N7PD : Miniprojet BMC

Author : Rémi Delmas & Christophe Garion
Public : N7 2SN
Date : 24/05/23

Résumé

L'objectif de ce miniprojet est de résoudre un problème en utilisant Z3 comme solveur SMT.

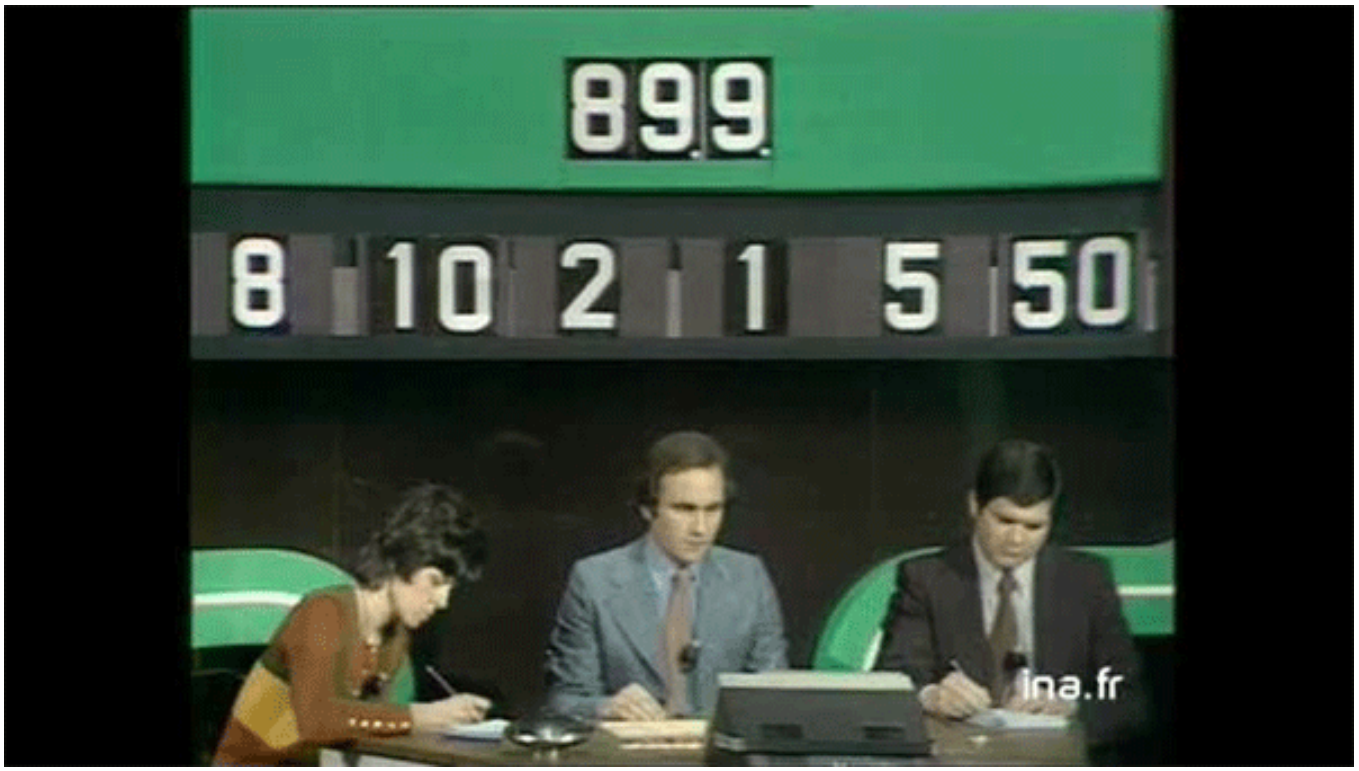

FIGURE 1 – Nous allons remplacer ces personnes venues des années 1980 par un programme des années 2020

