Une approche combinant les techniques BMC et un problème CSP pour l'aide à la localisation d'erreurs

Mohammed Bekkouche

Une approche combinant les techniques BMC et un problème CSP pour l'aide à la localisation d'erreurs

Mohammed Bekkouche[1,2], sous la direction de Yahia Lebbah[1,2], Hélène Collavizza[2], Michel Rueher[2]

[1] Université d'Oran 1, Lab. LITIO, BP 1524, El-M'Naouer, 31000 Oran, Algérie.
[2] Université de Nice, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France.

Résumé Un vérificateur de modèle peut produire une trace de contre-exemple, pour un programme erroné, qui est souvent inutile (longue et difficile) pour localiser les erreurs dans le code source. Dans ma thèse, nous avons proposé un algorithme de localisation d'erreurs à partir de contre-exemples, nommé LocFaults, combinant les approches de Bounded Model-Checking (BMC) avec un problème de satisfaction de contraintes (CSP). Cet algorithme analyse les chemins du CFG (Control Flow Graph) du programme erroné pour calculer les sous-ensembles d'instructions suspectes permettant de corriger le programme. En effet, nous générons un système de contraintes pour les chemins du graphe de flot de contrôle pour lesquels au plus k instructions conditionnelles peuvent être erronées. Ensuite, nous calculons les MCSs (Minimal Correction Sets) de taille limitée sur chacun de ces chemins. La suppression de l'un de ces ensembles de contraintes donne un sous-ensemble satisfiable maximal, en d'autres termes, un sous-ensemble maximal de contraintes satisfaisant la post-condition. Pour calculer les MCSs, nous étendons l'algorithme générique proposé par Liffiton et Sakallah dans le but de traiter des programmes avec des instructions numériques plus efficacement.

1 Problématique

Les erreurs dans un programme sont inévitables, elles peuvent nuire à son bon fonctionnement et avoir des conséquences financières extrêmement graves et présenter une menace pour le bien-être humain [16]. Le lien suivant [2] cite des histoires récentes de bugs logiciels. Conséquemment, le processus de débogage (la détection, la localisation et la correction d'erreurs) est essentiel. La localisation d'erreurs est l'étape qui coûte le plus. Elle consiste à identifier l'emplacement exact des instructions suspectes [9] afin d'aider l'utilisateur à comprendre pourquoi le programme a échoué, ce qui lui facilite la tâche de la correction des erreurs. En effet, quand un programme P est non conforme vis-à-vis de sa spécification (P contient des erreurs), un vérificateur de modèle peut produire une trace d'un contre-exemple, qui est souvent longue et difficile à comprendre même pour les programmeurs expérimentés.

2 Etat de l'art

Différentes approches ont été proposées pour aider le programmeur dans l'aide à la localisation d'erreur dans la communauté test et vérification.

Bal et al[1] utilisent plusieurs appels à un Model Checker et comparent les contre-exemples obtenus avec une trace d'exécution correcte. Les transitions qui ne figurent pas dans la trace correcte sont signalées comme une possible cause de l'erreur. Ils ont implanté leur algorithme dans le contexte de SLAM, un model checker qui vérifie les propriétés de sécurité temporelles de programmes C.

Dans [10], les auteurs partent de la trace d'un contre-exemple, mais ils utilisent la spécification pour dériver un programme correct pour les mêmes données d'entrée. Chaque instruction identifiée est un candidat potentiel de faute et elle peut être utilisée pour corriger les erreurs. Cette approche garantit que les erreurs sont effectivement parmi les instructions identifiées (en supposant que l'erreur est dans le modèle erroné considéré). En d'autres termes, leur approche identifie un sur-ensemble des instructions erronées. Pour réduire le nombre d'erreurs potentielles, le processus est redémarré pour différents contre-exemples et les auteurs calculent l'intersection des ensembles d'instructions suspectes. Cependant, cette approche souffre de deux problèmes majeurs :

- Elle permet de modifier n'importe quelle expression, l'espace de recherche peut ainsi être très grand ;
- Elle peut renvoyer beaucoup de faux diagnostics totalement absurdes car toute modification d'expression est possible (par exemple, changer la dernière affectation d'une fonction pour renvoyer le résultat attendu).

