N7PD: Quelques théories dans Z3

Author : Rémi Delmas & Christophe Garion
Public : N7 2SN
Date :

Résumé

L'objectif de ce TP est de prendre en main l'API Java du solver Z3 comme solver SMT à travers la théorie des entiers, la théorie des bitvectors et la théorie des tableaux.

1 La théorie des entiers dans Z3

Z3 permet de manipuler la théorie des entiers avec addition et multiplication. Attention, cette théorie n'est pas décidable, donc il se peut que si vous l'utilisez dans un problème celui-ci ne le soit pas non plus.
Vous trouverez dans l'archive disponible un exemple d'utilisation la théorie des entiers dans la classe `SimpleIntegerProblem` (dans le paquetage `fr.n7.smt`). L'initialisation du solver est toujours la même. Vous trouverez dans la suite quelques remarques et explications :

- on peut créer des variables entières grâce au contexte via `mkIntConst` :

```
IntExpr a = context.mkIntConst("a");
IntExpr b = context.mkIntConst("b");
IntExpr c = context.mkIntConst("c");
IntExpr d = context.mkIntConst("d");
IntExpr e = context.mkIntConst("e");
```

- on peut créer des constantes entières via `mkInt` et des expressions arithmétiques via les méthodes `mkAdd`, `mkMul` etc. Les prédicats habituels sur l'arithmétique des entiers sont disponibles via `mkEq`, `mkGe` etc.

```
// create and add constraint a > b + 2
System.out.print("  creating a > b + 2: ");
BoolExpr c1 = context.mkGt(a, context.mkAdd(b, context.mkInt(2)));
solver.add(c1);
System.out.println(c1);

// create and add constraint a = 2 * c + 10
System.out.print("  creating a = 2 * c + 10: ");
BoolExpr c2 = context.mkEq(a,
                           context.mkAdd(context.mkMul(c, context.mkInt(2)),
                                         context.mkInt(10)));
solver.add(c2);
System.out.println(c2);

// create and add constraint b + c <= 1000
System.out.print("  creating b + c <= 1000: ");
BoolExpr c3 = context.mkLe(context.mkAdd(b, c), context.mkInt(1000));
solver.add(c3);
System.out.println(c3);

// create and add constraint d >= e
System.out.print("  creating d >= e: ");
BoolExpr c4 = context.mkGe(d, e);
solver.add(c4);
System.out.println(c4);
```

- on appelle ensuite le solver avec `check` et si le problème est satisfaisable on peut récupérer un modèle pour trouver l'interprétation des variables entières :

```
if (solver.check() == Status.SATISFIABLE) {
    System.out.println("  Problem is SAT!");
```

```
            Model m = solver.getModel();

            System.out.println("  - value of a: " + m.getConstInterp(a));
            System.out.println("  - value of b: " + m.getConstInterp(b));
            System.out.println("  - value of c: " + m.getConstInterp(c));
            System.out.println("  - value of d: " + m.getConstInterp(d));
            System.out.println("  - value of e: " + m.getConstInterp(e));
        } else {
            System.out.println("  UNSAT or UNKNOWN problem!");
        }
```

2 La théorie des bitvectors dans Z3

En plus des entiers mathématiques, Z3 permet de manipuler des vecteurs de bits (*bitvectors*) qui vont servir en particulier à représenter des entiers machine, i.e. codés sur un nombre fixé de bits.

