Analyse de programmes par SMT-Solving

Julien Henry

Julien.Henry@ensimag.imag.fr

Encadrants :

David Monniaux

David.Monniaux@imag.fr

Matthieu Moy

Matthieu.Moy@imag.fr

Travaux d'études et de recherche 2010

École Nationale Supérieure d'Informatique et de Mathématiques Appliquées, Grenoble

Préambule

Ce stage de Travaux d'Études et de Recherche a été effectué au laboratoire Verimag au sein de l'équipe Synchrone.

Je tiens à remercier toutes les personnes qui m'ont aidé durant ce TER, en particulier mes deux tuteurs, David Monniaux et Matthieu Moy, chercheurs à Verimag, pour leurs disponibilités dès lors que je rencontrais des difficultés, ainsi que pour leur aide à la rédaction de cet article.

Je voudrais également remercier en particulier Kevin Marquet, post-doctorant au laboratoire Verimag, pour sa très précieuse aide en ce qui concerne la prise en main du code de LLVM.

1 Introduction

Certains secteurs de l'industrie nécessitent des programmes informatiques d'un très haut niveau de fiabilité. Pour cela, il existe différentes méthodes de vérification de programmes. Ces méthodes sont très variées, de la simple revue de code à la démonstration formelle de la correction du programme. Dans cet article, nous nous intéresserons à l'analyse par SMT-Solving (Satisfiability Modulo Theory).

Le Bounded Model Checking par SMT consiste à modéliser l'exécution d'un programme par une formule booléenne, dont la résolution est donnée à un programme (appelé SMT-solver) qui calcule la satisfiabilité de la formule. Si la formule que l'on obtient est satisfiable, cela veut dire que le programme possède une exécution qui engendre une erreur. Dans le cas où la formule est insatisfiable, alors on a prouvé que le programme est correct pour tous les cas d'utilisation possibles de longueur bornée par un entier N. Malgré tout, le problème de satisfiabilité d'une formule propositionnelle est bien connu comme étant $\mathcal{NP}$ -complet. Cela impose une limite à cette méthode, qui peut alors être limitée par des raisons d'efficacité. De plus, le problème de désicion SMT est plus complexe que le problème de désision SAT. En effet, il doit tenir compte de la logique de l'égalité, des fonctions non interprétées etc. [4]

Toutefois, la recherche sur les SMT-solvers fait d'énormes progrès, ce qui permet à cette méthode de vérification d'être de plus en plus performante.

Ma contribution est la création d'un outil permettant de lire en entrée un programme en langage C, sans boucle et à une seule fonction. Ce programme C est annoté d'un ensemble d'hypothèses initiales sur ses variables, et de plusieurs assertions à vérifier à divers points du programme. L'outil permet alors de montrer, pour les hypothèses données, si les assertions à vérifier sont toujours correctes où non. Dans le cas où une assertion peut être fausse, il peut donner un exemple qui produit l'erreur.

Dans notre outil, nous avons besoin de lire un fichier C, de le transformer en un fichier contenant la formule logique associée, puis de vérifier la satisfiabilité de cette formule. Pour la première tranformation, nous avons besoin d'une infrastructure de compilation pour «parser» le programme C. Nous avons donc utilisé le compilateur LLVM.

LLVM [5] est un compilateur qui permet de transformer des programmes de plusieurs langages (tels que C, C++ etc) en programmes exécutables. Il a la particularité d'être très modulaire : il commence par transformer le programme en une représentation intermédiaire, que nous appellerons "Bitcode", lors d'une phase appelée Frontend, d'y appliquer des optimisations, puis de compiler le Bitcode vers les cibles grâce au Backend. LLVM est codé en C++.

L'outil de preuve que nous avons implémenté est basé sur le Frontend de LLVM. Il permet de faire une transformation du Bitcode en un fichier benchmark de format SMT-Lib. Ce dernier format de fichier est un format d'entrée standardisé lisible par les principaux SMT-solvers actuels [6]. Notre outil est implémenté en C++ pour pouvoir utiliser les bibliothèques de lecture du Bitcode de LLVM.

Ici, nous utiliserons le SMT-solveur Z3 [3].

Dans cet article, nous commencerons par présenter un pré-requis sur LLVM dont nous avons besoin pour notre programme, et le principe du SMT-solving. Ensuite, nous verrons de façon plus détaillée le fonctionnement du programme. Enfin, nous verrons les résultats obtenus, en faisant des tests de fichiers benchmarks. Nous verrons alors un exemple complet expliquant les résultats obtenus, puis nous ferons une comparaison des performances du programme, en particulier le temps d'exécution, avec un autre programme existant, CBMC (C Bounded Model Checker) [2].

