CaFE : un model-checker collaboratif

Steven de Oliveira[1], Virgile Prevosto[1], Saddek Bensalem[2]

1 : CEA, LIST, Software Reliability and Security Laboratory,
P.C. 174, 91191 Gif-sur-Yvette, France ;
2 : Verimag, Université Grenoble Alpes

Résumé

La logique temporelle linéaire (LTL) est utilisée dans de nombreux travaux pour décrire formellement le comportement attendu d'un programme. Une extension particulièrement intéressante de LTL pour l'analyse de programmes est la logique *CaRet*, qui autorise des raisonnements explicites sur la pile d'appel. Ce papier présente *CaFE*, un greffon de la plate-forme *Frama-C* qui vérifie automatiquement de manière approchée qu'un programme C vérifie une formule logique *CaRet* donnée. Il décrit également la mise en œuvre de *CaFE* et ses interactions avec le reste de la plate-forme.

1 Introduction

Le bon séquencement des événements au fil du temps fait partie des nombreux sujets d'analyse de programmes, par exemple pour l'étude de protocoles d'échange d'information ou de systèmes embarqués.

En particulier, la logique temporelle linéaire (LTL) [10] permet de décrire formellement le comportement attendu d'un système, sous la forme d'une succession d'actions distinctes. Une extension particulièrement intéressante de LTL pour l'analyse de programmes est la logique *CaRet*, qui autorise des raisonnements explicites sur la pile d'appel. Au sein de la plate-forme *Frama-C* [9], le greffon *CaFE* vérifie automatiquement de manière approchée (mais correcte) qu'un programme C, qui termine ou non, vérifie une formule logique *CaRet* donnée. *CaFE* tire parti des autres greffons de *Frama-C*, notamment *EVA* et *Pilat*, afin de limiter le problème d'explosion combinatoire. Cet article présente la mise en œuvre de *CaFE* et ses interactions avec le reste de la plate-forme.

Ce travail a été financé et fait parti du projet FUI INGO PCS.

2 La logique temporelle *CaRet*

La logique *CaRet* [2] est une extension de LTL [10], dont l'intérêt principal est de fournir des modalités adaptées à l'étude de la trace d'exécution d'un programme. Celle-ci est en effet structurée par les différents appels de fonctions intervenant à partir du point d'entrée principal. Pour faire apparaitre cette structure au niveau logique, *CaRet* distingue trois types de trace, associés chacun à un jeu d'opérateurs temporels classiques (X, désignant l'état suivant de la trace, U pour définir une propriété vraie jusqu'à un certain point, et les opérateurs dérivés G pour une propriété vraie à chaque instant et F pour une propriété qui sera vraie au moins un instant dans le futur). Ces trois types de trace sont les suivants.

- La trace d'exécution générale (X^g, U^g), i.e. instruction par instruction, correspond à des propriétés LTL pures.
- La trace abstraite (X^a, U^a) suit les instructions du module courant, mais ne descend pas dans les sous-modules. Ainsi, sur un nœud d'appel à un module M, $X^a p$ indique que la propriété p sera vraie lorsqu'on sort de M.
- la trace d'appels (X^-, U^-), remonte le long de la pile d'appel vers les états appelant le module courant. Plus précisément, $X^- p$ indique que p doit être vraie au niveau du site d'appel du module courant.

Cette logique est au moins aussi expressive que LTL, mais est plus concise et modulaire.

Exemples de spécifications. La logique *CaRet* permet d'exprimer des propriétés explicites sur la pile d'appel du programme. Elle peut représenter des propriétés de sûreté, de vivacité et contextuelles :

1. $G^a\ (p)$, exprime qu'en tout point de la trace abstraite partant du début de l'exécution (c'est-à-dire en tout point de la fonction `main` du programme, mais pas forcément dans les fonctions appelées) p doit être vérifiée ;
2. $G^g\ (X^- X^g F^a p)$ indique que toute fonction doit vérifier au moins une fois p au cours de son exécution : à tout instant, l'instant suivant sur la trace générale le dernier point d'appel, c'est-à-dire le point d'entrée de la fonction courante, doit vérifier $F^a p$, soit le fait que p doit être vraie avant que la fonction termine ;
3. $G^g\ (p \Rightarrow X^- X^a q)$, spécifie qu'à tout moment, si p est vérifiée, q doit être vérifiée à la fin de l'exécution de la fonction courante (plus précisément, l'instant suivant dans la trace abstraite de la fonction appelante le point d'appel de la fonction courante).

Si le premier exemple peut être vu comme un invariant de données faible en ACSL [3], le langage de spécifications formelles pris en charge par le noyau de Frama-C, le deuxième, qui demande qu'une propriété soit vérifiée en un point indeterminé d'une fonction, et le troisième, qui conditionne une post-condition q à un événement p interne à la fonction ne peuvent eux pas être directement représentés en ACSL.

