Vérification à contraintes de programmes avec tableaux

Peyiu Li

To cite this version:

Peyiu Li. Vérification à contraintes de programmes avec tableaux. Langage de programmation [cs.PL]. 2010. dumas-00530744

HAL Id: dumas-00530744

https://dumas.ccsd.cnrs.fr/dumas-00530744v1

Submitted on 29 Oct 2010

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

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

Vérification à contraintes de programmes avec tableaux

LI PeiYu
Encadreurs : Arnaud GOTLIEB et Sébastien BARDIN

4 juin 2010

Résumé

Ce stage de master de recherche a été effectué dans l'équipe Celtique du centre de recherche IRISA/INRIA Rennes pendant cinq mois (01/02/10-30/06/10). Dans ce stage, on s'intéresse à la vérification à contraintes de programmes avec tableaux. En combinant les points forts de deux méthodes existantes : la programmation par contraintes et le SAT/SMT, on crée un nouveau solveur qui teste la satisfiabilité des formules et qui renvoie un certificat pour les formules non satisfiables.

Résumé

This internship has been done in the team Celtique of IRISA/INRIA Rennes research center during five months (01/02/2010-30/06/2010). During this internship, we focus on the verification based on constraint of the programs with tables. Combining the advantages of two methods : constraint programming and SAT/SMT, we create a new solver which tests the satisfiability of different formulas and return a certificate for those who are non-satisfiable.

Table des matières

Remerciements **4**

1 Introduction **6**
1.1 Contexte 6
1.2 Problématique du stage 6

2 Etat de l'art **8**
2.1 Généralité sur la satisfiabilité 8
2.1.1 Logique du premier ordre et satisfiabilité 8
2.1.2 Théorie et les notions associées 8
2.1.3 Quelques exemples des théories 9
2.2 Procédure de décision de SMT 9
2.2.1 Procédure de décision de SAT 10
2.2.2 Congruence closure 10
2.2.3 Combinaison des théories 11
2.2.4 Procédure de décision de théorie des tableaux 13
2.3 Z3 : une procédure SMT exemplaire 14
2.4 Logiciels existants dans ces domaines 15

3 Technologies utilisées **17**
3.1 Les informations associées à chaque élément 17
3.1.1 L'algorithme congruence closure 17
3.1.2 Théorie des tableaux 19
3.1.3 Génération du certificat au cas unsatisfiable 19
3.2 Structures de données générales à mémoriser 21
3.3 Complémentarité des approches SMT et CLP 22

4 Implémentation **24**
4.1 Coder en Prolog 24
4.2 Implémentation de la version actuelle 25
4.3 Utilisation de clpfd 26

5 Résultat **28**

6 Travaux futurs **29**
6.1 Ajout de typage 29
6.2 Formules disjonctives 29

7 Conclusion **31**

Bibliographie **35**

Remerciements

Je tiens à remercier dans un premier temps, toute l'équipe et les intervenants de la formation Master de Recherche en informatique de Université de Rennes 1.

J'aimerais toute particulièrement remercier les personnes suivantes qui ont partagé leur expérience avec moi durant ces cinq mois :

Monsieur Arnaud GOTLIEB pour toute l'aide qu'il m'a donnés tout au long de ce stage, de l'encouragement aux critiques précieuses.

Monsieur Sébastien BARDIN pour m'avoir guidée et pour les conseils qu'il m'a donnée pour la correction du rapport.

Et Matthieu Carlier, mon collègue de bureau, pour le relecture de mon rapport final.

Je remercie également mes amis français qui m'ont aidé à corriger les fautes d'orthographe et de grammaire de mon rapport.

Chapitre 1

Introduction

1.1 Contexte

Mon stage de master est co-encadré par M. Arnaud GOTLIEB (équipe-projet INRIA Celtique) et M. Sébastien BARDIN (du laboratoire CEA LIST/LSL). Le programme de recherche de l'équipe Celtique vise à proposer des méthodes de certification sémantique de logiciels à l'aide d'analyse sémantique. Cette EPI (Équipe Projet INRIA) comprend un groupe travaillant dans le domaine du « constraint-based Testing », c'est à dire l'utilisation de techniques de résolution de contraintes pour la génération automatique de tests. Au CEA, les mêmes thématiques sont abordées mais sous un angle plus appliqué, avec la volonté de créer des outils ayant vocations à être transférés dans le monde industriel.

1.2 Problématique du stage

La vérification automatique de programmes est un sujet essentiel où certains comportements du programme sont évalués en testant la satisfiabilité des « conditions de vérification ». Ces conditions de vérification sont exprimées sous forme de formules. Lors de ce test, des tableaux de grande taille peuvent être manipulés. À cause des nombreux accès et mises à jour, les preuves deviennent alors coûteuses en consommation mémoire et en temps d'accès.

Ayant une formule qui représente des conditions de vérification, plusieurs techniques sont développées de nos jours. On s'intéresse en particulier à deux d'entre elles : l'approche CLP (Constraint Logic Programming) et l'approche SMT (Satisfiability Modulo Theories).

La CLP capture les relations entre variables et permet donc, de formuler des propriétés permettant le calcul de solutions. Dans CLP, les variables ont chacune leur propre domaine. Trouver une solution à un problème de CLP consiste donc à trouver une affectation de toutes les variables dans leur

domaine pour que toutes les contraintes soient satisfaites. Chaque fois que l'on apprend une nouvelle contrainte, on essaye de restreindre le domaine des variables. L'approche SMT consiste à déterminer la satisfiabilité de formules logiques du premier ordre en respectant certaines théories. Nous nous focaliserons de manière plus détaillée sur ce problème dans la partie état de l'art de ce rapport. Le domaine de SMT est bien développé. Une compétition sur les différents solveurs a lieu tous les ans (SMT-COMP [1]). Une librairie de benchmarks a été créée pour évaluer les meilleurs (SMT-LIB [2] et [7]). Les solveurs SMT sont réputés efficaces pour le traitement de tableaux [3]. Il est donc intéressant de pouvoir s'en servir pour améliorer l'approche CLP. Ainsi, nous nous sommes basés sur les connaissances de l'équipe Celtique en CLP. Puis nous avons apporté les techniques SMT sur le traitement des tableaux dans le domaine CLP. Cette combinaison permet d'atteindre l'objectif d'optimiser la façon de traiter les tableaux de grande taille venant de la vérification.

Ce rapport est organisé de la façon suivante. Tout d'abord, on verra quelques notions de SMT, et les études bibliographiques dans ces domaines de recherche. Puis, en utilisant ces notions, on éclairciera le sujet de stage et on expliquera les technologies utilisées dans la troisième partie. On expliquera aussi de quelques détails dans l'implémentation et les résultats. Ensuite, on discutera des travaux futurs avant de conclure.

---

1. http ://www.smtcomp.org/2010/
2. http ://goedel.cs.uiowa.edu/smtlib/
3. http ://www.smtexec.org/exec/divisionResults.php ?jobs=529&division=QF_AUFLIA avec la division QF_AUFLIA (quantifier-free, array, uninterpreted function and linear integer arithmetic)

Chapitre 2

Etat de l'art

2.1 Généralité sur la satisfiabilité

Les notions dans cette partie sont établies à l'aide de deux livres [1, 2].

2.1.1 Logique du premier ordre et satisfiabilité

Une formule de **la logique du premier ordre** (*FOL : First Order Logic*) est construite à partir d'un ensemble de variables ; de symboles de logique : $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$, (,) ; de symboles non logiques : constantes, prédicats et symboles de fonction. La différence sémantique entre une formule de la logique du premier ordre et celle de la logique propositionnelle (*PL : Propositional Logic*) est due à l'utilisation des quantificateurs ($\forall$, $\exists$) et de différentes théories dans FOL.