1 Présentation

Le but de ce miniprojet est de faire une IA pour la partie « Chiffres » du jeu « Chiffres et des Lettres ».
Étant donnés un ensemble de constantes entières $\{c_i \mid i \in [1, N]\}$ et un résultat final entier lui aussi, le but du jeu est de trouver une séquence d'opérations arithmétiques qui s'approche le plus possible en valeur absolue du résultat et où chaque constante n'est utilisée qu'une fois.
Pour résoudre ce jeu à l'aide d'un solveur SMT, nous allons modéliser le processus de calcul à l'aide d'un automate à pile implémentant la méthode de calcul polonaise inverse.
L'état de l'automate est représenté comme un couple (s, i) où s est un tableau encodant la pile, et i est l'index de la prochaine cellule libre du tableau.
Initialement, la pile est vide. Étant donnés une liste de constantes $\mathcal{C}$ et un resultat $\mathcal{R}$, on cherche une séquence de transitions de l'automate représentant un calcul de $\mathcal{R}$ avec les constantes de $\mathcal{C}$.
Les actions possibles de l'automate sont les suivantes :

- $\{push\_c_i \mid c_i \in \mathcal{C}\}$: pousse la constante c_i sur la pile. Chacune de ces actions ne peut être utilisée qu'une fois sur toute trace d'execution de la machine.
- *mult* : si la pile contient au moins deux éléments, pop les deux éléments du dessus de la pile, et push le résultat de leur multiplication sur la pile.
- *add* : si la pile contient au moins deux éléments, pop les deux éléments du dessus de la pile, et push le résultat de leur addition sur la pile.

- *sub* : si la pile contient au moins deux éléments, pop les deux éléments du dessus de la pile, et push le résultat de leur soustraction sur la pile. La soustraction n'étant pas commutative, l'ordre des arguments est important, l'élément du dessus de la pile est le premier opérande de l'opération.
- *div* : si la pile contient au moins deux éléments, pop les deux éléments du dessus de la pile, et push le résultat de leur division sur la pile. La division n'étant pas commutative, l'ordre des arguments est important, l'élément du dessus de la pile est le premier opérande de l'opération. On doit interdire la division par 0.

Pour ce faire, nous allons construire une représentation logique de l'automate à pile et utiliser la technique de *model checking borné* (*Bounded Model Checking* ou BMC). On représente l'automate sous la forme $\langle S, I, T, P\rangle$, avec :

- S : l'ensemble des états du système. Chaque état est un couple (pile, pointeur de pile).
- $I(.)$: 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 correspondant à une action de la machine (i.e. s' est l'image de s par une action de jeu).
- $P(.)$ est la propriété à prouver (ou plutôt ici, la condition à satisfaire).

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.

2 Code fourni

Vous trouverez deux classes dans l'archive fournie :

- `Main` qui est un programme principal utilisant l'algorithme de BMC sur des exemples. Dans un premier temps, vous pouvez commenter l'appel à `testAll` dans la méthode `main`
- `Chiffres` qui est la classe implantant l'algorithme de BMC pour le problème donné.

Vous pouvez découvrir les attributs de la classe `Chiffres` à travers son constructeur. Il s'agit essentiellement des nombres pouvant être utilisés, du résultat à atteindre, du nombre de bits des bitvectors utilisés et de la possibilité de détecter les overflows.
La classe `Chiffres` implante un système de cache pour les constantes Z3 booléennes, entières, bitvectors et tableaux. Les méthodes correspondantes s'appellent `XXXConst` et prennent un nom sous forme de chaîne de caractères en paramètre.
La méthode `toBvNum` permet de convertir un entier Java en bitvector. La méthode vérifie que l'entier peut bien être encodé dans le bitvector sans dépassement de capacité.
Les méthodes `XXXVar` fournissent des variables Z3 booléennes représentant les actions possibles sur la pile (pousser une valeur, additionner, soustraire, multiplier, diviser). Elles peuvent donc être utilisées dans la modélisation de la relation de transition.
`stackStateVar` et `idxStateVar` sont des variables Z3 représentant l'état de la pile et l'indice de dessus de pile à un pas d'exécution donné.
Enfin, des fonctions de *pretty-printing* sont fournies.

3 Questions

1. combien d'étapes de calcul peut-il y avoir au plus en fonction du nombre de constantes données en entrée ?
2. écrire la méthode `pushNumFormula(step, num)`. Cette méthode renvoie une formule booléenne qui sera vraie si et seulement si en partant de l'état s de l'automate au pas `step` et en effectuant l'action de pousser la valeur

`num` sur la pile on arrive à l'état s' au pas `step + 1`. Il faudra bien évidemment lier s et s' avec des formules Z3. Attention, il faudra modéliser le fait qu'on ne peut utiliser un nombre qu'une seule fois.

