'Alt-Ergo : sans le calcul de pertinence(`cdcl`), avec ce calcul pour la théorie et l'instanciation(`cdcl_tableaux`), uniquement pour la théorie (`tab_th`) et uniquement pour l'instanciation(`tab_inst`).

d'instanciation.

Pertinence pour les instances ou les théories uniquement

La figure 4.22 montre l'impact du calcul de pertinence des littéraux sur le même ensemble de benchs que précédemment. Nous avons montré l'impact de ce calcul si nous l'utilisons sur le solveur de théories et sur le moteur d'instanciation, mais nous n'avons pas étudié l'impact individuel sur ces parties d'un solveur SMT. En étudiant les travaux de Z3 [33] et CVC4 [46] il nous est apparu que ces résultats manquaient dans leurs études. Il semblerait qu'ils aient conclu qu'une technique de réduction de modèle ne soit pas toujours rentable pour les théories efficaces et qu'il vaut mieux privilégier ce calcul pour la phase d'instanciation et certaines théories moins efficaces. Nous avons donc décidé de tester deux possibilités, l'une avec le calcul de pertinence uniquement activé pour le moteur de théorie et l'autre pour le moteur d'instanciation.

Nous comparons dans la figure 4.22 et la figure 4.23 ces deux possibilités avec les deux configurations d'Alt-Ergo équipées d'un solveur sat CDCL présenté précédemment. Ces résultats montrent une baisse du nombre de buts résolus si les deux parties du solveur SMT n'utilisent pas le calcul de pertinence. Cette baisse est plus importante pour la configuration n'utilisant le calcul de pertinence que pour les théories. Elle donne d'ailleurs de moins bons résultats (sauf pour `p9`) que la configuration sans calcul de pertinence (`cdcl`) en nombre de buts résolus. Du fait que ces deux configurations travaillent sur le modèle booléen sans réduction pour l'instanciation, les différences sont dues au solveur de théories. Il est en effet possible qu'en envoyant moins de contexte (littéraux) au solveur de théories, celui-ci ne donne pas autant d'informations en retour qu'avec l'intégralité du modèle booléen. Il s'avère pourtant que lorsqu'on utilise la pertinence des littéraux sur les théories et les instances, nous avons de meilleurs résultats que pour l'instance

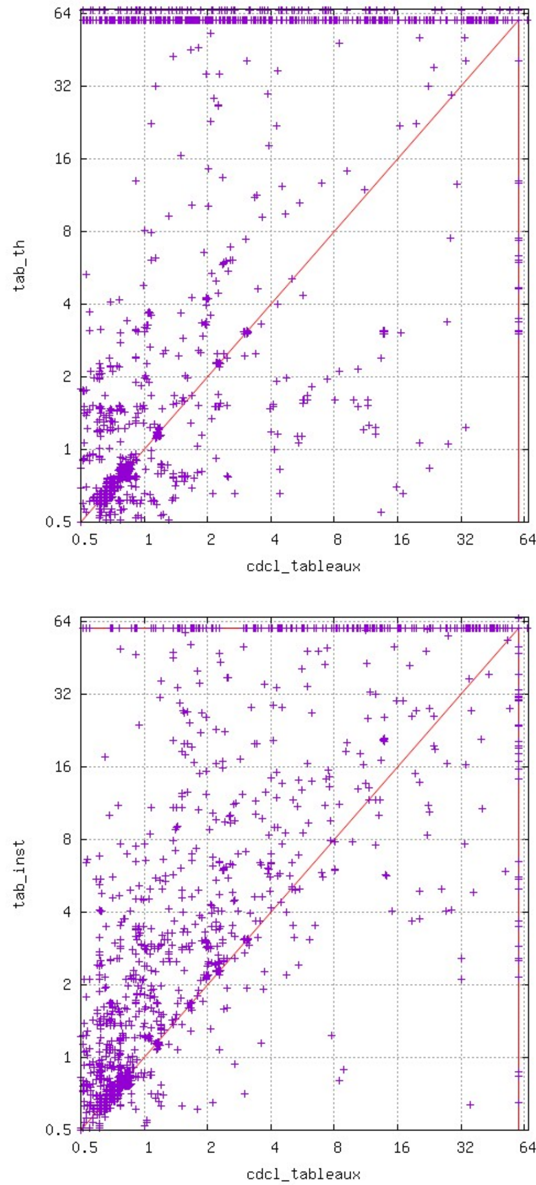

Figure 4.23 – Comparaison de l'impact du calcul de la pertinence uniquement sur l'instanciation et uniquement sur le solveur de théorie par rapport au solveur utilisant ce calcul de pertinence pour l'instanciation et les théories.

	`cdcl`	`cdcl_tableaux`	`vsids_guided`
BWARE–DAB	849(258s)	**860(47s)**	860(49s)
BWARE–RCS3	2228(742s)	2234(725s)	**2235(735s)**
BWARE–p4	9195(2097s)	9287(790s)	**9297(1358s)**
BWARE–p9	240(1104s)	268(492s)	**268(408s)**
EACSL	725(64s)	**886(293s)**	885(363s)
SPARK	13498(1769s)	**14089(2298s)**	13713(4653s)
WHY3	780(616s)	**1399(1471s)**	1292(1696s)
Total	27515(6652s)	**29023(6119s)**	28550(9265s)

Figure 4.24 – Résultats sur des fichiers issus de la preuve de programmes des solveurs SAT d'Alt-Ergo : avec l'heuristique VSIDS guidée par le calcul de pertinence.

seule (`tab_inst`). Cette dernière configuration, bien que moins performante que `cdcl_tableaux` propose de meilleurs résultats que les deux autres. Les plus gros gains sont donc dus au calcul de pertinence pour l'instanciation qui peut encore être améliorée lorsque ce calcul est utilisé aussi pour réduire le modèle envoyé aux théories. Ces résultats mériteraient d'être approfondis sur des ensembles de problèmes triés par théorie utilisée. Nous pourrions ainsi déceler quelles sont les théories qui bénéficient le plus du calcul de pertinence.

Heuristique de décision VSIDS

Le solveur historique fonctionnant par méthode des tableaux, ces décisions y sont étroitement liées. Nous avons décidé d'implémenter une modification de la fonction de décision sur les littéraux dans le solveur sat CDCL d'Alt-Ergo pour obtenir une heuristique guidant les décisions grâce à la méthode des tableaux. Pour cela, nous allons guider l'heuristique VSIDS du solveur SAT CDCL. Lorsqu'un littéral est choisi par l'heuristique nous vérifions qu'il est présent dans la frontière. Si il est présent dans la frontière il est candidat pour être pertinent, et la décision est faite sur ce littéral. Si ce n'est pas le cas on demande à VSIDS de nous fournir un autre candidat pour la décision. Nous considérons que si aucun littéral n'est dans la frontière nous passons alors à une phase d'instanciation. Cette expérience est très succincte et peu aboutie, et n'a pour but que de nous éclairer sur l'intérêt potentiel que pourrait avoir un changement d'heuristique de décision.

La figure 4.24 nous présente les résultats de cette expérience. Bien que les résultats soient globalement moins bons avec la modification de l'heuristique VSIDS on remarque quelques gains sur les buts issus de BWARE. Sur la figure 4.25 nous observons ces quelques buts résolus plus rapidement avec l'heuristique VSIDS modifiée. Ces buts nous montrent qu'il y a un potentiel intérêt à guider nos décisions grâce à la pertinence des littéraux. Avec plus

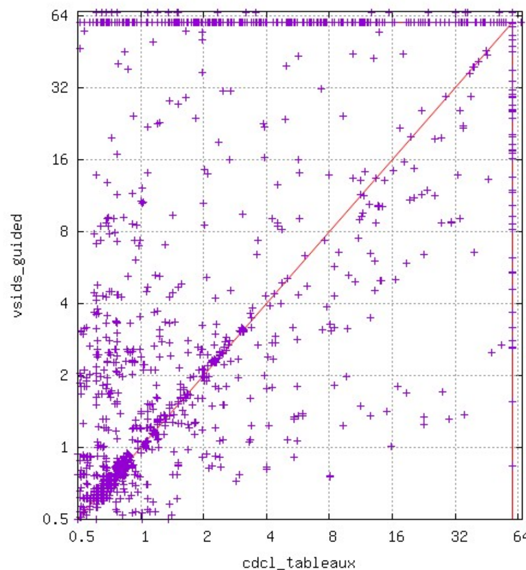

Figure 4.25 – Comparaison de l'impact du guidage de l'heuristique VSIDS avec le calcul de la pertinence.

de travail nous pourrions remplacer l'heuristique VSIDS par une heuristique utilisant notre frontière de pertinence. Chaque littéral serait ainsi pondéré par le nombre de disjonctions présentes dans la frontière dans lequel il est lui aussi présent. Cette heuristique choisirait ainsi en priorité les littéraux dont l'assignation ferait avancer notre frontière et donc notre raisonnement par méthode de tableaux.

Modèles CDCL pour l'instanciation

La dernière heuristique que nous avons testée est une heuristique concernant l'ensemble des littéraux envoyés au moteur d'instanciation. Comme nous l'avons présenté en section 3.3, Alt-Ergo peut utiliser plusieurs ensembles de littéraux pour l'instanciation. L'heuristique par défaut d'Alt-Ergo utilisée pour la configuration `cdcl_tableaux`, consiste en une combinaison des trois heuristiques précédemment présentées. Tout d'abord les instanciations sont effectuées sur l'ensemble des littéraux de l'heuristique $frugal$. Si aucune nouvelle instance n'est trouvée, on effectue alors une nouvelle passe d'instance en utilisant l'ensemble des littéraux de l'heuristique *normal*. Là encore, si aucune nouvelle instance n'est trouvée, on utilise alors les littéraux de *greedy*. Cette heuristique nous permet de travailler dans un premier temps sur un ensemble réduit de littéraux pour ensuite l'agrandir si cela n'est pas suffisant pour trouver des instances.

	`cdcl_tableaux`	`frugal`	`normal`	`greedy`
BWARE–DAB	**860 (47s)**	860 (49s)	837 (395s)	667 (1401s)
BWARE–RCS3	2234 (725s)	2233 (685s)	**2236 (801s)**	2137 (2490s)
BWARE–p4	9287 (790s)	**9291 (1070s)**	9031 (2484s)	7904 (26932s)
BWARE–p9	**268 (492s)**	268 (501s)	255 (743s)	102 (397s)
EACSL	886 (293s)	886 (381s)	**895 (630s)**	672 (3590s)
SPARK	**14089 (2298s)**	14027 (1900s)	14046 (2329s)	13216 (8403s)
WHY3	**1399 (1471s)**	1302 (858s)	1340 (1575s)	948 (1779s)
Total	**29023 (6119s)**	28867 (5447s)	28640 (8960s)	25646 (44995s)

Figure 4.26 – Résultats sur des fichiers issus de la preuve de programmes des solveurs SAT d'Alt-Ergo : avec différentes heuristiques concernant l'ensemble des littéraux envoyés au moteur d'instanciation.

La figure 4.26 nous montre les résultats obtenus par cette heuristique ainsi que les heuristiques `frugal`, `normal` et `greedy` testées individuellement. Ces heuristiques impactent aussi le nombre de déclencheurs (triggers) utilisés pour le calcul des instances de chaque terme quantifié. L'heuristique `normal` utilise plus de déclencheurs que `frugal` et `greedy` en utilise plus que `normal`. On remarque sur cette figure que notre heuristique combinant les trois autres, offre les meilleurs résultats globaux. Les heuristiques `frugal` et `normal` offrent de meilleurs résultats que `cdcl_tableaux` sur certains ensembles de problèmes. Les bons résultats de `frugal` par rapport à `greedy` nous permettent de démontrer que travailler sur un ensemble réduit de littéraux pour l'instanciation est bénéfique.

La figure 4.27 présente trois graphiques où les trois heuristiques `frugal`, `normal` et `greedy` sont comparées à l'heuristique par défaut d'Alt-Ergo. On remarque sur le premier graphique que les résultats de l'heuristique `frugal` sont très proches de ceux de l'heuristique par défaut. On remarque aussi que sur un bon nombre de buts, l'heuristique `frugal` répond Unknown (barre horizontal à 64s). Cela signifie que dans la majorité des cas où la combinaison d'heuristique est activée, l'heuristique `frugal` permet de trouver des instances sans avoir à travailler sur un ensemble de littéraux plus important. Le deuxième graphique confirme cela, nous pouvons voir que l'heuristique `normal` est plus lente que la combinaison d'heuristique. Ce résultat est aussi confirmé par le troisième graphique ou l'on peut remarquer que `greedy` est bien moins efficace que la combinaison d'heuristiques.

5 Conclusion

Nous avons présenté nos travaux concernant l'optimisation d'un solveur SAT dans un contexte de solveur SMT. Nous avons expliqué le cheminement de pensée nous ayant amené à élaborer une solution optimisant la combinaison d'un solveur SAT CDCL performant au sein d'un solveur SMT. Cette

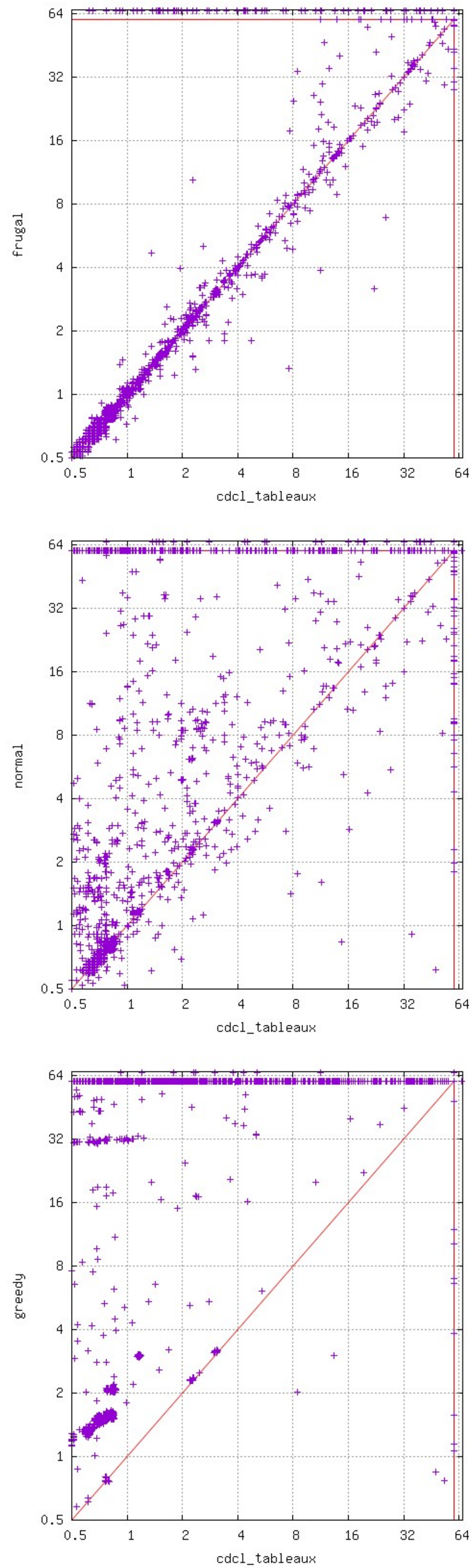

Figure 4.27 – Comparaison de l'impact des heuristiques d'instanciation comparées à la l'heuristique combinant les trois heuristiques.