Satisfiabilité : Une formule est satisfiable s'il existe une structure qui peut la satisfaire. Cette structure s'appelle un modèle.

2.1.2 Théorie et les notions associées

Une **théorie** est définie par sa signature $\sum$ (l'ensemble des symboles non logiques) et son interprétation. Elle peut être décrite par un ensemble de phrases. Une théorie du premier ordre est donc un ensemble de phrases écrites en logique du premier ordre. Dans la suite, nous allons aussi parler de théorie du premier ordre contenant l'égalité. Ceci est une théorie du premier ordre contenant le symbole «=» et ses axiomes de réflexivité, de symétrie et de transitivité.

Nous avons aussi besoin de comprendre la validité de formules. Une $\sum$-formule θ est **T-valide** si toutes les structures qui satisfont les phrases définissant la théorie T satisfont aussi θ. Cela est noté : T $\vDash \theta$. Un **fragment** sans quantificateur libre d'une théorie T est l'ensemble des formules sans quantificateurs qui sont T-valides, sachant qu'une formule θ sans quantificateurs est valide si et seulement si sa fermeture $\forall X.\theta$ est valide (avec X

l'ensemble des variables dans θ).

Combinaison des théories : Etant donné deux théories T1 et T2 avec les signatures $\sum 1$ et $\sum 2$ respectivement, la combinaison T1$\oplus$T2 est une théorie avec signature $\sum 1 \cup \sum 2$ définie par l'ensemble d'axiomes T1$\cup$T2. Le problème de combinaison des théories consiste donc à décider si une $\sum 1 \cup \sum 2$-formule θ est T1$\oplus$T2-valide.

2.1.3 Quelques exemples des théories

Théorie d'égalité

$\sum_E$: =, a, b, c, ..., f, g, h, ..., p, q, r, ... qui est construit à partir du prédicat binaire «=» et tous les symboles de constante, fonction, prédicat. Les axiomes sont les suivants :

$\forall$x. x=x	(Réflexivité)
$\forall$x,y. x=y $\Rightarrow$ y=x	(Symétrie)
$\forall$x,y,z. x=y $\wedge$ y=z $\Rightarrow$ x=z	(Transitivité)
$x = y \Rightarrow \mathrm{f}(x) = \mathrm{f}(y)$	(Congruence)

Théorie des tableaux

La théorie des tableaux peut être extensionnelle ou non-extensionnelle. Dans l'approche extensionnelle, on peut raisonner au sujet des tableaux et de leurs éléments. On a donc la comparaison entre deux tableaux. Quant à l'approche non-extensionnelle, nous pouvons raisonner uniquement au sujet des éléments des tableaux. Le moyen indirect de les comparer sous la théorie non-extensionnelle est de comparer leurs éléments.

La théorie des tableaux a la signature $\sum_A$: { read, write, =}. La fonction read(t,i) renvoie la valeur t[i]. La fonction write(t,i,e) renvoie le tableau t en écrivant e à l'indice i et en conservant les autres éléments inchangés.

On conserve les trois premiers axiomes de la théorie d'égalité. De plus, on ajoute ces axiomes suivants :

(A1) i = j $\Rightarrow$ read(a,i) = read(a,j)	Congruence
(A2) i = j $\Rightarrow$ read(write(a,i,e),j) = e	read-over-write (1)
(A3) i $\neq$ j $\Rightarrow$ read(write(a,i,e),j) = read(a,j)	read-over-write (2)

Si on supporte aussi l'égalité entre les tableaux, on ajoute un quatrième axiome :

(A4) a = b $\Leftrightarrow$ $\forall$i (read(a,i) = read(b,i))   Extensionalité

2.2 Procédure de décision de SMT

Une procédure de décision de SMT prend une formule FOL et nous renvoie si elle est satisfiable ou pas. Il existe deux types de procédures pour

le SMT : «lazy» et «eager». Pour l'approche «lazy», on transforme la formule en une formule sur-approximée exprimée par des variables booléennes, et on passe la nouvelle formule dans un solveur SAT. S'il renvoie «unsat», la formule d'origine est aussi unsatisfiable ; sinon, on la raffine avec la théorie correspondante, et on répète les étapes précédentes. Cette approche est «lazy» car la formule propositionnelle est construite progressivement. Quant à l'approche «eager», on se sert de la connaissance de théorie de suite pour déduire une formule équisatisfiable. Cette approche est «eager» car la formule propositionnelle (avec des variables booléennes) est construite en une seule étape (Voir [2]).

En général, les formules testées par une procédure SMT ne sont pas exprimées dans une seule théorie ; elles sont souvent exprimées comme combinaison des formules de plusieurs théories. Entre deux théories, seule l'égalité de variables est partagée. Ainsi, la propagation de connaissances sur les termes d'une théorie à l'autre se fait uniquement au travers des égalités de variables.

2.2.1 Procédure de décision de SAT

Un solveur SMT est contruit à partir d'un solveur SAT et des traitements des différentes théories.

Le problème SAT (Boolean satisfiability problem) consiste à déterminer si les variables d'une formule booléenne peuvent être affectées de telle façon que la formule s'évalue à vrai. Imaginons une formule : $(x \lor y) \land (\neg x \lor y)$. Le problème SAT consiste à trouver la satisfiabilité de cette formule. Et on voit que si on pose x=vrai, y=vrai, la formule vaut vrai. Donc cette formule est satisfiable.

Beaucoup de travaux ont été fait pour résoudre les problèmes SAT, ses procédures de décision sont très robustes de nos jours. Un des algorithmes (procédures de décision) les plus connus est l'algorithme DPLL (*Davis-Putnam-Logemann-Loveland algorithm*) [1]. Cet algorithme vise à décider la satisfiabilité des formules de PL, et non de FOL. Le DPLL essaie de construire un modèle pour la formule donnée. Le modèle est construit en déduisant la vraie valeur d'un littéral à partir du modèle de l'étape précédente et de la formule, ou en devinant. Au cas où ceci implique une inconsistance, la procédure backtrack et essaie le sens opposé.

2.2.2 Congruence closure

La propagation d'égalité est un des points importants des procédures de décision SMT. Pour ceci, on utilise en général l'algorithme «congruence closure»(Voir [9] et Cours en ligne [2] de L. De Moura). Cet algorithme nous

1. http ://en.wikipedia.org/wiki/DPLL_algorithm
2. http ://research.microsoft.com/en-us/um/people/leonardo/oregon08.pdf

permet de maintenir les égalités et les diségalités entre les variables et de dire si une formule est satisfiable.

Une structure de données qui optimise cet algorithme est la structure de données «union-find» (Voir [2]). Le principe est de construire une structure (F) qui maintient les classes d'équivalence (qui varie d'une étape à l'autre bien entendu) et un ensemble de diségalité (D). La structure F nous permet de mémoriser «le représentant» de chaque classe d'équivalence. F(x) nous renvoie le représentant de la classe d'équivalence contenant x. Donc si pour deux variables, on a le même représentant, on sait qu'ils sont dans la même classe d'équivalence.

Quand on a une nouvelle égalité à ajouter sur les termes, on utilise l'opération «union» qui combine les différentes classes d'équivalence correspondantes. On stocke aussi une relation *size(F,x)* qui renvoie la taille de la classe d'équivalence qui contient x. *union(F,x,y)* change le représentant de la classe contenant x au représentant de la classe contenant y si *size(F,x)* est inférieur à *size(F,y)* ; sinon, l'inverse. On ne prend pas alternativement une classe à ajouter sur l'autre parce qu'on souhaite ajouter la plus courte sur la plus longue, ceci est moins couteux.

De plus, on stocke des ensembles des termes supérieurs (e.x. f(x) est terme supérieur de x) de tous les termes par π*(terme)*.