2 Pré-requis

2.1 SMT-Solving

Le problème SMT est un problème de décision pour des formules logiques. Ces formules logiques sont exprimées dans la logique du premier ordre avec l'égalité. Pour résoudre ce problème de décision, on se base sur une théorie donnée, qui peut être par exemple la théorie des entiers, des tableaux, des pointeurs, etc [4], afin de déterminer si une formule logique est satisfiable ou non, modulo la théorie que l'on a choisi.

Ainsi, le résultat *satisfiable* pour une formule logique donnée dépend de la théorie que l'on utilise.

Exemple *Soit la formule logique :*

$$F = x > 0 \land y \leq 1 \land z > x \land z < y$$

Si on utilise la théorie des nombres réels, cela revient à rechercher $x, y, z \in \mathbb{R}$ tel que F soit vraie. Cette formule est alors satisfiable, en prenant par exemple $x = \frac{1}{10}, y = 1$ et $z = \frac{1}{2}$.

En revanche, en résolvant cette formule en travaillant avec des relations linéaires sur les entiers, on veut trouver $x, y, z \in \mathbb{Z}$ tel que F soit vraie. On a alors une formule insatisfiable.

La recherche sur le SMT-solving est en progrès depuis quelques années, et il existe maintenant des programmes permettant de résoudre ce problème de décision en un temps raisonnable.

Les SMT-solvers sont très complexes, et sont composés d'algorithmes de décision tels que l'algorithme du simplexe [1] ou l'algorithme DPLL [4].

L'intérêt de ce type de vérification formelle est de faire une véritable preuve de la correction d'un programme pour toutes les entrées possibles.

1. L'algorithme du simplexe est en fait un algorithme d'optimisation sur des inégalités linéaires. Ici, on s'en sert uniquement pour trouver un résultat, c'est à dire une affectation des variables vérifiant les inégalités.

2.2 LLVM

LLVM est une infrastructure de compilateur très modulaire, implémentée en C++, qui permet de compiler les programmes C en plusieurs étapes. LLVM lit le fichier C et crée une représentation intermédiaire du programme appelée Bitcode. Ce Bitcode peut ensuite être optimisé puis utilisé pour générer un fichier de sortie grâce à un Back-end.

2.2.1 Forme SSA

La forme SSA (Static Single Assignment form) est une forme de représentation des variables du programme source. Dans cette forme, une variable ne peut être définie que par une unique instruction.

Ainsi, la suite d'instructions suivante, utilisant uniquement une variable `z` :

```
z = x + 5;
z = z + 2;
```

se transforme en la suite d'instructions suivante :

$$z_1 = x + 5;$$
$$z_2 = z_1 + 2;$$

Cette transformation est essentielle et permet à une variable de ne plus changer de valeur au cours de l'exécution du programme.

Le Bitcode de LLVM est sous la forme SSA. Cette forme va nous permettre d'obtenir facilement une variable différente par état d'une variable du programme de départ. Cela va nous servir pour pouvoir garder en mémoire la suite des états de chacunes des variables de notre programme.

2.2.2 Phase d'optimisation

Le compilateur LLVM utilise par défaut la mémoire pour stocker les variables du programme source. Ainsi, chaque lecture d'une variable requiert un appel à une instruction `load` dans la mémoire, et chaque écriture dans une variable requiert un appel à une instruction `store`.

De ce fait, dans notre programme, il faudrait pouvoir faire une modélisation de la mémoire, et implémenter les traitements des instructions `store` et `load`.

Dans cet article, nous utiliserons systématiquement la phase d'optimisation de LLVM `-mem2reg`. Cette optimisation permet de ne plus stocker les variables dans la mémoire mais plutôt dans des registres SSA (chaque registre n'est affecté qu'une seule fois). Ainsi, il n'y a plus d'appels aux instructions `load` et `store`, et cela simplifie considérablement le traitement. Avec cette optimisation, on utilise désormais des noeuds *Phi* (voir 3.2.2) si besoin, pour associer les bonnes valeurs aux variables.