Natif	`Alt-Ergo_2.3`	`CVC4_1.6/1.5`	`Vampire_4.3`	`Z3_4.8.5`
BWARE–DAB	**860(47s)**	813(1295s)	401(3040s)	704(128s)
BWARE–RCS3	**2234(725s)**	2040(2845s)	1075(455s)	2191(229s)
BWARE–p4	**9287(790s)**	9109(9754s)	3043(24878s)	7962(1170s)
BWARE–p9	**268(492s)**	187(1018s)	55(1426s)	238(464s)
EACSL	886(293s)	**895(388s)**	434(4999s)	744(619s)
SPARK	14089(2298s)	14267(558s)	12228(16725s)	**15150(1545s)**
WHY3	**1399(1471s)**	1321(632s)	623(6301s)	1196(1429s)
Total	**29023(6119s)**	28632(16492s)	17859(57827s)	28185(5586s)

FIGURE 4.28 – Résultats du solveur Alt-Ergo dans sa version 2.3 (avec un solveur SAT CDCL équipé du calcul de pertinence) comparé au solveur de l'état de l'art sur leur version les plus récentes sur des fichiers dans les formats natifs de ces solveurs.

solution a été par la suite formalisée sous forme d'un algorithme, que nous avons raffiné pour des questions de performances. Nous avons ensuite étudié l'impact et les performances de notre solution via différentes configurations nous permettant de conclure que notre solution est performante et permet une meilleure combinaison d'un solveur SAT efficace dans un solveur SMT. Une heuristique concernant les décisions sur les littéraux a été étudiée et possède un potentiel qu'il nous faudrait approfondir.

Au vu des résultats de notre optimisation, nous avons souhaité nous comparer aux solveurs de la communauté pouvant traiter des problèmes contenant des quantificateurs très présents dans notre ensemble de benchs. L'outil Why3 permettant de générer des fichiers au format natif d'Alt-Ergo, que nous avons précédemment utilisé, et aussi au format natif pour les solveurs Z3 [32], CVC4 [10] et Vampire [68], nous avons pu comparer Alt-Ergo à ces solveurs en générant des benchs au format natif de chaque solveur basé sur le même ensemble de problèmes.

La figure 4.28 nous montre les résultats de ces différents solveurs. Nous utilisons les dernières versions officielles pour chacun de ces solveurs. Nous avons cependant dû lancer la version antérieure de CVC4 sur le benchs SPARK. En effet ces problèmes ont été générés pour la version 1.5 du solveur et ne sont plus compatibles avec la version 1.6 dû à un changement de syntaxe. Alt-Ergo est le solveur qui résout le plus de buts bien qu'il soit devancé sur la catégories SPARK par Z3 et par CVC4 pour la catégorie EACSL. Bien que ces résultats soient issus des mêmes problèmes en entrée de l'outil WHY3, les buts générés peuvent ne pas être équitables dû aux différentes modifications apportées par l'outil pour satisfaire chaque solveur. Pour obtenir des résultats les plus équitables possibles il faudrait donc générer des buts identiques et compréhensibles par tous ces solveurs. Malheureusement nous arrivons à un défaut d'Alt-Ergo, il ne supporte pas le standard SMT, supporté par les autres solveurs. Nous avons ainsi décidé d'ajouter le support

du standard de la SMT-LIB 2 au solveur Alt-Ergo.

Deuxième partie

SMT-LIB et SMT-COMP

Chapitre 5

Extension du standard SMT-LIB2 avec du polymorphisme

Ce chapitre a pour but de présenter les travaux effectués concernant le support du standard SMT-LIB2 [6, 9, 7, 5, 8] par Alt-Ergo. Comme expliqué auparavant en section 2 du chapitre 2, Alt-Ergo ne supporte pas le standard de la communauté et il utilise un langage d'entrée proche du langage de l'outil Why [44] [14]. Une des raisons pour laquelle Alt-Ergo ne supporte pas le standard est le manque de support pour le polymorphisme de celui-ci que nous demandons depuis des années. Le besoin pour Alt-Ergo de supporter à la fois le standard de la communauté SMT et une syntaxe polymorphe [12] ainsi que les récents efforts de la communauté pour supporter une forme de polymorphisme nous ont amené à proposer une extension conservatrice du standard SMT-LIB 2 avec un système de type polymorphe.

Nous présenterons dans ce chapitre les travaux effectués sur cette extension, sa syntaxe et son typage en section 1. Nous présenterons ensuite dans la section 2 les détails d'implémentation ainsi que les difficultés rencontrées pour ajouter ce langage à Alt-Ergo. Concernant les résultats expérimentaux de la section 3 nous présenterons tout d'abord notre méthodologie de création de bancs de tests nous ayant permis de comparer plusieurs points significatifs tels que l'impact du polymorphisme. Nous conclurons sur une comparaison équitable d'Alt-Ergo par rapport aux autres solveurs performants de l'état de l'art.

```
(declare-datatype Color ((red) (green) (blue)))

(declare-datatype List (par (A) ((nil) (cons (head A) (tail (List A))))))

(declare-datatypes ((X 1)(Y 2))
 ((par (A) ((cons_x (dest_x (Y A A)))))
  (par (A B) ((cons_y (dest_y (Y (X A) (X B))))))))

(declare-datatypes
 ((Option 1)(Tree 1))
 ((par (A) ((none) (some (Option A))))
  (par (B) ((leaf) (node (left (Tree B)) (elem B) (right (Tree B)))))))

(check-sat)
```

FIGURE 5.1 – Liste de commandes déclarant des types de données algébrique au format SMT-LIB 2.

1 Extension polymorphe

1.1 Syntaxe

Dans cette section, nous allons présenter notre proposition d'extension conservative du standard SMT-LIB 2 avec polymorphisme. Les travaux préliminaires dans ce sens ont été initiés dès 2014. Cela consistait à fournir un parser et un typechecker pour SMT-LIB 2 étendu avec du polymorphisme. Bonichon et al. ont aussi proposé une extension polymorphe au standard [16] présente de manière ad-hoc dans le solveur VeriT. Durant cette thèse, nous avons repris nos anciens travaux et avons considéré leur intégration dans Alt-Ergo. Des modifications ont aussi été apportées au langage lors de sa dernière mise à jour. En effet la version 2.6 [7] du standard apporte de nouvelles fonctions de déclarations de types permettant la création de types énumérés ou de types de données algébriques, comme les listes. Avec ces additions la communauté a décidé d'ajouter la syntaxe `(par (⟨symbol⟩))` permettant d'ajouter un polymorphisme ad-hoc lors des déclarations de types algébriques. Par exemple dans la figure 5.1, la première ligne présente la syntaxe permettant de déclarer un type `Color` possédant trois constructeurs. La deuxième montre, quant à elle, la syntaxe permettant de déclarer une liste de données récursives polymorphes. Ici le `A` est équivalent à la variable de type `'a` de OCaml. Enfin, la troisième ligne montre une déclaration de type mutuellement récursif .

Dans le standard, il existait déjà depuis le passage de la version 1.2 à la version 2.0 une forme de polymorphisme pour les fonctions au sein des déclarations de théories. L'exemple 5.2 montre une partie de la déclaration de la théorie Core. Cette théorie a pour but de définir le type `Bool` et ses fonctions associées. Les trois dernières fonctions sont paramétrées par une variable de type `A` avec la syntaxe suivante : `(par (⟨symbol⟩)⟨function declaration⟩)`

```
(theory Core
...
 :sorts ((Bool 0))
 :funs ((true Bool)
        (false Bool)
        (not Bool Bool)
        (=> Bool Bool Bool :right-assoc)
        (and Bool Bool Bool :left-assoc)
        (or Bool Bool Bool :left-assoc)
        (xor Bool Bool Bool :left-assoc)
        (par (A) (= A A Bool :chainable))
        (par (A) (distinct A A Bool :pairwise))
        (par (A) (ite Bool A A A))
       )
...
)
```

Figure 5.2 – Exemple de déclaration de théories au format SMT-LIB 2. Théorie comprenant des fonctions polymorphes.

La communauté utilise la notation `(par (⟨symbol⟩)⟨function declaration⟩)` pour exprimer la quantification universelle prenexe implicite sur des variables de type polymorphe depuis la version 2.0 [9] de la SMT-LIB 2. Cette notation n'a été introduite dans les fichiers à prouver (scripts) que depuis la version 2.6 apparue il y a un an environ. Cette notation se limite maintenant à la déclaration de types algébriques polymorphes. Pour rester conservatif notre extension de syntaxe se base donc sur les travaux de la communauté en utilisant cette notation du standard pour déclarer des variables de types polymorphes au sein des commandes.

Notre extension de syntaxe est présentée sur les exemples de la figure 5.3. Elle montre les différentes commandes dont nous avons modifié la syntaxe pour prendre en charge l'ajout du polymorphisme prenexe à la ML. La syntaxe devant être conservative nous présentons dans la figure 5.4 la grammaire correspondante permettant d'obtenir à la fois notre syntaxe polymorphe et de garder la syntaxe du standard. Nous avons essayé de rester le plus proche possible de la grammaire du standard. Chacune des déclarations ou définitions des commandes peut être paramétrée par des variables de type polymorphes.

Chaque commande dans laquelle un ⟨`sort`⟩ peut apparaître peut maintenant être annotée avec un `(par (⟨symbol⟩)..)`. Nous avons choisi une syntaxe avec quantification des variables de types explicites comme cela est fait dans la version 2.6 de SMT-LIB 2. Nous pensons que cela facilite la lecture et l'expressivité du langage pour les néophytes du polymorphisme. La commande `assert` peut elle aussi être paramétrée par une variable de type quantifié du fait que l'on peut trouver des types dans les quantificateurs universels comme

```
(declare-const y (par (A) A))
(declare-fun f (par (B C) (B C Int) B))

(assert (par (A) (forall ((x A)(y A)) (= x y))))

(define-fun g (par (A B) ((x A)(y B)) B) (f x y 3))
(define-fun-rec h (par (C) ((c C)) C) (h c))

(define-funs-rec
  ((v (par (X) ((a X)(b Int)) Int))
   (w (par (Y) ((c Int)(d Y)) Int)))
    ((w b a)
     (v d c)))
```

Figure 5.3 – Exemple de commandes au format SMT-LIB 2 étendues avec notre extension de syntaxe polymorphe PSMT2.

```
⟨par_dec⟩       ::= par (⟨symbol⟩+)

⟨const_dec⟩     ::= ⟨sort⟩
                 |  (⟨par_dec⟩ ⟨sort⟩)

⟨fun_dec⟩       ::= (⟨sort⟩*) ⟨sort⟩
                 |  (⟨par_dec⟩ (⟨sort⟩*) ⟨sort⟩)

⟨fun_def⟩       ::= (⟨sorted_var⟩*) ⟨sort⟩
                 |  (⟨par_dec⟩ (⟨sorted_var⟩*) ⟨sort⟩)

⟨fun_def_term⟩  ::= (⟨sorted_var⟩*) ⟨sort⟩ ⟨term⟩
                 |  (⟨par_dec⟩ (⟨sorted_var⟩*) ⟨sort⟩ ⟨term⟩)

⟨assert_dec⟩    ::= ⟨term⟩
                 |  (⟨par_dec⟩ ⟨term⟩)


⟨command⟩       ::= (declare-const ⟨symbol⟩ ⟨const_dec⟩)
                 |  (declare-fun ⟨symbol⟩ ⟨fun_dec⟩)
                 |  (define-fun ⟨symbol⟩ ⟨fun_def_term⟩)
                 |  (define-fun-rec ⟨symbol⟩ ⟨fun_def_term⟩)
                 |  (define-funs-rec ((⟨symbol⟩ ⟨fun_def⟩)^(n+1)) (⟨term⟩^(n+1)))
                 |  (assert ⟨assert_dec⟩)
```

Figure 5.4 – Grammaire conservative de l'extension polymorphe PSMT2 au standard SMT-LIB 2 permettant de paramêtrer certaines commandes avec des variables de type polymorphes.

```
(declare-sort Pair 2)
(define-sort I () Int)
(define-sort PII () (Pair I I))
(define-sort PA (A) (Pair A A))

(assert (forall ((x (Pair Int Int))(y PII)(z (PA Int))) (= x y z)))
(check-sat)
```

FIGURE 5.5 – Exemple d'utilisation de la commande define-sort utilisant un polymorphisme ad-hoc pour la définition d'alias de type.

dans l'exemple 5.3.

Par souci de compatibilité avec le standard, nous avons choisi de laisser inchangée la commande `define-sort` du standard qui permet de définir un alias de type. Cette notation permet une sorte de polymorphisme qui n'est pas explicite. En effet, comme le montre l'exemple 5.5, nous pouvons ainsi créer un alias pour des types définis comme `Int` ou des types polymorphes déclarés implicitement comme `A`. Nous aurions cependant pu ajouter un `define-sort` avec la notation `(par (`⟨`symbol`⟩`)`⟨...⟩`)` tout en gardant l'ancienne construction.

1.2 Typage

Nous avons présenté notre extension de syntaxe de SMT-LIB 2 avec polymorphisme en essayant de respecter au mieux les travaux déjà effectués par la communauté. Il nous faut maintenant en présenter le typage. Par souci de performance et de simplicité d'implémentation, nous avons choisi d'utiliser de l'unification destructive plutôt que de la comparaison de type. Le principal avantage est de ne pas avoir à recréer d'arbre de syntaxe abstraite lors du typage (AST). Le principal défaut de ce choix est le manque de possibilités de backtracking permettant d'avoir des messages d'erreur plus clairs ce qui peut être contraignant pour les erreurs de type.

Notre algorithme de typage fut brièvement présenté dans l'article [24]. Cet algorithme par unification destructive est basé sur des travaux de typage présentés par B. Pierce dans Types and Programming Languages [60], ces travaux sont quant à eux basés sur ceux de Hindley [47] Milner [54] et Robinson [63].

Typage de l'environnement

Nous allons dès à présent présenter l'algorithme de typage utilisés sous forme de règles de typage. Ces règles se lisent de bas en haut. Notre environnement de typage est représenté par le couple $\langle \Gamma \mid \Delta \rangle$. Γ est un $\mathsf{Dictionnaire}$ de symboles de types vers des arités. Elle comprend aussi bien des constructeurs

$$\frac{s \notin \Gamma \qquad n \geq 0}{\langle\, \Gamma \,|\, \Delta \,\rangle \xrightarrow[\texttt{(declare-sort s n)}]{} \langle\, \Gamma \cup (s, n) \,|\, \Delta \,\rangle} \; \textit{déclaration d'un sort}$$

$$\frac{\Gamma \vdash u_i \qquad \Gamma(s) = n}{\Gamma \vdash (s\ u_1 \cdots u_n)} \; \textit{typage d'un type}$$

$$\frac{}{\Gamma \vdash \alpha} \; \textit{typage d'une variable d'unification}$$

FIGURE 5.6 – Règles de typage des déclarations et utilisation de sorts.

de sorts que des variables de types locales. Δ est un Dictionnaire de symboles de fonctions et de variables vers un tuple $(par^*, type^n, type)$. Par exemple la fonction `g` de l'exemple 5.3 sera représentée par le tuple $([A; B], [A; B], B)$. Le premier argument correspond aux variables de types, le deuxième correspond aux types des paramètres. Le dernier argument correspond au type de retour du *symbol*. On note $\Gamma \vdash \tau$ un type τ bien formé dans Γ. On écrit $\langle\, \Gamma \,|\, \Delta \,\rangle \underset{c}{\rightarrow} \langle\, \Gamma' \,|\, \Delta' \,\rangle$ l'effet qu'a une commande de la SMT-LIB 2 étendue c sur notre environnement en plus de l'unification destructive.