Le système entier :

1. Quand on a une égalité x=y,
   - Si F(x) = F(y), ne rien faire ;
   - Sinon, F' = union(F,x,y),
     - Si F'(u)=F'(v) pour certain u$\neq$v dans D, unsat ;
     - Sinon, pour tout couple (u,v) avec u $\in \pi$(F(x)) et v $\in \pi$(F(y)), si Congruent (u,v) = Vrai, on ajoute une nouvelle égalité u=v.
2. Quand on a une diségalité x$\neq$y,
   - Si F(x) = F(y), unsat ;
   - Sinon,
     - s'il existe un u, un v tels que F(x)=F(u), F(y)=F(v), et u$\neq$v $\in$ D, ne rien faire ;
     - Sinon, D=D $\cup$ {x$\neq$y}.
3. Congruent(u,v) :
   - Si u=f(x1,...,xn), v=f(y1,...,yn), si quelque soit i, F'(xi) = F'(yi), rend vrai ;
   - Sinon, rend faux.

2.2.3 Combinaison des théories

Comme expliqué plus haut, les solveurs SMT coupent la formule sous différentes théories. Pour que les théories s'interrogent entre elles, l'algorithme

le plus répandu est celui proposé par Nelson et Oppen (Voir [6]). Pour expliquer comment cela fonctionne, prenons le cas de combinaison des deux théories, sachant que cela fonctionne de la même façon pour une combinaison de plusieurs théories.

Cet algorithme travaille sous certaines contraintes. Premièrement la formule, exprimée par deux théories T1 et T2, doit être sans quantificateur. Deuxièmement, la signature des deux théories ne peut partager que l'égalité. Troisièmement, les deux théories doivent être infiniment stables, c'est-à-dire qu'elles sont interprétées sous un domaine infini. De plus, une théorie T est convexe si pour une formule F exprimée par cette théorie implique une disjonction d'égalités (e1$\vee \ldots \vee$en), alors F implique e_i pour un i $\in$ 1..n. Notons que la complexité de la partie exprimée sous T1 est *O(T1(n))*, celle de la deuxième est *O(T2(n))*. Cet algorithme a deux versions, une version avec la complexité $O(n^3 \times (T1(n) + T2(n)))$ pour des théories convexes, et une autre avec la complexité $O(2^{n^2} \times (T1(n) + T2(n))$ pour des théories non convexes.

Algorithme de Nelson & Oppen version convexe

1. Purification : on transforme la formule d'origine F en deux formules CNF (Conjunctif Normal Form) : une $\sum$1-formule F1 et une $\sum$2-formule F2 telles que F et F1$\wedge$F2 sont équisatisfiables.
2. Appliquer la procédure de décision de Ti à Fi. S'il existe un i tel que Fi est unsatisfiable sous Ti, renvoie «unsatisfiable».
3. Propagation d'égalité : après avoir déduit une égalité x=y sous T1, on met à jour F2 à F2$\wedge$x=y. Et vice versa. Retourne à l'étape 2.
4. Renvoie «satisfiable».

Algorithme de Nelson & Oppen version non-convexe

1. Purification.
2. Appliquer la procédure de décision de Ti à Fi. S'il existe un i tel que Fi est unsatisfiable sous Ti, renvoie «unsatisfiable».
3. Propagation d'égalité : après avoir déduit une égalité x=y sous T1, on met à jour F2 à F2$\wedge$x=y. Et vice versa. Retourne à l'étape 2.
4. «Splitting» : S'il existe un i tel que
   $F_i \Rightarrow (x_1{=}y_1 \vee \ldots \vee x_k{=}y_k)$ et $\forall$j$\in$1,..,k. $F_i \not\Rightarrow x_j{=}y_j$,
   on testera séparément les sous problèmes avec $F_{\neg i} \wedge x_1{=}y_1, \ldots, F_{\neg i} \wedge x_k{=}y_k$.
   S'il existe un de ces sous problèmes qui est satisfiable, renvoie «satisfiable» ; sinon, renvoie «unsatisfiable».
5. Renvoie «satisfiable».

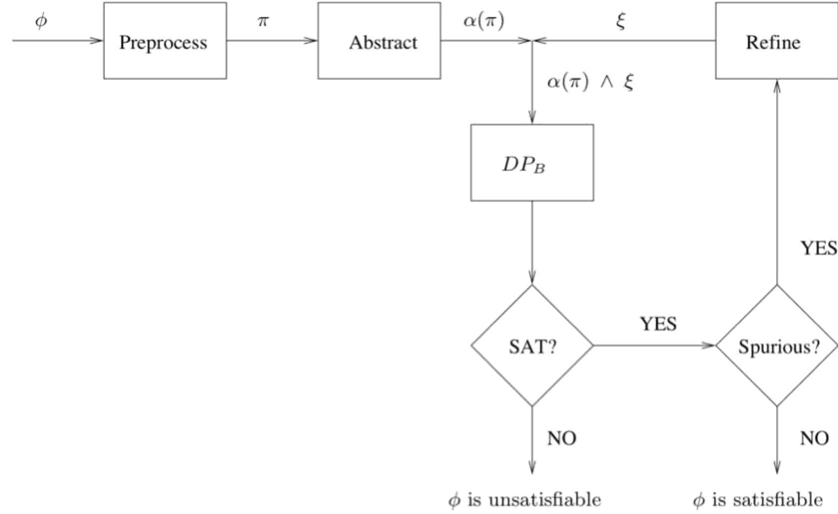

FIGURE 2.1 – Présentation graphique de la procédure proposée.

2.2.4 Procédure de décision de théorie des tableaux

Un exemple concret de solveur SMT pour la théorie des tableaux est présenté dans l'article [3].

La théorie des tableaux extensionnelle T_A a la signature $\sum_A$. On suppose qu'on a une théorie de quantificateur libre du premier ordre T_B avec la signature $\sum_B$. On suppose que $\sum_B \cap \sum_A = \{=\}$. Pour trouver la procédure de décision DP_A, on utilise la procédure de décision de T_B (DP_B). DP_B prend une $\sum_B$-formule et calcule la satisfiabilité selon le T_B. Si une $\sum_B$-formule est satisfiable, DP_B retourne un B-modèle σ avec l'affectation sur les variables, les termes et les formules à valeurs concrètes.

DP_A décide donc la satisfiabilité d'une $\sum_B \cap \sum_A$ -formule Φ selon le $T_A \cup T_B$. De plus, comme la plupart des procédures, on suppose que le type des indices et des valeurs d'un tableau sont tous «Base». Une variable de tableau et le «write» sont de type «Array». Le «read» est de type «Base». La figure 2.1 montre une présentation de la procédure proposée.

Tout d'abord, dans l'étape de préprocessing, on transforme la formule Φ en une formule Π équisatisfiable en faisant :

1. Pour toute égalité entre deux tableaux $a = c \in \Phi$, on introduit une nouvelle variable λ de type «Base» et deux «read» virtuels. On ajoute une nouvelle contrainte :

   (ci) $a \neq c \rightarrow \exists\lambda.\ read(a,\lambda) \neq read(c,\lambda)$

2. Pour tout $write(a,i,e) \in \Phi$, on ajoute une nouvelle contrainte :

   (cj) $read(write(a,i,e),i) = e$

La nouvelle formule à analyser est $\Pi := \Phi \wedge \bigwedge_{i=1}^{n} c_i$.

Deuxièmement, on applique une fonction partielle d'abstraction α. Cela a pour but de réduire la formule Φ. C'est une technique bien connue, mais la différence ici est qu'on utilise un squelette T_B comme abstraction de formule, à la place d'un squelette de P_L (logique propositionnelle). On obtient $\alpha(\pi)$ au final avec π comme entrée. On démontre que $\alpha(\pi)$ est une sur-approximation de π, c'est à dire $\alpha(\pi)$ peut inclure des modèles additionnels erronés par rapport à π. Donc si $\alpha(\pi)$ n'est pas satisfiable, π ne l'est non plus. Mais on doit trouver un moyen pour éliminer les modèles erronés quand $\alpha(\pi)$ est satisfiable.