Exemple *Soit le programme C suivant :*

```
int G;
int H;
int test(int Cond) {
  int x;
  if (Cond) {
    x = G;
  } else {
    x = H;
  }
  return x;
}
```

Le Bitcode de LLVM associé à ce programme est le suivant, lorsque l'on n'utilise pas l'optimisation `-mem2reg`*.*

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Cond) {
entry:
  %X = alloca i32        ; type of %X is i32*
  br i1 %Cond, label %c_true, label %c_false

c_true:
  %X.0 = load i32* @G
        store i32 %X.0, i32* %X   ; Update X
  br label %c_next

c_false:
  %X.1 = load i32* @H
        store i32 %X.1, i32* %X   ; Update X
  br label %c_next

c_next:
  %X.2 = load i32* %X          ; Read X
  ret i32 %X.2
}
```

Le même Bitcode auquel on applique l'optimisation `-mem2reg` *devient :*

```
@G = weak global i32 0
@H = weak global i32 0

define i32 @test(i1 %Cond) {
entry:
  br i1 %Cond, label %c_true, label %c_false

c_true:
  %X.0 = load i32* @G
  br label %c_next

c_false:
  %X.1 = load i32* @H
  br label %c_next

c_next:
  %X.2 = phi i32 [%X.1,%c_false],[%X.0,%c_true]
  ret i32 %X.2
}
```

2.3 Notion de BasicBlock

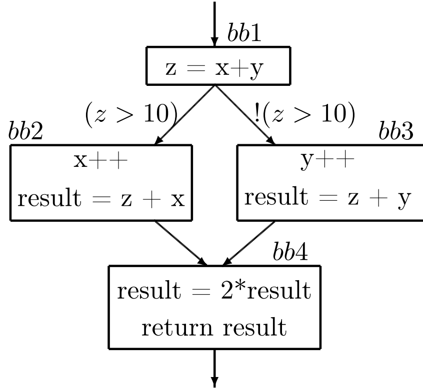
Un programme est composé traditionnellement d'un ensemble de fonctions. Dans cet article, nous traitons uniquement le cas où le programme est composé d'une unique

fonction. On définit un basicblock comme étant une suite d'instructions qui s'exécutent de façon linéaire. Les basicblocks forment alors les noeuds d'un graphe dont les arêtes signifient que l'on peut passer d'un basicblock à un autre. Une trace d'exécution du programme est alors une suite de basicblocks exécutés.

Prenons l'exemple de la fonction suivante :

```
int function(int x, int y, int z) {
  int result;
  z = x + y;
  if (z > 10) {
    x++;
    result = z + x;
  } else {
    y++;
    result = z + y;
  }
  result = 2*result;
  return result;
}
```

On peut alors distinguer les basicblocks suivants :



On constate que certaines transitions entre basicblocks sont soumises à des conditions provenant d'instructions *if* ou *switch*.

On peut donc noter BB l'ensemble des basicblocks d'une fonction.

L'ensemble des transitions entre les blocs est un sous ensemble de $BB \times BB$ noté T. À chaque transition $(BB_i, BB_j) \in T$, on peut associer un prédicat noté $P_{i,j}$, qui correspond en fait aux conditions nécessaires pour pouvoir suivre la transition.

L'ensemble des prédécesseurs d'un basicblock BB_j a donc une expression de la forme :

$$pred(BB_j) = \{BB_i \in BB / (BB_i, BB_j) \in T\}$$

Ainsi, si bb_i est une variable booléenne qui vaut *true* si et seulement si la trace d'exécution passe par le bloc BB_i, alors

$$bb_i = \bigvee_{j \in pred(BB_i)} (bb_j \wedge P_{j,i}) \quad (1)$$

Sachant que le basicblock bb_1 correspondant à l'entrée du programme est toujours parcouru, on peut dire que $bb_1 = true$. Par construction, on peut donc calculer tous les prédicats relatifs aux bb_i.

Dans notre implémentation, on utilise une variable booléenne par basicblock. La valeur de cette variable booléenne est déterminée par le résultat que l'on vient de voir.

3 Traduction

3.1 Vue d'ensemble

Nous voulons faire des preuves sur les états de variables de notre programme à certains instants de l'exécution, en considérant un ensemble d'hypothèses initiales sur ces variables. Par exemple :

