EJCIM 2018

Loria - Nancy (26 - 30 mars)

Satisfiabilité Modulo Théories (SMT)

Sylvain Conchon

LRI (UMR 8623), Université Paris-Sud
Équipe Toccata, INRIA Saclay – Île-de-France

Plan du cours

INTRODUCTION

Le problème SMT

Satisfiabilité Modulo Théories

La problème SMT consiste à décider de la satisfiabilité de formules logiques contenant des symboles de théories particulières

Par exemple :

$$\exists x, y, z.\ x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

Le problème SMT

Satisfiabilité Modulo Théories

La problème SMT consiste à décider de la satisfiabilité de formules logiques contenant des symboles de théories particulières

Par exemple :

$$\exists x, y, z.\ x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

(par la suite, les variables libres seront *implicitement* quantifiées avec $\exists$)

Quelles formules ? Quelles contraintes ?

En toute généralité, les solveurs SMT peuvent prendre en entrée des formules quelconques de la logique du premier ordre

En pratique, il y a des solveurs SMT qui prennent en entrée uniquement des formules booléennes avec une seule théorie (par ex. l'arithmétique linéaire sur les entiers)

Mais il y a aussi des solveurs qui traitent toute la logique du premier ordre avec quantificateurs, des langages de types riches (polymorphisme, types algébriques, etc.) et des théories sur des objets complexes (par ex. ODEs)

Exemples de théories

La théorie de l'égalité avec symboles non interprétés

$$f(f(f(x))) = x \wedge f(f(f(f(f(x))))) = x \wedge f(x) \neq x$$

La théorie de l'arithmétique linéaire sur les rationnels

$$x + y = 19 \wedge x - y = 7 \wedge x \neq 13$$

La théorie des tableaux

$$a[i \leftarrow x] = b \wedge a = b \wedge b[i] = y \wedge b[i \leftarrow x][j] = y \wedge i = j$$

Comment résoudre le problème SMT ?

Soit $\mathcal{F}$ la formule suivante avec des symboles de l'arithmétique

$$x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

$\mathcal{F}$ est-elle satisfiable ?

Comment résoudre le problème SMT ?

Soit $\mathcal{F}$ la formule suivante avec des symboles de l'arithmétique

$$x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

$\mathcal{F}$ est-elle satisfiable ?

Pour répondre à cette question, on applique l'algorithme suivant:

Comment résoudre le problème SMT ?

Soit $\mathcal{F}$ la formule suivante avec des symboles de l'arithmétique

$$x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

$\mathcal{F}$ est-elle satisfiable ?

Pour répondre à cette question, on applique l'algorithme suivant:

1. convertir $\mathcal{F}$ en une CNF

Comment résoudre le problème SMT ?

Soit $\mathcal{F}$ la formule suivante avec des symboles de l'arithmétique

$$x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

$\mathcal{F}$ est-elle satisfiable ?

Pour répondre à cette question, on applique l'algorithme suivant:

1. convertir $\mathcal{F}$ en une CNF
2. remplacer chaque litéral par une variable booléenne

Comment résoudre le problème SMT ?

Soit $\mathcal{F}$ la formule suivante avec des symboles de l'arithmétique

$$x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

$\mathcal{F}$ est-elle satisfiable ?

Pour répondre à cette question, on applique l'algorithme suivant:

1. convertir $\mathcal{F}$ en une CNF
2. remplacer chaque litéral par une variable booléenne
3. appeler un solveur SAT afin d'obtenir un modèle $\mathcal{M}$ booléen de la formule

Comment résoudre le problème SMT ?

Soit $\mathcal{F}$ la formule suivante avec des symboles de l'arithmétique

$$x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

$\mathcal{F}$ est-elle satisfiable ?

Pour répondre à cette question, on applique l'algorithme suivant:

1. convertir $\mathcal{F}$ en une CNF
2. remplacer chaque litéral par une variable booléenne
3. appeler un solveur SAT afin d'obtenir un modèle $\mathcal{M}$ booléen de la formule
4. convertir $\mathcal{M}$ en contraintes dans l'arithmétique et le faire vérifier par la procédure de décision

Comment résoudre le problème SMT ?

Soit $\mathcal{F}$ la formule suivante avec des symboles de l'arithmétique

$$x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

$\mathcal{F}$ est-elle satisfiable ?

Pour répondre à cette question, on applique l'algorithme suivant:

1. convertir $\mathcal{F}$ en une CNF
2. remplacer chaque litéral par une variable booléenne
3. appeler un solveur SAT afin d'obtenir un modèle $\mathcal{M}$ booléen de la formule
4. convertir $\mathcal{M}$ en contraintes dans l'arithmétique et le faire vérifier par la procédure de décision

Si $\mathcal{M}$ est satisfiable dans la théorie de l'arithmétique, alors $\mathcal{F}$ l'est également, sinon on ajoute $\neg\mathcal{M}$ à $\mathcal{F}$ et on retourne en 2.

Résoudre le problème SMT sur un exemple

$$x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

Résoudre le problème SMT sur un exemple

$$x + y \geq 0 \wedge (x = z \Rightarrow y + z = -1) \wedge z > 3t$$

1. conversion en CNF

Résoudre le problème SMT sur un exemple

$$x + y \geq 0 \wedge (x \neq z \vee y + z = -1) \wedge z > 3t$$

1. conversion en CNF

Résoudre le problème SMT sur un exemple

$$x + y \geq 0 \wedge (x \neq z \vee y + z = -1) \wedge z > 3t$$

1. conversion en CNF
2. remplacement des contraintes arithmétiques par des variables booléennes

Résoudre le problème SMT sur un exemple

$$p_1 \wedge (p_2 \vee p_3) \wedge p_4$$

1. conversion en CNF
2. remplacement des contraintes arithmétiques par des variables booléennes

Résoudre le problème SMT sur un exemple

$$p_1 \wedge (p_2 \vee p_3) \wedge p_4$$

1. conversion en CNF
2. remplacement des contraintes arithmétiques par des variables booléennes
3. appel du solveur SAT: un modèle $\mathcal{M}$ est renvoyé si la formule est satisfiable

Résoudre le problème SMT sur un exemple

$$\mathcal{M} = \{p_1 = true,\, p_2 = false,\, p_3 = true,\, p_4 = true\}$$

1. conversion en CNF
2. remplacement des contraintes arithmétiques par des variables booléennes
3. appel du solveur SAT: un modèle $\mathcal{M}$ est renvoyé si la formule est satisfiable

Résoudre le problème SMT sur un exemple

$$\mathcal{M} = \{p_1 = true,\, p_2 = false,\, p_3 = true,\, p_4 = true\}$$

1. conversion en CNF
2. remplacement des contraintes arithmétiques par des variables booléennes
3. appel du solveur SAT: un modèle $\mathcal{M}$ est renvoyé si la formule est satisfiable
4. convertir $\mathcal{M}$ vers l'arithmétique

8

Résoudre le problème SMT sur un exemple

$$\mathcal{M} = \{x + y \geq 0,\, x = z,\, y + z = -1,\, z > 3t\}$$

1. conversion en CNF
2. remplacement des contraintes arithmétiques par des variables booléennes
3. appel du solveur SAT: un modèle $\mathcal{M}$ est renvoyé si la formule est satisfiable
4. convertir $\mathcal{M}$ vers l'arithmétique

8

Résoudre le problème SMT sur un exemple

$$\mathcal{M} = \{x + y \geq 0,\, x = z,\, y + z = -1,\, z > 3t\}$$

1. conversion en CNF
2. remplacement des contraintes arithmétiques par des variables booléennes
3. appel du solveur SAT: un modèle $\mathcal{M}$ est renvoyé si la formule est satisfiable
4. convertir $\mathcal{M}$ vers l'arithmétique
5. vérifier si $\mathcal{M}$ est cohérent modulo arithmétique à l'aide d'une procédure de décision

8

Résoudre le problème SMT sur un exemple

M n'est pas cohérent dans l'arithmétique!