Pour typer un fichier respectant le standard SMT-LIB 2 étendu, la première étape est de typer la logique utilisée. Cette logique est définie par la commande `set-logic` et peut comprendre plusieurs théories. En fonction des théories présentes il nous faut ajouter à notre environnement de typage les types et fonctions prédéfinies par celles-ci. Une fois cette étape réalisée, nous pouvons typer une à une les commandes SMT-LIB 2 du fichier d'entrée. Pour simplifier nos règles de typage, nous ne présenterons pas celles de la commande `set-logic` et partirons du principe que notre environnement contient les types et fonctions de la logique déclarée.

La commande `declare-sort` présentée à la figure 5.6 permet d'ajouter à l'environnement des symboles de constructeurs de `sort`. Ces symboles de sort ne doivent pas être déjà présents dans l'environnement et doivent avoir une arité positive. Cette figure nous présente aussi le typage d'un type. Son nombre d'arguments doit correspondre à son arité et les types le comprenant doivent être définis dans Γ. La dernière règle représente le fait qu'une variable d'unification est bien typée dans Γ. Une variable d'unification est une variable issue d'une variable de type *par*.

Ces `sort` peuvent ensuite être utilisés dans les déclarations et définitions de fonctions grâce aux commandes `declare-fun` et `define-fun` donnés sur la figure 5.7. Pour qu'une fonction soit ajoutée à l'environnement, son `symbol` ne doit pas déjà être présent dans l'environnement, et ses types en paramètre comme de retour doivent être correctement définis dans Γ.

Pour une fonction définie par une expression e, le type de retour déclaré

$$\frac{f \notin \Delta \qquad \Gamma \vdash \tau_i \qquad \Gamma \vdash \gamma}{\langle \Gamma \,|\, \Delta \rangle \xrightarrow[\texttt{(declare-fun f (}\tau_1 \cdots \tau_m\texttt{)}\ \gamma\texttt{)}]{} \langle \Gamma \,|\, \Delta \cup (f, (\emptyset, \vec{\tau}, \gamma)) \rangle}$$

$$\frac{\texttt{distinct}(\vec{A}) \qquad f \notin \Delta \qquad \Gamma \cup \bigcup (A_i, 0) \vdash \tau_i \qquad \Gamma \cup \bigcup (A_i, 0) \vdash \gamma}{\langle \Gamma \,|\, \Delta \rangle \xrightarrow[\texttt{(declare-fun f (par (}A_1 \cdots A_n\texttt{) (}\tau_1 \cdots \tau_m\texttt{)}\ \gamma\texttt{))}]{} \langle \Gamma \,|\, \Delta \cup (f, (\vec{A}, \vec{\tau}, \gamma)) \rangle}$$

$$\frac{f \notin \Delta \qquad \Gamma \vdash \tau_i \qquad \Gamma \vdash \gamma \quad \langle \Gamma \,|\, \Delta \cup \bigcup (t_i, (\emptyset, \emptyset, \tau_i)) \rangle \vdash e : \gamma' \quad unify(\gamma, \gamma')}{\langle \Gamma \,|\, \Delta \rangle \xrightarrow[\texttt{(define-fun f ((}t_1\ \tau_1\texttt{)} \cdots \texttt{(}t_m\ \tau_m\texttt{))}\ \gamma\ \texttt{e)}]{} \langle \Gamma \,|\, \Delta \cup (f, (\emptyset, \vec{\tau}, \gamma)) \rangle}$$

$$\frac{\begin{array}{lll} \texttt{distinct}(\vec{A}) & f \notin \Delta & \Gamma \cup \bigcup (A_i, 0) \vdash \tau_i \quad \Gamma \cup \bigcup (A_i, 0) \vdash \gamma \\ \langle \Gamma \cup \bigcup (A_i, 0) \,|\, \Delta \cup \bigcup (t_i, (\emptyset, \emptyset, \tau_i)) \rangle \vdash e : \gamma' & \texttt{type-vars}(e) \in \{\vec{A}\} & unify(\gamma, \gamma') \end{array}}{\langle \Gamma \,|\, \Delta \rangle \xrightarrow[\texttt{(define-fun f (par (}A_1 \cdots A_n\texttt{) ((}t_1\ \tau_1\texttt{)} \cdots \texttt{(}t_m\ \tau_m\texttt{))}\ \gamma\ \texttt{e))}]{} \langle \Gamma \,|\, \Delta \cup (f, (\vec{A}, \vec{\tau}, \gamma)) \rangle}$$

Figure 5.7 – Règles de typage des déclarations et définition de fonctions avec et sans variables de type polymorphes.

```
1  let rec unify t1 t2 =
2    match t1, t2 with
3     | TDummy, TDummy -> t1 <- t2
4     | TDummy, _ -> t1 <- t2
5     | _, TDummy -> t2 <- t1
6
7     | TVar(s1), TVar(s2) ->
8       if s1 <> s2 then
9         error (Type_clash_error(t1, t2))
10
11    | TInt, TInt | TReal, TReal | TBool, TBool -> ()
12
13    | TInt, TReal | TReal, TInt ->
14      error (Type_clash_error(t1, t2))
15
16    | TArray(t1a,t1b), TArray(t2a,t2b) ->
17      unify t1a t2a pos; unify t1b t2b pos
18
19    | TSort(s1,tl1), TSort(s2,tl2) ->
20      if s1 <> s2 then
21        error (Type_clash_error(t1, t2));
22      begin
23        try
24          List.iter2 (fun l l' ->
25              unify l l' pos) tl1 tl2
26        with Invalid_argument _ ->
27          error (Type_clash_error(t1, t2))
28      end
29
30    | TFun(pars1,ret1), TFun(pars2,ret2) ->
31      begin
32        try
33          List.iter2 (fun l l' ->
34              unify l l' pos) pars1 pars2;
35          unify ret1 ret2 pos
36        with Invalid_argument _ ->
37          error (Type_clash_error(t1, t2))
38      end
39      ...
40    | _ , _ -> error (Type_clash_error(t1, t2))
41 end
```

Figure 5.8 – Fonction d'unification des types.

doit être unifiable avec celui inféré du typage de e. Il faut pour cela ajouter à notre environnement de fonction les paramètres nommés de la définition de fonction. La fonction `unify` retourne une erreur qui fait échouer le typage si les deux types en argument ne sont pas unifiables. Elle est présentée de manière simplifiée sur la figure 5.8. `Unify` est une fonction récursive qui prend en argument deux type à unifier. Si un des deux types n'est pas encore assigné (`Tdummy`) il prend la valeur de l'autre type. Si les deux types à unifier sont des variables de types alors on retourne une erreur si il ne sont pas identique. Nous interdisons ainsi la possibilité d'unifier `'a` avec `'b`. Nous avons aussi une erreur si nous essayons de typer un entier avec un réel (ligne 14).Pour ce qui est du typage des tableaux nous typons récursivement les types les comprenant. De même pour les sort déclaré par l'utilisateur. Pour les fonctions nous typons deux à deux leurs arguments puis nous typons leurs types de retour.

Évoquons maintenant les cas où ces déclarations et définitions de fonction sont paramétrées par des variables de type. Il faut tout d'abord vérifier que ces variables sont bien distinctes, puis les ajouter à Γ avec une arité de zéro. Le tuple ajouté à Δ, correspondant à la fonction f, contient les variables de type paramétrés. Dans le cas d'une définition d'une fonction paramétrée par des variables de type, on observe la présence de la fonction `type-vars`. Cette fonction permet de vérifier que nous n'avons pas d'échappement de variable de type dans l'expression e. Toutes les variables de type contenues dans cette expression doivent être déclarées dans la commande de définition de la fonction.

Les dernières commandes (voir figure 5.9) de déclarations sont celles de types de données algébriques. Elles déclarent quant à elles des types et des fonctions liées à ceux-ci. Nous présentons dans la figure 5.9 les règles de typage de ces commandes. La première règle présente une déclaration d'un types de données algébriques n'étant pas paramétré, comme `Color` dans l'exemple 5.1. Le type `t` est ajouté à l'environnement avec une arité de zéro. Les constructeurs $constr_i$ prennent en paramètre les types de retour des destructeurs $destr_{in}$ associés. Les destructeurs prennent quant à eux en paramètre le type de données algébriques `t`. Nous présentons ensuite une déclaration de type de données algébriques paramétré. Le type est ajouté à Γ avec comme arité le nombre de variables de type A_i. Les constructeurs et destructeurs sont paramétrés par ces arguments de type comme des fonctions paramétrées.

La dernière règle présente cette fois-ci la commande simplifiée de déclaration de types de données algébriques pouvant être mutuellement récursifs comme `X` et `Y` dans l'exemple 5.1. Avant de pouvoir vérifier le typage des τ_{i_p} il nous faut ajouter à Γ les types de données algébriques ainsi que leur arité.

Avant de présenter en détail le typage des termes, nous allons présenter dans la figure 5.10 le typage de la dernière commande, la commande `assert`. Comme pour les commandes précédentes nous présentons dans un premier

$$\frac{t \notin \Gamma \quad constr_i \notin \Delta \quad destr_{in} \notin \Delta \qquad \Gamma \vdash \tau_{in}}{\langle\, \Gamma \,|\, \Delta \,\rangle \xrightarrow[\substack{\text{(declare-datatype t} \\ ((constr_i((destr_{i1}\ \tau_{i1}) \cdots (destr_{in}\ \tau_{in}))) \\ ...(constr_m((destr_{m1}\ \tau_{m1}) \cdots (destr_{mo}\ \tau_{mo}))))))}]{} \langle\, \Gamma \cup (t,0) \,|\, \Delta^{\cup \bigcup (constr_i,(\emptyset,\vec{\tau}_{in},t))}_{\cup \bigcup (destr_{in},(\emptyset,t,\tau_{in}))} \,\rangle}$$

$$\frac{\texttt{distinct}(\vec{A}) \quad t \notin \Gamma \quad constr_i \notin \Delta \quad destr_{in} \notin \Delta \qquad \Gamma \cup \bigcup (A_j, 0) \vdash \tau_{in}}{\langle\, \Gamma \,|\, \Delta \,\rangle \xrightarrow[\substack{\text{(declare-datatype t (par } (A_1 \cdots A_j)) \\ ((constr_i((destr_{i1}\ \tau_{i1}) \cdots (destr_{in}\ \tau_{in}))) \\ ...(constr_m((destr_{m1}\ \tau_{m1}) \cdots (destr_{mo}\ \tau_{mo}))))))}]{} \langle\, \Gamma \cup (t,j) \,|\, \Delta^{\cup \bigcup (constr_i,(\vec{A},\vec{\tau}_{in},t))}_{\cup \bigcup (destr_{in},(\vec{A},t,\tau_{in}))} \,\rangle}$$

$$\frac{\texttt{distinct}(\vec{A}_{in}) \quad t_i \notin \Gamma \quad constr_{i_j} \notin \Delta \quad destr_{i_p} \notin \Delta \quad a_i = \#A_{i_n} \quad \Gamma \cup \bigcup (t_i, a_i) \cup \bigcup (A_{i_n}, 0) \vdash \tau_{i_p}}{\langle\, \Gamma \,|\, \Delta \,\rangle \xrightarrow[\substack{\text{(declare-datatypes } ((t_i\ a_i)^+) \\ ((\text{par } (A_n)((constr_j((destr_p\ \tau_p)^*))^+)) \cdots \\ (\text{par } (A_m)((constr_k((destr_q\ \tau_q)*))^+))))}]{} \langle\, \Gamma \cup \bigcup (t_i, a_i) \,|\, \Delta^{\cup \bigcup (constr_{i_j},(\vec{A}_{i_n},\vec{\tau}_{i_p},t_i))}_{\cup \bigcup (destr_{i_p},(\vec{A}_{i_n},t_i,\tau_{i_p}))} \,\rangle}$$

FIGURE 5.9 – Règles de typage des déclarations de types de données algébriques.

$$\frac{\langle\, \Gamma \mid \Delta \,\rangle \vdash e : \texttt{Bool}}{\langle \Gamma \mid \Delta \rangle \vdash \texttt{(assert e)} : ok}$$

$$\frac{\texttt{distinct}(\vec{A}) \qquad \langle\, \Gamma \cup \bigcup(A_i, 0) \mid \Delta \,\rangle \vdash e : \texttt{Bool} \qquad \texttt{type-vars}(e) \in \{\vec{A}\}}{\langle \Gamma \mid \Delta \rangle \vdash \texttt{(assert (par (}A_1 \cdots A_n\texttt{) e))} : ok}$$

Figure 5.10 – Règles de typage des assertions pouvant être paramêtrées par des variable de types et règle d'annotation de type.

```
(define-fun f (par (B) ((s B)) Bool) (forall ((x B)) (= s x)))
(declare-const c (par (C) C))

(assert (f c))
; Typing error : Escaped type variables

(assert (par (A) (f (as c A))))

(check-sat)
```

Figure 5.11 – Exemple d'echappement de variable de type au sein d'une assertion.

temps la commande sans paramètre puis paramétrée avec des variables de type polymorphes. Le typage de la commande non paramétrée requiert uniquement que `e` soit bien typé dans l'environnement et qu'il soit du type `Bool`.

Pour la commande paramétrée nous vérifions comme pour les autres commandes que les variables de type A sont bien toutes distinctes. Nous les ajoutons ensuite à Δ comme des types d'arité 0. Comme expliqué auparavant, la fonction `type-vars` permet d'extraire les variables d'unification libres de l'expression à typer. Nous adoptons une discipline de typage stricte empêchant l'échappement de variables d'unification non déclarées. Dans l'exemple 5.11, la première commande `assert` va échouer car des variables de type peuvent s'en échapper. En effet `c` est une famille de constantes paramétrée par une variable de type qui n'est pas déclarée dans la commande. Pour que la commande soit correcte, il faut la paramétrer avec une variable de type et spécifier le type de `c` avec l'annotation SMT-LIB 2 `(as ⟨symbol⟩ ⟨sort⟩)`.

Nous avons aussi souhaité éviter l'unification entre deux variables de type polymorphes introduites par `par`. Nous considérons, par souci de sûreté, que si `A` est unifiable avec `B` dans la déclaration `(assert (par (A B) (forall ((x A)(y B)) (= x y))))` alors il est préférable de n'utiliser qu'une variable de type. Une commande forçant l'unification de deux variables de types va donc échouer au typage.

$$\frac{\Gamma \vdash \tau \quad \langle\, \Gamma \mid \Delta \,\rangle \vdash e : \tau' \quad unify(\tau, \tau')}{\langle\, \Gamma \mid \Delta \,\rangle \vdash \texttt{(as e } \tau\texttt{)} : \tau}$$

$$\frac{\begin{array}{cccc} \langle\, \Gamma \mid \Delta \,\rangle \vdash t_i : \tau_i & \Delta(f) = (\vec{A}, \vec{\mu}, \nu) & & |\mu| = n \\ \sigma = \{A_i \mapsto \alpha_i\} & \vec{\delta} := \vec{\mu}\,\sigma & \gamma := \nu\,\sigma & unify(\tau_i, \delta_i) \end{array}}{\langle\, \Gamma \mid \Delta \,\rangle \vdash (f\ t_1 \cdots\ t_n) : \gamma}$$

FIGURE 5.12 – Typage d'une application de symbole de fonction.

$$\frac{\Gamma \vdash \tau_x \quad \langle\, \Gamma \mid \Delta \cup (x, \tau_x) \,\rangle \vdash e : \texttt{Bool}}{\langle\, \Gamma \mid \Delta \,\rangle \vdash \texttt{(forall ((x } \tau_x\texttt{)}^{+}\texttt{) e)} : \texttt{Bool}}$$

$$\frac{\Gamma \vdash \tau_x \quad \langle\, \Gamma \mid \Delta \cup (x, \tau_x) \,\rangle \vdash e : \texttt{Bool}}{\langle\, \Gamma \mid \Delta \,\rangle \vdash \texttt{(exists ((x } \tau_x\texttt{)}^{+}\texttt{) e)} : \texttt{Bool}}$$

$$\frac{\langle\, \Gamma \mid \Delta \,\rangle \vdash e_x : \tau_x \quad \Gamma \vdash \tau_x \quad \langle\, \Gamma \mid \Delta \cup (x, \tau_x) \,\rangle \vdash e : \gamma \quad \Gamma \vdash \gamma}{\langle\, \Gamma \mid \Delta \,\rangle \vdash \texttt{(let ((x } e_x\texttt{)}^{+}\texttt{) e)} : \gamma}$$

FIGURE 5.13 – Typage des différents types.

