N7PD : Implantation d'un solver BMC utilisant 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 Algorithme de BMC

Cette section va vous présenter quelles sont les classes Java qui sont mise à votre disposition pour réaliser votre algorithme de BMC avec le langage Java. Un exemple minimaliste est également fourni pour vous permettre de tester votre implantation.
Trois classes sont fournies qui seront détaillées dans ce qui suit :

- `Z3Context` est une classe singleton permettant de partager facilement un contexte Z3
- `TransitionSystem` est une classe abstraite permettant de représenter un système de transition
- `BMC` est une classe qui représente l'algorithme de BMC à exécuter

1.1 Rappels sur l'algorithme de Bounded Model Checking

On représente un système de transition sous la forme $\langle S, I, T, P\rangle$, avec :

- S : l'ensemble des états du système.
- $I(.)$: le prédicat sur S caractérisant les états initiaux de la machine.
- $T(.,.)$: le prédicat sur $S \times S$ caractérisant la relation de transition de la machine. $T(s, s')$ s'évalue à vrai ssi s et s' sont des états liés par une transition.
- $P(.)$ est la propriété à prouver (ou plutôt ici, la condition à satisfaire).

On rappelle qu'un problème de BMC pour un nombre de transitions n donné est modélisé par la formule suivante :

$$BMC(\langle S, I, T, P\rangle, n) = I(s_0) \wedge T(s_0, s_1) \wedge ... \wedge T(s_{n-1}, s_n) \wedge P(s_n)$$

Un problème de BMC est usuellement construit incrémentalement par ajout de nouvelles variables et assertions dans le solveur SMT (on parle d'*unrolling* de la relation de transition) en partant de 0 transitions.
A chaque pas de déroulement $n \rightsquigarrow n+1$ de l'algorithme de BMC, la partie $P(S_n)$ de la formule courante est remplacée par la formule $T(s_n, s_{n+1}) \wedge P(s_{n+1})$.
Quand le but est de chercher une trace d'éxécution partant d'un état initial et atteignant un état particulier, le déroulement de la relation de transition continue tant que la formule courante est UNSAT. Le critère d'arrêt du BMC est soit d'obtenir un résultat SAT, soit d'avoir atteint le nombre de transitions correspondant au *seuil de complétude* de l'analyse pour le système donné : si on n'a pas réussi à satisfaire la formule avant d'atteindre ce seuil, il n'est pas la peine d'aller plus loin car tous les états possibles ont été considérés.
Pour un système à un nombre d'états fini, un seuil de complétude acceptable est le *diamètre de réocurrence* du système, c'est à dire la longueur du plus long chemin sans cycle possible dans ce système.
Il peut parfois être calculé à partir des données initiales du problème, mais nécessite dans la plupart des cas de résoudre un problème de BMC particulier pour le déterminer, voire un calcul de point fixe complexe.
On peut également fixer le seuil de complétude de façon arbitraire.

1.2 La classe `Z3Utils`

La classe `Z3Utils` propose des méthodes statiques utiles :

- `getZ3Context` renvoie un contexte Z3 unique. La configuration étant la même pour tous les TP et le projet, cela ne pose pas de problème.
- `atMostOne` prend un tableau ou une liste d'expressions booléennes Z3 et renvoie une expression Z3 représentant la formule « au moins une des expressions passées en paramètres est vraie ».
- `exactlyOne` prend un tableau ou une liste d'expressions booléennes Z3 et renvoie une expression Z3 représentant la formule « exactement une des expressions passées en paramètres est vraie ».

1.3 La classe abstraite `TransitionSystem`

La classe abstraite `TransitionSystem` représente un système de transition. Il faudra donc en hériter et implanter ses méthodes lorsque vous voudrez modéliser un système de transition.

Lorsque vous hériterez de cette classe, il faudra bien sûr définir l'état de votre système en utilisant des types Z3.

Les méthodes abstraites à redéfinir sont les suivantes :
- `BoolExpr transitionFormula(int step)` qui renvoie une expression booléenne Z3 représentant une transition entre l'étape `step` et l'étape `step + 1`. Cette formule sera donc une **contrainte** qui sera ajoutée à Z3 pour « expliquer » le lien entre les états du système à l'étape `step` et l'étape `step + 1`.
- `BoolExpr initialStateFormula()` qui renvoie une expression booléenne Z3 représentant une assertion que doit vérifier l'état initial du système.
- `BoolExpr finalStateFormula(int step)` qui renvoie une expression booléenne Z3 représentant une assertion que doit vérifier l'état du système à l'étape `step` si c'est l'état final.
  Si l'expression renvoyée est `null`, cela signifie qu'il ne faut pas considérer l'étape concernée comme pouvant être finale.
- `void printParams()` permet d'afficher les paramètres du système de transition (par exemple l'état initial, l'objectif à atteindre etc).
- `void printModel(Model m, int steps)` permet d'afficher un modèle renvoyé par Z3 pour `steps` étapes.