Model Checking. Un algorithme de model-checking pour la vérification de formules *CaRet* sur des machines à états récursifs est décrit en détail dans [2]. Il s'agit de l'adaptation à la logique *CaRet* de l'algorithme classique de model-checking qui consiste à calculer l'intersection entre un automate représentant la négation d'une formule $\mathcal{F}$ et le modèle étudié, et à vérifier si cette intersection est vide. Les modèles étudiés en *CaRet* sont les *machines à états récursifs* [1] (ou *MER*), c'est à dire un ensemble d'automates (ou modules) contenant plusieurs entrées et sorties. Certains états de ces modules permettent d'appeler d'autres modules.

L'algorithme commence par intersecter chaque module avec l'automate généré à partir de la négation de la formule $\mathcal{F}$, c'est à dire générer un automate acceptant l'intersection de l'ensemble des mots acceptés par $\mathcal{F}$ avec l'ensemble des exécutions possibles des modules. Il prend en compte les chemins infinis en marquant chaque état de l'automate par *fin* si l'exécution courante termine ou *inf* si elle ne termine pas. Ce nouvel automate contient plusieurs états acceptants correspondants aux endroits où $\mathcal{F}$ est violée par une exécution de la *MER* initiale. S'il en existe, les exécutions passant par ces états acceptants représentent des contre-exemples à la formule $\mathcal{F}$ du modèle étudié. Si au contraire l'intersection est vide, $\mathcal{F}$ est vérifiée par la *MER* analysée.

3 *CaFE* : un model checker de programmes C

CaFE (CaRet Frama-C's extension) [8] est un model-checker de programmes C développé sous la forme d'un greffon de la plate-forme *Frama-C*. Son objectif est de vérifier automatiquement des propriétés écrites dans la logique temporelle *CaRet* par l'algorithme de [2] tout en optimisant la tâche par l'appel à d'autres greffons de la plateforme. La figure 1 présente le fonctionnement de *CaFE* et ses interactions avec différents greffons de la plateforme et un prouveur externe. Nous présentons dans la suite de cette section les différents outils utilisés.

Frama-C. *Frama-C* [9] est une plateforme open-source extensible et collaborative dédiée à l'analyse de programmes C et développée en OCaml. Son architecture modulaire permet la collaboration de différents outils internes et externes.

***CaFE*.** Dans *CaFE*, les programmes sont représentés sous la forme de *MER* où les fonctions représentent les différents modules et les états sont les instructions du programme C. Afin de pallier au problème d'explosion combinatoire propre au model checking, *CaFE* collabore avec plusieurs autres greffons de *Frama-C* qui lui fournissent de nombreuses informations, réduisant drastiquement la taille de l'automate.

- *EVA* [4] : un interpreteur abstrait capable de combiner différents domaines numériques. Ses résultats sont stockés sous forme de tables, et il est possible de calculer la valeur abstraite d'une expression en un point quelconque du programme potentiellement atteignable depuis le point d'entrée initial.

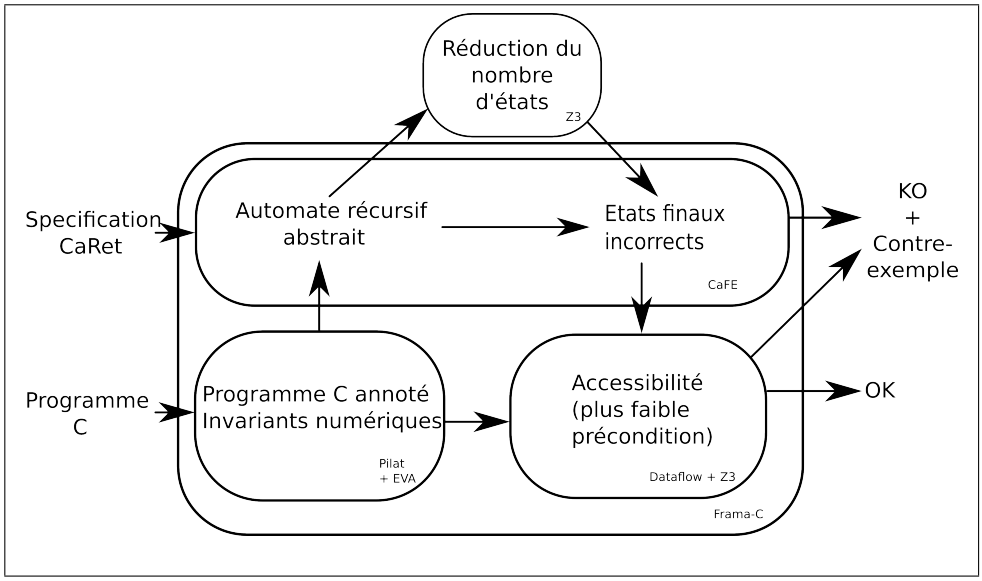

FIGURE 1: Fonctionnement de *CaFE* en interaction avec les autres greffons de *Frama-C*

- *Pilat* [7] : un générateur d'invariants inductifs de boucle. L'outil rajoute des annotations ACSL [3] aux boucles linéaires et polynomiales du programme.
- Le module Dataflow : un outil générique de parcours de programmes C en avant ou en arrière.