Typage des termes

La figure 5.12 présente notre règle pour le typage d'une application $(f\ t_1 \cdots t_n)$. La première étape est de typer ses arguments ; une fois cette étape effectuée nous recherchons la signature de la fonction f au sein de l'environnement Δ. Après avoir vérifié que l'arité de la signature récupérée correspond au nombre d'arguments, nous créons une signature fraîche $f : \vec{\delta} \rightarrow \gamma$. Pour cela, nous créons une substitution des types paramétrés A_i de la fonction par des variables fraîches d'unification α_i. Cette substitution permet d'obtenir une nouvelle instance des paramètres de type de la fonction ainsi que de son type de retour. Nous pouvons maintenant unifier les types τ_i des arguments t_i avec les arguments de la signature fraîche δ_i. Si les unifications n'échouent pas, nous associons le type γ au type de l'application. Il est important de noter que les α_i sont physiquement partagés entre les $\delta_1, \cdots, \delta_n, \gamma$ pour activer l'unification destructive et propager les modifications sur γ pendant l'unification des δ_i avec les τ_i. Notons aussi que durant le processus de typage, seules les variables fraîches d'unification peuvent être substituées. Les types monomorphes et les paramètres de type ne sont jamais modifiés.

Il nous reste maintenant à présenter les règles de typage pour les autres termes possibles. La figure 5.13 représente ces règles. Les deux premières règles sont triviales et correspondent à la quantification universelle et existentielle. Pour les termes comprenant des `let`, il nous faut tout d'abord typer les expressions e_x puis typer l'expression e en ajoutant à Δ les nouvelles variables x associées à leurs types respectifs τ_x.

2 Implémentation

Maintenant que nous avons présenté notre extension polymorphe pour la SMT-LIB 2 nous allons présenter l'implémentation d'une bibliothèque en OCaml pouvant lire et typer cette SMT-LIB 2 étendue. Nous présenterons ensuite l'intégration de cette bibliothèque dans le solveur Alt-Ergo pour lui permettre de traiter des fichiers au format SMT-LIB 2.

2.1 Création d'une bibliothèque "frontend" pour le langage

Pour la création de cette bibliothèque en OCaml nous avons dans un premier temps développé un analyseur lexical et syntaxique permettant d'accepter un fichier au format de la SMT-LIB 2 natif. Nous l'avons ensuite étendu pour y ajouter notre extension de syntaxe. Il nous a suffi pour cela de modifier la grammaire pour que les expressions paramétrables acceptent la syntaxe précédemment présentée. Notre parser traite les commandes au format SMT-LIB 2 les unes après les autres. Nous créons alors un AST représentant la liste des commandes du fichier d'entrée au format SMT-LIB 2.

Pour notre vérificateur de type, nous nous sommes tout d'abord penchés sur la déclaration de la logique utilisée. Tel qu'expliqué auparavant, la déclaration de la logique utilisée va nous permettre d'ajouter dans notre environnement de typage ses types et fonctions. Les différentes théories gérées par le standard ont été présentées dans l'état de l'art. Les fonctions et types déclarés par les théories sont ajoutés à notre environnement de typage en accord avec les règles présentées précédemment. Une fois la définition des logiques traitées, nous avons fait de même pour les autres commandes du standard en respectant notre système de type.

Cette bibliothèque, appelée `psmt2-frontend` [1] permet donc de lire et vérifier les types d'une partie du standard de la SMT-LIB 2 avec une extension polymorphe. En effet tout le standard de la SMT-LIB 2 n'est pas géré actuellement par cette bibliothèque. Nous ne nous sommes penchés que sur les commandes que nous avons présentées dans ce document. Nous ne supportons pas l'ensemble de théories du standard comme les théories des bitvecteurs et des flottants. Le but était d'obtenir rapidement une bibliothèque permettant de gérer le format SMT-LIB 2 polymorphe dans Alt-Ergo.

2.2 Intégration à Alt-Ergo

Une fois notre bibliothèque développée, nous avons pu brancher l'AST typé qu'elle génère dans Alt-Ergo comme représenté sur la figure 5.14. Du fait qu'au moment où nous avons développé le frontend, Alt-Ergo utilisait la phase de typage pour calculer les triggers des termes quantifiés, nous avons

1. https ://github.com/Coquera/psmt2-frontend

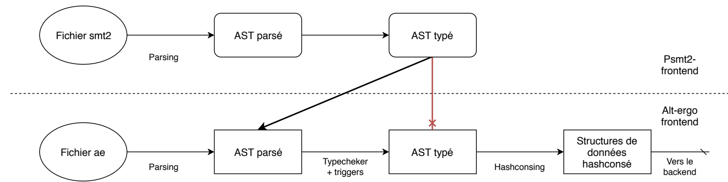

FIGURE 5.14 – Architecture du branchement de psmt2-frontend à Alt-Ergo.

dû nous brancher sur l'AST non typé de celui-ci. À partir de notre AST typé du fichier *smt2* nous générons un AST parsé dans Alt-Ergo. Alt-Ergo étant un solveur de logique du premier ordre, il fait une différence stricte entre les termes et les formules (contrairement à SMT-LIB 2), nous avons dû effectuer quelques modifications pour pouvoir générer un AST au format d'Alt-Ergo.

Cette différenciation amène Alt-Ergo à générer deux types de fonctions, les `predicate`, qui retournent une proposition de logique du premier ordre, et les `logic` qui sont des fonctions pouvant retourner tout autre type. Prenons par exemple l'exemple suivant : `(<= (f (or A B)) c)`. `(or A B)` va être interprété comme une formule par Alt-Ergo, or il n'est pas possible de déclarer une formule au sein d'un terme, ici `(f (or A B))`. Pour résoudre ce problème il a fallu abstraire les formules contenues dans les termes par des variables booléennes fraîches. Nous avons utilisé pour cela des `let-in`. Avec une telle abstraction nous obtenons `(let ((fresh_b (or A B))) (<= (f fresh_b) c))` pour l'exemple précédent.

Ces abstractions nous ont aussi permis d'étendre le support des `if-then-else` et des `let-in` d'Alt-Ergo pour pouvoir supporter toute forme de ces expressions comme par exemple `(<= x (ite (A or B) 1 (+ y 2)))`. Ces abstractions pouvant être effectuées au typage cela nous permet de contourner les spécificités d'Alt-Ergo sans pour autant modifier toutes les structures et raisonnements internes de ce dernier, comme les structures qvec du partage (hashconsing) du backend représentant les termes, littéraux et formules.

Bien que notre travail diffère peu d'un point de vue syntaxe et typage du travail de Bonichon et al [16], il diffère complètement concernant le raisonnement modulo polymorphisme. En effet, comme présenté dans l'état de l'art, Alt-Ergo a été conçu pour supporter nativement le polymorphisme au sein du moteur d'instanciation. Le raccordement de l'aspect polymorphe de notre frontend à Alt-Ergo s'est donc fait de façon triviale. Aucun solveur SMT à notre connaissance ne sait faire cela aujourd'hui. Des expériences ont été menées dans `CVC4` mais sans succès pour l'instant.

3 Résultats expérimentaux

Alt-Ergo supportant maintenant partiellement le standard SMT-LIB 2 avec une extension polymorphique de celle-ci, nous avons souhaité pouvoir montrer l'avantage de traiter des fichiers utilisant du polymorphisme ainsi que son traitement natif par le solveur. Pour cela il nous a fallu générer un ensemble de bancs de tests contenant des fichiers au format `psmt2`, notre format SMT-LIB 2 avec polymorphisme. Nous présenterons comment l'outil Why3 nous a permis de générer nos jeux de tests. Puis, nous présenterons les résultats concernant l'impact du polymorphisme sur les performances de notre solveur. Enfin nous pourrons comparer Alt-Ergo avec d'autres solveurs SMT sur des fichiers d'entrée identiques.

Les fichiers que nous avons utilisés sont des problèmes fournis en partie par nos partenaires industriels comme les fichiers du banc d'essai `SPARK` [48] qui proviennent de la société AdaCore. Les exemples de la catégorie `EACSL` proviennent de fichiers C du projet "EACSL by Exemple" de *Fraunhofer Fokus*. Les quatre catégories `BWARE` [35] proviennent de projet industriels ayant été pré-traités par l'Atelier-B [21, 53]. `DAB` et `RCS3` ont été fournis par *Mitsubishi Electric R&D Centre Europe* tandis que la société *ClearSy* a fourni les catégories `P9` et `P4`. Enfin, nous utilisons aussi les fichiers issus de la galerie de vérification de programmes de Why3.

En ce qui concerne les spécifications techniques, nous avons utilisé comme pour les résultats expérimentaux précédents la plate-forme *Starexec*. Là encore, nous utiliserons une limite de temps de 60 secondes et une limite de consommation mémoire de 2 Go.

3.1 Génération de bancs de tests avec l'outil Why3

Why3 est un outil d'aide à la preuve formelle de programmes. Cet outil est capable de générer des fichiers pouvant être soumis à des solveurs automatiques sous différents formats. Il peut en effet générer des fichiers au format natif d'Alt-Ergo, au format SMT-LIB 2 ou encore au format `TPTP`. Il utilise pour cela des printers transformant la structure du problème d'entrée au format souhaité. Il dispose aussi de multiples transformations permettant de modifier cette structure selon les besoins. Par exemple, la transformation `eliminate-let` supprime les `let-in` et la transformation `eliminate-poly` monomorphise le problème initial si il contenait des variables de types polymorphes.

Dans Why3, un driver contient un printer vers un langage comme SMT-LIB 2. Il contient aussi diverses transformations, et permet de parser et interpréter les réponses du solveur. La figure 5.15 montre une partie du driver d'Alt-Ergo. On peut voir que le printer utilisé est celui d'Alt-Ergo et que le format de réponse d'Alt-Ergo est parsé pour faire une correspondance avec les réponses de Why3. Nous pouvons ensuite remarquer quelques

```
printer "alt-ergo"
filename "%s.why"

valid
 "^File \".*\",line [0-9]+,characters [0-9]+-[0-9]+:Valid"
invalid
 "^File \".*\",line [0-9]+,characters [0-9]+-[0-9]+:Invalid"
unknown
 "^File \".*\",line [0-9]+,characters [0-9]+-[0-9]+:I don't know"
timeout
 "^File \".*\",line [0-9]+,characters [0-9]+-[0-9]+:Timeout"
...

transformation "inline_trivial"
transformation "eliminate_builtin"
transformation "eliminate_if"
transformation "eliminate_let"
transformation "simplify_formula"
...

theory int.Int

  syntax function zero "0"
  syntax function one  "1"

  syntax function (+)  "(%1 + %2)"
  syntax function (-)  "(%1 - %2)"
  syntax function (*)  "(%1 * %2)"
  syntax function (-_) "(-%1)"

  meta "invalid trigger" predicate (<=)
  meta "invalid trigger" predicate (<)
...
```

Figure 5.15 – Driver Why3 pour Alt-Ergo.

```
type 'a option

logic None : 'a option
logic Some : 'a -> 'a option
logic match_option : 'a option, 'b, 'b -> 'b

axiom match_option_None :
(forall z:'a. forall z1:'a. (match_option((None : 'b option), z, z1) = z))

axiom match_option_Some :
(forall z:'a. forall z1:'a. forall u:'b. (match_option(Some(u), z,z1) = z1))
```

Figure 5.16 – Output Why3 au format natif d'Alt-Ergo.

```
(declare-sort option 1)

(declare-const None (par (A) (option A)))
(declare-fun Some (par (A) (A) (option A)))
(declare-fun match_option (par (B A) ((option A) B B) B))

;; match_option_None
(assert (par (B A) (forall ((z B) (z1 B))
        (= (match_option (as None (option A)) z z1) z))))

;; match_option_Some
(assert (par (B A) (forall ((z B) (z1 B) (u A))
        (= (match_option (Some u) z z1) z1))))
```

Figure 5.17 – Output Why3 au format psmt2.

transformations utilisées par Alt-Ergo, et enfin des correspondances entre les théories de Why3 et celles built-in dans Alt-Ergo.

Nous avons dans un premier temps modifié le printer de la SMT-LIB 2 de Why3 pour qu'il supporte notre nouvelle syntaxe. Ensuite, nous avons créé un ensemble de drivers possédant les mêmes transformations. Le but était d'obtenir trois drivers : un pour *smt2*, un pour *psmt2* et un au format natif d'Alt-Ergo dont les seules différences sont le polymorphisme et la syntaxe via des printers différents. Nous essayons ainsi d'avoir des comparaisons les plus équitables possible. Pour le driver au format *smt2*, il devait être capable de générer des fichiers compréhensibles par les autres solveurs multithéories de l'état de l'art tel que `Z3` et `CVC4`.

Nous présentons sur les figures 5.165.17,5.18 un exemple d'encodage avec ces différents formats. Ces exemples représentent l'axiomatisation du pattern matching sur le type "option". On remarque que l'exemple au format natif d'Alt-Ergo est très semblable modulo syntaxe à celui au format *psmt2*. Pour l'exemple au format *smt2* par contre, on remarque que la monomorphisation de Why3 a introduit des nouveaux types `uni` et `ty`. Cette monomorphisa-

```
(declare-sort uni 0)
(declare-sort ty 0)

(declare-fun sort (ty uni) Bool)
(declare-fun option (ty) ty)

(declare-fun None (ty) uni)
;; None_sort
(assert (forall ((a ty)) (sort (option a) (None a))))

(declare-fun Some (ty uni) uni)
;; Some_sort
(assert (forall ((a ty)) (forall ((x uni)) (sort (option a) (Some a x)))))

(declare-fun match_option (ty ty uni uni uni) uni)
;; match_option_sort
(assert
 (forall ((a ty) (a1 ty))
        (forall ((x uni) (x1 uni) (x2 uni)) (sort a1 (match_option a1 a x x1 x2)))))

;; match_option_None
(assert
 (forall ((a ty) (a1 ty))
        (forall ((z uni) (z1 uni))
               (=> (sort a1 z) (= (match_option a1 a (None a) z z1) z)))))

;; match_option_Some
(assert
 (forall ((a ty) (a1 ty))
        (forall ((z uni) (z1 uni) (u uni))
               (=> (sort a1 z1) (= (match_option a1 a (Some a u) z z1) z1)))))
```

Figure 5.18 – Output Why3 au format smt2.