La partie de raffinement est la partie de la procédure qui nous permet d'éliminer les modèles erronés. Quand on trouve un modèle erroné, on ajoute un lemme pour raffiner l'abstraction. La liste des lemmes se trouve dans l'article [5].

Cette procédure est correcte et complète. C'est à dire si $DP_A(\Phi)$ renvoie non-satisfiable, Φ est non-satisfiable selon $T_A \cup T_B$; si Φ est non-satisfiable selon $T_A \cup T_B$, $DP_A(\Phi)$ renvoie unsatisfiable. La complexité de cette procédure dépend de borne supérieure du nombre de lemmes à générer. Ce nombre est borné par $O(n^2 * 2^n)$ avec $n = |\Phi|$.

2.3 Z3 : une procédure SMT exemplaire

Beaucoup de papiers avec de nouvelles techniques de SMT ont été publiés pendant ces dernières années. Parmi eux, un des papiers avec un outil bien connu est celui développé par Leonardo de Moura et Nikolaj Bjørner de Microsoft Research qui s'appelle Z3. Il est utilisé dans beaucoup des logiciels de tests chez Microsoft. Pex, mentionné plus haut, en fait partie.

La structure interne de Z3 est présentée en figure 2.2 [5].

Z3 est capable d'analyser plusieurs sortes de langage en entrée : format SMT-LIB, format Simplify [6], etc. Tout d'abord, une formule à tester est passée dans l'étape Simplifer qui simplifie la formule d'une façon incomplète, mais efficace. Puis elle est convertie en plusieurs clauses des théories différentes. Ensuite comme d'autres solveurs, ces clauses sont analysées dans leurs propres solveurs en respectant les égalités et les diségalités propagés entre eux.

L'équipe de développement de Z3 n'a pas simplement pris les méthodes existantes pour construire ce solveur. Elle a créé ses propres algorithmes. Ici on ne va pas analyser tout le solveur entier. On ne regarde que son algorithme pour la combinaison des théories et son solveur de théorie des tableaux.

Plus loin en haut, on a distingué la différence entre une théorie convexe et une théorie non convexe. La combinaison des théories Nelson-Oppen a une façon efficace de travailler avec des théories convexes. Mais avec les théories

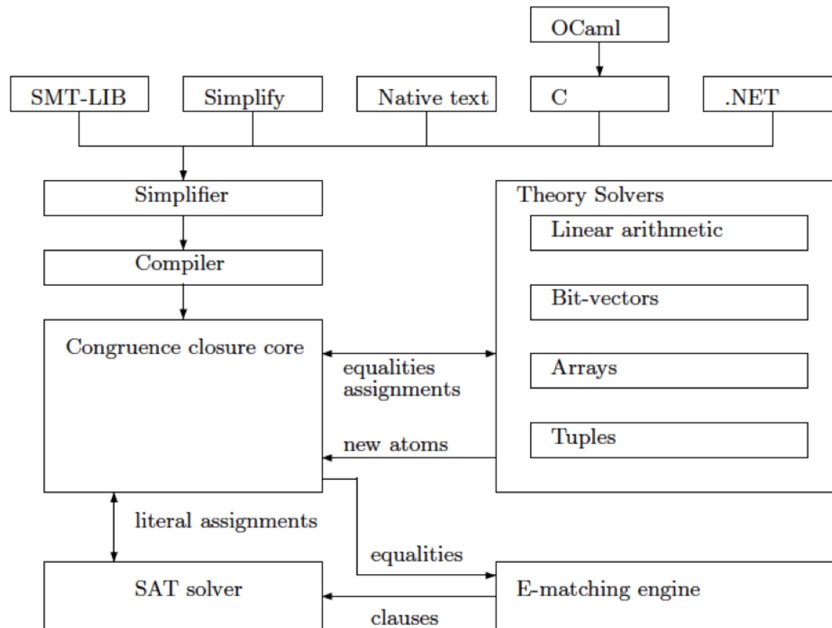

FIGURE 2.2 – Structure interne de Z3.

non convexes, cet algorithme obtient une disjonction des égalités, et il doit les tester un par un, ce qui n'est pas assez efficace. Leonardo et al. ont donc proposé une nouvelle façon de combiner les théories dans [4]. Ceci fonctionne de façon similaire pour les deux types de théories.

Pour chaque théorie Ti, on maintient un modèle Mi pour la signature Γi ou un sous ensemble de Γi. De temps en temps, si on a $u^{Mi} = v^{Mi}$, on crée le cas u=v, et on teste avec l'égalité d'abord. On propage u=v, si cela implique insatisfiabilité, on backtrack et on insère u≠v. Puis on reconstruit un modèle et on recommence.

Quant au solveur de théorie des tableaux présenté dans l'article [7], Z3 génère des règles d'inférence pour transformer étape par étape les clauses qu'il reçoit.

2.4 Logiciels existants dans ces domaines

Dans le domaine de la vérification et du test, de nombreux travaux ont déjà été faits. Des outils ont également été créés à partir de ces connaissances. Par exemple, PEX [3], développé par Microsoft, est un outil de test unitaire automatique pour le langage .NET. Le test unitaire est un procédé permettant de s'assurer du fonctionnement correct d'une partie d'un logiciel ou d'une portion de programme. Il est intégré avec l'un des solveurs SMT :

3. http ://research.microsoft.com/en-us/projects/pex/

Z3 [4]. PEX utilise Z3 pour générer les données de tests unitaires.

Un autre exemple est le logiciel OSMOSE [5]. OSMOSE est un outil de génération automatique de cas de test développé par Sébastien Bardin qui travaille au niveau du binaire. Cet outil utilise une approche qui combine l'exécution symbolique et l'exécution concrète pour trouver des cas de test qui activent les chemins d'un code binaire. OSMOSE utilise une résolution SMT, en particulier sur la théorie des "bit-vectors", pour identifier de nouveaux cas de test.

Euclide [6] est un outil de test basé sur les contraintes pour vérifier les programmes C critiques. Cet outil est développé par Arnaud Gotlieb. Lorsque Euclide travaille sur un test de satisfiabilité des conditions de vérifications, des structures de données de grande taille peuvent entrer en jeu.

4. http ://research.microsoft.com/en-us/um/redmond/projects/z3/index.html
5. http ://sebastien.bardin.free.fr/Research-executables.html
6. http ://euclide.gforge.inria.fr

Chapitre 3

Technologies utilisées

3.1 Les informations associées à chaque élément

3.1.1 L'algorithme congruence closure

Dans la suite, considérons seulement des formules conjonctives.

Comme dit plus haut, on souhaite mémoriser des classes d'équivalence et un ensemble de diségalités. Si on utilisait une liste ou un tableau, on ne pourrait pas maintenir facilement la relation entre un élément et son représentant. Une structure de donnée qui modélise bien cet algorithme est la table de hachage.

Dans la suite, considérons un exemple «$a = b, b = c, f(c) = f(d), f(f(a)) \neq f(f(d))$ ».

Solution 1 : on stocke une table de hachage qui maintient la relation entre un élément (clé) et son représentant (valeur).

Élément	Représentant
a	a
b	a
...	...

L'inconvénient : on ne peut pas obtenir facilement la classe d'équivalence quand on donne un représentant.

Solution 2 : on stocke une table de hachage qui a des éléments comme clés et un ensemble signifiant la classe d'équivalence quand l'élément est un représentant, un ensemble vide sinon.

Élément	Classe d'équivalence
a	{a,b,c}
b	{}
...	...

L'inconvénient : on ne peut pas savoir facilement le représentant d'un élément.