3. écrire la méthode `actionFormula` qui renvoie une expression booléenne Z3 représentant le lien existant entre l'état de la pile au pas `step` et au pas `step + 1` si on exécute une action d'addition, de soustraction, de multiplication ou de division.

   L'action à effectuer est encapsulée dans un objet de type `ActionVar`, les préconditions de l'action dans un objet de type `ActionPrecondition` et les postconditions dans un objet de type `ActionResult`. On remarquera que les 4 actions en question utilisent toutes les deux éléments du haut de la pile.

4. écrires les méthodes `addFormula`, `subFormula`, `mulFormula` et `divFormula` qui renvoient une expression booléenne Z3 représentant la formule de transition pour l'opération considérée. On utilisera la méthode `actionFormula` définie précédemment.

   Pour définir les objets de type `ActionVar` on utilisera les méthodes `addVar` etc. déjà définies. Par exemple

   ```
   ActionVar a = this::subVar
   ```

   crée un objet de type `ActionVar` dont la méthode `get` est « identique » à `subVAr`.

   Pour définir les objets de type `ActionPrecondition` et `ActionResult`, on pourra utiliser des lambda-expressions plutôt que de créer des classes anonymes réalisant les interfaces correspondantes (car les interfaces ne définissent qu'une seule méthode).

5. écrire la méthode `transitionFormula` qui renvoie une expression booléenne Z3 signifiant que les états aux pas `step` et `step + 1` sont liés par une transition d'action. Attention, il faudra bien préciser qu'exactement une action parmi celles possibles est exécutée.

6. écrire la méthode `initialStateFormula` qui renvoie une expression booléenne Z3 vraie si et seulement si la pile et le pointeur de pile sont dans leur état initial.

7. écrire la méthode `finalStateFormula` qui renvoie une expression booléenne Z3 vraie si et seulement si la valeur du dessus de pile à l'instant `step` est la valeur attendue. La pile doit avoir une taille de 1.

8. écrire la méthode `solveExact` qui essaye de résoudre le problème de façon exacte. L'idée est d'essayer de résoudre le problème en 1 étape, puis 2 si cela n'est pas possible etc. On utilise pour cela le solveur SMT en mode incrémental à l'aide des fonctions $push()$ et $pop()$ qui permettent de gérer la pile d'assertions du solveur afin de réaliser l'algorithme de BMC décrit en section 1.

   Dans les tests fournis, la méthode `testSimple` utilise une instance qui possède une solution exacte pour 120 et ne possède pas de solution exacte pour 119. Servez-vous en pour tester votre algorithme.

9. on s'intéresse maintenant au cas où la solution exacte n'existe pas. Le but devient donc, *à chaque étape de l'unrolling* jusqu'au seuil de complétude, de trouver la séquence d'actions qui produit le résultat *le plus proche en valeur absolue du résultat demandé*.

   (a) écrire la méthode `finalStateApproxFormula` qui renvoie une formule vraie si et seulement si la pile n'est pas dans l'état final attendu au pas `step`.

   (b) écrire la méthode `finalStateApproxCriterion` qui renvoie le critère d'optimisation, i.e. une expression de type `BitVecExpr` représentant la valeur absolue de la différence entre le résultat approché et le résultat attendu.

   (c) écrire la méthode `solveApprox` qui permet de trouver la meilleure solution approchée. On utilisera un solveur de type `Optimize`. Attention, il n'est pas incrémental (on ne peut pas utiliser `push` et `pop` dessus), donc il faut recréer un solveur à chaque étape du BMC et lui ajouter toutes les formules représentant les transitions jusqu'à l'étape courante.

10. on s'intéresse maintenant à la gestion des dépassements avec les bitvectors. Modifiez votre implantation pour tenir compte des dépassements si le booléen `noOverflows` est vrai.

4 Documents à rendre

Vous devez déposer pour le **4/06/23 à 23h59** sur le dépôt Moodle du cours le source d'un programme Java répondant aux questions ci-dessus. **Le fichier source devra impérativement comporter les logins de votre binôme dans un commentaire en début de fichier.** Les réponses aux questions pourront y être insérées sous forme de commentaires.

License CC BY-NC-SA 3.0