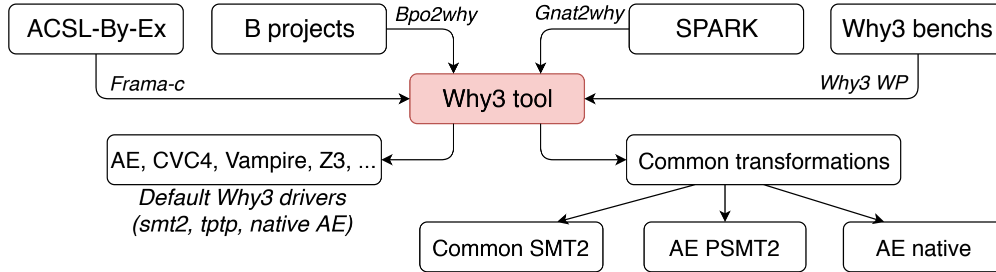

Figure 5.19 – Traduction des problèmes de preuves de programme grâce à Why3.

	`Alt-Ergo_2.3_smt2`	`Alt-Ergo_2.3_psmt2`	`Alt-Ergo_2.3_nat`
BWARE–DAB	765(712s)	844(77s)	**844(59s)**
BWARE–RCS3	2220(1746s)	2227(975s)	**2228(732s)**
BWARE–p4	8789(8186s)	9300(1706s)	**9304(1064s)**
BWARE–p9	223(826s)	268(540s)	**268(475s)**
EACSL	870(476s)	**885(377s)**	881(366s)
SPARK	**14079(1881s)**	14077(1895s)	14069(2615s)
WHY3	1320(1343s)	**1407(1082s)**	1396(1324s)
Total	28266(15173s)	**29008(6656s)**	28990(6636s)

Figure 5.20 – Résultats d'Alt-Ergo sur des fichiers issus de la preuve de programmes avec différent format.

tion introduit aussi de nouvelles fonctions et donc de nouveaux axiomes. La monomorphisation est un problème complexe, celle de Why3 est basée sur les travaux suivants [27] et [13]. Du fait qu'Alt-Ergo supporte nativement le polymorphisme [12], cela permet de ne pas avoir à gérer les fonctions et axiomes générés spécifiquement par Why3 par la monomorphisation.

Une fois ces trois drivers créés, nous avons pu générer des ensembles de fichiers de test dans ces trois formats ainsi que dans les formats natifs des solveurs que nous souhaitions étudier. Nous avons pour cela utilisé les drivers fournis par Why3 pour ces outils. La figure 5.19 représente le schéma de fonctionnement que nous avons adopté pour générer ces bancs de tests. Why3 nous permet ainsi de créer deux ensembles de bancs de test. Le premier consiste à utiliser les drivers natifs des différents solveurs. Le deuxième regroupe les trois formats de bancs de test générés par les trois drivers présentés ci-dessus. Ce sont les fichiers au format natif générés par Why3 que nous avons utilisés dans nos précédents résultats.

3.2 Impact du polymorphisme

Nous allons commencer par étudier l'impact du polymorphisme sur les performances de résolution de notre solveur. La figure 5.20 présente les ré-

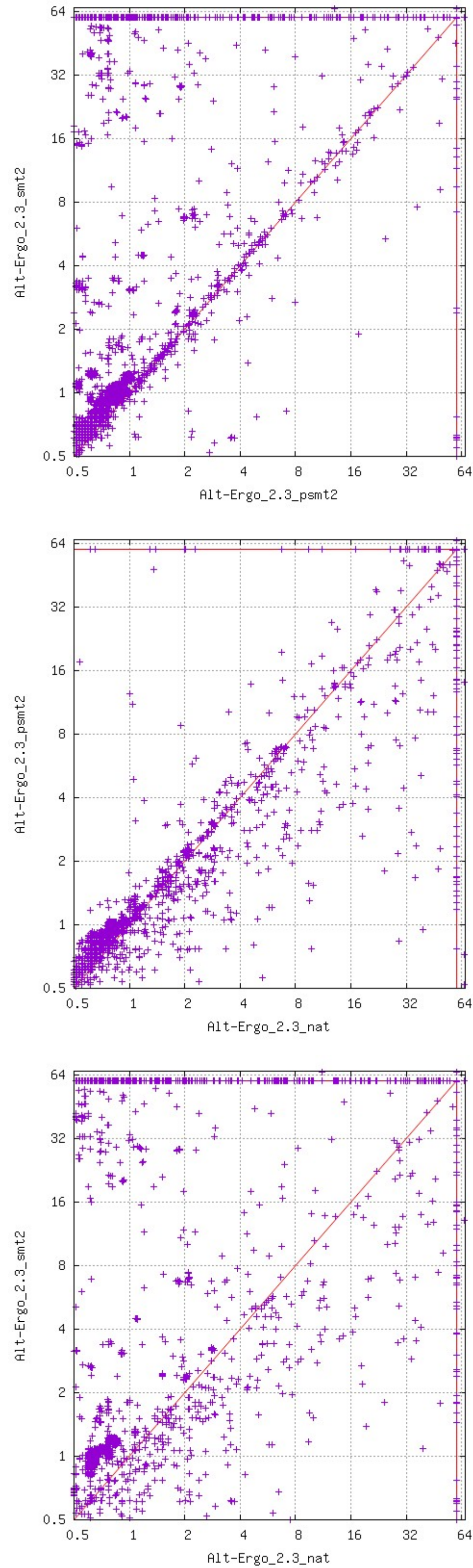

Figure 5.21 – Comparaison des résultats d'Alt-Ergo selon le format d'entrée. Les axes représentent le temps d'exécution d'Alt-Ergo avec les différents format

sultats obtenus par Alt-Ergo sur les bancs de tests présentés précédemment aux formats *smt2*, *psmt2* et natif. On remarque qu'Alt-Ergo résout moins de problèmes monomorphisés au format *smt2* et prend plus de temps. Bien que l'écart de buts résolus ne soit pas très important (0.4%) c'est le temps de résolution qui est le plus frappant : la résolution au format *smt2* polymorphe prend moitié moins de temps que celle au format monomorphe. Si l'on regarde en détail, le format *smt2* polymorphe s'en sort bien mieux que celui monomorphe sur toutes les catégories excepté sur `SPARK`. Si nous regardons maintenant le premier graphique de la figure 5.21 nous observons que peu de fichiers sont résolus plus rapidement voire uniquement résolus au format monomorphe. Le polymorphisme semble être un avantage certain pour Alt-Ergo.

Pour ce qui est de la comparaison des deux formats polymorphes, on remarque que là aussi les résultats diffèrent. Il semblerait qu'Alt-Ergo s'en sorte mieux en termes de temps et de nombre de buts résolus sur les problèmes provenant de `BWARE` au format natif qu'au format *smt2* polymorphe. Cette tendance s'inverse cependant pour les autres catégories où le format *smt2* polymorphe est meilleur. Cela paraît surprenant : en effet, du fait que nous typons deux fois notre AST, nous nous attendions à une consommation temporelle supérieure pour le format *psmt2*. Nous expliquons ces bons résultats par le fait que la résolution de fichiers au format natif est guidée par le but et les termes le contenant. Cette heuristique et le calcul des triggers peuvent influer sur la vitesse de résolution d'Alt-Ergo. On remarque sur le graphique central de la figure 5.21 que de nombreux fichiers sont résolus plus rapidement par le format *psmt2*. On remarque aussi que le nuage de points des fichiers résolus en moins d'une seconde penche en faveur du format natif. Nous expliquons cela par la surcharge de notre double typage. Cette différence entre le langage natif qui est typé une fois et *psmt2* qui l'est deux fois est visible sur les fichiers simples où la majeure partie du temps est passée dans le frontend.

Pour conclure, notre frontend avec extension polymorphe de la SMT-LIB 2 nous permet d'obtenir des résultats proches de ceux du langage natif d'Alt-Ergo. Nous avons pu montrer l'impact qu'a le polymorphisme sur les capacités de résolution d'Alt-Ergo. Maintenant qu'Alt-Ergo supporte le langage de la SMT-LIB 2, nous allons pouvoir le comparer aux autres solveurs de l'état de l'art en utilisant exactement les mêmes fichiers d'entrée.

3.3 Alt-Ergo et les autres solveurs SMT

Nous allons maintenant présenter les résultats que nous avons obtenus en comparant Alt-Ergo avec les solveurs de l'état de l'art. Contrairement aux résultats présentés, sur la figure 4.28 du chapitre précédent, nous avons essayé d'être le plus équitable possible. Nous utilisons pour cela les bancs de tests que nous avons générés au format *smt2*. La figure 5.22 rappelle sur le

Natif	`Alt-Ergo_2.3`	`CVC4_1.6/1.5`	`Vampire_4.3`	`Z3_4.8.5`
BWARE–DAB	**860(47s)**	813(1295s)	401(3040s)	704(128s)
BWARE–RCS3	**2234(725s)**	2040(2845s)	1075(455s)	2191(229s)
BWARE–p4	**9287(790s)**	9109(9754s)	3043(24878s)	7962(1170s)
BWARE–p9	**268(492s)**	187(1018s)	55(1426s)	238(464s)
EACSL	886(293s)	**895(388s)**	434(4999s)	744(619s)
SPARK	14089(2298s)	14267(558s)	12228(16725s)	**15150(1545s)**
WHY3	**1399(1471s)**	1321(632s)	623(6301s)	1196(1429s)
Total	**29023(6119s)**	28632(16492s)	17859(57827s)	28185(5586s)

Smt2 commun	`Alt-ergo_2.3`	`CVC4_1.6`	`vampire_4.3`	`Z3_4.8.5`
BWARE–DAB	**765(712s)**	512(499s)	350(3638s)	722(110s)
BWARE–RCS3	**2220(1746s)**	2039(2056s)	513(7304s)	2169(363s)
BWARE–p4	8789(8186s)	**9044(7723s)**	1905(59618s)	7871(3994s)
BWARE–p9	223(826s)	**260(1560s)**	18(305s)	238(505s)
EACSL	**870(476s)**	683(50s)	433(585s)	679(202s)
SPARK	**14079(1881s)**	14012(612s)	12847(3351s)	13821(324s)
WHY3	**1320(1343s)**	1206(650s)	817(1453s)	1179(2331s)
Total	**28266(15173s)**	27756(13154s)	16883(76257s)	26679(7832s)

Figure 5.22 – Résultats des solveurs de l'état de l'art sur des fichiers issus de la preuve de programmes sur leur formats natif et sur des fichiers *smt2* commun.

premier tableau les résultats des solveurs sur des fichiers obtenus avec leur driver natif. Cette même figure présente sur le second tableau, les résultats avec le driver commun au format *smt2*.

C'est la première fois que nous pouvons comparer de manière équitable nos solveurs car ils travaillent sur des fichiers identiques. Tous les solveurs perdent globalement en nombre de preuves et temps de résolution en passant du fichier au format natif à un format *smt2* commun. `Z3` perd un nombre important de preuves dans la catégorie `SPARK` ; ces pertes sont probablement dues à la présence de bit vecteurs qui sont éliminés dans le format *smt2* commun pour pouvoir être supportés par Alt-Ergo. `CVC4` semble quant à lui profiter des transformations présentes dans le format smt2 commun. En particulier sur la catégorie `BWARE-P9` où il résout plus de soixante-dix buts supplémentaires comparé au nombre de buts qu'il résout sur des fichiers au format natif.

Si nous analysons maintenant les résultats de la figure 5.23 nous remarquons que `CVC4` et Alt-Ergo sont complémentaires et résolvent tous les deux des problèmes que l'autre solveur ne résout pas. Cela est moins le cas lorsque l'on compare Alt-Ergo à `Z3` où l'on remarque que `Z3` dépasse la limite de temps ou répond “Unknown” là où Alt-Ergo répond UNSAT. Nous expliquons les différences de résultats de `Vampire` par le fait que le solveur guide fortement son raisonnement sur le but à prouver. Or, dans le format *smt2*, le but est encodé comme dans une assertion et non une commande

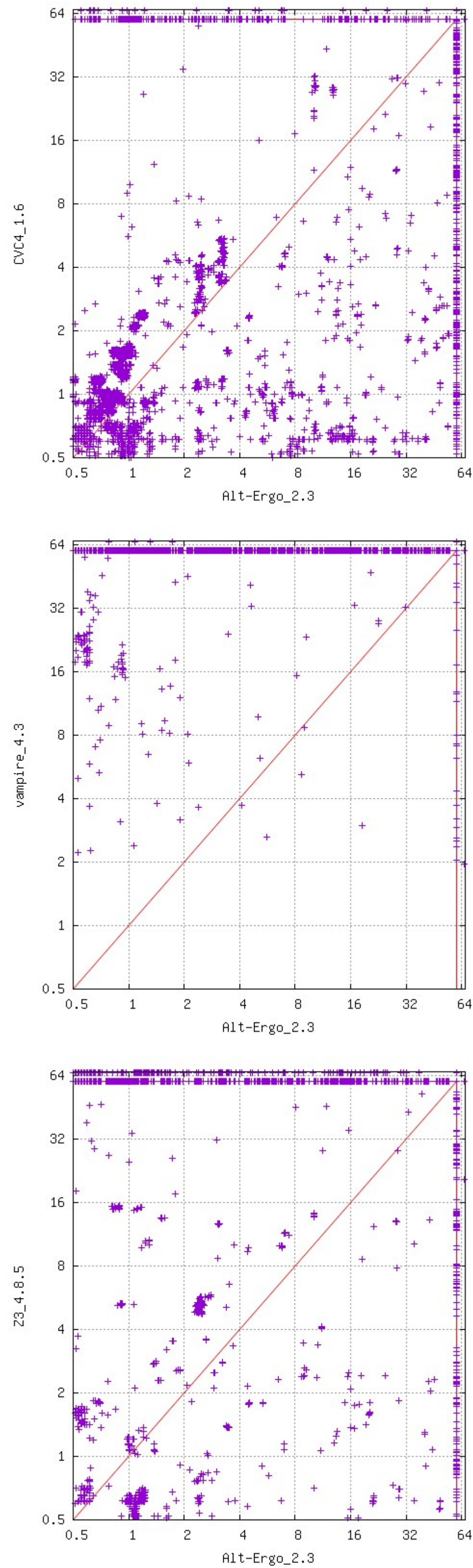

Figure 5.23 – Comparaison des résultats d'Alt-Ergo comparé aux autres solveurs.

spécifique qui permettrait au solveur de se guider.

3.4 Conclusion

Dans ce chapitre, nous avons présenté une extension polymorphe et non intrusive à la syntaxe SMT-LIB 2. Nous avons présenté un ensemble de règles de typage pour cette extension, règles que nous avons ensuite implémentées au sein d'une bibliothèque OCaml. Nous avons ensuite branché cette bibliothèque dans le solveur Alt-Ergo pour enrichir ses formats d'entrée avec une extension polymorphe de SMT-LIB 2. Grâce à l'outil Why3, nous avons pu étudier l'impact du polymorphisme sur les capacités de résolution d'Alt-Ergo et de le comparer aux solveurs de l'état de l'art. Nous avons longtemps milité pour du polymorphisme à la ML dans le standard de la communauté SMT. Un tel polymorphisme devrait apparaître dans la version 3.0 du standard.

Alt-Ergo supportant maintenant suffisamment le standard de la SMT-LIB 2, nous avons pu tester notre solveur sur des fichiers issus de l'ensemble des tests de la communauté. Nous allons présenter des résultats sur la suite de bancs de test de SMT-LIB 2 dans le prochain chapitre consacré à notre participation à la compétition internationale SMT-COMP 2018.