Pour remédier à ces inconvénients, Zhang et al [17] proposent de modifier uniquement les prédicats de flux de contrôle. L'intuition de cette approche est qu'à travers un switch des résultats d'un prédicat et la modification du flot de contrôle, l'état de programme peut non seulement être modifié à peu de frais, mais qu'en plus, il est souvent possible d'arriver à un état succès. Liu et al [14] généralisent cette approche en permettant la modification de plusieurs prédicats. Ils proposent également une étude théorique d'un algorithme de débogage pour les erreurs *RHS*, c'est à dire les erreurs dans les prédicats de contrôle et la partie droite des affectations.

Récemment, Manu Jose et Rupak Majumdar [11,12] ont abordé ce problème différemment : ils ont introduit un nouvel algorithme qui utilise un solveur MAX-SAT pour calculer le nombre maximum des clauses d'une formule booléenne qui peut être satisfaite par une affectation. Leur algorithme fonctionne en trois étapes :

1. Comme dans l'exécution symbolique, ils encodent une trace d'un programme par une formule booléenne F qui est satisfiable si et seulement si la trace est satisfiable ;
2. ils construisent une formule fausse F' en imposant que la post-condition soit vraie (la formule F' est insatisfiable car la trace correspond à un contre-exemple qui viole la post-condition) ;

3. Ils utilisent `MAXSAT` pour calculer le nombre maximum de clauses pouvant être satisfaites dans F' et affichent le complément de cet ensemble comme une cause potentielle des erreurs. En d'autres termes, ils calculent le complément d'un MSS (Maximal Satisfiable Subset).

3 Résultats et contributions

L'idée de notre approche est de réduire le problème de la localisation d'erreurs vers celui qui consiste à calculer un ensemble minimal qui explique pourquoi un CSP (Constraint Satisfaction Problem) est infaisable. Le CSP représente l'union des contraintes du contre-exemple, du programme et de l'assertion violée. L'ensemble calculé peut être un MCS (Minimal Correction Subset) ou MUS (Minimal Unsatisfiable Subset). En général, tester la faisabilité d'un CSP sur un domaine fini est un problème NP-Complet (intraitable) [3], la classe des problèmes les plus difficiles de la classe NP. Cela veut dire, expliquer l'infaisabilité dans un CSP est aussi dur, voire plus (on peut classer le problème comme NP-Difficile). `BugAssist` [11,12] est une méthode de localisation d'erreurs qui utilise un solveur Max-SAT pour calculer la fusion des MCSs de la formule Booléenne du programme en entier avec le contre-exemple. Elle devient inefficace pour les programmes de grande taille. `LocFaults` [4,5,7,6] travaille aussi à partir d'un contre-exemple pour calculer les MCSs. La contribution de notre approche par rapport à `BugAssist` peut se résumer dans les points suivants :

* Nous ne transformons pas la totalité du programme en un système de contraintes, mais nous utilisons le CFG du programme pour collecter les contraintes du chemin du contre-exemple et des chemins dérivés de ce dernier, en supposant qu'au plus k instructions conditionnelles sont susceptibles de contenir les erreurs. Nous calculons les MCSs uniquement sur le chemin du contre-exemple et les chemins qui corrigent le programme ;
* Nous ne traduisons pas les instructions du programme en une formule SAT, mais plutôt en contraintes numériques qui vont être manipulées par des solveurs de contraintes ;
* Nous n'utilisons pas des solveurs MaxSAT comme boîtes noires, mais plutôt un algorithme générique pour calculer les MCSs par l'usage d'un solveur de contraintes.

Pour évaluer la méthode que nous avons proposée, nous avons comparé les performances de `LocFaults` et de `BugAssist` [11,12] sur un ensemble de programmes. Comme `LocFaults` est basé sur `CPBPV` [8] qui travaille sur des programmes Java et que`BugAssist` travaille sur des programmes C, nous avons construit pour chacun des programmes :

– une version en Java annotée par une spécification JML ;
– une version en ANSI-C annotée par la même spécification mais en ACSL.

Les deux versions ont les mêmes numéros de ligne et les mêmes instructions. La précondition est un contre-exemple du programme, et la postcondition cor-

3. Si ce problème pouvait être résolu en temps polynomial, alors tous les problèmes NP-Complet le seraient aussi.

respond au résultat de la version correcte du programme pour les données du contre-exemple. Comme dit précédemment, nous avons considéré qu'au plus une condition pouvait être fausse sur un chemin. Par ailleurs, nous n'avons pas cherché de MCS de cardinalité supérieure à 3. Les expérimentations ont été effectuées avec un processeur Intel Core i7-3720QM 2.60 GHz avec 8 GO de RAM.

