Projet RNTL AVERILES
Fourniture F1.3 : Algorithmes de Vérification

LIAFA, CRIL, EDF, LSV, VERIMAG

1 Introduction

Dans le cadre du projet AVERILES, des algorithmes de vérification symboliques ont été développés par les différents partenaires académiques. Ces algorithmes permettent de vérifier des programmes mono-tâche manipulant des structures de donnés à un ou plusieurs sélecteurs ainsi que certains programmes multi-tâches manipulant des structures à un sélecteur. La plupart des ces algorithmes ont de plus été implémenté dans les différents outils développés au sein du projet.

Nous donnons ici une idée du fonctionnement des différents algorithmes, les détails plus techniques étant disponibles dans les publications associées, dont la liste est fournie en bibliographie.

2 Traduction vers des systèmes à compteurs

2.1 Cadre de l'algorithme de vérification

Dans [2] et [5], les auteurs proposent une méthode pour analyser les programmes manipulant des structures de type listes simplement chaînées pouvant être décrites comme indiqué par la figure 1 (selon la syntaxe du C). Le champ `next` pointe vers la cellule suivante dans la liste simplement chaînée et le champ `data` pointe vers la donnée contenue dans la cellule. Dans [2], les programmes considérés ne contiennent pas d'appel au champ `data` (ceux-ci sont donc abstraits du programme original), en revanche dans [5], il est possible de considérer les données incluses dans les cellules afin de pouvoir

```
typedef struct node
{
 struct node *next;
 int data;
} *List
```

FIG. 1 – Structure de liste utilisée

parler de listes ordonnées. Les programmes considérés ne sont pas concurrents et ne contiennent pas d'appels récursifs de fonctions, ils peuvent ainsi être considérés comme des procédures travaillant sur des variables globales.

Les variables peuvent être soit des variables de pointeurs, soit des variables entières. Sur les variables de pointeurs, les instructions suivantes peuvent être réalisées : les mises à jour de variables tels que `u :=null`, `u :=w` ou `u :=w.next`, les mises à jours de sélecteurs `u.next=null` ou `u.next :=w`, les créations `u :=new` et les destructions de cellules `free(u)`. En ce qui concerne les variables entières, les instructions considérées sont l'incrémentation `i :=i+1`, la décrémentation `i :=i-1` et la mise à zéro `i :=0`. Il est également possible de réaliser des tests sur ces variables tels que l'égalité de pointeurs `u=w` ou `u=null` et le test à zéro sur les entiers `i=0`. Les programmes sont alors représentés sous forme d'automates finis étendus dans lesquels on associe à chaque transition une combinsaison booléenne de tests et une instruction.

Les propriétés que l'on cherche à vérifier sur ces programmes sont, dans un premier temps, l'absence de violation mémoire et l'absence de fuite mémoire, mais il s'avère que l'algorithme proposé permet aussi de vérifier des propriétés plus complexes sur la forme du tas mémoire (par exemple qu'un programme ne crée jamais de liste cyclique) et sur le nombre de cellules manipulées (par exemple deux listes ont toujours la même longueur). Dans certains cas, cet algorithme permet également de vérifier la terminaison du programme.

2.2 Représentation symbolique de la mémoire

Le tas mémoire est représenté sous forme d'un graphe (appelé graphe mémoire) où chaque noeud a un unique successeur et peut être étiqueté par une variable de pointeurs. Il existe aussi un noeud spécial appelé `null` (correspondant à l'adresse mémoire `NULL`). Les différentes instructions du programme modifient alors soit la forme du graphe, soit la position des variables sur les noeuds du graphe.

Dans [3], une représentation abstraite de tels graphes a été proposée. Les graphes mémoire abstraits consistent en des graphes dans lesquels chaque noeud est soit pointé par au moins deux noeuds, ou est étiqueté par une variable. On associe de plus à chaque noeud du graphe mémoire abstrait une variable entière. Une propriété intéressante de ces graphes mémoire abstraits est que pour un nombre donné de variables de pointeurs, il n'existe qu'un nombre fini de graphes mémoire abstraits (modulo isomorphisme de graphes et renommage des variables entières). De plus en associant une valeur (strictement positive) à chaque variable entière, il est possible d'obtenir un unique graphe mémoire, la valeur donnant le nombre de successeurs du noeud auquel la variable correspondante est associée.