Chapitre 6

Participation à la compétition SMT-COMP

Maintenant que Alt-Ergo supporte le langage standardisé de la SMT-LIB 2, il nous a semblé naturel de nous tester sur les bancs de test de la communauté. Nous avons donc souhaité participer à la SMT-COMP 2018.

Nous présenterons dans un premier temps les grandes lignes de la compétition, ses règles, ses spécifications et ses bancs de test dans la section 1. Dans un deuxième temps dans la section 2, nous expliquerons la préparation d'Alt-Ergo pour la compétition et les modifications que nous avons apportées pour tirer le meilleur de ses performances. Ensuite, nous montrerons les résultats de la compétition dans la section 3. Enfin, nous conclurons ce chapitre et présenterons les résultats de la SMT-COMP 2019 dans la section 4.

1 SMT-COMP

SMT-COMP est une compétition organisée par la communauté SMT depuis 2005 qui possède deux buts principaux. Le premier est de faire adopter la SMT-LIB 2 ainsi que les modifications apportées à celle-ci à un plus grand nombre de solveurs en proposant un nombre conséquent de bancs de test. Le second objectif est orienté vers les performances et les résultats : une compétition pousse les développeurs à améliorer et proposer de nouvelles optimisations pour le monde SMT.

1.1 Les théories et logiques de la SMT-LIB 2

Le terme logique est utilisé par la communauté SMT pour décrire un ensemble de théories mathématiques. Le standard de la SMT-LIB 2 version 2.6 prend en charge les théories mathématiques sur les booléens, les entiers, les réels, les flottants, les tableaux, les vecteurs de bits...

FIGURE 6.1 – Quelques logiques des plus restrictives aux moins contraignantes.

On utilise des abréviations pour définir des logiques et des restrictions sur les problèmes. Pour faciliter la compréhension des combinaisons de logique, on appelle ces combinaisons des catégories. L'ensemble des logiques est présenté ci-dessous :

— QF : formules sans quantificateurs
— AX, A : pour la théorie des tableaux ; avec ou sans extentionnalité
— BV : les vecteurs de bits (ou bit-vecteurs)
— DT : les types de données définis
— FP : les flottants
— IA : l'arithmétique des entiers
— RA : l'arithmétique des réels
— IRA : l'arithmétique des entiers et des réels
— IDL : les Inéquations Différences Logiques sur entiers
— RDL : les Inéquations Différences Logiques sur réels
— LIA, LRA ou LIRA signifient que l'on travaille en arithmétique linéaire
— NIA, NRA ou NIRA signifient que l'on travaille en arithmétique non linéaire
— UF désigne le fait que l'on puisse utiliser des symboles de fonctions non interprétées

La théorie Core sur les booléens est comprise dans toutes les catégories. On obtient donc des catégories simples comme `QF_UF` traitant des problèmes sans quantificateurs sur des symboles non interprétés. Comme des catégories plus complexes telle `AUFBVDTNIRA`, qui traite des problèmes avec quantifica-

teurs, tableaux, bit-vecteurs, datatypes, symboles non interprétés et arithmétique sur les entiers et rationnels. Pour résoudre cette dernière, il faut donc à la fois gérer de nombreuses théories, savoir les combiner de manière performante ainsi qu'avoir une gestion des formules quantifiées performante. La figure 6.1 [1] représente une partie des ces catégories. Plus la catégorie est à droite du graphique et plus elle contient de théorie.

1.2 Bancs de tests de la communauté SMT

La communauté SMT a à sa disposition plusieurs outils permettant son bon développement. Elle dispose en effet d'un ensemble important de bancs de test, disponibles à tous. Ces tests sont principalement fournis par des acteurs de la communauté SMT. Toute personne peut proposer des problèmes au format de la SMT-LIB 2. Ces problèmes, s'ils sont conformes à la SMT-LIB 2, sont triés selon les logiques nécessaires pour leur résolution, puis ajoutés à l'ensemble des bancs de test. Ces problèmes sont dits non incrémentaux lorsqu'ils ne possèdent qu'une commande `check-sat`, sinon ils sont catégorisés en `incrémental` s'ils alternent assertions et commande `check-sat`. Il peut exister différents types de problèmes. Les `crafted` sont des fichiers créés à la main ou générés automatiquement et ayant pour but de vérifier un problème complexe ou une nouveauté de la SMT-LIB 2. Les problèmes `random` sont des problèmes auto-générés aléatoirement, et les problèmes `industrial` sont issus du monde industriel. La quantité de fichiers de tests de la SMT-LIB 2 s'élève aujourd'hui à plus de 300 000, répartis en 51 logiques différentes.

La communauté SMT dispose aussi d'un accès au cluster `Starexec` [2], financé par la *National Science Foundation*. Cette plate-forme utilisée pour la résolution de problèmes est partagée avec d'autres communautés comme `TPTP`. Le cluster est composé de 192 machines [3] équipées de deux processeurs `Intel Xeon E5-2609` (10 MiB Cache, 2.4 GHz) et ce, depuis sa création en 2013. Chacun de ces processeurs possède 4 cœurs et dispose de 64 Go de RAM. Cette constance dans l'infrastructure permet de mieux évaluer les améliorations au fur et à mesure des années et des compétitions.

1.3 Règles de la compétition

Nous allons présenter succinctement les règles de la compétition SMT-COMP 2018 (voir ici pour plus de détails [4]).

Chaque membre de la communauté SMT peut soumettre son solveur à la compétition en spécifiant les catégories dans lesquelles il souhaite participer.

1. http ://smtlib.cs.uiowa.edu/logics.shtml
2. https ://www.starexec.org/starexec/public/about.jsp
3. http ://smtcomp.sourceforge.net/2018/specs.shtml
4. http ://smtcomp.sourceforge.net/2018/rules18.pdf

Les solveurs doivent être déposés sur la plate-forme `Starexec` joints d'un descriptif succinct de ses capacités (théories supportées, fonctionnement sat).

La compétition est divisée en trois sections, `main`, `unsat-core` et `application`. La section `main` permet d'évaluer les performances d'un solveur sur des fichiers pouvant être SAT ou UNSAT et ne comprenant qu'une commande `check-sat`. La section `unsat-core` permet de tester la capacité d'un solveur à générer un unsat-core (ensemble de minimal de formules rendant le problème insatisfiable). tandis que la section `application` permet de tester les capacités incrémentales des solveurs. Nous nous intéresserons à la section `main` à laquelle nous avons participé.

Pour chaque catégorie, le but est de résoudre un maximum de problèmes, c'est-à-dire répondre `UNSAT/SAT` dans le temps imparti. Un système de score permet de classer les solveurs. Ce score est basé sur le nombre de buts résolus correctement ainsi que le poids de chaque but. Ce poids est calculé par rapport au nombre de buts issus de la même famille par rapport au nombre total de buts de la catégorie. La catégorie `AUFNIRA` contient par exemple quatre familles de problèmes : `aviation, FFt, nasa` et `why`. La famille `why` contient 13 buts sur un ensemble de 1480. Un problème résolu de cette famille donnera un score plus important qu'un problème résolu de la famille `Nasa` ($\sim$ 1000 benchs). Le score est fortement impacté lorsqu'une réponse est incorrecte, car elle inflige une perte de quatre points.

Les solveurs disposeront d'un nœud de `Starexec` comprenant quatre cœurs et de 20 Go de `RAM`. Concernant le temps de résolution imparti à chaque problème, chaque solveur aura 20 minutes pour retourner un statut. Les scores sont divisés en deux parties, la première concerne les résultats obtenus de manière séquentielle. Le solveur dispose de 20 minutes qu'il peut consommer sur un ou plusieurs `threads`. Ces 20 minutes sur un `thread` équivaux à 5 minutes sur 4 `threads`. La deuxième partie concerne les résultats obtenus de manière parallèle. Ici le solveur dispose encore de 20 minutes de temps réel, mais peut consommer 20 minutes par `thread` soit jusqu'à 80 minutes s'il fait tourner quatre `threads` en parallèle.

Un score global de la compétition permet de récompenser les meilleurs solveurs toutes catégories confondues. Plus un solveur participe à plusieurs catégories sans faire d'erreur dans ses résultats, plus son score sera élevé.

2 Préparation pour la compétition

Alt-Ergo ne supportant pas toutes les théories du standard, nous ne nous sommes intéressés qu'aux logiques pouvant être supportées par celui-ci. Alt-Ergo étant un solveur SMT dédié à la preuve de programme et ce domaine nécessitant des combinaisons de plusieurs théories ainsi qu'une gestion des formules quantifiées performante, nous avons choisi de nous intéresser aux catégories utilisant de multiples théories ainsi que des formules quantifiées.

Dans cette section nous présenterons les modifications apportées à Alt-Ergo pour obtenir les meilleurs résultats possibles lors de la compétition.

2.1 Amélioration des heuristiques

Dès lors que nous nous sommes confrontés aux problèmes de la SMT-LIB 2 nous avons remarqué une faiblesse concernant le calcul de déclencheur d'Alt-Ergo. En effet, ce calcul étant fait au typage, les déclencheurs ne sont pas modifiables au cours de la résolution. Quelques modifications ont permis d'améliorer ce calcul en permettant de rechercher en profondeur des déclencheurs dans les termes quantifiés ainsi que de mieux traiter la taille de ces derniers.

Pour étudier et comparer l'impact des différentes heuristiques d'Alt-Ergo, nous avons lancé plusieurs instances avec différentes options sur les catégories où nous participons. Grâce à une analyse des résultats de `Starexec`, nous avons pu remarquer que certaines options permettaient de résoudre plus rapidement, voire d'éviter un timeout, un statut Unknown ou de rendre la main précipitamment sans résoudre le but.

Certains problèmes sont par exemple résolus instantanément (moins de 1 seconde) avec le SAT solveur historique d'Alt-Ergo alors que l'on a un timeout avec le solveur SAT CDCL optimisé. Sur l'ensemble des catégories que nous avons testées, le solveur SAT optimisé nous permet de prouver le plus de buts en un temps réduit.

Une modification de notre analyseur de résultats provenant de `Starexec` nous a permis d'obtenir le nombre total de buts résolus en combinant plusieurs jeux d'options. Cela nous a permis de connaître le nombre maximum de buts qu'Alt-Ergo est capable de résoudre par catégorie. Nous en avons déduit qu'il serait judicieux de lancer non pas une configuration d'Alt-Ergo mais plutôt un ensemble d'options pour ce dernier.

`Starexec` permettant de lancer le solveur via un script, nous avons dans un premier temps écrit un script lançant séquentiellement plusieurs instances d'Alt-Ergo avec un jeu d'options permettant d'approcher nos résultats optimums. Il a fallu pour cela gérer du mieux possible le temps de calcul imparti. Nous avons donc lancé en priorité des options avec des petits timeout. Le but était d'obtenir rapidement ($\sim$ 5 secondes) le maximum de résultats possibles. Puis nous avons relancé ces mêmes options avec de plus grands timeout. Le but était d'éviter qu'un problème résolu en une seconde avec une option soit résolu en mille secondes car il fallait attendre que l'option de base se termine avant de pouvoir tester une autre option.

2.2 Implémentation d'une version distribuée d'Alt-Ergo

Le problème de la solution précédente est le fait que l'on doive potentiellement refaire le même travail plusieurs fois en relançant le même jeu

```
1200 -enable-inst-gen 2 -triggers-var
150  -triggers-var -normalize-instances
1200 -triggers-var -normalize-instances -enable-inst-gen 2
150  -sat-detection -sat-solver tableaux -enable-inst-gen 2
150  -nb-triggers 2
150  -nb-triggers 1 -no-Ematching
150  -enable-inst-gen 2 -normalize-instances -enable-restarts
150  -normalize-instances -disable-flat-formulas-simplification
150  -normalize-instances -triggers-var -no-minimal-bj
1200 -no-tableaux-cdcl -normalize-instances -no-minimal-bj
     -disable-flat-formulas-simplification -enable-restarts
1200 -no-tableaux-cdcl -normalize-instances -no-minimal-bj
     -disable-flat-formulas-simplification -enable-restarts
     -nb-triggers 20 -max-multi-triggers-size 20 -triggers-var
     -enable-inst-gen 2 -greedy
```

FIGURE 6.2 – exemple de fichier de configuration d'heuristiques.

d'options avec plusieurs timeout, car on ne peut pas mettre en pause une exécution d'Alt-Ergo pour la relancer ultérieurement. Comme la compétition autorise l'utilisation de plusieurs threads, nous avons décidé de paralléliser cette séquence d'options. Cela nous permet d'éviter le problème de calcul fait plusieurs fois et nous évite aussi qu'une option doive attendre les précédents timeout pour répondre instantanément `UNSAT`. Prenons par exemple la figure 6.2 où le premier argument de chaque ligne est le temps maximal accordé par instance d'Alt-Ergo et les arguments suivants représentent les jeux d'options. L'option à la quatrième ligne est lancée en même temps que les trois premières, au bout d'une seconde elle va répondre UNSAT, alors que les autres feront timeout.

Pour arriver à cette solution, nous avons ajouté l'option `distribute n` à Alt-Ergo. Cette option prend un nombre de threads maximum `n`. Nous souhaitons que le nombre de threads maximum lancé sur chaque nœud corresponde au nombre maximum de cœurs de chaque processeur, ici quatre. Cette option prend aussi un fichier de la forme de la figure 6.2. Les n premières commandes du fichier de configuration sont lancées en parallèle dès le début. Lorsqu'une configuration se termine et qu'elle retourne UNSAT ou SAT, nous arrêtons toutes les autres exécutions en cours. Sinon cela veut dire que nous avons atteint la limite de temps ou que nous avons retourné le statut Unknown. Dans ces cas, nous lançons une nouvelle configuration d'Alt-Ergo avec un autre timeout et un autre jeu d'options qui n'est pas encore lancé en respectant l'ordre du fichier de configuration. Nous n'avons pas remarqué d'augmentation du nombre de memory out malgré le fait que nous consommions plus de mémoire, car nous lançons plusieurs instances d'Alt-Ergo en parallèle.

Prenons maintenant l'exemple 6.3 où les trois premières commandes

```
10 −enable−inst−gen 2 −triggers−var
42  −triggers−var −normalize−instances
50 −triggers−var −normalize−instances −enable−inst−gen 2
100  −sat−detection −sat−solver tableaux −enable−inst−gen 2
```

Figure 6.3 – exemple de fichier de configuration d'heuristiques simplifié.

se lancent en parallèle. La troisième retourne `UNSAT` après 10 secondes de résolution et le programme va alors arrêter tous les `threads`. L'exécution complète aura pris 40 secondes en temps CPU mais 10 secondes en temps réel. Si nous comparons cela à une exécution séquentielle où seule cette troisième option retourne `UNSAT` il faudrait 62 secondes pour obtenir ce résultat.

3 Résultats

Nous allons maintenant aborder les résultats de la compétition SMT 2018 (disponibles ici [5]). Nous avons participé dans trois catégories : AUFNIRA, catégorie comprenant des symboles de la théorie des tableaux (A), des symboles de fonctions non interprétés (UF), et de l'arithmétique non linéaire sur des entiers et réels (NIRA), AUFLIRA(LIRA : arithmétique linéaire sur des entiers et réels), AUFLIA (LIA : arithmétique linéaire sur les entiers), ALIA. Les figures 6.4, 6.5 et 6.6 présentent les résultats dans ces différentes catégories. Nous ne présenterons pas les résultats de la catégorie `AUFLIA` où seulement 3 buts ont été gardés, les autres étant des ajouts récents à la catégorie et non conformes aux règles de la compétition. Dans les résultats qui seront présentés, le solveur `Z3` a été ajouté au solveur participant à la compétition par les organisateurs à titre de référence et n'est pas compté comme participant. Il est aussi important de noter que les scores prennent en compte les résultats valides, qu'ils soient UNSAT ou SAT. Ce manque de distinction entre les deux réponses a une importance pour Alt-Ergo qui ne peut pas répondre SAT.