Solution retenue : on stocke une table de hachage qui a des éléments comme clés et un terme *t(Rep, ClassEq)* associé à chaque terme. *Rep* signifie son représentant, *ClassEq* est la classe d'équivalence si cet élément est un représentant ou ensemble vide sinon.

Élément	Terme
a	t(a,{a,b,c})
b	t(a,{})
...	...

Comme d'autres techniques de «union-find », à l'ajout d'une classe d'équivalence sur une autre, on choisit de garder la classe la plus grande comme racine et on lui rajoute la deuxième classe. Ceci nous demande d'avoir la taille de chaque classe d'équivalence. Une façon efficace est de mémoriser cette valeur (*taille*) directement dans le terme t des représentants.

Élément	Terme
a	t(a,{a,b,c},3)
b	t(a,{},0)
...	...

Au moment de combiner deux classes, sachant que ces deux classes ne partagent aucun élément commun, la valeur à ajouter sur l'attribut taille de la plus grande classe parmi les deux est la taille de la plus petite, i.e. l'attribut *taille* de la deuxième.

Cette solution couvre l'axiome de réfléxivité, de symétrie et de transitivité. Mais elle ne couvre pas l'axiome de congruence. Pour ce dernier, on mémorise les termes supérieurs. On ajoute donc un troisième paramètre (*termeSup*) dans le terme t qui indique les termes supérieurs correspondants. Pour faciliter la recherche de correspondance plus tard, on stocke les termes supérieurs d'un élément dans le terme t de son représentant.

Élément	Terme
a	t(a,{a,b,c},3,{f(a),f(c)})
b	t(a,{},0,{})
...	...

3.1.2 Théorie des tableaux

Tout d'abord, nous remarquons que la théorie des tableaux n'est pas convexe. Par exemple, considérons la formule $read(write(a,i,e),j) = e$. Ceci implique $i = j \bigvee read(a,j) = e$. Mais nous n'avons ni $i = j$, ni $read(a,j) = e$. Nous voulons trouver une solution pour ne pas utiliser la disjonction, parce que ceci n'est pas encore traité dans notre solveur. Nous avons ainsi inventé une notion de *evaluation retardée*. Ce que nous appellons une evaluation retardée est une contrainte qui ne sera ajoutée dans la liste de contraintes que si nous observons une diségalité entre un élément et l'indice de tableau sur laquelle nous avons fait une mise à jour.

D'après l'axiome de la théorie des tableaux, quand nous avons un $a' = write(a,i,e)$, une égalité est déduite directement : $read(a',i) = e$. De plus, nous avons $\forall k.\, k \neq i \Rightarrow read(a,k) = read(a',k)$. Mais tant que nous n'avons pas ce k, nous ne pouvons pas écrire l'égalité impliquée. Au moment où nous détectons un $j \neq i$, nous pouvons réveiller la contrainte $read(a',j) = read(a,j)$, et nous l'ajoutons dans la pile de contraintes. Nous devons donc enregistrer ce *i, a', a* quelque part. Pour cela, nous stockons ces couples de tableaux (nom de cet attribut : *contrainteRetardee*) sur qui les contraintes retardées apportent dans le terme t associé à l'indice concernée.

Élément	Terme
i	t(i,{i},1,{read(a',i)},{(a,a')})
...	...

Un élément i peut être utiliser comme index plusieurs fois. Par exemple, $write(t,\ i,\ e) = t1$, $write(t,\ i,\ f) = t2$. Dans cet exemple, nous avons deux contraintes retardées : $\forall k.\, k \neq i \Rightarrow read(t,k) = read(t1,k)$ et $\forall k.\, k \neq i \Rightarrow read(t,k) = read(t2,k)$. Nous avons donc deux couples (t,t1) et (t, t2) à associer à i. C'est pour cela que cet nouvel attribut est sous forme d'un ensemble de couples.

3.1.3 Génération du certificat au cas unsatisfiable

On ne se contente pas de savoir seulement si une formule est satisfiable ou pas. On souhaite aussi obtenir un certificat lorsque la formule est unsatisafiable.

Par exemple, si on a une formule : $a = b$, $b = c$, $d = e$, $f(a) \neq f(c)$. Cette formule est unsatisfiable, car $a = b$ et $b = c$ impliquent $a = c$ qui implique $f(a) = f(c)$. Ceci contredit $f(a) \neq f(c)$. Puisque $d = e$ ne joue aucun rôle ici, le certificat est « $a = b$, $b = c$, $f(a) \neq f(c)$ ».

Mais comment savoir quelles égalités sont utiles ? Robert Nieuwenhuis et al. ont proposé une solution dans [10]. Dans cet article, ils présentent les classes d'équivalence sous forme des arbres (une forêt) comme présentés sur

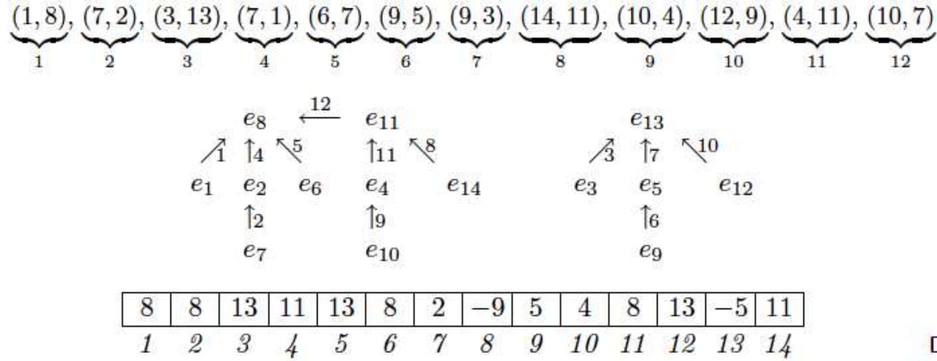

Figure 3.1 – Les classes d'équivalence sous forme d'arbres.

la figure 3.1. Ici on montre un exemple avec douze opérations de «union ». La première ligne représente les couples d'égalité (« union ») entre les éléments avec leur numéro d'affectation. Puis on les présente sous forme d'arbres. On maintient aussi un tableau qui indique le parent de chaque fils et une valeur négative pour un élément qui n'a pas de parents. Attention, le mot « *parent* » ici est au point de vue d'égalité, c'est différent de la notion « *terme supérieur* » mentionnée plus haut.

L'opération qu'on souhaite supporter est $explain(e, e')$ qui renvoie le sous ensemble minimal E d'une séquence « $(e1, e1'), \dots, (ep, ep')$» tel que (e, e') appartient à la relation d'équivalence générée par E et renvoie$\perp$ sinon. Par exemple, $explain(e2, e10) = \{(e10, e7), (e7, e2)\}$.

Pour trouver ce sous ensemble minimal E, on a un lemme important : considérons une structure de donnée sous la forme forêt. Pour chaque couple des éléments (e, e') avec le parent commun le plus proche c, le plus récent union (a, b) parmi les chemins de e à c et de e' à c appartient à $explain(e, e')$. De plus, (a, b) est le plus récent union dans $explain(e, e')$.

Avec ce lemme, une fois trouvé le plus récent union (a, b) pour $explain(e, e')$, il suffit de continuer avec les deux appels récursifs $explain(e, a)$ et $explain(b, e')$ pour trouver le reste des unions. Cela peut aussi être $explain(e, b)$ et $explain(a, e')$. Pour savoir quel cas il s'agit entre les deux, il faut respecter le fait que le chemin le plus récent des deux sous appels ne peut pas être plus récent que celui de $explain(e, e')$.