```
void function(int x, int y, int z) {
  // Hyp : x > 5 ; y > 8
  ...
  ...
  // z > 0 ?
}
```

Nous voulons ici montrer, sachant qu'en entrée $x > 5$ et $y > 8$, que $z > 0$ à la sortie de la fonction.

Il faut pouvoir spécifier dans le code C les hypothèses que l'on veut faire, ainsi que les formules logiques que l'on veut prouver à certains endroits du code. Pour cela, les programmes à vérifier devront utiliser le fichier `smt.h`, qui dispose des définitions de macros suivantes :

- *assume*(x), où x est une formule booléenne que l'on suppose vraie à l'endroit du code où on le spécifie.
- *check*(x), où x est une formule booléenne que l'on veut prouver comme correcte.

Ces macros font en fait un test sur leur paramètre, et dans le cas ou la formule est fausse, appellent des fonctions dont les noms sont reconnus par notre outil, qui peut alors effectuer un traitement particulier et que nous verrons dans la suite.

Nous allons construire une formule logique associée au Bitcode LLVM source telle que cette formule soit insatisfiable si et seulement si le programme source est correct.

Pour celà, il faut que cette formule soit satisfiable si et seulement si il existe un trace d'exécution du programme qui aboutit à un *check* faux avec les *assume* vrais.

La formule logique qui va être construite représente donc notre programme, et les différentes affectations des variables permettent de tester les différentes traces d'exécution possibles. L'objectif est alors de trouver une affectation des variables de notre formule qui prouve qu'à partir d'une entrée vérifiant les hypothèses (*assume* vrai), on arrive à un *check faux*. C'est le SMT-solver qui va trouver cette affectation dans le cas où il est possible. Celui-ci sera alors un exemple d'exécution du programme qui engendre l'erreur. À l'inverse, si le SMT-solver établit que la formule est insatisfiable, cela signifie qu'aucune exécution du programme soumise aux hypothèses n'aboutira à un *check* faux.

3.2 Constructions

Afin de générer la formule logique et le fichier de sortie SMT-Lib, nous parcourons le programme à deux reprises. La première passe sert à mettre en place les variables booléennes relatives aux basicblocks, ainsi que toutes les variables entières et booléennes intermédiaires. La seconde passe crée la formule logique à proprement parler, en utilisant les variables crées lors de la passe précédente.

3.2.1 Première passe

La première passe du programme permet de créer chacunes des variables booléennes et entières dont on va avoir besoin dans le fichier SMT-Lib à générer.

Pour créer ces variables, on parcourt chacune des instructions de chacun des basicblocks du programme. Pour chaque basicblock, on crée une variable booléenne qui correspond au bb_i vu précédemment, et qui permettra de savoir si l'exécution passe par le basicblock ou non.

On crée également une variable booléenne par condition à vérifier ou à supposer (avec *check* ou *assume*).

Enfin, on parcourt chacune des instructions, et selon le type de l'instruction, on crée ou non une variable intermédiaire qui correspond au résultat renvoyé par l'instruction.

Cette première passe du programme source aurait pu être évitée. En effet, il est possible, lors de la création de la formule (dans la passe 2), de créer les variables temporaires dont on a besoin au fur et à mesure. Cependant, il est plus clair pour le code SMT-Lib de déclarer les variables avant la formule logique. Ainsi, le fait d'avoir fait deux passes permet au fichier SMT-Lib d'être écrit dans l'ordre.

3.2.2 Seconde passe

La seconde passe est uniquement destinée à créer la formule logique associée au programme. Dans le fichier généré, on retrouve cette formule après la balise `:formula`. Elle effectue un parcours de chacun des basicblocks de la fonction.

Pour chacun de ces blocs, on commence par affecter le booléen associé au bloc avec la bonne formule (formule (1)).

Ensuite, on doit écrire une formule logique qui correspond au basicblock. Toutefois, il faut tenir compte du fait qu'on ne passe pas nécessairement par ce bloc, donc la formule logique associée au basicblock BB_i est de la forme :

```
(or (not bb_i) (...F...))
```

F est alors une formule logique qui est déterminée en parcourant la suite des instructions effectuées et en leur appliquant un traitement particulier, selon la nature de cette instruction. Chacune des instructions du basicblock peut alors être associée à une formule logique traduite en SMT-Lib. La formule F est alors la conjonction de chacune de ces formules, par propriété d'un basicblock.

Binary Operator Ce type de noeud est rencontré lors d'une opération binaire, qu'elle soit booléenne ou entière. On crée donc une variable intermédiaire spécifique pour le résultat de l'opération, que l'on affecte à la bonne valeur, en faisant simplement une réécriture de l'opération avec la nouvelle syntaxe.

Exemple *en C :*

```
z = x+y;
```

Bitcode LLVM associé :

```
%0 = add nsw i32 %x, %y ; <i32> [#uses=0]
```

Traduction en SMT-Lib :