1. conversion en CNF
2. remplacement des contraintes arithmétiques par des variables booléennes
3. appel du solveur SAT: un modèle $\mathcal{M}$ est renvoyé si la formule est satisfiable
4. convertir $\mathcal{M}$ vers l'arithmétique
5. vérifier si $\mathcal{M}$ est cohérent modulo arithmétique à l'aide d'une procédure de décision

Résoudre le problème SMT sur un exemple

M n'est pas cohérent dans l'arithmétique!

1. conversion en CNF
2. remplacement des contraintes arithmétiques par des variables booléennes
3. appel du solveur SAT: un modèle $\mathcal{M}$ est renvoyé si la formule est satisfiable
4. convertir $\mathcal{M}$ vers l'arithmétique
5. vérifier si $\mathcal{M}$ est cohérent modulo arithmétique à l'aide d'une procédure de décision
6. ajout de $\neg\mathcal{M}$ à $\mathcal{F}$ et retourner à l'étape 2

Résoudre le problème SMT sur un exemple

$$x + y \geq 0 \wedge (x \neq z \vee y + z = -1) \wedge z > 3t \wedge$$
$$\neg(x + y \geq 0 \wedge x = z \wedge y + z = -1 \wedge z > 3t)$$

1. conversion en CNF
2. remplacement des contraintes arithmétiques par des variables booléennes
3. appel du solveur SAT: un modèle $\mathcal{M}$ est renvoyé si la formule est satisfiable
4. convertir $\mathcal{M}$ vers l'arithmétique
5. vérifier si $\mathcal{M}$ est cohérent modulo arithmétique à l'aide d'une procédure de décision
6. ajout de $\neg\mathcal{M}$ à $\mathcal{F}$ et retourner à l'étape 2

Combinaison de théories

En pratique, les formules à traiter sont souvent formées d'un mélange de symboles appartenant à plusieurs théories

Par exemple, la formule suivante mélange des symboles de la théorie des tableaux, de l'arithmétique linéaire sur les entiers et de la théorie de l'égalité avec symboles non interprétés:

$$x + 2 = y \wedge f(a[x \leftarrow 3][y - 2]) \neq f(y - x + 1)$$

Les principales difficultés du problème SMT

- La taille des formules (parfois plusieurs Giga-octets !)
- La complexité de la structure booléenne (par exemple quand les formules encodent des circuits logiques de micro-processeurs)
- La combination de théories
- L'efficacité des procédures de décision
- Des théories sur des objets mathématiques de plus en plus complexes
- La gestion des quantificateurs
- ...

Les principales difficultés du problème SMT

- La taille des formules (parfois plusieurs Giga-octets !)
- La complexité de la structure booléenne (par exemple quand les formules encodent des circuits logiques de micro-processeurs)
- La combination de théories
- L'efficacité des procédures de décision
- Des théories sur des objets mathématiques de plus en plus complexes
- La gestion des quantificateurs
- ...

C'est un domaine de l'informatique passionnant qui mélange logique, mathématique et programmation

10

DE PETITS MOTEURS DE PREUVE

La théorie de l'égalité avec symboles non interprétés

Appelée également théorie libre de l'égalité, cette théorie donne un sens au prédicat d'égalité $=$ en présence de symboles de fonctions d'arité quelconque dont le sens n'est pas défini

La théorie de l'égalité avec symboles non interprétés

Appelée également théorie libre de l'égalité, cette théorie donne un sens au prédicat d'égalité $=$ en présence de symboles de fonctions d'arité quelconque dont le sens n'est pas défini

Les termes de cette théorie appartiennent à deux catégories syntaxiques :

(1) les variables $x, y, z,$ etc.
(2) les applications de fonctions $f(x), g(x, y),$ etc.

La théorie de l'égalité avec symboles non interprétés

Appelée également théorie libre de l'égalité, cette théorie donne un sens au prédicat d'égalité $=$ en présence de symboles de fonctions d'arité quelconque dont le sens n'est pas défini

Les termes de cette théorie appartiennent à deux catégories syntaxiques :

(1) les variables $x, y, z,$ etc.
(2) les applications de fonctions $f(x), g(x, y),$ etc.

Les contraintes élémentaires sont soit des égalités ou des différences entre les termes comme par exemple:

$$x = f(y, z) \quad g(x) \neq h(y) \quad x = y$$

12

Axiomes de la théorie de l'égalité

La théorie est définie à partir de trois axiomes et d'un schéma d'axiomes.

(Réflexivité) $\forall x.\ x = x$

(Symétrie) $\forall xy.\ x = y \Rightarrow y = x$

(Transitivité) $\forall xyz.\ x = y \wedge y = z \Rightarrow x = z$

(Congruence) Pour tout symbole de fonction f d'arité n

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n.$$
$$x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

Axiomes de la théorie de l'égalité

La théorie est définie à partir de trois axiomes et d'un schéma d'axiomes.

(Réflexivité) $\forall x.\ x = x$

(Symétrie) $\forall xy.\ x = y \Rightarrow y = x$

(Transitivité) $\forall xyz.\ x = y \wedge y = z \Rightarrow x = z$

(Congruence) Pour tout symbole de fonction f d'arité n

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n.$$
$$x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

Exemples:

$$g(x, y) = x \wedge g(g(x, y), y) \neq x$$

13

Axiomes de la théorie de l'égalité

La théorie est définie à partir de trois axiomes et d'un schéma d'axiomes.

(Réflexivité) $\forall x.\ x = x$

(Symétrie) $\forall xy.\ x = y \Rightarrow y = x$

(Transitivité) $\forall xyz.\ x = y \wedge y = z \Rightarrow x = z$

(Congruence) Pour tout symbole de fonction f d'arité n

$$\forall x_1, \ldots, x_n, y_1, \ldots, y_n.$$
$$x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

Exemples:

$$g(x, y) = x \wedge g(g(x, y), y) \neq x$$
$$f(f(f(x))) = x \wedge f(f(f(f(f(x))))) = x \wedge f(x) \neq x$$

Procédure de décision pour la théorie de l'égalité

La procédure décide de la satisfiabilité d'une conjonction $\mathcal{C}$ de contraintes élémentaires de la forme

$$\bigwedge_i t_i \bowtie u_i \qquad \text{où } \bowtie \in \{=, \neq\}$$

Procédure de décision pour la théorie de l'égalité

La procédure décide de la satisfiabilité d'une conjonction $\mathcal{C}$ de contraintes élémentaires de la forme

$$\bigwedge_i t_i \bowtie u_i \qquad \text{où } \bowtie \in \{=, \neq\}$$

L'algorithme commence par séparer $\mathcal{C}$ en deux ensembles de contraintes $\mathcal{E}$ (égalités) et $\mathcal{D}$ (différences)

$$\overbrace{\bigwedge_i t_i = u_i}^{\mathcal{E}} \wedge \overbrace{\bigwedge_j t_j \neq u_j}^{\mathcal{D}}$$

Procédure de décision pour la théorie de l'égalité

La procédure décide de la satisfiabilité d'une conjonction $\mathcal{C}$ de contraintes élémentaires de la forme

$$\bigwedge_i t_i \bowtie u_i \qquad \text{où } \bowtie \in \{=, \neq\}$$

L'algorithme commence par séparer $\mathcal{C}$ en deux ensembles de contraintes $\mathcal{E}$ (égalités) et $\mathcal{D}$ (différences)

$$\overbrace{\bigwedge_i t_i = u_i}^{\mathcal{E}} \wedge \overbrace{\bigwedge_j t_j \neq u_j}^{\mathcal{D}}$$

La procédure de décision va calculer la **fermeture par congruence** de $\mathcal{E}$ et s'assurer qu'elle est compatible avec les contraintes dans $\mathcal{D}$

Fermeture par congruence

Soit $\mathcal{R}$ une relation d'équivalence sur les termes. Le domaine de $\mathcal{R}$, noté $\mathtt{dom}(\mathcal{R})$, est l'ensemble de tous les termes et sous-termes de R

Deux termes t et u sont congruents par $\mathcal{R}$ s'ils sont respectivement de la forme $f(t_1, \ldots, t_n)$ et $f(u_1, \ldots, u_n)$ et $\forall i.\ (t_i, u_i) \in \mathcal{R}$

$\mathcal{R}$ est fermée par congruence si pour tous les termes $t, u \in \mathtt{dom}(\mathcal{R})$ congruents par $\mathcal{R}$ on a $(t, u) \in \mathcal{R}$

Fermeture par congruence:

La fermeture par congruence de $\mathcal{R}$ est la plus petite relation contenant $\mathcal{R}$ et qui est close par congruence

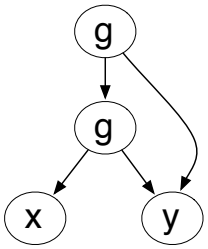
Représentation des termes

DAG des termes : on construit un DAG pour représenter tous les termes (et sous-termes) apparaissant dans l'ensemble $\mathcal{E} \cup \mathcal{D}$.