2.3 Algorithme de vérification

L'algorithme de vérification proposé repose sur une traduction des programmes vers un système à compteurs bisimilaire. Un système à compteurs est un automate fini étendu manipulant uniquement des variables entières et dont les transitions sont étiquetés par un test (combinaison booléenne de tests à zéro sur les variables entières) et par une opération sur les différentes variables entières (incrémentation ou décrémentation). Notons, qu'en réalité, la syntaxe des opérations sur les variables entières peut être étendue à des fonctions linéaires de façon à avoir un automate plus concis. Les variables entières du système à compteurs correspondent ainsi aux variables entières du programme et aux variables entières présentes sur les différents graphes mémoire abstraits.

La vérification des propriétés telles que l'absence de violation mémoire ou l'absence de fuite mémoire sur le programme se ramène alors à un problème d'accessibilité d'un état de contrôle sur le système à compteurs obtenu. Remarquons que bien que ce dernier problème soit dans le cas général indécidable, il existe des outils tels que FAST [1, 4] qui permettent d'analyser les systèmes à compteurs. Bien que FAST soit basé sur un semi-algorithme et qu'il se puisse que son analyse ne termine pas, dans la pratique, ce model-checker a permis d'analyser un grand nombre de cas d'études.

L'algorithme de vérification se décompose donc en deux étapes :

1. Traduire le programme dans un système à compteurs
2. Analyser le système à compteurs obtenu avec un model-checker de systèmes à compteurs comme FAST

Un point important est que la construction du système à compteurs bisimilaire est toujours possible et lors de cette phase il est parfois possible de vérifier l'absence d'erreurs sans analyse ultérieure.

2.4 Résultats obtenus

La traduction vers des systèmes à compteurs a permis d'analyser la plupart des programmes classiques travaillant sur les listes simplement chaînées comme les fonctions `reverse`, `delete`, `merge`, etc. De plus, les outils L2CA et TOPICS développés dans le cadre du projet AVERILES utilisent cet algorithme de traduction.

3 Model-checking régulier abstrait

Dans le cadre du projet nous avons poursuivi des travaux basés sur le model-checking régulier abstrait [11, 8]. Ces travaux ont permis d'obtenir des résultats sur les programmes avec structures de mémoire dynamique en forme

de *listes simplement chaînées*. Dans [10] nous avons étendu ces résultats pour traiter des structures de mémoire plus complexes.

3.1 Modèle pour le programme et propriétés vérifiées

Nous considérons des programmes C non récursifs qui manipulent des structures dynamiques avec possiblement *plusieurs* sélecteurs et avec des données sur un domaine fini permettant par exemple de décrire des listes doublement chaînées ou des arbres avec leurs feuilles liées entre elles. Les propriétés vérifiées sont les propriétés de base (pas de déréférencement de pointeur null, etc.) ainsi que des invariants de forme (par exemple à la fin d'un programme on obtient une liste doublement chaînée, etc.). Les invariants de forme traités sont ceux dont la violation peut être spécifiée dans le fragment existentiel d'une logique du premier ordre sur les graphes que nous avons définie. Cette logique, appelée LBMP (*Logic of Bad Memory Patterns*), permet de décrire de *mauvaises situations* (par exemple qu'à un endroit la liste n'est pas correctement doublement chaînée).

Les propriétés décrites en LBMP peuvent être traduites vers des morceaux de programmes C (appelés testeurs), qui testent ces propriétés et qui vont vers un état d'erreur si une *mauvaise situation* est atteinte. Ainsi, la vérification se réduit à l'accessibilité de l'état d'erreur. Les propriétés peuvent aussi être directment décrites comme des testeurs.