Pour ceci, on modifie notre structure. On ajoute un attribut dans le terme t de chaque élément qui mémorise son parent (*parent*), et un autre attribut qui mémorise le numéro d'union qui a affecté cette relation(*numeroEg*). Par défaut, l'attribut *parent* d'un élément neutre est lui-même et l'attribut *numeroEg* est 0. On arrive avec une table de hachage avec des éléments comme clés et un terme *t(rep, classEq, taille, termeSup, contrainteRetardee, parent, numeroEg)*. Prenons l'exemple : «$a = b, f(a) = f(c), write(tab, i, e) = tab'$ », la table de hachage est présentée sur la figure 3.2.

Élément	Terme
a	t(a,{a,b},2,{f(a)},{},a,0)
b	t(a,{},0,{},{},a,1)
c	t(c,{c},1,{f(c)},{},c,0)
f	t(f,{f},1,{f(a),f(c)},{},f,0)
f(a)	t(f(a),{f(a),f(c)},2,{},{},f(a),0)
f(c)	t(f(a),{},0,{},{},f(a),2)
i	t(i,{i},1,{read(tab',i)},{(tab,tab')},i,0)
tab	t(tab,{tab},1,{},{},tab,0)
e	t(e,{e,read(tab',i)},2,{},{},e,0)
read	t(read,{read},1,{read(tab',i)},{},read,0)
read(tab',i)	t(e,{},0,{},{},e,3)
tab'	t(tab',{tab'},1,{read(tab',i)},{},tab',0)

FIGURE 3.2 – Table de hachage d'éléments

Avec cette structure, on voit qu'un nouvel élément peut apparaitre soit via une égalité avec un autre élément, soit via une diségalité avec un autre élément, soit via l'apparition d'un terme supérieur. Au moment où on ajoute un nouveau élément dans notre table de hachage, on ajoute automatiquement tous ses termes inférieurs, et on doit regarder s'il existe une congruence entre lui et un élément existant.

3.2 Structures de données générales à mémoriser

Tout d'abord, la table de hachage présentée ci dessus doit être maintenue tout au long de la résolution (*classe_eq*). Pour l'algorithme de congruence closure, on stocke aussi un ensemble de couples de diségalité (*diff*). De plus, on ne veut pas que l'unsatisfiabilité de la formule provoque l'arrêt de notre solveur, on stocke la satisfiabilité dans un attribut (*satisfiability*). Cet attribut est mis à « satisfiable » par défaut. Ainsi, la procédure continue à analyser la suite de formule, mais dès qu'elle rencontre un endroit qui provoque l'insatisfiabilité de la formule, elle met l'attribut à « unsatisfiable ». Et on stocke les certificats pour les cas unsatisfiables dans un ensemble des ensembles car il peut y avoir plusieurs raisons d'insatisfiabilité pour une formule. Par exemple, pour la formule « $a = b$, $x = y$, $f(a) \neq f(b)$, $g(a,x) \neq g(a,y)$ », le certificat est $\{\{a = b,\ f(a) \neq f(b)\},\{x = y,\ g(a,x) \neq g(a,y)\}\}$.

Comme présenté dans la partie 3.1.3, il existe un ordre d'égalité pour savoir quelle égalité est plus récente qu'une autre, et cela influence la génération de certificats. On stocke donc une table de hachage des égalités (*table_egal*) et une valeur qui indique l'ordre courant et qui incrémente à chaque apprentissage d'une nouvelle égalité (*ordre_egal*). Mais on sait aussi que certaines égalités après l'analyse de l'algorithme congruence closure sont déduites et

ajoutées sur la pile. Ces égalités prennent aussi un numéro d'ordre, alors qu'elles ne doivent pas être envoyées comme une partie de certificat si c'est le cas, il faut continuer à chercher les égalités d'origine (Voir l'exemple 3.2). La table de hachage a donc le couple d'égalité comme clé et un terme qui inclut son numéro d'égalité et un ensemble des égalités qui ont deduit cette égalité. Prenons l'exemple 3.2, ceci donne :

Égalité	Terme
egal(a,b)	terme(1,{})
egal(x,y)	terme(2,{})
egal(f(a),f(b))	terme(3,{egal(a,b)})
egal(g(a,x),g(a,y))	terme(4,{egal(x,y)})

3.3 Complémentarité des approches SMT et CLP

Après avoir lu les articles et après avoir créé notre solveur, on voit qu'une procédure de décision de l'approche SMT reconstruit la formule sous différentes théories. Elle résout chaque partie de la formule dans sa théorie correspondante et propage l'égalité ou la diségalité détectée dans les autres parties de la formule. Cela fonctionne en conservant des classes d'équivalence et un ensemble de diségalité. Quant à l'approche CLP, chaque fois où on apprend une nouvelle contrainte, on essaye de restreindre le domaine des variables. Grâce à ceci, elle est plus intéressante que l'approche SMT dans certains types de problèmes. Par exemple, x et y se trouvent tous les deux entre 3 et 7. Quand on a une contrainte « $x < y$ », CLP est capable de restreindre le domaine de x à [3..6] et celui de y à [4..7]. L'approche SMT ne peut pas le déduire.

La propagation d'égalité est donc un avantage de l'approche SMT par rapport à l'approche CLP. Par exemple, sous l'approche CLP, quand on apprend une contrainte $x = y$, on peut restreindre le domaine de x et de y en prenant l'intersection de leur domaine initial ; quand on ajoute la contrainte $x \neq y$, puisque pour chaque élément du domaine de x, on peut trouver un élément du domaine de y qui est différent de ce premier, donc on n'apprend rien de nouveau sur le domaine des variables et on n'est pas capable de dire que cette formule ($x = y$, $x \neq y$) est insatisfiable. Si on utilise l'approche SMT, à la première étape, on combine la classe d'équivalence de x et de y. On est donc capable de renvoyer « unsatisfiable » au moment où on ajoute le diségalité entre des éléments de même classe d'équivalence.

On déduit qu'avec le SMT, on peut ajouter facilement la procédure d'une théorie sur une procédure SMT déjà faite, car les procédures de différentes théories sont indépendantes entre eux, et il suffit d'ajouter ses axiomes et le reste n'a pas besoin d'être touché. C'est aussi pour cela qu'il manque

des communications directes entre le solveur de chaque théorie. Ensuite, on observe une grande capacité de CLP pour résoudre la théorie LIA (*Linear Integer Arithmetic*). C'est pour cela qu'on peut se servir de CLP pour implémenter un solveur de la théorie LIA dans notre solveur principal. Pour mieux comprendre, nous le présentons plus en détails dans la partie « 4.3 Utilisation de clpfd ».

Chapitre 4

Implémentation

Comme dit plus haut, on souhaite profiter des points forts de l'approche SMT et s'en servir dans la programmation par contraintes. Un des langages connus pour la programmation par contraintes est Prolog.

4.1 Coder en Prolog

Prolog a été créé par A. Colmerauer et P.Roussel vers 1972. Prolog est très différent des autres langages de programmation, car il utilise directement l'expressivité de la logique. On l'a utilisé pour coder notre solveur SMT.

Il existe plusieurs IDE pour Prolog : Eclipse, SwI-Prolog, etc. L'IDE utilisé ici est SICStus. Si on compare SICStus avec les autres IDE, une des raisons principales pour lesquelles on l'a choisit est la licence et la pérennité. La licence académique de SICStus est peu onéreuse ($<$1000EUR) et nous autorise à redistribuer des runtime gratuitement. SICStus est développé par un institut de recherche semi-public en Suède mais qui met un point d'honneur à maintenir correctement ses logiciels. De plus, SICStus nous fournit beaucoup de librairies qui nous facilitent l'implémentation. Certaines librairies sont présentées dans la section suivante. Il existe plusieurs versions de SICStus, il y a beaucoup de changements entre chaque version. Pour rester cohérent avec certains librairies, on utilise la version SICStus 3.