Représentation des termes

DAG des termes : on construit un DAG pour représenter tous les termes (et sous-termes) apparaissant dans l'ensemble $\mathcal{E} \cup \mathcal{D}$.

Par exemple, pour la formule $g(x, y) = x \wedge g(g(x, y), y) \neq x$, on obtient:



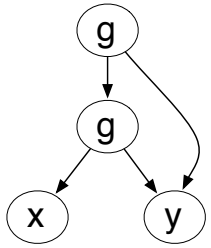
Représentation de la relation d'équivalence

L'algorithme de la fermeture par congruence consiste à utiliser une structure union-find pour maintenir des classes d'équivalence entre les nœuds du graphe

Deux nœuds n et m étant dans la même classe quand les termes t_1 et u_a qu'ils représentent respectivement sont égaux soit:

1. soit parce que $t_1 = u_1 \in \mathcal{E}$
2. soit parce que l'égalité $t_1 = u_1$ est une conséquence de $\mathcal{E}$ et des axiomes de la théorie de l'égalité.

Cette relation est représentée dans le DAG par des traits en pointillés (ex. $g(x, y) = x$)



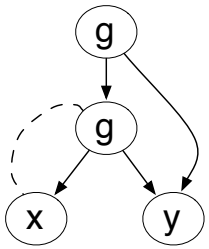
Représentation de la relation d'équivalence

L'algorithme de la fermeture par congruence consiste à utiliser une structure union-find pour maintenir des classes d'équivalence entre les nœuds du graphe

Deux nœuds n et m étant dans la même classe quand les termes t_1 et u_a qu'ils représentent respectivement sont égaux soit:

1. soit parce que $t_1 = u_1 \in \mathcal{E}$
2. soit parce que l'égalité $t_1 = u_1$ est une conséquence de $\mathcal{E}$ et des axiomes de la théorie de l'égalité.

Cette relation est représentée dans le DAG par des traits en pointillés (ex. $g(x, y) = x$)



Structure Union-Find

Il s'agit d'une structure de données pour résoudre le problème des classes disjointes

- Ce problème consiste à maintenir dans une structure de données une partition d'un ensemble fini.
- Sans perte de généralité, on peut supposer qu'il s'agit de l'ensemble des n entiers $\{0, 1, \ldots, n-1\}$.

Signature pour *union-find*

```
type t
val create :  int -> t
val find :  t -> int -> int
val union :  t -> int -> int -> unit
```

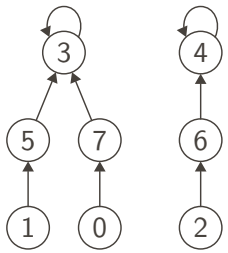
- L'opération `create n` construit une nouvelle partition de $\{0, 1, \ldots, n-1\}$ où chaque élément forme une classe à lui tout seul.
- L'opération `find` détermine la classe d'un élément, sous la forme d'un entier considéré comme le représentant de cette classe.
- Enfin l'opération `union` réunit deux classes de la partition, la structure de données étant modifiée en place.

Réprésentation des partitions

L'idée principale est de lier entre eux les éléments d'une même classe

- Dans chaque classe, ces liaisons forment un graphe acyclique où tous les chemins mènent au représentant, qui est le seul élément lié à lui-même.

La schéma suivant illustre une situation où l'ensemble $\{0, 1, \ldots, 7\}$ est partitionné en deux classes dont les représentants sont respectivement 3 et 4.

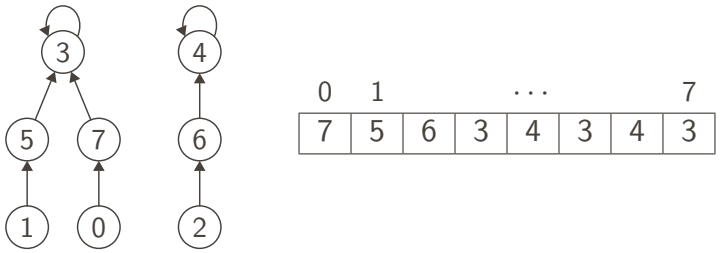


La structure est donc une forêt de graphes acycliques

Implémentation avec des tableaux

Une manière simple de réaliser cette forêt consiste à utiliser un tableau qui lie chaque entier i à un autre entier de la même classe.

Ainsi la partition du schéma de gauche est représentée par le tableau de droite :

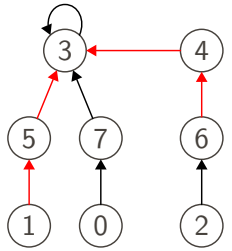


0	1		···		7
7	5	6	3	4	3	4	3

Réalisation de `find` et `union`

- L'opération `find` se contente de suivre les liaisons jusqu'à trouver le représentant d'un élément.
- L'opération `union` commence par trouver les représentants des deux éléments, puis lie l'un des deux représentants à l'autre.

Exemple: `find 1` suit les liaisons rouges à partir du nœud `1` pour trouver son représentant (ici `3`). De même, `union 1 6` trouve tout d'abord les représentants de `1` et `6` (ici `3` et `4`) puis lie le nœud `4` au nœud `3`.



Amélioration

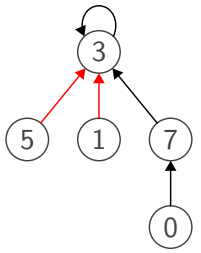
Afin d'atteindre de bonnes performances, on apporte deux améliorations.

- La compression de chemins
- L'équilibrage des classes

La compression de chemins

On *compresse les chemins* pendant la recherche effectuée par `find`

Cela consiste à lier directement au représentant tous les éléments trouvés sur le chemin parcouru pour l'atteindre



L'équilibrage des classes

- On maintient pour chaque représentant une approximation de la taille de sa classe, comme un un majorant de la longueur du plus long chemin dans sa classe.
- Cette information est stockée dans un second tableau et utilisée par la fonction `union` pour choisir le représentant d'une union.

Type de la structure *union-find* (en OCaml)

Le type `t` est un enregistrement contenant deux tableaux :

- `rank` qui contient l'information sur la taille de chaque classe;
- `parent` qui contient les liaisons.

```
type t = {
 rank :  int array;
 parent :  int array;
}
```


Classes initiales

- Chaque élément forme une classe à lui tout seul, c'est-à-dire que chaque élément est son propre représentant.
- La taille de chaque classe vaut 0.

```
let create n =
 { rank = Array.create n 0;
   parent = Array.init n (fun i -> i) }
```


La fonction `find`

```
let rec find t i =
 let p = t.parent.(i) in
 if p = i then
  i
 else begin
  let r = find t p in
  t.parent.(i) <- r;
  r
 end
```

Un appel `find t i` commence par calculer le parent `p` de `i`.

- Si c'est `i` lui-même, on a terminé et `i` est le représentant de la classe.
- Sinon, il suffit de calculer récursivement le représentant `r` comme `find t p`.

Cependant, pour réaliser la compression de chemins, on modifie le parent de `i`, pour lui donner la valeur `r`, avant de renvoyer `r`.

La fonction union (1/3)

Pour réaliser l'union des classes de deux éléments `i` et `j`, on commence par calculer leurs représentants respectifs `ri` et `rj`.

S'ils sont égaux, il n'y a rien à faire.

```
let union t i j =
 let ri = find t i in
 let rj = find t j in
 if ri <> rj then begin
 ...
```

Sinon, on compare la taille des deux classes...