3.2 Représentation symbolique de la mémoire

Notre méthode de vérification est basée sur l'approche du model-checking régulier abstrait ou *Abstract Regular Tree Model Checking* (ARTMC) [9]. Dans ARTMC les configurations d'un système sont des arbres sur un alphabet gradué fini, les ensembles de configurations sont décrits par des automates d'arbre et les transitions du système sont données comme des transducteurs d'arbre. Ensuite, l'ensemble des configurations atteignables à partir d'un ensemble initial est calculé en appliquant d'une façon répétitive le transducteur d'arbre sur les configurations atteignables jusqu'à présent. Pour que ce calcul s'arrête le plus souvent possible, plusieurs abstractions sur les automates d'arbre sont utilisées en ARTMC. Ces abstractions peuvent être automatiquement raffinées si nécessaire.

Pour pouvoir appliquer cette méthode dans notre cadre, nous définissons un codage des configurations de mémoire (qui sont des graphes généraux) comme des arbres. Un graphe mémoire est représenté par un squelette (un arbre) et les arêtes qui ne sont pas présentes dans ce squelette sont données par des *expressions de routage* sur le squelette. Ces expressions sont des expressions régulières sur les directions de l'arbre (par exemple gauche, droite, haut, etc.) et elles indiquent les extrémités possibles des arêtes représentées. Par exemple, une liste doublement chaînée peut être représentée par une liste

simple (qui est un arbre) *et* des arêtes supplémentaires qui décrivent le fait que chaque cellule (mise à part la première) a aussi un pointeur qui pointe vers son prédécesseur.

Pour toutes les opérations de manipulation de pointeurs utilisées en C, nous définissons des transducteurs d'arbre correspondants. Nous pouvons ainsi appliquer ARTMC.

3.3 Algorithme de vérification

Nous utilisons la méthode d'ARTMC décrite ci-dessus pour la vérification qui consiste à montrer qu'un état d'erreur n'est pas atteignable.

3.4 Résultats obtenus

Nous avons appliqué notre méthode à plusieurs études de cas : listes doublement chaînées, arbres (notamment l'algorithme Deutsch-Schorr-Waite), listes de listes et arbre avec feuilles liées entre eux.

4 Analyse d'accessibilité de programmes multithread avec appels de procédures et structures de données dynamiques

L'analyse d'accessibilité avec un changement de contexte borné (ou bounded context switch), est une approche efficace pour la détection d'erreurs dans les programmes multi-thread. En effet, il s'avère que dans beaucoup de cas, les erreurs apparaissent après un nombre assez faible de changements de contexte. Noter qu'il s'agit ici de borner le nombre de changements de contexte d'une thread à une autre, sans borner le nombre d'étapes de calcul de chacune des threads.

Dans [7], nous étudions l'application de cette approche à l'analyse de programmes multi-thread avec :

1. appels de procédures (potentiellement récursives), et
2. manipulation de structures de données dynamiques (création dynamique d'objets et manipulation de pointeurs)

Nous définissons une sémantique des programmes basée sur les automates à pile concurrents, ayant comme symboles de pile ce que nous appelons des *tas mémoire visibles* (ou visible heaps). Un tas mémoire visible est la partie du tas mémoire du programme qui est accessible (à un moment donné) à partir des variables globales et des variables locales (de la procédure qui s'exécute à ce moment là).

Nous utilisons des techniques d'analyse d'automates à pile pour définir un algorithme qui explore tout l'espace des configurations accessibles du programme, ceci en fixant :

1. une borne sur le nombre des changements de contexte, et
2. une borne sur la taille des tas mémoire visibles.

Remaquons que, du fait que nous permettons des procédures récursives, d'une part (1) la taille de la pile des appels est non bornée (car, comme il est mentionné plus haut, le nombre des étapes de calcul des threads n'est pas borné entre les changements de contexte), et d'autre part (2) la taille du programme analysé peut ne pas être bornée, même en fixant une borne sur la taille des tas visibles.