Pour calculer les MCSs, nous avons utilisé les solveurs IBM ILOG MIP et CP [4] de CPLEX. Nous avons adapté et implémenté l'algorithme de Liffiton et Sakallah [13], voir alg. 1. Cette implémentation prend en entrée l'ensemble de contraintes infaisable qui correspond au chemin identifié (C), et b_{mcs} : la borne sur la taille des MCSs calculés. Chaque contrainte c_i dans le système construit C est augmentée par un indicateur y_i pour donner $y_i \rightarrow c_i$ dans le nouveau système de contraintes C'. Affecter à y_i la valeur $Vrai$ implique la contrainte c_i ; en revanche, affecter à y_i la valeur $Faux$ implique la suppression de la contrainte c_i. Un MCS est obtenu en cherchant une affectation qui satisfait le système de contraintes avec un ensemble minimal d'indicateurs de contraintes affectés avec $Faux$. Pour limiter le nombre de variables indicateurs de contraintes qui peuvent être assignées à Faux, on utilise la contrainte $AtMost(\neg y_1, \neg y_2, ..., \neg y_n, k)$ (voir la ligne 5), le système créé est noté dans l'algorithme C'_k (ligne 5). Chaque itération de la boucle While (lignes $6-19$) permet de trouver tous les MCSs de taille k, k est incrémenté de 1 après chaque itération. Après chaque MCS trouvé (lignes $8-13$), une contrainte de blocage est ajoutée à C'_k et C' pour empêcher de trouver ce nouveau MCS dans les prochaines itérations (lignes $15-16$). La première boucle (lignes $4-19$) s'itère jusqu'à ce que tous les MCSs de C soient générés (C' devient infaisable) ; elle peut s'arrêter aussi si les MCSs de taille inférieure ou égale b_{mcs} sont obtenus ($k > b_{mcs}$).

Algorithm 1: Algorithme de Liffiton et Sakallah

```
1  Fonction MCS(C,b_mcs)
   Entrées: C : Ensemble de contraintes infaisable, b_mcs : Entier
   Sorties: MCS : Liste de MCSs de C de cardinalité inférieure à b_mcs
2  début
3      C' ← AddYVars(C) ; MCS ← ∅ ; k ← 1 ;
4      tant que SAT(C') ∧ k ≤ b_mcs faire
5          C'_k ← C' ∧ AtMost({¬y_1, ¬y_2, ..., ¬y_n},k)
6          tant que SAT(C'_k) faire
7              newMCS ← ∅
8              pour chaque indicateur y_i faire
9                  % y_i est l'indicateur de la contrainte c_i ∈ C, et val(y_i) la valeur de y_i dans la
                   solution calculée de C'_k.
10                 si val(y_i) = 0 alors
11                     newMCS ← newMCS ∪ {c_i}.
12                 fin
13             fin
14             MCS.add(newMCS).
15             C'_k ← C'_k ∧ BlockingClause(newMCS)
16             C' ← C' ∧ BlockingClause(newMCS)
17         fin
18         k ← k + 1
19     fin
20     retourner MCS
21 fin
```

4. Disponible à http ://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/

Nous expliquerons, dans ce rapport, nos résultats de l'évaluation de notre approche sur les programmes TCAS (Traffic Collision Avoidance System) de la suite de test Siemens [15]. Il s'agit d'un benchmark bien connu qui correspond à un système d'alerte de trafic et d'évitement de collisions aériennes. Il y a 41 versions erronées et 1608 cas de tests. Nous avons utilisé toutes les versions erronées sauf celles dont l'indice $AltLayerValue$ déborde du tableau $PositiveRAAltThresh$ car les débordements de tableau ne sont pas traités dans `CPBPV`. Plus précisément, voici les versions **TcasKO...TcasKO41**. Les erreurs dans ces programmes sont provoquées dans des endroits différents. 1608 cas de tests sont proposés, chacun correspondant à contre-exemple. Pour chacun de ces cas de test T_j, on construit un programme $TcasV_iT_j$ qui prend comme entrée le contre-exemple, et dont postcondition correspond à la sortie correcte attendue. Le code source de l'ensemble des programmes est disponible à l'adresse `http://www.i3s.unice.fr/~bekkouch/Bench_Mohammed.html`.

