Analyse de Code Automatique: Revisiter l'Inférence de Préconditions via l'Acquisition de Contraintes

Grégoire Menguy, Sébastien Bardin, Nadjib Lazaar, Arnaud Gotlieb

Analyse de Code Automatique: Revisiter l'Inférence de Préconditions via l'Acquisition de Contraintes *

G. Menguy[1], S. Bardin[1], N. Lazaar[2], A. Gotlieb[3]

[1] Université Paris-Saclay, CEA, List, Palaiseau, France
[2] LIRMM, University of Montpellier, CNRS, Montpellier, France
[3] Simula Research Laboratory, Oslo, Norway

Résumé

Les annotations de programme, sous forme de pré/post-conditions de fonctions, sont cruciales pour accomplir différentes tâches, de l'ingénierie logicielle à la vérification de code. Malheureusement, ces annotations sont rarement fournies et doivent donc être rétro-ingéniées manuellement. Dans notre article, nous étudions comment l'acquisition de contraintes peut être utilisée pour inférer des préconditions. Cela a conduit à PRECA*, un outil qui infère des préconditions à partir d'observations d'exécution du code uniquement, et assurant des garanties claires de correction.*

Mots-clés

Acquisition de contraintes, analyse de code, préconditions

Abstract

Program annotations under the form of function pre/post-conditions are crucial for many software engineering and program verification applications. Unfortunately, these are rarely available and must be retrofit by hand. This paper explores how Constraint Acquisition (CA) can be leveraged to automatically infer program preconditions. This leads to PRECA*, which infers preconditions from input-output observations only, and presents clear correctness guarantees.*

Keywords

Constraint acquisition, code analysis, preconditions

1 Introduction

Les annotations de code sous la forme de pré- et post-conditions [8, 5, 3] sont cruciales en ingénierie logicielle et en vérification formelle de code. Elles permettent d'améliorer la compréhension du code pour les utilisateurs et pour les outils d'analyse automatique de code [9, 7]. Malheureusement, ces annotations sont rarement fournies par les développeurs et doivent donc être rétro-ingéniées à la main, ce qui limite leur intérêt, en particulier pour les composants tiers dont le code source n'est pas disponible.

Problème. De nombreuses méthodes ont été proposées pour inférer automatiquement des préconditions à partir du code. Cependant, l'état de l'art n'est toujours pas satisfaisant. En effet, même si les approches en *boîte blanche* basées sur l'analyse statique du code [8, 5, 3, 2] peuvent être utiles, elles présentent de nombreuses limites en termes de précision et ont du mal à être scalables. De plus, la gestion de structures de code complexes comme les boucles, la récursion et la mémoire dynamique représente un véritable défi. Les approches basées sur les exemples d'exécution du code pour inférer les annotations, appelées "boîte noire", ont été proposées pour dépasser les limites des approches basées sur l'analyse statique du code, appelées "boîte blanche" [4, 11, 6]. Cependant, la qualité des annotations inférées dépend fortement de la qualité des cas de test utilisés. Les cas de test peuvent être générés aléatoirement, fournis par l'utilisateur ou générés pendant l'inférence. Malheureusement, l'état de l'art ne fournit pas de garanties claires de correction.

Acquisition de contraintes. La programmation par contraintes (CP) [12] a connu de considérables avancées ces quarante dernières années. Cependant, modéliser un problème comme un réseau de contraintes reste une tâche difficile. Des méthodes d'acquisition de contraintes (CA) ont donc été proposées pour aider les utilisateurs non-experts. Par exemple, CONACQ infère un réseau de contraintes représentant le concept utilisateur à partir de solutions et de non-solutions classifiées par l'utilisateur. Le domaine de la recherche en CA est actif et a proposé de nombreuses extensions, telles que l'utilisation de requêtes partielles [1]. Bien que CONACQ offre des garanties théoriques fortes, ce type de système est difficile à utiliser en pratique car il nécessite de soumettre un grand nombre d'exemples (appelés requêtes) à l'utilisateur. Toutefois, en analyse de code, le nombre de requêtes n'est pas limitant car elles peuvent être classifiées automatiquement.

Objectifs et contributions. Dans ce papier, nous proposons une nouvelle approche pour l'inférence de préconditions en boîte noire basée sur l'acquisition de contraintes active. À notre connaissance, il s'agit de la première application de cette méthode en analyse de code. Notre approche, PRECA, bénéficie de meilleures propriétés théoriques que l'état de l'art. En effet, si notre langage est suffisamment expressif pour représenter la weakest-precondition (WP) [8] de la fonction (i.e., la plus générale et donc la meilleure), alors PRECA est sûr de l'inférer.

Nous décrivons également une spécialisation de PRECA

*Cet article se base sur des résultats publiés à IJCAI-ECAI 2022 [10].