5 Model-checking de logiques temporelles linéaires sur la mémoire

Une approche de la vérification des programmes à pointeurs basée sur la logique de séparation et la logique temporelle a été explorée durant le début de la thèse de Rémi Brochenin (LSV, CNRS-DGA), co-encadré par Stéphane Demri et Etienne Lozes. Nous détaillons ci-dessous cette approche et les résultats obtenus, qui ont été présentés à la conférence LFCS à New York, en juin 2007 [13].

5.1 Cadre (Modèle pour le programme et propriétés vérifiées)

La logique de séparation [15] est une logique dédiée à l'annotation de programme manipulant des pointeurs, dans la tradition de la preuve de programme à la Hoare-Floyd. La spécificité de cette logique est de permettre d'exprimer simplement des propriétés de non aliasing. Par exemple, le triplet $\{x \mapsto 3 \star y \mapsto 3\}\mathsf{free}\ x\{y \mapsto 3\}$ exprime que si x et y pointent sans aliasing vers 3 avant libération de x, alors y pointera toujours sur 3 après libération de x. La logique de séparation est donc une logique qui décrit des états mémoires.

Nous avons cherché à l'étendre pour pouvoir spécifier des propriétés sur les exécutions de programmes vues comme des suites d'états ou traces. Une extension naturelle est de rajouter les connecteurs de la logique temporelle linéaire (LTL), qui permettent d'exprimer qu'une certaine propriété ϕ est vérifiée tout le temps ($G\phi$), au bout d'un certain temps ($F\phi$), voire au bout d'un certain temps sachant qu'en attendant une autre propriété ψ est vérifiée ($\psi U \phi$). Nous avons ainsi proposé un formalisme mélangeant les caractéristiques de la logique de séparation et la logique temporelle, appelé ci-dessous LTL_{mem}.

La première étape de notre travail a consisté à définir correctement la sémantique de LTL_{mem}, le modèle de la mémoire, et la sémantique des programmes. Nous avons proposé un modèle formel très général qui rend compte

à la fois de l'arithmétique des pointeurs (tableaux, matrices,...) et des structures récursives (listes, arbres,...). Au niveau logique, nous avons pu exprimer des propriétés statiques telle que la reconnaissance de listes chaînées, doublement chaînées, ou encore avec pointeur de tête, et des propriétés dynamiques telles que la terminaison du programme, l'invariance d'un pointeur, ou la propriété que deux variables restent constamment non aliasées.

5.2 Représentation symbolique de la mémoire

Nous avons considéré plusieurs problèmes de décision liés à LTL_{mem} : le model-checking, autrement dit savoir si un programme P donné vérifie une spécification ϕ de LTL_{mem}, et la conséquence logique, autrement dit savoir si une spécification en induit automatiquement une autre. Plusieurs variantes de ces problèmes ont été dégagées, selon que l'on considère des exécutions de vrais programmes ou des exécutions arbitraires, selon que l'on traite d'arithmétique ou non, de plusieurs sélecteurs ou d'un seul, de tas dont la structure évolue au cours de l'exécution ou non, etc.

Le problème central qui a permis de résoudre les autres positivement (autrement dit, lorsque la décidabilité du problème de décision considéré a été établie) est celui de la conséquence logique pour des exécutions arbitraires. Pour ce problème, nous avons fait appel à une représentation symbolique de la mémoire comme ensemble maximalement consistant de formules d'états. Cette technique logique relativement classique soulève quelques subtilités non triviales dans le cadre de la logique de séparation. L'intérêt essentiel de cette abstraction est de rester correcte et complète dans la plupart des cas (elle échoue cependant pour certaines propriétés temporelles sur l'arithmétique des pointeurs), et de fournir une représentation finie de toutes les formes mémoires qui doivent être considérées.

5.3 Algorithme de vérification

Notre technique de vérification du problème de conséquence logique est basée sur une traduction de la logique temporelle vers les automates de Büchi (voir [16]) : à toute spécification ϕ on associe un automate A_ϕ capable de reconnaître exactement les mots, c'est-à-dire les séquences d'états mémoires symboliques, qui satisfont la spécification ϕ. Une fois obtenu l'automate, le problème de la conséquence logique se ramène à celui du test du vide du langage reconnu, qui est connu pour être décidable.