La fonction union (2/3)

Si la classe de `ri` est strictement plus petite que celle de `rj`, on fait de `rj` le représentant de l'union, *i.e.* le parent de `ri`.

```
if t.rank.(ri) < t.rank.(rj) then
 t.parent.(ri) <- rj
```

Si en revanche la classe de `rj` est la plus petite, on procède symétriquement.

```
else begin
 t.parent.(rj) <- ri;
```


La fonction union (3/3)

L'information de taille n'a besoin d'être mise à jour que dans le cas où les deux classes ont la même taille, car c'est dans ce cas que la longueur du plus long chemin est susceptible d'augmenter.

```
 if t.rank.(ri) = t.rank.(rj) then
  t.rank.(ri) <- t.rank.(ri) + 1
 end
end
```

Il est important de noter que la fonction `union` utilise la fonction `find` et donc réalise deux compressions de chemin, même dans le cas où il s'avère que `i` et `j` sont dans la même classe.

Algorithme de fermeture par congruence (Python like)

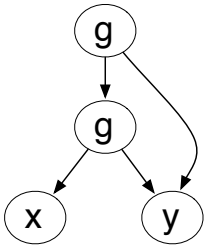
L'algorithme de fermeture par congruence s'implémente facilement avec les deux boucles for suivantes:

```
for every nodes n, m ∈ G
    labeled with the same symbol:
  if find(n) == find(m) and find(n_i) == find(m_i)
              for every children of n and m :
     union(n,m)

for every difference t_i ≠ u_i in 𝒟:
   if find(t_i) == find(u_i):
      return UNSAT
   return SAT
```

Exemple de fermeture par congruence

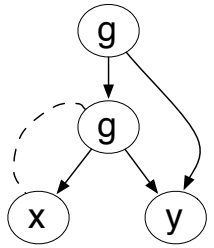
On souhaite vérifier la satisfiabilité de la formule:
$g(x,y) = x \wedge g(g(x,y),y) \neq x$



(1) DAG initial

Exemple de fermeture par congruence

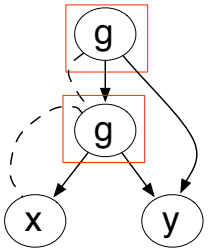
On souhaite vérifier la satisfiabilité de la formule:
$g(x,y) = x \wedge g(g(x,y),y) \neq x$



(2) $g(x,y) = x$

Exemple de fermeture par congruence

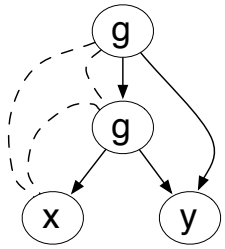
On souhaite vérifier la satisfiabilité de la formule:
$g(x,y) = x \wedge g(g(x,y),y) \neq x$



(3) $g(x,y) = x \wedge y = y$
$\Rightarrow g(g(x,y),y) = g(x,y)$

Exemple de fermeture par congruence

On souhaite vérifier la satisfiabilité de la formule:
$g(x,y) = x \wedge g(g(x,y),y) \neq x$



(4) $g(g(x,y),y) = g(x,y) \wedge g(x,y) = x$
$\Rightarrow g(g(x,y),y) = x$

On en déduit que $g(g(x,y),y) \neq x$ ne peut être satisfait

La théorie des inéquations

Cette théorie, appelée en anglais Difference Logic, est un fragment de l'arithmétique linéaire (sur $\mathbb{Z}$ ou $\mathbb{Q}$)

Les contraintes élémentaires sont restreintes à la forme $x - y \leq c$, avec x et y deux variables et c une constante numérique

La théorie des inéquations

Cette théorie, appelée en anglais Difference Logic, est un fragment de l'arithmétique linéaire (sur $\mathbb{Z}$ ou $\mathbb{Q}$)

Les contraintes élémentaires sont restreintes à la forme $x - y \leq c$, avec x et y deux variables et c une constante numérique

Une solution à un ensemble de telles contraintes est représentée par un dictionnaire σ qui à chaque variable x associe un entier ou un rationnel $\sigma(x)$

34

Encodage

Bien que restrictive, d'autres contraintes peuvent être encodées dans ce fragment

(1) Encodage des contraintes strictes:

- sur $\mathbb{Z}$, $x - y < c$ est remplacée par $x - y \leq c - 1$
- sur $\mathbb{Q}$, $x - y < c$ est remplacée par $x - y \leq c - \delta$, où δ est une constante suffisamment petite (en pratique, δ est simplement une constante *symbolique*)

Encodage

Bien que restrictive, d'autres contraintes peuvent être encodées dans ce fragment

(1) Encodage des contraintes strictes:

- sur $\mathbb{Z}$, $x - y < c$ est remplacée par $x - y \leq c - 1$
- sur $\mathbb{Q}$, $x - y < c$ est remplacée par $x - y \leq c - \delta$, où δ est une constante suffisamment petite (en pratique, δ est simplement une constante *symbolique*)

(2) Encodage des contraintes de la forme $x \leq c$

À partir d'une solution σ, on peut construire une solution σ' en décalant $\sigma(x)$ par une même constante k, pour tout x.

Ainsi, $x \leq c$ est encodée par $x - y_{zero} \leq c$, où y_{zero} est une nouvelle variable, et pour toute solution σ, on construit une solution σ' telle que $\sigma'(y_{zero}) = 0$, i.e. on décale de $k = -\sigma(y_{zero})$

35

Procédure de décision pour la théorie des inéquations

Soit un ensemble de variables $\{x_1, \ldots, x_n\}$ et une conjonction $\mathcal{C}$ de contraintes élémentaires de la forme $\bigwedge_i x_i - y_i \leq c_i$

La procédure de décision pour cette théorie consiste à construire un graphe pondéré $\mathcal{G}(V, E)$ tel que:

- $V = \{s, x_1, \ldots, x_n\}$

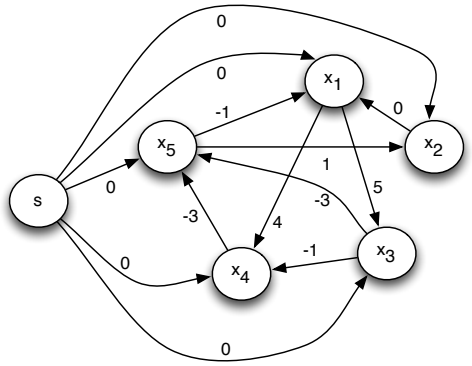
  Chaque nœud correspond à une variable du problème plus une nouvelle variable s

- $E = \{y_i \xrightarrow{c_i} x_i \mid x_i - y_i \leq c_i \in \mathcal{C}\} \cup \{s \xrightarrow{0} x_i \mid 1 \leq i \leq n\}$

  Chaque arête correspond à une contrainte du problème plus une arête de poids nul entre s et chaque variable du problème

Exemple de graphe pondéré des inéquations

$$
\begin{aligned}
x_1 - x_2 &\leq 0 \\
x_1 - x_5 &\leq -1 \\
x_2 - x_5 &\leq 1 \\
x_3 - x_1 &\leq 5 \\
x_4 - x_1 &\leq 4 \\
x_4 - x_3 &\leq -1 \\
x_5 - x_3 &\leq -3 \\
x_5 - x_4 &\leq -3
\end{aligned}
$$



Chemins dans un graphe

Un *chemin* entre deux nœuds a et b est une séquence d'arêtes de la forme $a \xrightarrow{c_1} v_1 \xrightarrow{c_2} \cdots \xrightarrow{c_{n-1}} v_{n-1} \xrightarrow{c_n} b$

Le *poids* d'un chemin est la somme $c_1 + \cdots + c_n$

Parmis tous les chemins entre a et b, on distingue le plus court chemin, i.e. le chemin ayant le poids le plus petit

Procédure de décision pour la théorie des inéquations

La procédure de décision pour cette théorie repose sur le théorème suivant.