Sous Prolog, les variables sont écrites en commençant par une majuscule, sinon ce sont des atomes. Un prédicat (*predicat/n*) a n attributs parmi lesquels il y en a qui sont en entrée, il y en a qui sont en sortie et il y en a qui peuvent être les deux.

4.2 Implémentation de la version actuelle

Prolog nous permet d'avoir des variables attribuées [1]. Avec la librairie « *atts* », on déclare les attributs qui peuvent exister pour les variables apparues dans le codage, dans la suite, on peut les récupérer ou les mettre à jour avec des simples commandes. Il y a une risque d'utiliser cette librairie. Lorsqu'on a certains attributs déclarés pour un type de variables, et certains autres attributs déclarés pour un autre type de variables, la librairie ne peut pas les distinguer facilement, et cela peut provoquer parfois des erreurs. Pour notre solveur, je souhaite simplement maintenir une seule variable attribuée, c'est la variable d'environnement (*ENV*). Ceci ne pose pas de problème à la librairie. C'est dans cette variable où on stocke toutes nos structures de données.

Pour implémenter les ensembles, on avait le choix parmi plusieurs librairies. Finalement on a pris la librairie *Ordsets* qui donne des ensembles ordonnés non-dupliqués. C'est moins coûteux qu'utiliser la librairie *lists* car si on prenait la deuxième, on devrait parcourir l'ensemble plus de fois pour supprimer les redondances.

De plus, la table de hachage est déjà implémentée en SICStus Prolog. Elle est codée dans la librairie *modpls*. Les prédicats qui nous intéressent sont :

i_new/2 : créer une table de hachage.

i_get/3 : récupérer la valeur ayant une clé.

i_del/3 : supprimer la valeur associée à une clé.

i_set/3 : ajouter une nouvelle couple(clé, valeur) dans la table.

Voici le déroulement du test de satisfiabilité. Avant de tester la satisfiabilité, on initialise d'abord l'environnement (*ENV*). On crée les tables de hachage (*classe_eq, table_egal*). On associe les attributs, qui sont des ensembles, à un ensemble vide (*diff, certificat*). Et on met l'attribut *satisfiability* à « sat » et le compteur du numéro d'égalité (*ordre_egal*) à 0. À chaque apprentissage d'une clause de la formule à tester, on ajoute *ENV* dans l'appel, pour récupérer les informations stockées et pour le mettre à jour.

Quand on a une clause d'égalité, si les deux éléments sont déjà dans la même classe d'équivalence, on ignore cette égalité redondante. Les axiomes d'égalité doivent être testés à la fin de chaque nouvelle clause d'égalité ou de diségalité. Quant à l'implémentation de disjonction, puisque le changement affecté sur ENV par un appel de disjonction n'est pas définitif, les appels à l'intérieur ne peuvent pas modifier directement ENV, ils travaillent avec une copie de la variable d'environnement.

1. http ://www.sics.se/sicstus/docs/latest3/html/sicstus.html/Attributes.html#Attributes

4.3 Utilisation de clpfd

Sous Prolog, la librairie sur CLP qui travaille avec des entiers s'appelle «clpfd », avec fd qui signifie «finite domaine ». Dans clpfd, il existe deux types de contraintes : contraintes globales et contraintes primitives. On peut ajouter des contraintes du premier type en passant par une interface de programmation, et ces contraintes sont ajoutées dans la pile de clpfd. Par exemple, pour le logiciel Euclide, Arnaud GOTLIEB a ajouté des contraintes globales pour travailler avec des tableaux. Le deuxième type de contraintes peuvent être ajouté sous une forme spéciale en utilisant des prédicats prédéfinis.

On observe qu'il existe le connecteur logique «ou » ($\#\backslash/$) comme prédicats en clpfd, mais on ne peut pas s'en servir directement pour analyser les disjonctions dans la formule. Par exemple, si on a une formule $a = b \vee a = c$, on ne peut pas obtenir la satisfiabilité de cette formule en faisant simplement $a = b \; \#\backslash/ \; a = c$. En effet, seulement des entiers et des variables sont permis dans les contraintes de domaine fini, alors que notre formule est basée sur des atomes. Donc nous gardons nos prédicats qui traite la disjonction des clauses dans la formule.

Donc pour combiner cette librairie avec notre solveur initial, nous corrigeons notre table de hachage de classes d'équivalence, nous associons à chaque élément une variable fraîche (nom d'attribut : *variable*) et on la stocke en plus dans le terme t. Ceci est présenté comme la table ci dessous.

Élément	Terme
a	t(A,_,_,_,_,_,_,_)
b	t(B,_,_,_,_,_,_,_)

Remarque : en réalité, le nom de variable fraîche n'a pas aucune lien avec le nom d'élément. Car cette variable est créée d'une façon aléatoire pour tout le monde. Donc pour l'élement a, la variable n'est pas forcément A.

On crée une variable pour chaque élément, on propage toutes les informations que clpfd peut traiter dans sa pile, les égalités, les relations arithmétiques, etc.

Pour la théorie des tableaux, considérons un exemple $read(t, i) = v1$, $read(t, j) = v2$, $i = j$. La librairie clpfd ne peut pas détecter $v1 = v2$. Alors que l'algorithme de congruence closure sous SMT nous permet de le déduire facilement. C'est pour cela que c'est intéressant que propager cette égalitée à clpfd. De plus, la contrainte *element/3* de la librairie clpfd peut traiter une lecture de tableau (ex : *read(t,i)=e*). Il faut faire attention que le tableau t doit être créer avant d'appeler cette contrainte. Mais cette contrainte ne peut pas travailler avec une écriture de tableau. Donc au moment d'une écriture, nous

traitons d'abord le tableau avec la façon proposée plus haut (contrainte retardée) dans la partie SMT, et qu'on envoie seulement des égalités de lecture dans la partie clpfd.

Pour traiter toutes les contraintes apparues dans les formules à tester, nous devons les redéfinir tout à la main. En effet, au moment d'entrer la formule à tester, elle doit être passée par la partie SMT et la partie clpfd, et que ces deux parties se propagent des informations pour s'aider. Par exemple, SMT envoie les égalités déduites à clpfd, et la capacité de réduire les domaines des variables de clpfd peut propager ses connaissances à SMT.

Si nous souhaitons avoir un modèle pour les formules satisfiables, nous pouvons nous servir du prédicat *labeling/2* de clpfd. Le pré-requis est de donner un domaine à toutes les variables avant de l'appeler. Mais nous pouvons aussi tenter à faire notre *labeling* dans la partie SMT.

Ce qui est présenté ici est notre première vision de traitement, cela peut évoluer éventuellement.

Chapitre 5

Résultat

En utilisant les techniques présentées ci dessus, on teste la satisfiabilité des formules.

- $a = b, c = d$.
  - Satisfiable. Temps d'exécution : 0.078sec. Temps d'exécution Z3 : 0.007sec.
- $a = b, c = d, d = e, b = e, b \neq c$.
  - Unsatisfiable. Certificat : $c = d, d = e, b = e, b \neq c$. Temps d'exécution : 0.078sec. Temps d'exécution Z3 : 0.003sec.
- $a = b, f(x) = f(y), g(a, x) \neq g(b, y)$.
  - Satisfiable. Temps d'exécution : 0.078sec. Temps d'exécution Z3 : 0.004sec.
- $read(t, i) = e, write(t, i, e) = t', i = j, read(t', j) = read(t, i)$.
  - Satisfiable. Temps d'exécution : 0.078sec.
- $write(t1, 1, a) = t2, write(t2, 2, b) = t3, write(t3, 3, c) = t4, write(t4, 4, d) = t5, write(t5, 5, e) = t6, write(t6, 6, f) = t7, write(t7, 7, g) = t8, write(t8, 8, h) = t9, read(t9, 1) = a, read(t9, 2) = b, read(t9, 3) = c, read(t9, 4) = d, read(t9, 5) = e, read(t9, 6) = f, read(t9, 7) = g, read(t9, 8) = h$.
  Satisfiable. Temps d'exécution : 0.078sec.