En pratique, nous n'avons pas cherché à implémenter notre algorithme, mais nous avons cherché à connaître sa complexité théorique : bien qu'élevée (PSPACE-complète), elle reste la même que celle de la logique de séparation "statique" et la logique temporelle propositionnelle, ce qui d'un certain point de vue valide l'extension que propose LTL_{mem}.

5.4 Résultats obtenus

Le principal résultat de ce travail est une cartographie précise de la décidabilité des problèmes de model-checking et de conséquence logique pour la logique LTL_{mem} selon différents critères sur les exécutions considérées. Résumons quelques-uns des résultats les plus pertinents :

- sans surprise, le model-checking de propriétés temporelles très élémentaires sur les programmes sur les listes a été (re)démontré indécidable
- le model-checking des programmes sans mise à jour du tas a été montré décidable, tant pour l'aspect arithmétique de pointeur que pour l'aspect structures récursives, et tant que certains opérateurs de la logique de séparation ne sont pas utilisés.
- la décidabilité de la conséquence logique pour les exécutions arbitraires a été établie ; bien que peu utilisable en pratique, cela reste un résultat théorique important.
- l'indécidabilité de la conséquence logique restreinte aux exécutions à tas constant (et donc différente de la précédente) a été établie ; ce résultat négatif montre que LTL_{mem} n'est malheureusement pas utilisable telle quelle pour spécifier des formes de récursivité de la mémoire, et suscite la recherche de fragments ou d'autres formalismes pour lesquels on saurait obtenir un résultat de décidabilité.

Références

[1] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast : Fast acceleration of symbolic transition systems. In *CAV'03*, volume 2725 of *LNCS*, pages 118–121. Springer, 2003.

[2] S. Bardin, A. Finkel, É. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. In *AVIS'06*, 2006.

[3] S. Bardin, A. Finkel, and D. Nowak. Toward symbolic verification of programs handling pointers. In *AVIS'04*, 2004.

[4] S. Bardin, J. Leroux, and G. Point. Fast extended release. In *CAV'06*, volume 4144 of *LNCS*, pages 63–66. Springer, 2006.

[5] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV'06*, volume 4144 of *LNCS*, pages 517–531. Springer, 2006.

[6] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In Th. Ball and R. B. Jones, editors, *CAV'06)*, volume 4144 of *LNCS*, pages 517–531, Seattle, Washington, USA, Aug. 2006. Springer.

[7] A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *CAV'07*, volume 4590. LNCS, 2007.

[8] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In N. Halbwachs and L. D. Zuck, editors, *TACAS'05*, volume 3440 of *LNCS*, pages 13–29, Edinburgh, UK, Apr. 2005. Springer.

[9] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. In J. Srba and S. A. Smolka, editors, *INFINITY'05*, volume 149 of *ENTCS*, pages 37–48, San Francisco, CA, USA, Feb. 2006. Elsevier Science Publishers.

[10] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. In K. Yi, editor, *SAS'06)*, volume 4134 of *LNCS*, pages 52–70, Seoul, Korea, Aug. 2006. Springer.

[11] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In R. Alur and D. A. Peled, editors, *CAV'04*, volume 3114 of *LNCS*, pages 372–386, Boston, Massachusetts, USA, July 2004. Springer.

[12] M. Bozga and R. Iosif. On flat programs with lists. In *VMCAI'07*, volume 4349 of *LNCS*, pages 122–136. Springer, 2007.

[13] R. Brochenin, S. Demri, and E. Lozes. Reasoning about sequences of memory states. In *LFCS'07*, volume 3634 of *LNCS*, pages 100–114. Springer, 2007.

[14] P. Habermehl, R. Iosif, and T. Vojnar. Automata-based verification of programs with tree updates. In H. Hermanns and J. Palsberg, editors, *TACAS'06*, volume 3920 of *LNCS*, pages 350–364, Vienna, Austria, Mar. 2006. Springer.

[15] J. Reynolds. Separation logic : a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE, 2002.

[16] M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115 :1–37, 1994.