Théorème : Étant donnée une conjonction d'inéquations $\mathcal{C}$ de la forme:

$$\bigwedge_i x_i - y_i \leq c_i$$

et $\mathcal{G}(V, E)$ le graphe correspondant:

1. Si $\mathcal{G}$ a un cycle *négatif*, i.e un chemin de la forme
   $$v_1 \xrightarrow{c_1} v_2 \xrightarrow{c_2} \ldots \xrightarrow{c_{n-1}} v_n \xrightarrow{c_n} v_1$$
   tel que $c_1 + c_2 + \cdots + c_{n-1} + c_n < 0$, alors $\mathcal{C}$ est insatisfiable
2. Sinon, une solution est
   $x_1 = \delta(s, x_1), \ldots, x_n = \delta(s, x_n)$
   où $\delta(s, x_i)$ est le chemin le plus court entre s et x_i.

Preuve (voir livre)

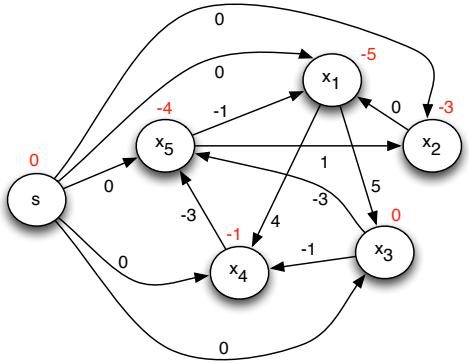
Détection des cycles négatifs

La détection des cycles négatifs peut être réalisée à l'aide de l'algorithme de Bellman-Ford

```
for each x_i in V:
    d[x_i] = ∞
d[s] = 0
for k in range(1, |V|):
    for each y_i --c--> x_i in E:
        if d[x_i] > d[y_i] + c:  # relaxation
            d[x_i] = d[y_i] + c
            π[x_i] = y_i
for each y_i --c--> x_i ∈ E:
    if d[x_i] > d[y_i] + c: # cela ferait diminuer d[x_i]
        return Negative Cycle Detected
               (use π to reconstruct the cycle)
```


Exemple

$$
\begin{aligned}
x_1 - x_2 &\leq 0 \\
x_1 - x_5 &\leq -1 \\
x_2 - x_5 &\leq 1 \\
x_3 - x_1 &\leq 5 \\
x_4 - x_1 &\leq 4 \\
x_4 - x_3 &\leq -1 \\
x_5 - x_3 &\leq -3 \\
x_5 - x_4 &\leq -3
\end{aligned}
$$



$$
\begin{aligned}
x_1 &= -5 \quad x_2 = -3 \\
x_3 &= 0 \quad x_4 = -1 \\
x_5 &= -4
\end{aligned}
$$

INTERFACER SAT AVEC DES PROCÉDURES DE DÉCISION

Restriction

On commence par s'intéresser au problème SMT en se restreignant à des formules d'une unique théorie T disposant d'une procédure de décision Tsolve

Nous verrons dans la partie suivante comment traiter le problème plus général avec plusieurs théories

La combinaison hors-ligne

Soit une formule ϕ, la technique la plus simple pour résoudre le problème SMT est de convertir ϕ en DNF de la forme $\bigvee_{i \in I} \mathcal{C}_i$

Ensuite, il suffit d'appeler `Tsolve`$(\mathcal{C}_i)$ sur chaque conjonction de contraintes $\mathcal{C}_i$, jusqu'à ce qu'une de ces contraintes soit T-satisfiable

Plutôt que de faire cette conversion en DNF (pas efficace), nous allons utiliser un solveur SAT pour traiter efficacement la structure booléenne de ϕ

Fonctions intermédiaires

- `T2B`(ϕ) calcule une abstraction booléenne de ϕ en remplaçant chaque litéral par une variable booléenne fraîche
- `B2T`(μ) fonction inverse pour retrouver les litéraux de départ
- `CDCL`(f) appelle le solveur SAT et renvoie un couple (r, μ), où r indique si f est satisfiable et, si c'est le cas, μ contient un modèle booléen pour f
- `Tsolve`(M) renvoie un couple (r, uc) où r indique si le modèle μ est satisfiable modulo `T` ; si $r =$ UNSAT, alors uc contient une explication (*unsat core*) qui peut être un sous-ensemble incohérent de μ

L'algorithme SMT hors ligne

```
def smtOffline(φ):
  f = T2B(φ)
  while True:
      (r, μ) = CDCL(f)
      if r == UNSAT:
        return UNSAT
      else:
        (r, uc) = Tsolve(B2T(μ))
        if r == SAT:
           return SAT
        else:
           f = f ∧ ¬T2B(uc)
```


Exemple avec SMT hors ligne

$$\neg(a = b)$$
$$(x = a \vee x = b)$$
$$(y = a \vee y = b)$$
$$(z = a \vee z = b)$$
$$\neg(x = y)$$

Exemple avec SMT hors ligne

$$\texttt{T2B} \quad \begin{array}{c} \neg P_1 \\ (P_2 \vee P_3) \\ (P_4 \vee P_5) \\ (P_6 \vee P_7) \\ \neg P_8 \end{array}$$

Exemple avec SMT hors ligne

$$\texttt{CDCL} \quad \left\{ \begin{array}{c} \neg P_1, \neg P_8 \\ \neg P_3, P_2 \\ \neg P_5, P_4 \\ \neg P_7, P_6 \end{array} \right\}$$

Exemple avec SMT hors ligne

$$\texttt{B2T} \quad \left\{ \begin{array}{c} \neg(a = b), \neg(x = y) \\ \neg(x = b), x = a \\ \neg(y = b), y = a \\ \neg(z = b), z = a \end{array} \right\}$$

Exemple avec SMT hors ligne

$$\texttt{Tsolve} \quad \text{UNSAT}, \left\{ \begin{array}{c} x = a, \\ y = a, \\ \neg(x = y) \end{array} \right\}$$

Exemple avec SMT hors ligne

T2B $\left\{ \begin{array}{c} P_2 \\ P_4 \\ \neg P_8 \end{array} \right\}$

Exemple avec SMT hors ligne

add $\neg uc$ $\begin{array}{c} \neg P_1 \\ (P_2 \vee P_3) \\ (P_4 \vee P_5) \\ (P_6 \vee P_7) \\ \neg P_8 \\ (P_8 \vee \neg P_2 \vee \neg P_4) \end{array}$

Exemple avec SMT hors ligne

CDCL $\left\{ \begin{array}{c} \neg P_1, \neg P_8 \\ \neg P_3, P_2 \\ \neg P_4, P_5 \\ \neg P_7, P_6 \end{array} \right\}$

Exemple avec SMT hors ligne

B2T $\left\{ \begin{array}{c} \neg(a = b), \neg(x = y) \\ \neg(x = b), x = a \\ \neg(y = a), y = b \\ \neg(z = b), z = a \end{array} \right\}$

Exemple avec SMT hors ligne

`Tsolve` SAT

CDCL + Théorie

La recherche du modèles dans un SMT hors ligne n'est pas guidée par la théorie (choix de variables, BCP en aveugle)

Pour remédier à cela, on va intégrer directement le raisonnement modulo théorie dans l'algorithme CDCL.

Cela nécessite quelques extensions:

CDCL + Théorie

La recherche du modèles dans un SMT hors ligne n'est pas guidée par la théorie (choix de variables, BCP en aveugle)

Pour remédier à cela, on va intégrer directement le raisonnement modulo théorie dans l'algorithme CDCL.

Cela nécessite quelques extensions:

- La fonction `theory_and_boolean_propagation`(ϕ,μ) réalise le BCP puis, si aucun conflit n'est détecté, appelle `Tsolve` pour vérifier la cohérence du modèle μ modulo `T`

  Point fixe : on propage les litéraux **déduits** par `Tsolve` pour trouver de nouvelles clauses unitaires, etc.

CDCL + Théorie

La recherche du modèles dans un SMT hors ligne n'est pas guidée par la théorie (choix de variables, BCP en aveugle)

Pour remédier à cela, on va intégrer directement le raisonnement modulo théorie dans l'algorithme CDCL.

Cela nécessite quelques extensions:

- La fonction `theory_and_boolean_propagation`(ϕ,μ) réalise le BCP puis, si aucun conflit n'est détecté, appelle `Tsolve` pour vérifier la cohérence du modèle μ modulo `T`

  Point fixe : on propage les litéraux **déduits** par `Tsolve` pour trouver de nouvelles clauses unitaires, etc.