Sur la figure 6.4 nous pouvons remarquer que les résultats obtenus sur la catégorie AUFNIRA sont prometteurs, nous nous plaçons derrière `CVC4` et `z3` mais sommes devant `Vampire` (gagnant 2017 sur les catégories AUFNIRA et AUFLIRA) en termes de nombre de buts résolus. Comme expliqué auparavant en section 1.3, le score ne dépend pas que du nombre de buts résolus, ce qui permet à `Vampire` d'obtenir un meilleur score qu'Alt-Ergo et même `z3`. Nous remarquons que notre score parallèle reste quasiment inchangé avec seulement un but supplémentaire résolu.

La figure 6.5 nous présente les résultats de la catégorie `AUFLIRA`. Là encore les résultats sont prometteurs : nous devançons `Vampire` et `VeriT` en score séquentiel, et `Vampire` reprend la seconde place sur le score parallèle. Encore

5. http ://smtcomp.sourceforge.net/2018/results-toc.shtml

AUFNIRA (Main Track)

Competition results for the AUFNIRA division as of Fri Jul 13 00:02:11 GMT

Benchmarks in this division : 1480
Time limit: 1200s

Winners:

Sequential Performances	Parallel Performances
CVC4	CVC4

Result table[1]

Sequential Performance

Solver	Error Score	Correctly Solved Score	CPU time Score	Solved	Unsolved
CVC4	0.000	1147.005	284.139	1072	408
Vampire 4.3	0.000	1060.560	354.508	1019	461
z3-4.7.1[n]	0.000	1057.460	194.776	1031	449
Alt-Ergo	0.000	1032.924	255.355	1024	456

Parallel Performance

Solver	Error Score	Correctly Solved Score	CPU time Score	WALL time Score	Solved	Unsolved
CVC4	0.000	1147.005	284.307	288.589	1072	408
Vampire 4.3	0.000	1125.865	1224.182	307.871	1038	442
z3-4.7.1[n]	0.000	1057.460	194.777	195.326	1031	449
Alt-Ergo	0.000	1033.428	745.706	252.515	1025	455

n. Non-competing.

FIGURE 6.4 – Résultats de la catégorie AUFNIRA lors de la SMT-COMP 2018.

AUFLIRA (Main Track)

Competition results for the AUFLIRA division as of Fri Jul 13 00:02:11 GMT

Benchmarks in this division : 20011
Time limit: 1200s

Winners:

Sequential Performances	Parallel Performances
CVC4	CVC4

Result table[1]

Sequential Performance

Solver	Error Score	Correctly Solved Score	CPU time Score	Solved	Unsolved
z3-4.7.1[n]	0.000	17811.287	108.778	19849	162
CVC4	0.000	16961.904	193.990	19763	248
Alt-Ergo	0.000	16407.063	159.101	19658	353
Vampire 4.3	0.000	16278.986	251.420	19532	479
veriT	0.000	15049.918	273.923	19317	694

Parallel Performance

Solver	Error Score	Correctly Solved Score	CPU time Score	WALL time Score	Solved	Unsolved
z3-4.7.1[n]	0.000	17811.287	108.779	108.797	19849	162
CVC4	0.000	16961.904	194.259	195.661	19763	248
Vampire 4.3	0.000	16868.126	784.406	197.366	19737	274
Alt-Ergo	0.000	16526.108	393.858	141.392	19672	339
veriT	0.000	15049.918	274.337	273.931	19317	694

n. Non-competing.

Figure 6.5 – Résultats de la catégorie AUFLIRA lors de la SMT-COMP 2018.

une fois nous gagnons quelques buts dans cette section parallèle, mais cela est bien moins important que pour `Vampire`.

Les derniers résultats sont présentés sur la figure 6.6. Alt-Ergo montre là encore son potentiel en se positionnant deuxième de la catégorie `ALIA` – notons que `Z3` n'y participe pas.

4 Conclusion

Participer à la compétition SMT nous a avant tout permis de situer Alt-Ergo dans la communauté SMT comme un solveur SMT performant et compétitif sur des problèmes proches de la preuve de programme. Les benchmarks de la communauté peuvent être assez différents de ceux de nos partenaires habituels. Nous comptons proposer nos bancs de tests à la communauté. Alt-Ergo montre de mauvais résultats sur des exemples fait-main qui ont pour but de mettre l'accent intensivement sur la théorie des tableaux (benchs peter). Notre frontend pour la syntaxe *SMT2* et son intégration sont encore très récents et nous gagnerions à améliorer son support ainsi qu'ajouter une phase de pre-processing. Nous avons remarqué des exemples pour lesquels la plupart du temps d'exécution est passé dans les phases précédant la résolution. Cela est en particulier criant sur des problèmes contenant beaucoup de `if-then-else` ou de `let-in`. Notre support de ceux-ci (présenté précédemment) peut alourdir la phase de résolution en ajoutant des littéraux sur lesquels raisonner. De plus, le calcul de déclencheurs n'étant pas optimum, cela peut avoir un impact important sur notre moteur d'instanciation.

Nous arrivons malgré tout à des résultats sans erreur et proche des solveurs de l'état de l'art, ce qui nous encourage et montre qu'Alt-Ergo est un solveur fiable et performant. Des participations aux futures compétitions SMT pourraient permettre d'améliorer les points faibles d'Alt-Ergo et d'étendre ses théories pour couvrir plus de logiques. Il serait aussi possible de participer dans d'autres sections telles que `unsat-core`.

SMT-COMP 2019. Nous avons également inscrit Alt-Ergo à la SMT-COMP 2019. Comme en 2018, nous avons participé dans la section non-incrémental `single query` (anciennement `main`). Cependant quelques changements sont à noter. Cette année une différenciation dans le calcul des points a été faite entre les résultats UNSAT et SAT. Cela nous a permis de lancer Alt-Ergo sur plus de catégories, comme des catégories sans quantificateurs qui comportent beaucoup de problème SAT. De plus l'ajout récent du raisonnement sur les types de données abstraits, `datatype`, nous permet de lancer Alt-Ergo sur des nouvelles catégories. La base de fichiers à résoudre à elle aussi été modifiée. Il a en effet été décidé que seuls les problèmes qui n'ont pas été résolus par tous les solveurs en 2018 seront étudiés en 2019

ALIA (Main Track)

Competition results for the ALIA division as of Fri Jul 13 00:02:11 GMT

Benchmarks in this division : 42
Time limit: 1200s

Winners:

Sequential Performances	Parallel Performances
CVC4	CVC4

Result table[1]

Sequential Performance

Solver	Error Score	Correctly Solved Score	CPU time Score	Solved	Unsolved
z3-4.7.1[n]	0.000	42.000	0.056	42	0
CVC4	0.000	40.000	0.677	40	2
Alt-Ergo	0.000	39.000	28.764	39	3
Vampire 4.3	0.000	27.000	495.046	27	15
veriT	0.000	27.000	428.554	27	15

Parallel Performance

Solver	Error Score	Correctly Solved Score	CPU time Score	WALL time Score	Solved	Unsolved
z3-4.7.1[n]	0.000	42.000	0.056	0.056	42	0
CVC4	0.000	40.000	0.677	0.677	40	2
Alt-Ergo	0.000	39.000	28.821	28.672	39	3
Vampire 4.3	0.000	35.000	1473.815	370.630	35	7
veriT	0.000	27.000	428.697	428.599	27	15

n. Non-competing.

Figure 6.6 – Résultats de la catégorie ALIA lors de la SMT-COMP 2018.

UNSAT Performance

Solver	Error Score	Correct Score	CPU Time Score	Wall Time Score	Solved	Solved SAT	Solved UNSAT	Unsolved	Timeout	Memout
Vampire	0	1283	86193.496	46991.541	1283	0	1283	355	251	0
2018-CVC4[n]	0	1246	138434.195	138748.646	1246	0	1246	392	231	0
CVC4-SymBreak[n]	0	1244	150692.748	151147.351	1244	0	1244	394	231	0
CVC4	0	1244	152707.489	153128.973	1244	0	1244	394	231	0
Alt-Ergo	0	1229	248456.484	182814.691	1229	0	1229	409	253	38
veriT	0	1169	310293.021	310255.595	1169	0	1169	469	287	15
Z3[n]	0	1145	290572.637	292456.469	1145	0	1145	493	201	24
SMTInterpol	0	802	1267718.696	1181493.656	802	0	802	836	638	0
UltimateEliminator+SMTInterpol	0	9	108577.91	88421.65	9	0	9	1629	57	0
UltimateEliminator+MathSAT-5.5.4	0	9	97069.412	88808.782	9	0	9	1629	60	0
UltimateEliminator+Yices-2.6.1	0	4	97292.151	88767.632	4	0	4	1634	61	0

24s Performance

Solver	Error Score	Correct Score	CPU Time Score	Wall Time Score	Solved	Solved SAT	Solved UNSAT	Unsolved	Timeout	Memout
Vampire	0	1351	9527.331	7707.288	1351	99	1252	287	287	0
Z3[n]	0	1271	9222.71	9222.746	1271	155	1116	367	342	24
2018-CVC4[n]	0	1239	9636.401	9635.936	1239	87	1152	399	391	0
CVC4-SymBreak[n]	0	1236	9695.328	9694.848	1236	88	1148	402	394	0
CVC4	0	1235	9705.25	9704.786	1235	88	1147	403	395	0
Alt-Ergo	0	1165	10881.024	9272.146	1165	0	1165	473	322	38
veriT	0	1108	10066.745	10049.121	1108	0	1108	530	365	15
SMTInterpol	0	823	20632.697	19458.572	823	93	730	815	754	0
UltimateEliminator+SMTInterpol	0	14	7675.815	5595.368	14	5	9	1624	83	0
UltimateEliminator+MathSAT-5.5.4	0	13	7663.916	5823.054	13	4	9	1625	82	0
UltimateEliminator+Yices-2.6.1	0	7	7559.95	5793.901	7	3	4	1631	81	0

n Non-competing.

FIGURE 6.7 – Résultats de la catégorie AUFLIA lors de la SMT-COMP 2019.

allégeant fortement la base de buts à prouver (détails des règles[6]).

Concernant les modifications apportées à Alt-Ergo pour la compétition, elles sont minimes. Nous nous sommes basés sur la dernière version du solveur et avons repris les modifications concernant la version distribuée. Les principales modifications entre les deux versions des compétitions sont donc celles apportées entre la version 2.2 et 2.3.

Du fait que peu d'efforts ont été produits pour cette édition 2019, nous ne détaillerons pas tous les résultats[7] des catégories où Alt-Ergo a concouru. La figure 6.7 représente une partie des résultats de la catégorie AUFLIA. Les deux tableaux présentent deux nouveautés de l'édition 2019. Le premier

6. https ://smt-comp.github.io/2019/rules19.pdf
7. https ://smt-comp.github.io/2019/results.html

ne présente que les performances des solveurs sur les résultats UNSAT. Le second présente lui les performances des solveurs avec une limite de temps de seulement 24s. Cela permet de distinguer les solveurs qui offrent rapidement une réponse.

Globalement Alt-Ergo a été aussi performant que l'an passé, bien que nous ne soyons pas très efficaces sur les problèmes sans quantificateur. Cela nous montre qu'encore beaucoup d'efforts restent à produire concernant nos théories.

Chapitre 7

Conclusion

Dans cette thèse nous avons présenté nos travaux d'analyse de performance entre deux implémentations du solveur Minisat, nous avons ensuite présenté un algorithme permettant de rendre efficace l'intégration d'un solveur SAT tel que Minisat dans un solveur SMT comme Alt-Ergo. Puis nous avons présenté une extension polymorphe du standard SMT-LIB 2 ainsi que son support par le solveur SMT Alt-Ergo qui nous a finalement amené à participer à la compétition de la communauté SMT.

Solveur SAT performant en OCaml

Dans le chapitre 3 nous avons dans un premier temps présenté les atouts du solveur Minisat, qui en font un solveur de référence. Nous avons ensuite expliqué la méthodologie nous ayant permis de comparer Minisat à une implémentation de ce même solveur en OCaml. Des résultats expérimentaux nous ont permis de conclure qu'une telle implémentation de Minisat en OCaml était moins performante que son homologue en C++. Ces résultats sont proches de ceux présentés par Cruanes[28]. Nous nous sommes assurés que les écarts n'étaient pas dus à une différence algorithmique en vérifiant que les décisions étaient identiques dans les deux implémentations. D'après nos analyses, nous avons conclu que les écarts de performance étaient principalement dus à la gestion mémoire du langage OCaml. En effet, d'une part la taille des données en OCaml est plus importante qu'en C++ et d'autre part à cause du gestionnaire de mémoire automatique d'OCaml. Ce dernier augmente la consommation de la mémoire et perturbe les chargements de données dans les différents caches du processeur.

En partant de ce constat, nous avons comparé les performances de ce solveur SAT à un solveur par méthode des tableaux au sein du solveur SMT Alt-Ergo. Les résultats nous ont montré que l'intégration d'un solveur SAT performant au sein d'Alt-Ergo nécessitait une optimisation de la combinaison entre les différentes parties d'un SMT. Le branchement d'un solveur SAT

performant au sein d'un solveur SMT n'est donc pas trivial.

Intégration du solveur SAT performant au solveur SMT Alt-Ergo

Nous avons présenté et formalisé dans le chapitre 4 une optimisation permettant une meilleure combinaison d'un solveur SAT CDCL performant dans Alt-Ergo. Cette optimisation se base sur une réduction du modèle booléen retourné par le solveur CDCL par un calcul de pertinence semblable aux travaux de Björner et de Moura[33] sur le solveur Z3. Pour effectuer ce calcul, nous utilisons un parcours par méthode des tableaux de la formule représentant le problème pour marquer et sélectionner les atomes pertinents du modèle booléen à envoyer au combinateur de théorie ou au moteur d'instanciation. Plusieurs itérations d'implémentation de cette optimisation furent présentées. Nous avons par la suite testé et comparé l'optimisation la plus aboutie au solveur SAT par méthode des tableaux (solveur SAT historique d'Alt-Ergo) et au solveur SAT CDCL sans l'optimisation. Nous obtenons de meilleurs résultats comparé à ces deux derniers solveurs. Nous avons étudié l'impact de cette optimisation sur les modèles booléens envoyés au combinateur de théories ainsi que le moteur d'instanciation. L'optimisation semble cruciale pour que le moteur d'instanciation tienne la charge.

Nous avons souhaité comparer notre solveur Alt-Ergo équipé de cette optimisation à des solveurs performants de l'état de l'art. Malgré les bons résultats de notre solveur, le fait que les fichiers traités n'étaient pas exactement les mêmes dus aux différences de syntaxe d'entrée des solveurs nous a poussés à ajouter le support du standard de la communauté SMT au sein d'Alt-Ergo. Cela pour nous permettre d'effectuer des comparaisons plus justes, sur la même base de problèmes, avec d'autres solveurs.

Extension du standard SMT-LIB 2 avec du polymorphisme

Le chapitre 5 présente notre extension du standard SMT-LIB 2 avec du polymorphisme. Nous avons proposé des modifications à la syntaxe du standard en nous basant sur des travaux de la communauté concernant le polymorphisme comme l'ajout récent d'une certaine forme de polymorphisme pour les types de données abstraits ou les travaux de Bonichon, Déharbe et Tavares[16]. Nous avons ensuite présenté les règles de typage correspondant pour les commandes polymorphes de l'extension.