2.1 Première utilisation

Vous trouverez dans l'archive disponible un exemple d'utilisation la théorie des bitvectors dans la classe `SimpleBVProblem` (dans le paquetage `fr.n7.smt`). L'initialisation du solver est toujours la même. Vous trouverez dans la suite quelques remarques et explications :

– on peut créer des variables entières grâce au contexte via `mkBVConst` (il faut préciser le nombre de bits utilisés) :

```
        BitVecExpr a = context.mkBVConst("a", 4);
        BitVecExpr b = context.mkBVConst("b", 4);
        BitVecExpr c = context.mkBVConst("c", 4);
```

– on peut créer des constantes via `mkBV` (le deuxième argument est le nombre de bits du bitvector) et des expressions arithmétiques sur les bitvectors via les méthodes `mkBVAdd`, `mkBVMul` etc. Les prédicats habituels sur les bitvectors sont disponibles via `mkEq`, `mkBVST` etc.

```
        System.out.print("  creating a > b + 2: ");
        BoolExpr c1 = context.mkBVSGT(a, context.mkBVAdd(b, context.mkBV(2, 4)));
        solver.add(c1);
        System.out.println(c1);

        // create and add constraint a = 2 * c + 10
        System.out.print("  creating a = 2 * c + 10: ");
        BoolExpr c2 = context.mkEq(a,
                           context.mkBVAdd(context.mkBVMul(c, context.mkBV(2, 4)),
                                           context.mkBV(10, 4)));
        solver.add(c2);
        System.out.println(c2);

        // create and add constraint b + c <= 10
        System.out.print("  creating b + c <= 10: ");
        BoolExpr c3 = context.mkBVSLE(context.mkBVAdd(b, c),
                                      context.mkBV(10, 4));
        solver.add(c3);
        System.out.println(c3);
```

– on appelle ensuite le solver avec `check` et si le problème est satisfaisable on peut récupérer comme d'habitude un modèle pour trouver l'interprétation des variables :

```
        if (solver.check() == Status.SATISFIABLE) {
            System.out.println("  Problem is SAT!");

            Model m = solver.getModel();

            System.out.println("  - value of a: " + m.getConstInterp(a));
```

```
            System.out.println("  - value of b: " + m.getConstInterp(b));
            System.out.println("  - value of c: " + m.getConstInterp(c));
        } else {
            System.out.println("  UNSAT or UNKNOWN problem!");
        }
```

2.2 Gérer les dépassements

Le problème précédent était satisfaisable, mais il se peut que des dépassements se produisent en utilisant les bitvectors. La classe `CheckingBVOverflow` présente comment détecter ces dépassements en utilisant Z3.
Dans un premier temps, on considére un problème simple à trois variables bitvectors `a`, `b` et `c` codées sur 4 bits. Si on ajoute les contraintes suivantes au solver

```
a = b + c
b = 8
c = 8
```

on obtient un problème satisfaisable, mais le modèle renvoyé par Z3 instancie `a` avec la valeur `0`. Ceci est parfaitement normal car `8 + 8` induit un dépassement si les entiers sont codés signés sur 4 bits. Dans le cadre de l'analyse de programmes, il faut bien entendu détecter ces dépassements.
La démarche que nous allons suivre est la suivante (tout est implanté dans la classe `CheckingOverflowBV`) :

- on crée tout d'abord 3 variables bitvectors `sa`, `sb` et `sc` sur **5** bits au lieu de 4. L'idée est de « refaire les opérations » effectuées avec les bitvectors codés sur 4 bits avec `sa`, `sb` et `sc` et de vérifier que les résultats sont les mêmes (pour l'addition il ne pourra pas y avoir de dépassements avec les variables sur 5 bits)
- on ajoute des contraintes stipulant que `sb` (resp. `sc`) est une extension de `b` (resp. `c`) à 5 bits. On utilise pour cela la méthode `mkSignExt` qui permet de construire un bitvector de taille `n + i` à partir d'un bitvector de taille `n` (`i` étant le premier paramètre de `mkSignExt`)
- on ajoute ensuite une contrainte stipulant que `sa` doit être différent de l'extension à 5 bits de `a`. Dans ce cas, si le problème est satisfaisable, cela voudra dire que `sa` et `a` sont différents et donc qu'il y a eu un dépassement lors de l'addition
- deux exemples sont donnés dans la classe

Bien évidemment, l'extension d'un bit est suffisante pour l'addition mais ne suffira pas pour la multiplication. Dans ce cas, il faut étendre les bitvectors de taille `n` à des bitvectors de taille `2n`.