```
(= x_0 (+ x y))
```

Cmp Ce type de noeud correspond aux instructions de comparaisons. On crée alors une variable booléenne dédiée au résultat de cette comparaison, que l'on affecte avec la bonne formule. Cette formule est encore une fois une réécriture du prédicat avec la bonne syntaxe.

Exemple *en C :*

```
z = (x < y);
```

Bitcode LLVM associé :

```
%0 = icmp slt i32 %x, %y ; <i1> [#uses=1]
```

Traduction en SMT-Lib :

```
(= x_0 (< x y))
```

Branch Cette instruction correspond à un branchement vers un autre basicblock. Ce branchement peut être conditionné ou non. Dans le cas où le branchement est inconditionnel, alors on écrit simplement que la variable du bloc vers lequel on se branche est vraie.

Dans le cas où le branchement est conditionnel, alors on écrit une instruction `ite` (*if-then-else* en SMT-Lib), qui teste la condition et qui force la variable du bloc vers lequel on se dirige à *vrai.*

Exemple *en C :*

```
if (z > 0) {
  a = 2; // bloc bb
} else {
  a = 3; // bloc bb1
}
```

Bitcode LLVM associé :

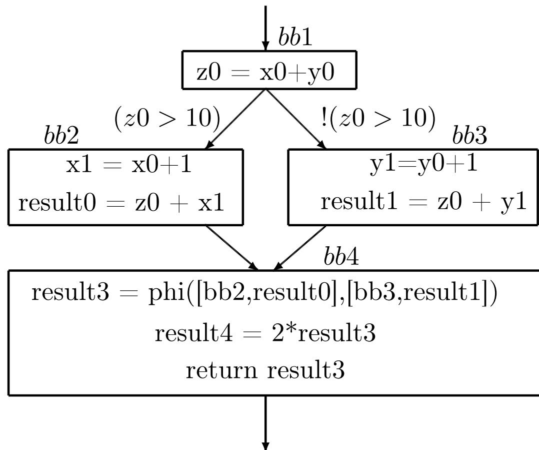
```
%1 = icmp sgt i32 %0, 0 ; <i1> [#uses=1]
br i1 %1, label %bb, label %bb1
```

Traduction en SMT-Lib :

```
(= x_1 (> x_0 0))    # x_0 est en fait z
(ite x_1 bb bb1)
```

Phi Le *Phi* est un noeud du graphe du bitcode LLVM particulier. En effet, il ne se rapporte directement à aucune instruction du programme en C de départ. Il s'agit en fait d'une fonction qui permet de donner une valeur à une variable de tout type, en fonction du basicblock exécuté précédemment. Autrement dit, si le *Phi* se trouve dans le bloc BB_i, alors on recherche $BB_j \in pred(BB_i)$ tel que bb_j soit évalué à vrai. *Phi* est une fonction de ces variables booléennes vers le type de la variable à affecter.

Pour illustrer l'instruction Phi, on peut reprendre l'exemple utilisé pour expliquer les basicblocks. On considère maintenant que le graphe est sous la forme SSA, donc dans *bb*4, la valeur de *result* dépend du basicblock parent.



On a donc ici une instruction phi pour commencer le basicblock *bb*4, qui permet de stocker dans *result*3 la valeur *result*0 si *bb*2 = *true*, et la valeur *result*1 si *bb*3 = *true*.

On arrive donc à attribuer la valeur à la variable en faisant une cascade de `ite` dans le code SMT-Lib, en testant les valeurs de chacune des variables booléennes correspondant aux blocs parents possibles.

Exemple *en C :*

```
if (z > 0) {
  a = 0;  // basicblock bb1
} else if (z < 0) {
  a = 2;  // basicblock bb2
} else {
  a = 1;  // basicblock bb3
}
z = a + 5;
```

Bitcode LLVM associé :

```
%a_addr.0 = phi i32 [0,%bb1],[2,%bb2],[1,%bb3];
%3 = add nsw i32 %a_addr.0, 5
```

Traduction en SMT-Lib :

```
(= a (ite bb3 1 (ite bb1 0 2)))
(= z (+ a 5))
```

Select L'instruction *select* sert à pouvoir affecter une valeur à une variable en dépendant d'une condition, tout en évitant un branchement. Elle remplace en fait le *Phi* dans les cas les plus simples. Ainsi, il est très simple de transformer ce noeud en un *if-then-else* en SMT. On crée une variable destinée à être le résultat du *select*, et on teste la condition avec un *ite* pour affecter la bonne valeur à notre variable.

Exemple *en C :*