- La fonction `theory_and_boolean_conflict_analysis`(ϕ,μ) utilise les explications renvoyées par la procédure de décision pour trouver le niveau et la clause conflit modulo `T`

CDCL(T)

```
def cdclT(φ):
  μ = []
  dl = 0
  while True:
    res = theory_and_boolean_propagation(φ,μ)
    if res == SAT:
      if all_variables_are_assigned(φ,μ):
        return SAT
       l = pick_a_branching_literal(φ,μ)
       dl = dl + 1
       push(μ,l@dl)
      else:
        (lvl, cls) = theory_and_boolean_conflict_analysis(φ,μ)
        if lvl < 0:
          return UNSAT
        backtrack(φ, μ, lvl)
        learn(cls)
        dl = lvl
```


Exemple

$$(x_2 - x_3 \leq 2) \vee A_1 \quad \neg A_2 \vee (x_1 - x_5 \leq 1) \quad (x_1 - x_2 \leq 3) \vee A_2$$
$$(x_3 + x_4 < 5) \vee (x_1 - x_3 > 6) \vee \neg A_1 \quad (x_1 - x_2 \leq 3) \vee A_1$$
$$(x_2 - x_4 \leq 6) \vee (x_5 = 5 - 3x_4) \vee \neg A_1 \quad (x_3 = x_5 + 4) \vee A_1 \vee A_2$$

Exemple

$$(x_2 - x_3 \leq 2) \vee A_1 \quad \neg A_2 \vee (x_1 - x_5 \leq 1) \quad (x_1 - x_2 \leq 3) \vee A_2$$
$$(x_3 + x_4 < 5) \vee (x_1 - x_3 > 6) \vee \neg A_1 \quad (x_1 - x_2 \leq 3) \vee A_1$$
$$(x_2 - x_4 \leq 6) \vee (x_5 = 5 - 3x_4) \vee \neg A_1 \quad (x_3 = x_5 + 4) \vee A_1 \vee A_2$$

@1	$(x_1 - x_3 > 6)$
@2	$(x_3 = x_5 + 4)$
@3	$(x_2 - x_4 \leq 6)$
@4	$(x_2 - x_3 \leq 2)$

Exemple

$$(x_2 - x_3 \leq 2) \vee A_1 \quad \neg A_2 \vee (x_1 - x_5 \leq 1) \quad (x_1 - x_2 \leq 3) \vee A_2$$
$$(x_3 + x_4 < 5) \vee (x_1 - x_3 > 6) \vee \neg A_1 \quad (x_1 - x_2 \leq 3) \vee A_1$$
$$(x_2 - x_4 \leq 6) \vee (x_5 = 5 - 3x_4) \vee \neg A_1 \quad (x_3 = x_5 + 4) \vee A_1 \vee A_2$$

@1	$(x_1 - x_3 > 6)$
@2	$(x_3 = x_5 + 4)$
@3	$(x_2 - x_4 \leq 6)$
@4	$(x_2 - x_3 \leq 2)$
T-propagate	$(x_1 - x_3 > 6) \wedge (x_2 - x_3 \leq 2) \Rightarrow (x_1 - x_2 > 3)$

Exemple

$$(x_2 - x_3 \leq 2) \vee A_1 \quad \neg A_2 \vee (x_1 - x_5 \leq 1) \quad (x_1 - x_2 \leq 3) \vee A_2$$
$$(x_3 + x_4 < 5) \vee (x_1 - x_3 > 6) \vee \neg A_1 \qquad (x_1 - x_2 \leq 3) \vee A_1$$
$$(x_2 - x_4 \leq 6) \vee (x_5 = 5 - 3x_4) \vee \neg A_1 \quad (x_3 = x_5 + 4) \vee A_1 \vee A_2$$

@1	$(x_1 - x_3 > 6)$
@2	$(x_3 = x_5 + 4)$
@3	$(x_2 - x_4 \leq 6)$
@4	$(x_2 - x_3 \leq 2)$
T-propagate	$(x_1 - x_3 > 6) \wedge (x_2 - x_3 \leq 2) \Rightarrow (x_1 - x_2 > 3)$
BCP	A_1
BCP	A_2
BCP	$(x_1 - x_5 \leq 1)$

Exemple

$$(x_2 - x_3 \leq 2) \vee A_1 \quad \neg A_2 \vee (x_1 - x_5 \leq 1) \quad (x_1 - x_2 \leq 3) \vee A_2$$
$$(x_3 + x_4 < 5) \vee (x_1 - x_3 > 6) \vee \neg A_1 \qquad (x_1 - x_2 \leq 3) \vee A_1$$
$$(x_2 - x_4 \leq 6) \vee (x_5 = 5 - 3x_4) \vee \neg A_1 \quad (x_3 = x_5 + 4) \vee A_1 \vee A_2$$

@1	$(x_1 - x_3 > 6)$
@2	$(x_3 = x_5 + 4)$
@3	$(x_2 - x_4 \leq 6)$
@4	$(x_2 - x_3 \leq 2)$
T-propagate	$(x_1 - x_3 > 6) \wedge (x_2 - x_3 \leq 2) \Rightarrow (x_1 - x_2 > 3)$
BCP	A_1
BCP	A_2
BCP	$(x_1 - x_5 \leq 1)$
T-conflict	$(x_1 - x_3 > 6) \wedge (x_3 = x_5 + 4) \wedge (x_1 - x_5 \leq 1)$

Exemple

$$(x_2 - x_3 \leq 2) \vee A_1 \quad \neg A_2 \vee (x_1 - x_5 \leq 1) \quad (x_1 - x_2 \leq 3) \vee A_2$$
$$(x_3 + x_4 < 5) \vee (x_1 - x_3 > 6) \vee \neg A_1 \qquad (x_1 - x_2 \leq 3) \vee A_1$$
$$(x_2 - x_4 \leq 6) \vee (x_5 = 5 - 3x_4) \vee \neg A_1 \quad (x_3 = x_5 + 4) \vee A_1 \vee A_2$$

@1	$(x_1 - x_3 > 6)$
@2	$(x_3 = x_5 + 4)$
@3	$(x_2 - x_4 \leq 6)$
@4	$(x_2 - x_3 \leq 2)$
T-propagate	$(x_1 - x_3 > 6) \wedge (x_2 - x_3 \leq 2) \Rightarrow (x_1 - x_2 > 3)$
BCP	A_1
BCP	A_2
BCP	$(x_1 - x_5 \leq 1)$
T-conflict	$(x_1 - x_3 > 6) \wedge (x_3 = x_5 + 4) \wedge (x_1 - x_5 \leq 1)$
backtrack	$x_1 - x_3 \leq 6 \vee (x_3 \neq x_5 + 4) \vee (x_2 - x_3 > 2)$
keep@1	$(x_1 - x_3 > 6)$
keep@2	$(x_3 = x_5 + 4)$
BCP	$(x_2 - x_3 > 2)$
...	

Procédures de décision pour SMT

Puisque la procédure de décision est maintenant partie prenante des choix et rebroussements du solveur SAT, il est fondamental qu'elle possède les propriétés suivantes:

- Incrémentalité, pour être appelée successivement sur une suite d'ensembles de contraintes $\mathcal{C}_\prime \subset \mathcal{C}_1 \subset \ldots \subset \mathcal{C}_k$ afin que le traitement d'un ensemble $\mathcal{C}_i$ ne nécessite pas de tout refaire depuis le début, mais réutilise le traitement effectué pour $\mathcal{C}_{i-1}$
- Backtrackabilité, pour revenir efficacement à un état antérieur de l'algorithme sans avoir à tout ré-initialiser
- Explications, pour calculer des *unsat cores*
- Propagations, pour détecter rapidement de nouveaux faits (ou contraintes) impliqués par un ensemble de contraintes

COMBINAISON DE PROCÉDURES DE DÉCISION

Union de deux théories

Un des principaux problèmes sous-jacents à l'implémentation d'un solveur SMT est le problème de la construction d'une procédure de décision pour l'union de théories

Mais, si $\mathcal{T}_1$ et $\mathcal{T}_2$ sont deux théories *cohérentes*:

1. Est-ce que l'union $\mathcal{T}_1 \cup \mathcal{T}_2$ est cohérente?
2. Peut-on construire une procédure de décision $\mathcal{T}_1 \cup \mathcal{T}_2$ à partir de procédures de décision pour $\mathcal{T}_1$ et $\mathcal{T}_2$?