3 La théorie des tableaux dans Z3

John McCarthy a proposé en 1962 une théorie élémentaire des tableaux qui a été étendue et implantée dans Z3 [1].
Cette théorie utilise deux fonctions *store* et *select* :

- $store(a, i, v)$ représente le tableau obtenu après stockage de la valeur v à l'index i dans le tableau a
- $select(a, i)$ représente la valeur stockée à l'index i dans le tableau a

Ces deux fonctions sont régies par les deux axiomes suivants :

$$\forall a \forall i \forall v \;\; select(store(a, i, v), i) = v$$
$$\forall a \forall i \forall j \forall v \;\; i = j \vee select(store(a, i, v), j) = select(a, j)$$

Le premier axiome indique que le tableau a' résultant du stockage de la valeur v à l'index i dans le tableau a est tel que que si on demande la valeur à l'index i dans a' on obtient v. Le deuxième indique que le stockage d'une valeur à l'index i ne modifie pas les valeurs aux autres index du tableau.
La classe `SimpleArrayProblem` est un programme utilisant Z3 avec la théorie des tableaux.

3.1 Création de variables représentant des tableaux

Pour pouvoir créer une variable de décision représentant un tableau, on utilise `mkArrayConst` en précisant quelles sont les sortes servant de domaine pour les indices et les valeurs du tableau. Par exemple, pour créer une variable représentant un tableau de valeurs codées sur des bitvectors de taille 8 indexées par des entiers :

```
BitVecSort bvSort = context.mkBitVecSort(8);

ArrayExpr my_array_v = context.mkArrayConst("my_array_v",
                                            context.getIntSort(),
                                            bvSort);
```

3.2 Utiliser `select` pour contraindre les valeurs du tableau

Afin de préciser les valeurs d'un tableau, on peut utiliser `select` afin d'ajouter des contraintes représentant les valeurs voulues. Par exemple, supposons que l'on veuille que le tableau représenté par la variable de décision `my_array_v` ait la valeur `1` à l'index `0` et la valeur `2` à l'index `1`. On pourra alors ajouter les contraintes suivantes :

```
solver.add(context.mkEq(context.mkSelect(my_array_v,
                                         context.mkInt(0)),
                        context.mkBV(1, 8)));
solver.add(context.mkEq(context.mkSelect(my_array_v,
                                         context.mkInt(1)),
                        context.mkBV(2, 8)));
```

3.3 Utiliser `store` pour mettre à jour les valeurs du tableau

Supposons maintenant qu'on veuille représenter le fait que l'on a mis à jour deux valeurs dans le tableau, à l'index 1 et à l'index 5. Dans ce cas, on peut écrire une contrainte utilisant `store`, mais l'appel à `store` renvoie un *nouveau* tableau. On crée donc ici deux nouvelles variables de décision représentant les deux tableaux (un par mise à jour) :

```
ArrayExpr my_array_v_up1 = context.mkArrayConst("my_array_v_up1",
                                                context.getIntSort(),
                                                bvSort);
solver.add(context.mkEq(my_array_v_up1,
                        context.mkStore(my_array_v,
                                        context.mkInt(1),
                                        context.mkBV(5, 8))));
ArrayExpr my_array_v_up2 = context.mkArrayConst("my_array_v_up2",
                                                context.getIntSort(),
                                                bvSort);
solver.add(context.mkEq(my_array_v_up2,
                        context.mkStore(my_array_v_up1,
                                        context.mkInt(2),
                                        context.mkBV(42, 8))));
```

On remarquera que l'on aurait pu se passer de la variable `my_array_v_up1` en chaînant les deux appels à `mkStore`.