```
a = 2;
if (z > 0) {
  a = 0;
}
return a;
```

Bitcode LLVM associé :

```
%0 = icmp sgt i32 %z, 0
%a_addr.0 = select i1 %0, i32 5, i32 7
ret i32 %a_addr.0
```

Traduction en SMT-Lib :

```
(= x_0 (> z 0))
(= x_1 (ite x_0 0 2))
```

Switch Le *switch* ne nécessite pas de traitement particulièrement différent d'une instruction *if then else*. En fait, pour chacune des conditions du switch, on associe un basicblock qui va contenir l'ensemble des instructions à effectuer, de la même façon que pour un *if then else* classique. Pour le dernier cas *default*, on a également un basicblock, dont la variable booléenne sera vraie si et seulement si les variables booléennes relatives aux cas précédemment traités sont évaluées à *faux*. Dans la plupart des cas, une instruction *switch* est suivie d'une instruction *Phi* dans le bitcode de LLVM.

Exemple *en C :*

```
switch (z) {
case 1:
     a = 0;
     break;
case 2:
     a = 1;
     break;
default:
     a = 42;
}
return a;
```

Bitcode LLVM associé :

```
switch i32 %z, label %bb2 [
     i32 1, label %bb
     i32 2, label %bb1
  ]
...
// un bloc par case, qui branchent vers bb3
...
```

```
bb3:
%a_addr.0 = phi i32 [42,%bb2],[1,%bb1],[0,%bb];
```

Return L'instruction *return* ne nécessite aucun traitement particulier dans le cas de notre programme, où l'on ne vérifie qu'une seule fonction. En effet, si on est arrivé à ce point du programme, alors il s'est à priori exécuté correctement. Le SMT-solver devrait alors renvoyer comme résultat que la formule n'est pas satisfiable, c'est-à-dire que les tests de conditions par *check* et *assume* sont corrects.

CallSite Un *Callsite* est une instruction d'appel à une fonction externe. Ceci n'est pas implémenté dans notre programme, sauf pour des cas particuliers. En effet, seuls les appels de fonction à *verif_stop* et *verif_warn* sont reconnus. Ces appels de fonction permettent de traiter les *check* et les *assume* que le programmeur à mis dans son code C. En effet, *check* et *assume* sont des macros définies comme suit :

- *assume*(x) :

```
do { if (!(x)) verif_stop();} while(0)
```

- *check*(x) :

```
do { if (!(x)) verif_warn();} while(0)
```

En fait, ces deux macros permettent d'ajouter une formule logique à la fin de notre formule associée au programme. Pour chaque instruction *check* ou *assume*, on associe une variable booléenne, qui vaudra *true* si et seulement si la trace d'éxécution passe par la fonction *verif_stop* ou *verif_warn* associée.

On note $(warn_i)_{i=1..n}$ les n variables booléennes associées aux *verif_warn* et $(verif_i)_{i=1..m}$ les m variables associées aux *verif_stop*.

Puisque notre programme veut trouver ces traces d'exécution, on ajoute donc en fin de notre formule l'expression :

$$\bigvee_{i\in 1..n}\left(\left(\bigwedge_{j\in 1..m} stop_j\right)\wedge warn_i\right)$$

Exemple *en C :*

```
check(x > 0);
```

Bitcode LLVM associé :

```
  %0 = icmp sle i32 %x, 0 ;
  br i1 %0, label %bb, label %bb1
bb:
  call void (...)* @verif_warn() nounwind
  br label %bb1
```

Traduction en SMT-Lib :

```
(= x_0 (<= x 0))
(ite x_0 bb return)

(= bb (or  ((and entry x_0))))
(or (not bb)
(and
(= warn_0 true)
))

...

((warn_0))
```

Instructions non traitées Le programme ne traite pas un grand nombre d'instructions, par exemple les *store* et les *load* dans la mémoire. En fait, seuls les programmes sans boucle, sans tableaux, avec des variables uniquement entières ou booléennes, peuvent être testés par la version actuelle du programme.

4 Résultats

4.1 Exemple

Nous allons maintenant voir un cas d'utilisation du programme pour une fonction simple, qui comporte trois variables, dont deux sont comditionnées, et deux tests *check*. On fait également une instruction *if* pour créer différents basicblocks.