Théories disjointes

Bien qu'il n'y ait pas de réponse dans le cas général, des résultats ont été donnés pour certaines classes de théories

Par exemple, quand $\mathcal{T}_1$ et $\mathcal{T}_2$ n'ont pas de symboles communs (i.e. quand leurs signatures sont disjointes), on a le résultat suivant

Théorème: Soient deux théories cohérentes et disjointes $\mathcal{T}_1$ et $\mathcal{T}_2$, si $\mathcal{T}_1$ et $\mathcal{T}_2$ admettent toutes les deux un modèle de cardinalité infinie, alors l'union $\mathcal{T}_1 \cup \mathcal{T}_2$ est cohérente.

Preuve (voir livre)

Combinaison naïve (1/2)

Même dans le cas simple de théories *disjointes*, la combinaison n'est pas une question triviale

Exemple: soit $\mathcal{T}_1$ la théorie de l'arithmétique linéaire, $\mathcal{T}_2$ la théorie de l'égalité et Φ la formule suivante:

$$f(x) - x = 0 \wedge f(2x - f(x)) \neq x$$

Une première idée pour décider la satisfiabilité de cette formule consiste à appliquer l'algorithme suivant:

Combinaison naïve (1/2)

Même dans le cas simple de théories *disjointes*, la combinaison n'est pas une question triviale

Exemple: soit $\mathcal{T}_1$ la théorie de l'arithmétique linéaire, $\mathcal{T}_2$ la théorie de l'égalité et Φ la formule suivante:

$$f(x) - x = 0 \wedge f(2x - f(x)) \neq x$$

Une première idée pour décider la satisfiabilité de cette formule consiste à appliquer l'algorithme suivant:

1. décomposer Φ en deux formules logiques pures Φ_1 et Φ_2

Combinaison naïve (1/2)

Même dans le cas simple de théories *disjointes*, la combinaison n'est pas une question triviale

Exemple: soit $\mathcal{T}_1$ la théorie de l'arithmétique linéaire, $\mathcal{T}_2$ la théorie de l'égalité et Φ la formule suivante:

$$f(x) - x = 0 \wedge f(2x - f(x)) \neq x$$

Une première idée pour décider la satisfiabilité de cette formule consiste à appliquer l'algorithme suivant:

1. décomposer Φ en deux formules logiques pures Φ_1 et Φ_2
2. conclure que Φ est satisfiable si et seulement si Φ_1 et Φ_2 sont satisfiables dans $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement

55

Combinaison naïve (2/2)

Après la phase de décomposition, on obtient les formules Φ_1 et Φ_2

$$\begin{array}{ll} (\Phi_1) & z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \\ (\Phi_2) & z_1 - x = 0 \wedge 2x - z_2 = z_3 \end{array}$$

Combinaison naïve (2/2)

Après la phase de décomposition, on obtient les formules Φ_1 et Φ_2

$$\begin{array}{ll} (\Phi_1) & z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \\ (\Phi_2) & z_1 - x = 0 \wedge 2x - z_2 = z_3 \end{array}$$

qui sont toutes les deux satisfiables dans $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement

56

Combinaison naïve (2/2)

Après la phase de décomposition, on obtient les formules Φ_1 et Φ_2

$$(\Phi_1) \quad z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x$$
$$(\Phi_2) \quad z_1 - x = 0 \wedge 2x - z_2 = z_3$$

qui sont toutes les deux satisfiables dans $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement

Mais $\Phi_1 \wedge \Phi_2$ n'est pas satisfiable !

Combinaison naïve (2/2)

Après la phase de décomposition, on obtient les formules Φ_1 et Φ_2

$$(\Phi_1) \quad z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x$$
$$(\Phi_2) \quad z_1 - x = 0 \wedge 2x - z_2 = z_3$$

qui sont toutes les deux satisfiables dans $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement

Mais $\Phi_1 \wedge \Phi_2$ n'est pas satisfiable !

$z_1 = f(x) \wedge z_2 = f(x) \Rightarrow z_1 = z_2$ (1)

Combinaison naïve (2/2)

Après la phase de décomposition, on obtient les formules Φ_1 et Φ_2

$$(\Phi_1) \quad z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x$$
$$(\Phi_2) \quad z_1 - x = 0 \wedge 2x - z_2 = z_3$$

qui sont toutes les deux satisfiables dans $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement

Mais $\Phi_1 \wedge \Phi_2$ n'est pas satisfiable !

$z_1 = f(x) \wedge z_2 = f(x) \Rightarrow z_1 = z_2$ (1)

$z_1 - x = 0 \Rightarrow z_1 = x$ (2)

Combinaison naïve (2/2)

Après la phase de décomposition, on obtient les formules Φ_1 et Φ_2

$$(\Phi_1) \quad z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x$$
$$(\Phi_2) \quad z_1 - x = 0 \wedge 2x - z_2 = z_3$$

qui sont toutes les deux satisfiables dans $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement

Mais $\Phi_1 \wedge \Phi_2$ n'est pas satisfiable !

$z_1 = f(x) \wedge z_2 = f(x) \Rightarrow z_1 = z_2$ (1)

$z_1 - x = 0 \Rightarrow z_1 = x$ (2)

Par (1), (2) et $2x - z_2 = z_3$, on a $z_1 = z_2 = z_3 = x$

Combinaison naïve (2/2)

Après la phase de décomposition, on obtient les formules Φ_1 et Φ_2

$$(\Phi_1) \quad z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x$$
$$(\Phi_2) \quad z_1 - x = 0 \wedge 2x - z_2 = z_3$$

qui sont toutes les deux satisfiables dans $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement

Mais $\Phi_1 \wedge \Phi_2$ n'est pas satisfiable !

$z_1 = f(x) \wedge z_2 = f(x) \Rightarrow z_1 = z_2$ (1)

$z_1 - x = 0 \Rightarrow z_1 = x$ (2)

Par (1), (2) et $2x - z_2 = z_3$, on a $z_1 = z_2 = z_3 = x$

Par congruence, on a $f(x) \neq z_2$ ce qui contredit $f(x) = z_2$

Combinaison naïve (2/2)

Après la phase de décomposition, on obtient les formules Φ_1 et Φ_2

$$(\Phi_1) \quad z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x$$
$$(\Phi_2) \quad z_1 - x = 0 \wedge 2x - z_2 = z_3$$

qui sont toutes les deux satisfiables dans $\mathcal{T}_1$ et $\mathcal{T}_2$, respectivement

Mais $\Phi_1 \wedge \Phi_2$ n'est pas satisfiable !

$z_1 = f(x) \wedge z_2 = f(x) \Rightarrow z_1 = z_2$ (1)

$z_1 - x = 0 \Rightarrow z_1 = x$ (2)

Par (1), (2) et $2x - z_2 = z_3$, on a $z_1 = z_2 = z_3 = x$

Par congruence, on a $f(x) \neq z_2$ ce qui contredit $f(x) = z_2$

Cette méthode naïve de combinaison ne fonctionne pas !

Interpolants

La non-satisfiabilité d'une telle conjonction $\Phi_1 \wedge \Phi_2$ est localisée dans une formule close, appelée interpolant, qui ne contient que des symboles communs aux deux théories

Théorème Cohérence jointe [Robinson, Craig].

