Le slicing améliore une méthode de vérification combinant l'analyse statique et l'analyse dynamique
(Extended Abstract)

Omar Chebaro[1], Nikolai Kosmatov[2], Alain Giorgetti[3,4], and Jacques Julliand[3]

[1] ASCOLA (EMN-INRIA, LINA), École des Mines de Nantes, 44307 Nantes, France
[2] CEA, LIST, Laboratoire Sûreté des Logiciels, PC 174, 91191 Gif-sur-Yvette, France
[3] University of Franche-Comté, FEMTO-ST, UMR 6174, Besançon, F-25030, France
[4] Inria, Villers-lès-Nancy, F-54600, France, CASSIS project
firstname.lastname@{mines-nantes[1], cea[2], femto-st[3]}.fr

La validation des logiciels est une partie cruciale de leur cycle de développement. Deux techniques de vérification et de validation se sont démarquées au cours de ces dernières années : l'analyse statique et l'analyse dynamique. Les points forts et faibles des deux techniques sont complémentaires.

Dans [1], nous avons présenté une combinaison originale de ces deux techniques. Dans cette combinaison, l'analyse statique signale les instructions risquant de provoquer des erreurs à l'exécution, par des alarmes dont certaines peuvent être de fausses alarmes, puis l'analyse dynamique (génération de tests) est utilisée pour confirmer ou rejeter ces alarmes. Appliquée à des programmes de grande taille, la génération de tests peut manquer de temps ou d'espace mémoire avant de confirmer certaines alarmes comme de vraies erreurs ou conclure qu'aucun chemin d'exécution ne peut atteindre l'état d'erreur de certaines alarmes et donc rejeter ces alarmes. Les expérimentations ont montré que la génération de tests sur le programme complet peut perdre beaucoup de temps en essayant de couvrir des chemins d'exécution ou des parties de code qui ne sont pas pertinentes pour les alarmes. Nous proposons de réduire la taille du code source par le *slicing* avant de lancer la génération de tests (cf Fig. 1). Le *slicing* transforme un programme en un autre programme plus simple, appelé *slice*, qui est équivalent au programme initial selon certains critères.

Une autre motivation importante de ce travail est de fournir automatiquement à l'ingénieur de validation autant d'informations que possible sur chaque erreur détectée.

Quatre utilisations du *slicing* sont étudiées. La première utilisation est nommée *all*. Elle consiste à appliquer le *slicing* une seule fois, le critère de simplification étant l'ensemble de toutes les alarmes du programme qui ont été détectées par l'analyse statique. L'inconvénient de cette utilisation est que la génération de tests peut manquer de temps ou d'espace et les alarmes les plus faciles à classer sont pénalisées par l'analyse d'autres alarmes plus complexes. Dans la deuxième utilisation, nommée *each*, le *slicing* est effectué séparément par rapport à chaque alarme. Cependant, la génération de tests est exécutée pour chaque programme et il y a un risque de redondance d'analyse si des alarmes sont incluses dans d'autres slices.

Pour pallier ces inconvénients, nous avons étudié les dépendances entre les alarmes et nous avons introduit deux utilisations avancées du *slicing*, nommées *min* et *smart*, qui exploitent ces dépendances. Dans l'utilisation *min*, le *slicing* est effectué par rapport à un ensemble minimal de sous-ensembles d'alarmes. Ces sous-ensembles sont

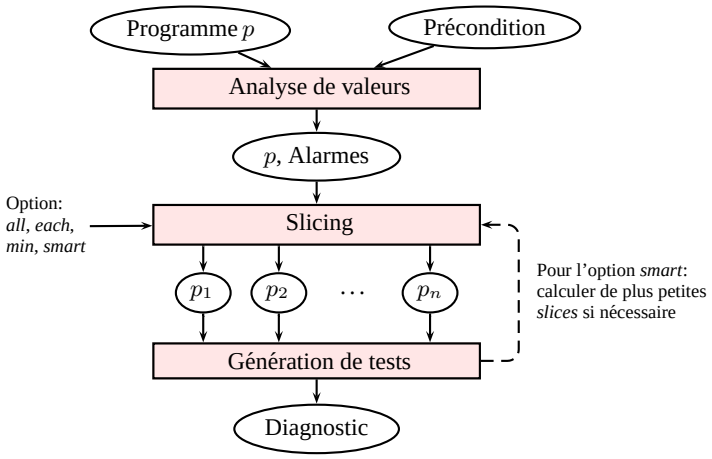

Fig. 1. Principe de la méthode de vérification

choisis en fonction de dépendances entre les alarmes et l'union de ces sous-ensembles couvre l'ensemble de toutes les alarmes. Avec cette utilisation, on a moins de *slices* qu'avec *each*, et des *slices* plus simples qu'avec *all*. Cependant, l'analyse dynamique de certaines *slices* peut manquer de temps ou d'espace mémoire pour classer certaines alarmes, tandis que l'analyse dynamique d'une *slice* éventuellement plus simple permettrait de les classer. L'utilisation *smart* consiste à appliquer l'utilisation précédente itérativement en réduisant la taille des sous-ensembles quand c'est nécessaire. Lorsqu'une alarme ne peut pas être classée par l'analyse dynamique d'une slice, des slices plus simples sont calculées.

Ces travaux sont implantés dans SANTE, notre outil qui relie l'outil de génération de tests PATHCRAWLER [2] et la plate-forme d'analyse statique FRAMA-C [3]. Des expérimentations ont montré que la génération de tests sur les programmes simplifiés est plus efficace (accélération moyenne autour de 43%, allant jusqu'à 99% pour certains exemples), et laisse moins d'alarmes non classées dans un temps donné. En outre, une erreur est signalée avec des informations plus précises et illustrée sur un programme plus simple. Ceci facilite l'analyse des erreurs détectées et des alarmes restantes. Les utilisations du slicing et les expérimentations sont présentées en détail dans [4].

References

1. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Combining static analysis and test generation for C program debugging. In: TAP. Volume 6143 of LNCS. (2010) 652–666
2. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST, Vancouver, Canada, IEEE Computer Society (May 2009) 70–78
3. Cuoq, P., Signoles, J.: Experience report: Ocaml for an industrial-strength static analysis framework. In: ICFP. (2009) 281–286
4. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC, ACM (2012) 1284–1291