La méthode `Expr<?> finalStateApproxCriterion(int step)` peut être redéfinie : elle renvoie une expression Z3 (souvent arithmétique) qui sert de critère d'optimisation lorsque l'on cherche à trouver une solution approchée (voir plus loin). Elle est définie dans la classe abstraite pour renvoyer toujours la même valeur Z3, un entier qui vaut 0.

1.4 Un exemple de système de transition : la classe `DummyTransitionSystem`

La classe `DummyTransitionSystem` qui vous est fournie représente un système de transition très simple qui va servir pour vos tests. L'état du système est représenté par un entier et la seule transition possible entre deux étapes amène à un état où la valeur de l'entier est augmentée de 2.
Regardez bien comment la classe est construite, en particulier l'utilisation des méthodes Z3 pour construire des variables ou des expressions.

1.5 La classe BMC

La classe `BMC` représente l'algorithme de Bounded Model Checking à implanter. Cette classe possède un constructeur qui initialise plusieurs attributs :
- un attribut représentant le système de transition considéré
- le nombre maximum d'étapes de BMC à effectuer
- un attribut booléen si on cherche à résoudre le problème de façon approchée si une résolution exacte n'est pas possible
- un attribut booléen indiquant que l'on veut simuler le système, i.e. on ne se préoccupe pas de l'état final à atteindre

La méthode `solveExact` est à définir (on laissera pour le moment la méthode `solveApprox` en l'état).

1. compléter la méthode `solveExact` pour qu'elle puisse simuler le système.
   Lors d'une simulation, on déroule le système étape par étape (en les affichant à chaque fois) sans ajouter de formules représentant l'état final à l'étape en cours. Le problème devrait être normalement satisfaisable à chaque étape si les transitions sont cohérentes. Comme d'habitude, on gérera quand même les cas UNSAT et UNKOWN. On affichera le modèle obtenu à chaque étape.
   On testera la méthode avec la classe `MainDummySimulation` qui devrait dérouler le système de transition présenté en section 1.4 sur un certain nombre d'étapes.

   ```
   make run-simulation
   ```

   compilera les classes et lancera la simulation.
2. compléter la méthode `solveExact` pour qu'elle déroule l'algorithme de BMC. On utilisera les méthodes `push` et `pop` de Z3 pour ajouter et retirer la formule représentant l'état final à une étape donné. On s'arrêtera si le problème est SAT à une étape donnée.

Si la méthode `finalStateFormula` du système de transition renvoie **`null`** pour une étape donnée, on ne cherchera pas à vérifier si le problème est satisfaisable ou non, mais on ajoutera les contraintes de la prochaine transition et on passera à l'étape suivante.

On testera la méthode avec la classe `MainDummyExact` qui teste le système de transition présenté en section 1.4 sur 3 problèmes (un satisfaisable, les deux autres non).

```
make run-exact
```

compilera les classes et lancera la résolution.

2 Possible en 3 coups ?

Nous cherchons dans cet exercice à manipuler des tableaux à travers la théorie des tableaux proposés par Z3. On considère 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 :

- `ArraySwapsTransitonSystem` qui représente le système de transition. Elle étend la classe `TransitionSystem` et vous devez compléter certaines de ses méthodes.
- `MainArrayCLI` qui récupère les entrées utilisateurs sur la ligne de commande et crée une instance de `BMC` utilisant `ArraySwapsTransitionSystem` pour résoudre le problème. Si aucune entrée n'est donnée, un tableau exemple est utilisé.

  Pour entrer un tableau en utilisant le Makefile fourni, utilisez la variable `ARRAY` :

  ```
  make run-array-swaps ARRAY="1 2 3 4 5 9 8"
  ```

1. l'attribut `arrays` est un tableau Java de 3 expressions tableau Z3 (de type `ArrayExpr<IntSort, IntSort` 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` dans le constructeur de la classe.
2. compléter la méthoe `initialStateFormula` 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` dans le constructeur de la classe.
   (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` ? Créer une méthode privée renvoyant une formule Z3 représentant les contraintes pour la transition à l'étape `s` lorsque l'on échange les indices `i` et `j`.
   (c) créer une méthode privée renvoyant une formule Z3 représentant la contrainte exprimant le fait qu'on doit choisir exactement une action à une étape donnée.
   (d) construire la méthode `transitionFormula` à partir des deux méthodes précédentes.
   (e) écrire la méthode `finalStateFormula`. Pour des étapes différentes de la dernière étape, on renverra **`null`** pour éviter de chercher à résoudre le problème.
4. vérifiez que le problème peut être résolu. Attention, si vous utilisez de grands tableaux qui ne peuvent pas être ordonnés, le temps de (non) résolution peut être très grand !

License CC BY-NC-SA 3.0