Soient deux théories cohérentes $\mathcal{T}_1$ et $\mathcal{T}_2$, l'union $\mathcal{T}_1 \cup \mathcal{T}_2$ est incohérente ssi il existe un interpolant Φ tel que $\mathcal{T}_1 \models \Phi$ et $\mathcal{T}_2 \models \neg\Phi$

Dans l'exemple précédent, un interpolant pour la conjonction $\Phi_1 \wedge \Phi_2$ est par exemple $x = z_1 \wedge x = z_3$

La méthode de Nelson-Oppen

Les algorithmes de combinaison de procédures de décision implémentent des techniques efficaces pour calculer ces interpolants, c'est le cas de la méthode de Nelson-Oppen

La méthode de Nelson-Oppen

Les algorithmes de combinaison de procédures de décision implémentent des techniques efficaces pour calculer ces interpolants, c'est le cas de la méthode de Nelson-Oppen

Cette méthode s'applique à des théories disjointes ayant les deux propriétés suivantes :

- **stables à l'infini**
  Toute formule satisfiable dans ces théories doit l'être au moins pour un modèle de cardinalité infinie
- **convexes**

  Si $\mathcal{T} \models \forall \vec{x}.(\phi \Rightarrow x_1 = y_1 \vee \ldots \vee x_n = y_n)$ alors
  il existe $1 \leq i \leq n$ tel que $\mathcal{T} \models \forall \vec{x}.(\phi \Rightarrow x_i = y_i)$

Algorithme de Nelson-Oppen

```
def NO(φ):
  (φ1, φ2) = purification(φ)
  while True:
    (res1, eq1) = Tsolve1(φ1)
    (res2, eq2) = Tsolve2(φ2)
    if res1 == UNSAT or res2 == UNSAT:
      return UNSAT
    if eq1 == eq2 == ∅:
       return SAT
    φ1 = φ1 ∧ eq1 ∧ eq2
    φ2 = φ2 ∧ eq1 ∧ eq2
```

Preuve de correction (voir livre, technique d'amalgamation)

NO : exemple

$$f(x) - x = 0 \wedge f(2x - f(x)) \neq x.$$

$$\begin{array}{lcl} \phi_1 & \equiv & z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \\ \phi_2 & \equiv & z_1 - x = 0 \wedge 2x - z_2 = z_3 \end{array}$$

NO : exemple

$$f(x) - x = 0 \wedge f(2x - f(x)) \neq x.$$

$$\begin{array}{lcl} \phi_1 & \equiv & z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \\ \phi_2 & \equiv & z_1 - x = 0 \wedge 2x - z_2 = z_3 \end{array}$$

$$(1) \quad \begin{array}{ll} \phi_1 & z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \\ \phi_2 & z_1 - x = 0 \wedge 2x - z_2 = z_3 \\ (res_1, eq_1) & (\mathsf{SAT}, z_1 = z_2) \\ (res_2, eq_2) & (\mathsf{SAT}, z_1 = x) \end{array}$$

NO : exemple

$$f(x) - x = 0 \wedge f(2x - f(x)) \neq x.$$

$$\begin{array}{lcl} \phi_1 & \equiv & z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \\ \phi_2 & \equiv & z_1 - x = 0 \wedge 2x - z_2 = z_3 \end{array}$$

---

(2)	ϕ_1	$z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \wedge$ $z_1 = z_2 \wedge z_1 = x$
	ϕ_2	$z_1 - x = 0 \wedge 2x - z_2 = z_3 \wedge$ $z_1 = z_2 \wedge z_1 = x$
	(res_1, eq_1)	$(\mathsf{SAT}, \emptyset)$
	(res_2, eq_2)	$(\mathsf{SAT}, x = z_3)$

NO : exemple

$$f(x) - x = 0 \wedge f(2x - f(x)) \neq x.$$

$$\begin{array}{lcl} \phi_1 & \equiv & z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \\ \phi_2 & \equiv & z_1 - x = 0 \wedge 2x - z_2 = z_3 \end{array}$$

---

(3)	ϕ_1	$z_1 = f(x) \wedge z_2 = f(x) \wedge f(z_3) \neq x \wedge$ $z_1 = z_2 \wedge z_1 = x \wedge x = z_3$
	ϕ_2	$z_1 - x = 0 \wedge 2x - z_2 = z_3 \wedge$ $z_1 = z_2 \wedge z_1 = x \wedge x = z_3$
	(res_1, eq_1)	$\mathsf{UNSAT}, -$
	(res_2, eq_2)	$\mathsf{SAT}, -$

Points faibles de la méthode de Nelson-Oppen

Cette méthode a deux points faibles:

- Elle nécessite d'instrumenter les procédures de décision pour déduire les égalités impliquées par une formule
- Le traitement des théories non-convexes nécessite
  1. d'étendre le mécanisme de déduction pour générer des disjonctions d'égalités
  2. de traiter ces disjonctions au sein même de l'algorithme de la combinaison de théories, alors que ces disjonctions seraient mieux traitées par un solveur SAT

Des variantes de cette méthode permettent d'éviter ces problèmes:

- L'**analyse pas cas à la demande** (*splitting on demand*)
- La **combinaison retardée** (*Delayed Theory Combination*)
- La **combinaison dirigée par les modèles** (*model-based* theory combination)

L'analyse par cas à la demande

Cette extension s'appuie sur une fonction `splitting_on_demand` pour ajouter à la formule d'entrée des clauses d'analyse par cas modulo théorie

```
def cdclT(φ):
  μ = []
  dl = 0
  while True:
    res = theory_and_boolean_propagation(φ,μ)
    if res == SAT:
      if all_variables_are_assigned(φ,μ):
        return SAT
        splitting_on_demand(φ,μ)
        ...
      else:
        ...
```

(restrictions pour garantir la terminaison et la correction)

La combinaison retardée (DTC)

Faire coopérer directement les procédures de décision avec le solveur SAT, ce dernier se chargeant de l'échange des égalités (entre variables d'interface) entre les procédures

Peut être vu comme l'intégration de NO dans CDCL

Après avoir purifié la formule en entrée ϕ en deux formules ϕ_1 et ϕ_2, DTC initialise une ensemble contenant l'ensemble des atomes sur lesquels effectuer des décisions, ainsi que toutes les égalités entre les variables partagées par ϕ_1 et ϕ_2

(voir algo dans le livre)

La combinaison dirigée par les modèles (MBTC)

DTC doit gérer un nombre quadratique d'égalités entre variables partagées et il choisit en aveugle parmi ces égalités

MBTC étend DTC afin que les procédures de décision de déduisent simplement des égalités impliquées par un modèle candidat

La combinaison dirigée par les modèles (MBTC)

DTC doit gérer un nombre quadratique d'égalités entre variables partagées et il choisit en aveugle parmi ces égalités

MBTC étend DTC afin que les procédures de décision de déduisent simplement des égalités impliquées par un modèle candidat

Par exemple, à partir de:

$$0 \leq x \leq 1 \wedge 0 \leq y \leq 1 \wedge z = y - 1$$

la théorie de l'arithmétique linéaire sur les entiers peut construire le modèle $\{x = 0, y = 0, z - 1\}$ et déduire $x = y$

Si ces égalités sont en contradiction avec les autres théories, il faut rebrousser. Pour cela, ces égalités dirigées par les modèles sont générées en appelant `theory_splitting_on_demand`

Par exemple, $x = y$ sera donnée par la clause $x = y \vee x \neq y$ (avec priorité de décision à $x = y$)

CONCLUSION

La révolution SMT

70's: Stanford Pascal Verifier (algorithm de Nelson-Oppen)
1984: Shostak algorithm
1992: Simplify
1995: SVC
2001: ICS
2002: CVC, haRVey
2004: CVC Lite
2005: Barcelogic, MathSAT
2005: Yices
2006: CVC3, Alt-Ergo
2007: Z3, MathSAT4
2008: Boolector, OpenSMT, Beaver,Yices2
2009: STP, VeriT
2010: MathSAT5, SONOLAR
2011: STP2, SMTInterpol
2012: CVC4
2015: OpenSMT2
2015: SMT-Rat
...

SMT-Lib

The Satisfiability Modulo Theory Library (SMT-Lib)

http://www.smtlib.org/

Une communauté internationale autour des solveurs SMT:

- Une description rigoureuse du langage logique d'entrée (SMT-Lib2) (avec une liste de théories)
- L'organisation d'une compétition (SMT-Comp) tous les ans avec plusieurs catégories (QF-LIA, AUFLIA, etc.)
- Une très grande bibliothèque de benchmarks (qu'il faut continuer à alimenter)