La table sur le lien suivant `http://www.i3s.unice.fr/~bekkouch/Bench_Mohammed.html#rtcas` donne les résultats pour les programmes de la suite `TCAS`. La colonne Nb_E indique pour chaque programme le nombre d'erreurs qui ont été introduites dans le programme alors que la colonne Nb_CE donne le nombre de contre-exemples. Les colonnes LF et BA indiquent le nombre de fois où `LocFaults` et `BugAssist` ont identifié l'instruction erronée. On remarquera que `LocFaults` se compare favorablement à `BugAssist` sur ce benchmark qui ne contient quasiment aucune instruction arithmétique ; comme précédemment le nombre d'instructions suspectes identifiées par `LocFaults` est dans l'ensemble nettement inférieur à celui de `BugAssist`.

Les temps de calcul de `BugAssist` et `LocFaults` sont très similaires et inférieurs à une seconde pour chacun des benchmarks de la suite `TCAS` très adaptés aux solveurs booléens.

4 Conclusion et perspectives

La méthode `LocFaults` détecte les sous-ensembles suspects en analysant les chemins du CFG pour trouver les déviations minimales qui corrigent le programme ainsi que les MCSs ; elle utilise des solveurs de contraintes. La méthode `BugAssit` calcule la fusion des MCSs du programme en transformant le programme complet en une formule booléenne ; elle utilise des solveurs Max-SAT. Les deux méthodes travaillent en partant d'un contre-exemple.

Dans le cadre de nos travaux futurs, nous envisageons de confirmer nos résultats sur des programmes avec boucles plus complexes (voir nos premiers résultats dans [3]). Nous développons une version interactive de notre outil qui fournit les sous-ensembles suspects l'un après l'autre : nous voulons tirer profit des connaissances de l'utilisateur pour sélectionner les conditions qui doivent être déviées. Nous réfléchissons également sur comment étendre notre méthode pour supporter les instructions numériques avec calcul sur les flottants. Il sera certainement aussi intéressant de calculer des MUS car ils apportent une information complémentaire à l'utilisateur.

Références

1. Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause : localizing errors in counterexample traces. In *Proceedings of POPL*, pages 97–105. ACM, 2003.
2. Mohammed Bekkouche. Bug stories. In `http://www.i3s.unice.fr/~bekkouch/Bug_stories.html`, 2015.
3. Mohammed Bekkouche. Exploration of the scalability of locfaults approach for error localization with while-loops programs. *arXiv preprint arXiv :1503.05508*, 2015.
4. Mohammed Bekkouche, Hélène Collavizza, and Michel Rueher. Une approche csp pour l'aide à la localisation d'erreurs. In *Dixièmes Journées Francophones de Programmation par Contraintes (JFPC 14)*, pages http–jfpc.
5. Mohammed Bekkouche, Hélene Collavizza, and Michel Rueher. Locfaults : A new flowdriven and constraint-based error localization approach*. In *ACM. SAC*, volume 15, 2015.
6. Mohammed Bekkouche, Hélène Collavizza, and Michel Rueher. Un algorithme incrémental dirigé par les flots et basé sur les contraintes pour l'aide à la localisation d'erreurs. *arXiv preprint arXiv :1505.06324*, 2015.
7. Mohammed Bekkouche, Michel Rueher, and Hélene Collavizza. Constraint-based error localization. In *2015 INFORMS Computing Society Conference (ICS)*, 2015.
8. Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Cpbpv : a constraint-programming framework for bounded program verification. *Constraints*, 15(2) :238–264, 2010.
9. Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7) :1165–1178, 2008.
10. Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for c programs. *Electr. Notes Theor. Comput. Sci.*, 174(4) :95–111, 2007.
11. Manu Jose and Rupak Majumdar. Bug-assist : Assisting fault localization in ansi-c programs. In *Proceedings of CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 504–509. Springer, 2011.
12. Manu Jose and Rupak Majumdar. Cause clue clauses : error localization using maximum satisfiability. In *Proceedings of PLDI*, pages 437–446. ACM, 2011.
13. Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1) :1–33, 2008.
14. Yongmei Liu and Bing Li. Automated program debugging via multiple predicate switching. In *Proceedings of AAAI*. AAAI Press, 2010.
15. David S. Rosenblum and Elaine J. Weyuker. Lessons learned from a regression testing case study. *Empirical Software Engineering*, 2(2) :188–191, 1997.
16. Wikipedia. List of software bugs — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=List_of_software_bugs&oldid=648559652`, 2015. [Online ; accessed 3-March-2015].
17. Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *Proceedings of ICSE*, pages 272–281. ACM, 2006.