```
#include "smt.h"

int exemple(int x, int y) {
      int z;

      assume(x < 10 && y < 50);
      z = x + y;

      check(z < 60);

      if (z > 30) {
            z -= 30;
      } else {
            z += 5;
      }
      check(z <= 30);
}
```

On utilise alors notre programme pour transformer ce code en formule SMT (Le contenu intégral du fichier SMT-Lib généré est en annexe page 10). On obtient alors une formule avec les variables suivantes :

- 13 variables temporaires entières ou booléennes servant aux calculs et aux évaluations de conditions.
- les variables x et y qui sont en paramètre de la fonction.
- 7 variables booléennes relatives à chacun des basicblocks du programme.

On remarque que 13 variables temporaires pour les calculs est un nombre assez important en comparant par rapport au nombre de variables initial. On pourrait diminuer ce

nombre en particulier lors des calculs, en testant si ceux-ci sont *inlinable* ou non.

On lance ensuite un SMT-solver, ici *Z3* [3], pour évaluer la satisfiabilité de la formule. On trouve alors que la formule est satisfiable, et le SMT-solver permet de trouver une trace d'exécution menant à une erreur.

Dans cet exemple, *z3* propose le chemin suivant :

1. On choisit $x = 9$ et $y = 21$.
2. Le calcul de z donne 30.
3. L'évaluation de la condition $z > 30$ est fausse.
4. z passe à 35
5. le *check* final renvoie l'erreur.

4.2 Comparaison avec CBMC

Nous voulons faire une comparaison de notre programme avec CBMC[2], notamment en terme de temps de calcul. Pour faire ces tests, on doit adapter le programme source en utilisant les *assert* classiques, contenus dans `assert.h` plutôt que notre `smt.h`. En effet, CBMC utilise ces `assert` pour générer sa formule logique.

De même, au lieu d'utiliser une fonction *assume(x)* pour donner les préconditions, nous les remplacerons par des éléments de la forme :

```
if (!x) return;
```

Nous ferons deux types de tests différents :

1. Nous commencerons par calculer le temps total d'exécution de CBMC, en partant du programme source jusqu'à la réponse de satisfiabilité de la formule. Nous comparerons ce temps avec le temps total d'exécution pour :
   – générer le Bitcode LLVM relatif au programme source
   – exécuter notre programme qui génère le fichier SMT-Lib
   – résoudre ce fichier SMT avec le SMT-solver *z3*.
2. Ensuite, nous génèrerons un fichier SMT-Lib à la fois avec notre programme, et avec CBMC. Ceci se fera avec l'option `-smt1` de CBMC. Cette option crée un fichier SMT-lib qui est ensuite lancé dans le SMT-solver *Boolector*.

4.2.1 Tests de performance

Nous testons le temps de calcul pour différents programmes sources. Pour plus de précision, nous ferons pour chacun des fichiers source 100 simulations, ce qui nous permettra d'avoir une moyenne du temps d'exécution plus correcte.

Voici un exemple en prenant un programme C générant un fichier SMT-Lib d'environ 10000 lignes :

Fichier	**Temps de calcul**		
	CBMC	CBMC –smt1	**llvm2smt**
counter.c	1.141s	1.142s	**1.050s**

En effectuant un certain nombre de tests, nous pouvons nous apercevoir que les résultats que l'on obtient sont relativement similaires à CBMC en terme de temps de calcul. Ce temps de calcul dépend en fait principalement du temps de résolution du SMT-solver. Malgré tout, ce résultat permet de montrer que notre programme a un coût raisonnable.

5 Perspectives

Notre programme est encore très incomplet et pourrait être compléter pour pouvoir vérifier des programmes plus complexes. Par exemple, on pourrait :

– traiter les programmes avec boucles en faisant du *bounded-model-checking*, en déroulant les boucles jusqu'à une profondeur donnée [1].
– permettre de tester des programmes avec des tableaux. Pour cela, on pourrait utiliser des fonctions non-interprétées (Voir [4]).
– permettre de traiter des programmes contenant des appels de fonctions. Pour cela, on pourrait "encapsuler" les formules logiques créées à partir de chacune des fonctions du programme.
– réduire le nombre de variables temporaires crées dans le fichier SMT-Lib en tirant parti des instructions *inlinable*.
– pour avoir une interface un peu plus conviviale, il faudrait «re-parser» la sortie du SMT-Solveur pour pouvoir afficher clairement la trace d'exécution qui crée l'erreur.

6 Conclusion

Dans cet article, nous avons présenté une méthode de vérification de programme par SMT-Solving. Nous avons expliqué notre approche et un programme permettant de transformer du Bitcode du compilateur LLVM en fichier SMT-Lib lisible par les SMT-solvers actuels. Les résultats de cette première version de notre programme sont plutôt encourageants en ce qui concerne le temps de calcul par rapport aux autres programmes déjà existants.

En complétant notre programme, en gérant les tableaux, les boucles (par Bounded Model Checking), les instructions non implémentées, etc, un tel outil pourrait être utilisé à l'échelle industrielle pour des programmes nécessitant une haute fiabilité à l'exécution.

7 Bilan sur ce TER

Initialement, j'avais naturellement choisi l'option Travaux d'Études et de Recherche puisque j'envisageais déjà de continuer mes études dans la Recherche. Ce TER m'a permis de confirmer ce souhait. En effet, j'y ai découvert plus précisément le travail de chercheur, en passant quelques heures par semaine au laboratoire. J'aime beaucoup l'aspect "international" de la Recherche : c'est un

effort international où chacun fait partager ses connaissances et ses idées. Ainsi, j'ai pu assister à une présentation du travail d'un chercheur autrichien dans le domaine de l'analyse de programmes, lire des articles sur ce qu'il se fait dans le monde à propos de ce sujet, etc. En plus d'avoir confirmé mon goût pour la recherche, j'ai découvert un sujet que je ne connaissais pas du tout et qui m'a beaucoup plu.

Le sujet que j'ai choisi m'a vraiment beaucoup intéressé, et je regrette ne pas avoir pu y consacrer plus de temps. En effet, j'aurais voulu pouvoir traiter certaines perspectives (5), qui ne sont hélas restées que des idées.

Références

[1] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58 :118–149, 2003.

[2] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. ISBN 3-540-21299-X.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3 : An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[4] Daniel Kroening and Ofer Strichman. *Decision procedures.* 2008. ISBN 978-3-540-74104-6.

[5] Chris Lattner and Vikram Adve. LLVM : A compilation framework for lifelong program analysis & transformation. In *CGO '04 : Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

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

8 Annexe

8.1 Code SMT-Lib de l'exemple 4.1