3.4 « Afficher » le contenu d'une variable de décision tableau

Lorsqu'un problème impliquant un tableau est satisfaisable, le modèle renvoyé ne permet de connaître directement les valeurs stockées dans le tableau. Une solution simple est d'ajouter des variables du type des éléments du tableau et des contraintes spécifiant que ces variables doivent être égales à certains éléments du tableau. On pourra ainsi récupérer les valeurs de ces variables supplémentaires dans le modèle.

```
BitVecExpr my_array_v_bv[] = new BitVecExpr[4];

for (int i = 0; i < 4; i++) {
    my_array_v_bv[i] = context.mkBVConst("my_array_v[" + i + "]", 8);
    solver.add(context.mkEq(my_array_v_bv[i], context.mkSelect(my_array_v,
                                                               context.mkInt(i))));
}
```

4 Résolution de grilles de Sudoku (encore)

Nous allons reprendre le problème du Sudoku que vous avez déjà résolu en utilisant Z3 comme solveur SAT. Vous pouvez partir de votre solution ou de la solution proposée. Un squelette de classe se trouve dans l'archive donnée. Placez les classes nécessaires dans le paquetage `fr.n7.smt`.

1. de quelle théorie avez-vous besoin pour modéliser le problème ?
2. modifier la solution « SAT » pour utiliser cette théorie et résoudre le problème en changeant les types des attributs si besoin et en complétant les méthodes.

   Pour la méthode `print`, regarder l'API Z3 pour bien comprendre comment retrouver une valeur « Java » pour une variable de décision à partir d'un modèle Z3 si nécessaire.

5 Possible en 3 coups ?

Nous cherchons dans cet exercice à manipuler des tableaux à travers la théorie des tableaux proposés par Z3. On considére un tableau à n valeurs entières indexé par des valeurs entières. L'objectif est de vérifier s'il est possible de rendre le tableau ordonné (par l'ordre naturel $\leq$ sur les entiers) en seulement 3 échanges de valeurs dans le tableau. On considère que l'on peut échanger une valeur avec elle-même pour considérer que 3 échanges se font obligatoirement.
Vous disposez de deux classes :

- `ArraySwaps` qui initialise et résoud le problème. Vous devez la compléter. La méthode `solveAndPrint` peut être appelée pour résoudre le problème et afficher les différentes étapes.
- `MainArrayCLI` qui récupère les entrées utilisateurs sur la ligne de commande crée une instance de `ArraySwaps` pour résoudre le problème.

1. l'attribut `arrays` est un tableau Java de 3 expressions tableau Z3 (de type `ArrayExpr` donc). Chaque tableau Z3 représente un état du tableau (initialement et après chaque échange). Quelle doit donc être la longueur de `arrays` ? Initialisez correctement `arrays`.
2. ajouter des contraintes Z3 pour représenter le fait que le premier tableau « contient » les valeurs contenues dans `values`.
3. le tableau à 3 dimensions `actions` représente les différentes actions possibles. La première dimension du tableau représente l'étape (donc 0, 1 ou 2), la deuxième dimension le premier indice à échanger et la troisième le deuxième indice à échanger. Si Z3 «choisit » par exemple `actions[1][8][6]`, cela veut dire que l'on va échanger les valeurs des cases d'indices 8 et 6 pour le deuxième échange.
   (a) initialisez correctement `actions`
   (b) si on choisit `actions[s][i][j]`, quelle(s) contrainte(s) doit-on définir sur le tableau de l'étape `s+1` par rapport au tableau de l'étape `s` ? Ajoutez les contraintes correspondantes dans le solver.
   (c) ajouter une contrainte exprimant le fait qu'on doit choisir exactement une action à chaque étape.
   (d) ajouter une contrainte exprimant le fait que le tableau obtenu après les 3 actions est ordonné.
4. utilisez Z3 pour résoudre le problème.

Références

[1] Lenoardo de Moura et Nikolaj Bjorner. *Generalized, efficient array decision procedures*. Technical Report MST-TR-2009-121. Microsft Research, 2009. URL : https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/fmcad09.pdf.

License CC BY-NC-SA 3.0