Software Verification

2023-2024

TP3: Bounded Model Checking

Vincent Penelle

Points abordés

— Programmation de l'algorithme du bounded model checking avec parcours en profondeur.
— Programmation de l'algorithme global du bounded model checking.

Le but de ce TP consiste à vous faire programmer deux modules DepthFirstBMC.ml et GlobalBMC.ml de notre outil Simple Program Analyser. Ces module implémenteront chacun un algorithme de Bounded Model Checking explorant l'arbre d'exécution, le premier en profondeur, le second niveau par niveau grâce au dépliage de la formule de pas.

Rappel:

Un *control-flow automaton* est un tuple $(Q, q_i, q_{bad}, \Delta)$ où Q est un ensemble fini d'états, $q_i \in Q$ est l'état initial, q_{bad} est l'état terminal (ou «mauvais»), et $\Delta \subseteq Q \times \text{Op} \times Q$ est l'ensemble des transitions, où Op est l'ensemble des opérations possibles de l'automate. Cet ensemble est défini ci-après. Autrement dit, c'est un automate étiqueté par des opérations (sur des variables).

On considère un ensemble de variables X. Une opération peut être soit `skip`, une garde ou une affectation de la forme $x := exp$, pour $x \in X$ et exp une expression arithmétique :

$$\text{op} ::= \texttt{skip} | \texttt{guard} | x := \texttt{exp}$$

(cf TP précédent pour plus de détails sur les opérations et leur sémantique)

Point technique:

Une *configuration* d'un control-flow automaton $\mathcal{A}$ est un couple (q, σ), où q est un état, et σ est une fonction de X dans $\mathbb{Z}$ appelée *valuation*.

La *sémantique* d'une opération op est la relation $[\![\text{op}]\!]$ qui contient les valuations (σ_1, σ_2) telles que σ_2 peut être obtenue en appliquant op à σ_1. Formellement, on a $[\![\texttt{skip}]\!] = \text{id}$, $[\![\text{guard}]\!]$ est l'ensemble des (σ, σ) tels que σ satisfait la garde, et $[\![\texttt{x} := \texttt{exp}]\!] = \{(\sigma, \sigma[x := exp])\}$. Cf TP précédent pour plus de détails et l'implémentation en formule de SMT.

La sémantique d'une transition $t = (q, \text{op}, q')$ est la relation binaire $\xrightarrow{t}$ sur les configurations définie par $c \xrightarrow{t} c'$ si $c = (q, \sigma)$, $c' = (q', \sigma')$ et $(\sigma, \sigma') \in [\![\text{op}]\!]$.

La *step relation* (ou relation de pas) $\rightarrow_{\mathcal{A}}$ est l'union de toutes les sémantiques

des transitions de $\mathcal{A}$: $\bigcup_{t\in\Delta_\mathcal{A}} \xrightarrow{t}$.

Une *éxecution* est une séquence $c_0, t_1, c_1, \ldots, t_n, c_n$ alternant configurations c_i et transitions t_i telles que $c_{i-1} \xrightarrow{t_i} c_i$ pour tout $0 < i \leq n$. Une telle éxecution est également écrite $c_0 \xrightarrow{t_1} c_1 \cdots \xrightarrow{t_n} c_n$ pour améliorer la lisibilité, et n est sa *longueur*.

Un chemin $q_0, \mathrm{op}_1, q_1, \ldots, \mathrm{op}_n, q_n$ est dit *éxécutable* si il existe des valuations $\rho_0, \ldots, \rho_n \in \mathbb{Z}^X$ telles que $(q_0, \rho_0) \xrightarrow{t_1} (q_1, \rho_1) \cdots \xrightarrow{t_n} (q_n, \rho_n)$ soit une éxecution, avec $t_i = (q_{i-1}, \mathrm{op}_i, q_i)$.

Point technique:

Le problème du bounded model-checking demande, étant donné un control-flow automaton $\mathcal{A}$ et une borne $k \in \mathbb{N}$, s'il existe deux configurations (q, ρ) et (q', ρ') telles que :

- $q = q_i$,
- $q' = q_{bad}$, et
- $(q, \rho) \underbrace{\to_\mathcal{A} \cdot \to_\mathcal{A} \cdots \to_\mathcal{A}}_{i \text{ fois}} (q', \rho')$ pour $i \leq k$.

Le problème peut être formulé de manière équivalente ainsi : Existe-t'il un chemin éxécutable de longueur au plus k de q_i à q_{bad}.

L'algorithme du bounded model-checking peut être simplement résumé ainsi : énumérer toutes les chemins de longueur au plus k, et dès qu'on en trouve un éxécutable de l'état initial à l'état cible, renvoyer ce chemin.

Bien évidemment, il existe plusieurs ordres d'explorations possible de ces chemins, et quelques optimisations simples, mais efficaces (notamment, il ne sert à rien de regarder un chemin si on sait que l'un de ses préfixes n'est pas éxécutable). On peut également demander au programme de déterminer si il a effectué une exploration exhaustive ou non (s'il existait des éxécutions plus longues que celles regardée ou non).