Nous avons présenté par la suite la bibliothèque que nous avons développée et qui permet de parser et typer cette extension. Le branchement de cette bibliothèque dans Alt-Ergo n'étant pas trivial, nous avons expliqué

les différences et les modifications que nous avons dû apporter pour que le solveur supporte notre bibliothèque. Grâce à l'outil *Why3*, nous avons généré des ensembles de fichiers sous différents formats à partir des mêmes problèmes initiaux. Ces différents formats nous ont permis de comparer les performances d'Alt-Ergo sur des fichiers dans son langage natif à des fichiers dans la syntaxe SMT-LIB 2 étendue avec du polymorphisme. Nous avons aussi comparé ces fichiers avec des fichiers monomorphisés par *Why3*[13].

Cela nous permet de conclure que le traitement natif du polymorphisme nous permet de résoudre plus de problèmes que lorsque ceux-ci sont monomorphisés. Les fichiers monomorphisés étant aussi compréhensibles par les solveurs de l'état de l'art, nous avons pu comparer Alt-Ergo à ces derniers. De cette expérimentation, nous pouvons conclure qu'Alt-Ergo est un solveur SMT performant.

SMT-COMP 2018

Les bons résultats d'Alt-Ergo comparé au solveur de l'état de l'art nous ont poussé à nous intéresser à la compétition annuelle de la communauté SMT. Nous nous sommes lancés dans les catégories se rapprochant du coeur de métier d'Alt-Ergo : la preuve de programme. Ces catégories requièrent le support de différentes théories ainsi qu'un support efficace pour les quantificateurs universels. Les règles de la compétition ainsi que les résultats sont présentés dans le chapitre 6. Nous avons également présenté dans ce chapitre les modifications que nous avons apportées au solveur Alt-Ergo pour optimiser ses résultats via différents jeux d'options et l'ajout d'un mécanisme de parallélisation permettant d'obtenir de meilleurs résultats globaux. Enfin, nous avons présenté les résultats obtenus sur les catégories dans lesquelles nous avons participé.

Perspectives

Les travaux présentés au cours de cette thèse nécessitent encore quelques améliorations et depuis leur réalisation, des modifications ont été apportées au solveur Alt-Ergo. Une réorganisation du code a ainsi permis d'extraire le calcul des déclencheurs du typage d'Alt-Ergo. Cela pourrait nous permettre de ne plus brancher la bibliothèque `psmt2-frontend` sur l'AST parsé d'Alt-Ergo mais directement sur son AST typé.

Des travaux supplémentaires peuvent être apportés à cette bibliothèque pour que celle-ci supporte l'intégralité du standard SMT-LIB 2, comme l'ajout du support pour les options ou l'analyse lexicale et syntaxique pour les théories des vecteurs de bits et de calcul flottant.

Des discussions[3] sur l'avenir du standard ont porté sur le support d'une certaine forme de polymorphisme. Des modifications mineures de la biblio-

thèque `psmt2-frontend` permettrait de supporter de telles modifications de la syntaxe. Nous devrions aussi pouvoir régénérer nos bancs de tests via l'outil Why3 en nous conformant à ces modifications de syntaxe, pour fournir à la communauté une base de tests polymorphes conséquente.

Enfin, comme présenté dans le chapitre 3, le ramasse-miettes d'OCaml pourrait profiter de l'utilisation d'instructions n'utilisant pas les caches mémoire[15] ce qui pourrait améliorer les performances des programmes en OCaml.

Bibliographie

[1] *First-Order Logic*, pages 35–68. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[2] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2) :129–168, Oct 2003.

[3] Clark Barett, Pascal Fontaine, and Cesare Tinelli. Smt-lib 3 : Bringing higher-order logic to smt. Technical report, 2018. `http://matryoshka.gforge.inria.fr/wait2018/slides/WAIT-2018-Fontaine-SMT-LIB-TIP.pdf`.

[4] Clark Barrett and Jacob Donham. Combining sat methods with non-clausal decision heuristics. *Electron. Notes Theor. Comput. Sci.*, 125(3) :3–12, July 2005.

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard : Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard : Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[8] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard : Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[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, UK)*, 2010.

[10] Clark Barrett and Cesare Tinelli. CVC4 : the SMT solver. `http://cvc4.cs.stanford.edu/web/`.

[11] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability : Volume 185 Frontiers in Artificial Intelligence and Ap-*

plications. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[12] François Bobot, Sylvain Conchon, Evelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In *SMT '08/BPR '08 : Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 1–5, New York, NY, USA, 2008. ACM.

[13] François Bobot and Andrei Paskevich. Expressing Polymorphic Types in a Many-Sorted Language. pages 87–102.

[14] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. Preserving User Proofs Across Specification Changes. In Ernie Cohen and Andrey Rybalchenko, editors, *Fifth Working Conference on Verified Software : Theories, Tools and Experiments*, volume 8164, pages 191–201, Atherton, United States, May 2013. Springer.

[15] Hans-J. Boehm. Reducing garbage collector cache misses, 2000.

[16] Richard Bonichon, David Déharbe, and Cláudia Tavares. Extending SMT-LIB v2 with λ-terms and polymorphism. In Philipp Rümmer and Christoph M. Wintersteiger, editors, *Proceedings of the 12th International Workshop on Satisfiability Modulo Theories, SMT 2014*, volume 1163, pages 53–62, 2014.

[17] Çağdaş Bozman. *Memory profiling of OCaml applications*. Theses, ENSTA ParisTech, December 2014.

[18] Çagdas Bozman, Grégoire Henry, Mohamed Iguernelala, Fabrice Le Fessant, and Michel Mauny. ocp-memprof : un profileur mémoire pour OCaml. In David Baelde and Jade Alglave, editors, *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, Le Val d'Ajol, France, January 2015.

[19] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11) :677–678, November 1970.

[20] Jean christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *In : Proceedings of the 2006 workshop on ML*, pages 12–19. ACM Press, 2006.

[21] ClearSy System Engineering. *Atelier B User Manual, version 4.0*. `http://tools.clearsy.com/wp-content/uploads/sites/8/resources/User_uk.pdf`.

[22] Sylvain Conchon, Evelyne Contejean, and Mohamed Iguernelala. Canonized rewriting and ground AC completion modulo shostak theories : Design and implementation. *Logical Methods in Computer Science*, 8(3), 2012.

[23] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Cc(x) : Semantic combination of congruence closure with solvable theories. *Electronic Notes in Theoretical Computer Science*, 198(2) :51–69, May 2008.

[24] Sylvain Conchon, Albin Coquereau, Mohamed Iguernelala, and Alain Mebsout. Alt-ergo 2.2. In *Proceedings of the 16th International Workshop on Satisfiability Modulo Theories, SMT 2018*, Oxford, UK, 2018.

[25] Sylvain Conchon, Mohamed Iguernlala, Kailiang Ji, Guillaume Melquiond, and Clément Fumex. A three-tier strategy for reasoning about floating-point numbers in smt. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 419–435, Cham, 2017. Springer International Publishing.

[26] Sylvain Conchon, Mohamed Iguernlala, and Alain Mebsout. AltGr-Ergo, a Graphical User Interface for the SMT Solver Alt-Ergo. In *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016.*, pages 1–13, 2016.

[27] Jean-François Couchot and Stéphane Lescuyer. Handling polymorphism in automated deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.

[28] Simon Cruanes. Faithful reimplementation of minisat 2.2 in ocaml. Technical report, 2018. `https://github.com/AestheticIntegration/minisat-ml/blob/master/docs/tech_report.md`.

[29] George B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1991.

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

[31] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, July 1960.

[32] L. de Moura and N. Bjørner. Z3, an efficient SMT solver. `http://research.microsoft.com/projects/z3`.

[33] Leonardo de Moura and Nikolaj Bjorner. Relevancy propagation. Technical report, Microsoft Research, October 2007.

[34] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. pages 46–52. IEEE, 2013.

[35] David Delahaye, Catherine Dubois, Claude Marché, and David Mentré. The bware project : Building a proof platform for the automated verification of b proof obligations. In Yamine Ait Ameur and Klaus-Dieter Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 290–293, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[36] David Detlefs, Greg Nelson, and James B. Saxe. Simplify : A theorem prover for program checking. *J. ACM*, 52(3) :365–473, May 2005.

[37] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 70–83, New York, NY, USA, 1994. ACM.

[38] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL 1993 : 20th symposium Principles of Programming Languages*, pages 113–123, Charleston, United States, January 1993.

[39] David Déharbe and Pascal Fontaine. The veriT SMT solver. `https://verit.loria.fr/`.

[40] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[41] Uwe Egly and Leopold Haller. A sat solver for circuits based on the tableau method. *KI - Künstliche Intelligenz*, 24(1) :15–23, Apr 2010.

[42] Jean-Christophe Filliâtre. One Logic To Use Them All. In Maria Paola Bonacina, editor, *CADE 24 - the 24th International Conference on Automated Deduction*, Lake Placid, NY, United States, June 2013. Springer.

[43] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[44] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792 of *LNCS*, Rome, Italy, March 2013. Springer.

[45] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, Cesare Tinelli, Rajeev Alur, and Doron Peled. Dpll(t) : Fast decision procedures. 06 2004.

[46] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. A tale of two solvers : Eager and lazy approaches to bit-vectors. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 680–695, Berlin, Heidelberg, 2014. Springer-Verlag.

[47] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146 :29–60, 1969.

[48] Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. SPARK 2014 and GNATprove. *International Journal on Software Tools for Technology Transfer*, 17(6) :695–707, Nov 2015.

[49] Mohamed Iguernelala. *Strengthening the heart of an SMT-solver : Design and implementation of efficient decision procedures.* Theses, Université Paris Sud - Paris XI, June 2013.

[50] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Number 248966-041. April 2019.

[51] Tommi A. Junttila and Ilkka Niemelä. Towards an efficient tableau method for boolean circuit satisfiability checking. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic — CL 2000*, pages 553–567, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[52] Leonid Khachiyan. *Fourier–Motzkin elimination method*, pages 1074–1077. Springer US, Boston, MA, 2009.

[53] David Mentré, Claude Marché, Jean-Christophe Filliâtre, and Masashi Asuka. Discharging Proof Obligations from Atelier B Using Multiple Automated Provers. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, pages 238–251, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[54] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348 – 375, 1978.

[55] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[56] Leonardo Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *Proceedings of the 21st International Conference on Automated Deduction : Automated Deduction*, CADE-21, pages 183–198, Berlin, Heidelberg, 2007. Springer-Verlag.

[57] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2) :356–364, April 1980.

[58] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205 :557–580, 04 2007.

[59] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories : From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6) :937–977, November 2006.

[60] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[61] Silvio Ranise and Cesare Tinelli. The smt-lib format : An initial proposal. In *In PDPAR*, pages 94–111, 2003.

[62] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard : Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[63] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, January 1965.

[64] Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1) :1–12, January 1984.

[65] João P. Marques Silva and Karem A. Sakallah. Grasp&mdash ;a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[66] Wayne Snyder. Efficient ground completion. In Nachum Dershowitz, editor, *Rewriting Techniques and Applications*, pages 419–433, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[67] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[68] Andrei Voronkov, Laura Kovács, Giles Reger, and Martin Suda. Vampire. `https://vprover.github.io/`.

[69] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Amélioration de performances du solveur SMT Alt-Ergo grâce à l'intégration d'un solveur SAT efficace

Mots clés : SMT, SAT, OCaml, Alt-Ergo, SMT-LIB, polymorphisme

Résumé : Ce manuscrit de thèse s'intéresse dans un premier temps à l'étude des performances d'une implémentation de Minisat, un solveur SAT de référence en OCaml, comparée à une mise en œuvre en C++. Une implémentation rigoureusement identique fut nécessaire pour garantir un fonctionnement équivalent de ces deux implémentations. Une étude approfondie de chacune d'elles grâce à une suite de tests alliée à des outils d'analyse nous a permis de comprendre la raison des écarts de performance entre les deux intégrations.
Malgré ces écarts de performance, cette nouvelle implémentation en OCaml montre de bien meilleurs résultats que le solveur SAT historique du solveur SMT Alt-Ergo. En conséquence, nous avons décidé d'optimiser la combinaison d'un tel solveur SAT performant au sein d'un solveur SMT. Pour cette optimisation de la combinaison, nous avons eu recours à un filtrage des littéraux envoyés aux théories et au moteur d'instanciation. Ce filtrage s'inspire du solveur SAT historique d'Alt-Ergo dont le fonctionnement, proche des solveurs SAT, se base sur la méthode des tableaux analytiques. Après avoir formalisé ce fonctionnement via des algorithmes et prouvé notre optimisation, nous en démontrons les gains de performance à l'aide de résultats expérimentaux.
Nous poursuivons nos travaux en présentant une extension polymorphe au langage standardisé de la communauté SMT. Dans un premier temps, nous soulignons les modifications apportées à la syntaxe et au typage de celle-ci. Nous avons développé une bibliothèque permettant d'effectuer l'analyse lexicale et syntaxique de ce standard étendu. Le branchement de cette bibliothèque au sein d'Alt-Ergo a permis à ce dernier de traiter des fichiers au format du standard SMT-LIB. De plus, grâce à l'outil Why3, nous avons généré des fichiers sous différents formats. Nous démontrons ainsi l'intérêt de travailler sur des fichiers contenant du polymorphisme face à des fichiers monomorphisés.
Enfin, l'ajout de ce support nous a permis de lancer Alt-Ergo sur l'ensemble de bancs de test de la communauté. Nous avons participé à la compétition annuelle de la communauté SMT. Nous exposons les modifications apportées à notre solveur dans ce cadre, avant de présenter les résultats de la compétition et de conclure ce document.

Title : Improving the SMT solver Alt-Ergo performance with a better integrated efficient SAT solver

Keywords : SMT, SAT, OCaml, Alt-Ergo, SMT-LIB, polymorphism

Abstract : The first focus of this PhD thesis is to study the performance metrics of an OCaml language implementation of Minisat, which serves as our referent SAT solver, compared to a C++ one. To ensure the relevance of our work, our implementation of the SAT solver strictly mirrored the one in C++. We rationalized the performance disparities that could be observed between them with an in-depth study based on a collection of test cases combined to analysis tools.
Despite these performance gaps, the results of this OCaml integration are significantly improved compared to the historical SAT solver of the SMT solver Alt-Ergo. This promising start lead us to optimize the combination of such a SAT solver within a SMT solver. To this end, we sent a reduced set of literals to the theories and the instanciation engine. This filtering is based on the method of analytic tableaux, as this decision procedure worked well on the historical SAT solver of Alt-Ergo. Algorithms formalized our proposed optimization, which we then prove before presenting the experimental results and benchmarks of these performance gains.
We continue our work by introducing SMT-LIB 2, a polymorphic extension of the standardized language of the SMT community. We begin this section by highlighting the changes we made regarding the syntax and typing. We developed a library for parsing and typing this extension. The integration of this library within Alt-Ergo is essential to process files in the standard SMT-LIB extended format. The Why3 platform enables SMT-LIB 2 to generate files in different formats. This shows the benefits of working on files relying on polymorphism rather than monomorphism.
The addition of the aforementioned support allowed us to launch Alt-Ergo on the entire test suite of the community. To this end, we entered the yearly SMT community competition. We explain the upgrades our solver undertook for this event, then present the competition results and, lastly, conclude this document.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France