Remarque : notre solveur est en cours de développement. Des exemples plus significatifs pourront être présentés au jour de la soutenance.

Chapitre 6

Travaux futurs

6.1 Ajout de typage

On voit sur la figure 3.2 que même les symboles de fonction sont pris en compte comme des éléments normaux, et qu'il n'y a pas de différence entre un élément signifiant un tableau et un élément d'indice. Du coup on peut travailler avec des formules comme « $f(a) = f(b)$, $f = c$, $write(t, i, e) = t'$, $diff(t, i)$ », alors que cette formule n'a aucune sens. On souhaite donc donner du typage pour les éléments. On ajoute un autre attribut dans le terme *t()* de chaque élément (*type*). Cet attribut peut prendre trois valeurs différentes : *neutre* (par défaut), *tableau*, *fonction*. Ceci peut évoluer en fonction de notre besoin. On dit que la valeur *neutre* est la racine de la valeur *tableau* et *fonction*. Les règles sont listées en bas.

1. On ne peut jamais avoir de comparaison entre un élément de type *fonction* et autre chose. Même une égalité entre deux fonctions n'est pas permise. C'est pour éviter l'égalité entre deux fonctions d'arité différente.

2. On ne peut jamais avoir de comparaison entre un élément de type *tableau* et autre chose. Même une égalité entre deux tableaux n'est pas permise. Ceci est dû au fait qu'on ne s'intéresse à la théorie des tableaux non-extensionnelle, donc on ne traite pas d'égalité entre les tableaux.

Quand on a une égalité ou une diségalité à ajouter, on vérifie d'abord si les deux règles ici sont respectées. Puis on continue sur la procédure normale.

Cette modification ne change pas la correction de notre solveur et de plus, elle rend les formules à tester plus significatives.

6.2 Formules disjonctives

Avec tout ce qu'on a présenté ci dessus, notre solveur ne peut résoudre que des formules conjonctives. Pour élargir le capacité de notre solveur, un

algorithme qui permet de traiter des formules avec des disjonctions est indispensable. On utilise l'algorithme DPLL présenté dans la partie « État de l'art ».

Tout d'abord, pour utiliser algorithme DPLL, il faut que la formule soit sous la forme CNF (Conjunctive Normal Form). Une formule est sous la forme CNF si elle est une conjonction de clauses où chaque clause est écrit en une disjonction des littéraux (des atomes ou de la négation des atomes). Dans le livre [11], un algorithme efficace pour traduire une formule FOL en forme CNF est proposé.

Quand le solveur de chaque théorie renvoie de nouvelles égalités ou des diségalités à l'algorithme de congruence closure, il transmet ces informations sous forme des variables propositionnelles au solveur DPLL et le solveur DPLL les affecte pour chercher la satisfiabilité.

En comparant les résultats obtenus par notre solveur et le solveur Z3, on voit que le nôtre est loin d'être optimisé. Notre solveur ne peut travailler qu'avec des formules de certaines théories. C'est insuffisant par rapport aux autres solveurs bien développés maintenant. De plus, toutes les techniques qu'on utilise ici ne sont pas originales. Certaines méthodes mentionnées plus haut ne sont pas implémentées dans le solveur à la livraison de ce rapport. Dans les travaux futurs, on pense à implémenter des algorithmes qui permettent à notre solveur de résoudre la satisfiabilité des formules plus compliquées. Et on doit enrichir notre algorithme, qui combine l'approche SMT et l'approche CLP, pour optimier le traitement.

Chapitre 7

Conclusion

Nous avons vu à travers de nombreux travaux existants sur le SMT leurs façons de traiter les formules, leurs méthodes de les découper et regrouper les informations utiles. En se basant sur ceci, nous avons implémenté notre propre solveur. Nous avons commencé par la base, puis élargir le domaine des formules qu'on peut traiter en ajoutant des traitement des différentes théories ou en ajoutant le traitement de disjonction. Mais on voit aussi que ceci n'est pas suffisant pour livrer un bon solveur. De nouveaux algorithmes optimisés doivent être créés, combinant les deux approches pour apporter une réelle innovation par rapport aux autres solveurs de la vérification.

Table de glossaire

CLP : Constraint Logic Programming
SAT : boolean Satisfiability problems
SMT : Satisfiability Modulo Theory problems
PL : Propositional Logic
FOL : First Order Logic
Algorithme DPLL : Algorithme Davis-Putnam-Logemann-Loveland algorithm

Annexe

Liste des prédicats reconnus par le solveur

Rappel : pour tous les prédicats, le premier attribut est la variable d'environnement

init/1

egal/3

diff/3

access/4

update/5

or/3

inf/3

sup/3

neg/3

add/4

multiply/4

Exemple d'appel : init(ENV), update(ENV, t, i, e, t'), or(ENV, egal(ENV, i, j), diff(ENV, read(t,i), read(t',i)).

Annexe

Exemple de prédicats en Prolog

```
% init/1 %
% init(ENV) initialise la variable d'environnement ENV %
init(ENV) :-
 put_atts(ENV,[+(certificat([]))]),
 put_atts(ENV,[+(diff([]))]),
 put_atts(ENV,[+(satisfiability('sat'))]),
 i_new(REP,97),
 put_atts(ENV,[+(classe_eq(REP))]),
 i_new(Tegal,97),
 put_atts(ENV,[+(table_egal(Tegal))]),
 put_atts(ENV,[+(ordre_egal(0))]).
```

```
% access/4 %
% access(ENV,T,I,E) affecte l'élément à l'indice I du tableau T à E %
access(ENV,T,I,E) :-
 egal(ENV,read(T,I),E).
```

```
% update/6 %
% update(ENV,REP,T,I,E,T1) crée un nouveau tableau T1 qui met E à l'indice I et qui recopie le reste du tableau T %
update(ENV,T,I,E,T1) :-
 get_atts(ENV,classe_eq(REP)),
 egal(ENV,read(T1,I),E),
 i_get(REP,I,t(RepI,_,_,_,_,_,_)),
 i_get(REP,RepI,
  t(RepI,Lrepi,Taillerepi,Lsuprepi,Lctrrepi,Parentrepi,NumEgalrepi)),
 i_del(REP,RepI,_),
 i_set(REP,RepI,
  t(RepI,Lrepi,Taillerepi,Lsuprepi,[(T1,T)|Lctrrepi],Parentrepi,NumEgalrepi)).
```

Bibliographie

[1] *The Calculus of Computation.* 2007. Decision Procedures with Applications to Verification.

[2] *Decision Procedures.* Texts in Theoretical Computer Science An EATCS Series. 2008. An Algorithmic Point of View.

[3] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. 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 6–11, New York, NY, USA, 2008. ACM.

[4] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electron. Notes Theor. Comput. Sci.*, 198(2) :37–49, 2008.

[5] Leonardo De Moura and Nikolaj Bjørner. Z3 : an efficient smt solver. In *TACAS'08/ETAPS'08 : Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[7] Leonardo De Moura and Bjørner Nikolaj. Generalized, efficient array decision procedures. 2009.

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

[9] Robert Nieuwenhuis and Albert Oliveras. Union-find and congruence closure algorithms that produce proofs. *2ND INTERNATIONAL WORKSHOP ON PRAGMATICS OF DECISION PROCEDURES IN AUTOMATED REASONING*, 2004.

[10] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. *16TH INTERNATIONAL CONFERENCE ON REWRITING TECHNIQUES AND APPLICATIONS*, pages 453–468, 2005.

[11] Stuart J. Russell and Peter Norvig. *Artificial Intelligence : A Modern Approach.* Pearson Education, 2003.