```
(benchmark VERIMAG
:category { random }
:status unknown
:source { VERIMAG, Synchrone }

:extrapreds ((entry))
:extrapreds ((bb3))
:extrapreds ((bb4))
:extrapreds ((bb5))
:extrapreds ((bb6))
:extrapreds ((bb10))
:extrapreds ((return))

:extrafuns ((x Int))
:extrafuns ((y Int))

:extrapreds ((x_0))
:extrapreds ((x_1))
:extrapreds ((x_2))
:extrapreds ((x_3))
:extrapreds ((x_4))
:extrapreds ((x_5))
:extrapreds ((x_6))
:extrafuns ((x_7 Int))
:extrapreds ((x_8))
:extrapreds ((x_9))
:extrafuns ((x_10 Int))
:extrafuns ((x_11 Int))
:extrafuns ((x_12 Int))
:extrapreds ((x_13))

:extrapreds ((stop_0))
:extrapreds ((warn_0))
:extrapreds ((warn_1))


:formula
(and
(= entry (or  (true)))
(or (not entry)
(and
(= x_0 (> x 9))
(= x_1 (> y 49))
(= x_2 (or x_0 x_1))
(= x_3 (<= x 0))
(= x_4 (or x_2 x_3))
(= x_5 (<= y 0))
(= x_6 (or x_4 x_5))
(ite x_6 bb3 bb4)
))

(= bb3 (or  ((and entry x_6))))
(or (not bb3)
(and
```

```
(= stop_0 true)
(bb4)))

(= bb4 (or  ((and entry (not x_6))) (bb3)))
(or (not bb4)
(and
(= x_7 (+ x y))
(= x_8 (> x_7 59))
(ite x_8 bb5 bb6)
))

(= bb5 (or  ((and bb4 x_8))))
(or (not bb5)
(and
(= warn_0 true)
(bb6)))

(= bb6 (or  ((and bb4 (not x_8))) (bb5)))
(or (not bb6)
(and
(= x_9 (> x_7 30))
(= x_10 (- x_7 30))
(= x_11 (+ x_7 5))
(= x_12 (ite x_9 x_10 x_11))
(= x_13 (> x_12 30))
(ite x_13 bb10 return)
))

(= bb10 (or  ((and bb6 x_13))))
(or (not bb10)
(and
(= warn_1 true)
))

(= return (or  ((and bb6 (not x_13)))))
(or (not return)
false)

(or (and (not stop_0) warn_0)(and (not stop_0) warn_1))))
```