pour l'inférence de préconditions sur la mémoire. Pour cela, nous avons développé un langage de contraintes dédiées gérant la validité, l'aliasing et le déréférencement des pointeurs. Par exemple, PRECA sur la fonction void find_first (int∗ a, int m, int∗ b, int n) infère la WP $(m > 0 \Rightarrow valid(a)) \wedge (m > 0 \wedge n > 0 \Rightarrow valid(b))$. De plus, nous avons proposé une stratégie pour accélérer l'inférence. Cette stratégie repose sur l'observation que les requêtes positives suppriment une plus grande partie de l'espace de recherche que les requêtes négatives. Nous générons donc en priorité des requêtes avec peu de pointeurs invalides et qui aliasent, ayant peu de chance de mener à un bug et qui seront donc probablement classées positives. Combinée avec un "background knowledge", cela permet d'accélérer significativement PRECA (cf. Table 1). Enfin, nous avons évalué expérimentalement notre approche sur un benchmark de 50 fonctions provenant de librairies standards (comme string.h) ou d'exemples présentés dans l'état de l'art. Nous avons comparé PRECA à trois autres méthodes d'inférence de préconditions en boîte noire, à savoir Daikon [4], PIE [11] et l'approche de Gehr et al. [6], que nous avons réimplémentée. Nous avons également comparé notre approche à l'approche d'inférence de préconditions en boîte blanche P-Gen [13]. Notre étude montre que PRECA est capable d'inférer plus de préconditions que ses concurrents (cf. Table 1). En particulier, nous avons observé que PRECA est capable d'inférer plus de weakest-preconditions en 5s que les concurrents en 1h. Cela reste vrai même face au concurrent en boîte blanche P-Gen qui a accès au code et est donc avantagé.

	1s		5s		5 mins		1h	
	$\#WP_\top$	$\#WP_Q$	$\#WP_\top$	$\#WP_Q$	$\#WP_\top$	$\#WP_Q$	$\#WP_\top$	$\#WP_Q$
Daikon	1.4/50	0.4/44	1.6/50	0.4/44	1.6/50	0.4/44	1.6/50	0.4/44
↳ PRECA	2/50	1/44	2/50	1/44	2/50	1/44	2/50	1/44
↳ Both	3.3/50	0/44	5.7/50	0/44	5.7/50	0/44	5.7/50	0/44
PIE	16.4/50	4.7/44	16.4/50	4.7/44	17.7/50	4.7/44	17.7/50	5.3/44
↳ PRECA	5/50	3/44	5/50	3/44	5/50	3/44	5/50	3/44
↳ Both	25.3/50	11.3/44	25.4/50	11.3/44	26.4/50	11.3/44	28.4/50	11.3/44
Gehr et al.	8.0/50	5.0/44	16.8/50	8.1/44	26.1/50	10.1/44	26.1/50	10.3/44
↳ PRECA	37/50	15/44	43/50	17/44	46/50	18/44	46/50	18/44
PRECA	29/50	11/44	**38/50**	**16/44**	**46/50**	**18/44**	**46/50**	**18/44**
↳ BK	15/50	8/44	38/50	16/44	45/50	18/44	46/50	18/44
↳ Preproc.	19/50	9/44	36/50	16/44	45/50	18/44	46/50	18/44
↳ ∅	13/50	7/44	35/50	15/44	45/50	18/44	46/50	18/44
↳ Random	29.9/50	12.1/44	29.9/50	12.1/44	30.0/50	12.1/44	30.0/50	12.1/44
P-Gen	**34/50**	**13/44**	37/50	15/44	37/50	15/44	37/50	15/44

Le nombre de Weakest Precondition inférées sans (resp. avec) une postcondition est représenté par $\#WP_\top$ (resp. $\#WP_Q$). Nous étudions trois variations de Daikon et PIE : (i) l'original (surligné) sur 100 exemples aléatoires ; (ii) sur les exemples générés par PRECA ; (iii) sur des exemples aléatoires et de PRECA. Nous examinons également la méthode actif originale de Gehr et al. (surlignée) et nous lui donnons les exemples de PRECA. Enfin, nous étudions PRECA avec son "background knowledge" et son prétraitement (surligné), avec seulement son "background knowledge" (BK), avec seulement le prétraitement (Preproc.), sans aucun des deux (∅) et en mode passif avec 100 requêtes aléatoires (Random). Étant donné que P-Gen est une méthode statique, nous ne considérons que sa forme originale.

TABLE 1 – Results depending on the time budget

2 Conclusion

Nous présentons la première application de l'acquisition de contraintes pour l'inférence de préconditions, qui est un problème majeur en analyse de code et en méthodes formelles. Cette approche représente la première méthode d'inférence de préconditions totalement boîte noire avec des garanties de correction solides. De plus, nos expériences sur l'inférence de préconditions orientées vers la mémoire ont montré que PRECA améliore significativement l'état de l'art, ce qui démontre l'intérêt de l'acquisition de contraintes dans ce domaine.

3 Remerciements

Ce travail a bénéficié du soutien de l'Institut de Cyber sécurité d'Occitanie (ICO), financé par la Région Occitanie en France, et du programme de recherche et d'innovation Horizon 2020 de l'Union européenne (projet TAILOR).

Références

[1] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In *IJCAI*, 2013.

[2] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *VMCAI'13*. Springer, 2013.

[3] Edsger W Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 1968.

[4] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *TSE*, 2001.

[5] Robert W Floyd. Assigning meanings to programs. In *Program Verification*. Springer, 1993.

[6] Timon Gehr, Dimitar Dimitrov, and Martin Vechev. Learning commutativity specifications. In *CAV'15*, 2015.

[7] Patrice Godefroid, Shuvendu K Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*. Springer, 2011.

[8] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *CACM*, 1969.

[9] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c : A software analysis perspective. *Formal Aspects of Computing*, 2015.

[10] Grégoire Menguy, Sébastien Bardin, Nadjib Lazaar, and Arnaud Gotlieb. Automated program analysis : Revisiting precondition inference through constraint acquisition. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 1873–1879. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Main Track.

[11] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices*, 2016.

[12] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[13] Mohamed Nassim Seghir and Daniel Kroening. Counterexample-guided precondition inference. In *ESOP*. Springer, 2013.