Ici, nous allons implémenter un ordre d'exploration en profondeur des chemins : cela consiste, étant donné un ordre (arbitraire) sur les transitions, regarder d'abord tous les chemins commençant par la première, puis ceux commençant par la seconde, etc. Dès que l'on atteint la borne où un chemin non-exécutable, on remonte au niveau supérieur et on continue l'exploration à partir de là. Dès qu'on atteint un mauvais chemin éxécutable, on stoppe l'exploration et on le renvoie.

Plus précisément, l'algorithme (récursif), reçoit en entrée, un CFA $\mathcal{A}$, un chemin τ, un état courant q, la profondeur courante ℓ, et la borne k, et peut être résumé comme suit :

- si $\ell > k$, renvoyer "non-exhaustive".
- si τ n'est pas éxécutable, renvoyer "exhaustive".
- si τ est éxécutable et que $q = q_{bad}$, on a trouvé une éxécution fautive : renvoyer τ.
- si τ est éxécutable et que $q \neq q_{bad}$, on continue l'exploration à partir du sommet courant de l'arbre : pour toute transition de la forme (q, op, q'), on appelle l'algorithme sur $(\mathcal{A}, \tau :: ((q, \mathrm{op}, q'), q'), q', \ell + 1, k)$. Si tous les appels renvoient "exhaustive", on renvoie "exhaustive". Si tous les appels renvoient "exhaustive" ou "non-exhaustive", on renvoie "non-exhaustive". Dès que l'on reçoit un τ' sur l'un des appels on renvoie ce τ' et on n'explore pas les autres exécutions.

L'algorithme est appelé sur $(\mathcal{A}, (q_i), q_i, 0, k)$ pour effectuer le bounded model-checking avec borne k sur l'automate $\mathcal{A}$.

Exercice 1: Depth-first search bounded model-checking Téléchargez l'archive du TP. Dans le fichier `DepthFirstBMC.ml`, vous avez à coder la fonction `dfs` qui réalise le parcours en profondeur de l'arbre décrit ci-dessus. Le commentaire au dessus du prototype la fonction contient tout ce qu'il faut pour vous aider à programmer cette algorithme, notamment en adaptant les arguments pour faciliter le travail du programme (en retenant une formule décrivant le chemin jusqu'ici plutôt que le chemin lui-même, par exemple). Il y aura quelques subtilités qui vous sont laissées pour renvoyer correctement le chemin fautif.

Vous pouvez également tester votre programme sur les exemples fournis, et comparer avec l'exécutable fonctionnel vous étant fourni `bmc.solution.d.byte`.

Point technique:

L'algorithme global consiste à déterminer, longueur par longueur jusqu'à la borne, s'il existe un chemin de cette longueur. Cela revient à se demander, pour une longueur k, s'il existe des états $q_1, \cdots, q_{k-1}$ de l'automates et des valuations $X_0, \cdots, X_k$, tels que $(q_{in}, X_0 \underset{\mathcal{A}}{\rightarrow} (q_1, X_1) \underset{\mathcal{A}}{\rightarrow} \cdots \underset{\mathcal{A}}{\rightarrow} (q_{k-1}, X_{k-1}) \underset{\mathcal{A}}{\rightarrow} (q_{bad}, X_k)$.

Pour cela, on commence par calculer la formule ϕ_{step} de l'automate, reliant deux configurations si et seulement si l'automate permet de passer de l'une à l'autre. Avec cette formule, pour chaque profondeur i jusqu'à la borne, on vérifie si la formule suivante est satisfaisable ou non :

$$\psi_i(q_0, X_0, \cdots, q_i, X_i) \overset{\text{def}}{=} q_0 = q_{in} \wedge \bigwedge_{j=1}^{i} \varphi_{\text{step}}(q_{j-1}, X_{j-1}, q_j, X_j) \wedge q_i = q_{bad}$$

Si elle l'est, on a l'existence d'un éxécution de bug (et il faut le reconstruire grâce au modèle de la formule (sachant qu'il manque des informations pour récupérer les transitions dans la formule donnée ici)). Si elle ne l'est pas, il n'y a aucun éxécution de bug à la profondeur i. Si on a fait les profondeurs une à une, on peut donc passer à la suivante.

À noter que évidemment, il ne sert à rien de tester ψ_i si aucune éxécution de longueur i n'existe. Pour le déterminer, il suffit de demander au solveur de tester ψ_i sans assurer que le dernier état est q_{bad} (c-à-d sans la partie rouge). Si oui, on doit continuer l'exploration, mais si elle est insatisfiable, il n'y a aucune éxécution de longueur i, et donc on peut dire que le programme est correcte (recherche exhaustive).

Exercice 2: Global bounded model-checking

Dans le fichier `GlobalBMC.ml`, implémentez les fonctions indiquées. Des indications y sont données. N'hésitez pas à découper votre code avec des fonctions auxiliaires. Dans `Z3Helper.mli`, vous avez des fonctions pour vous aider à manipuler les états et transitions de l'automate dans la formule.

Commencez par implémenter une version sans vous préoccuper de renvoyer le contre-exemple trouvé, puis modifiez la formule pour pour renvoyer ce contre-exemple.