L'intersection entre le programme et la négation de la spécification *CaRet* utilise les résultats des deux premiers greffons afin de supprimer les états ne pouvant pas correspondre à une exécution effective du programme. Lorsque ces derniers ne suffisent pas à conclure sur la cohérence d'un état de l'automate, le SMT solver Z3 [6] est utilisé pour vérifier ou infirmer sa validité. Le nouvel automate $\mathcal{A}$ est ensuite simplifié en supprimant les exécutions non-acceptantes et non-accessibles. S'il reste des états acceptants, un calcul de plus faible pré-condition est effectué à l'aide du module Dataflow à partir des instructions liées aux états finaux des exécutions acceptantes (car chaque état de $\mathcal{A}$ correspond également à une instruction du programme original). Ce calcul utilise les invariants générés par les deux greffons précédemment cités pour extraire un contre-exemple potentiel montrant l'invalidation de la formule CaRet initiale. Un solveur SMT est utilisé pour conclure sur la validité du contre-exemple.

L'implantation de *CaFE* est correcte, au sens où la spécification *CaRet* est valide s'il ne trouve pas de contre-exemple, mais pas complet. Il peut en effet y avoir des faux positifs, c'est-à-dire des cas où l'algorithme retourne un contre-exemple potentiel alors que l'intersection est effectivement vide.

Limitations. L'outil *CaFE* se heurte malgré tout à plusieurs limitations. Premièrement, les boucles qui n'entrent pas dans le cadre de *Pilat* ne sont pas automatiquement dotées d'invariants pour aider l'analyse. Même si *EVA* arrive à surapproximer l'état de la boucle, cet état est souvent très imprécis. Lorsque cela arrive, *CaFE* renvoie de nombreux faux positifs. L'utilisateur peut néanmoins utiliser les différentes options d'*EVA* pour essayer d'obtenir un niveau de précision satisfaisant.

Enfin, les cas d'étude de taille importante ne passent pas à l'échelle. En général, le problème est lié à la complexité de la formule *CaRet* représentant la spécification que l'on souhaite prouver. Il est nécessaire de diviser à la main la formule en plusieurs sous-formules plus petites.

4 Étude de cas : protocole MOESI

Avec l'arrivée des processeurs multi-cœurs, le concept de ressources partagées a soulevé plusieurs problèmes, notamment celui de l'accès par un cœur à des données dans le cache en cours de traitement par un autre processeur. Le protocole MOESI est un protocole de cohérence de cache répondant à ce problème. Chaque ligne du cache peut être dans l'état ***m****odified*, ***o****wned*, ***e****xclusive*, ***s****hared* ou ***i****nvalid*. Lorsqu'un cœur souhaite lire une donnée en mémoire, il effectue une requête afin de s'assurer l'exclusivité de la ligne, sauf si un autre cœur l'a déjà et effectue des calculs avec. Si le statut de chaque ligne peut être modifié, le protocole doit bien sûr par contre conserver la même quantité de lignes au total entre l'entrée et la sortie.

Pour les besoins de l'expérimentation, la représentation séquentielle du protocole en figure 2, inspirée de [5], a été utilisée. Il s'agit d'une boucle infinie simulant à chaque tour un appel à des fonctions manipulant l'état des données du cache. Le but de l'expérimentation est de comparer deux spécifications différentes testant si l'ensemble des ressources est conservé durant l'exécution du programme sous deux hypothèses :

- chaque instruction est observée séquentiellement : $G^g(m+o+e+s+i = c)$;
- les changements d'état du cache effectués par la boucle sont atomiques : $G^a(m+o+e+s+i = c)$.

En ACSL, ces spécifications peuvent être représentés par un ensemble d'assertions sur l'ensemble de la fonction *main*, chaque assertion étant à prouver séparément. Ce procédé est peu efficace pour le traitement automatique de programmes de taille conséquente. Dans *CaFE*, le programme est directement assimilé à un automate récursif où les états sont les instructions et les modules sont les différentes fonctions. L'utilisation de l'ensemble de la plateforme est ici vitale pour le traitement des spécifications.

1. *Pilat* commence à générer l'unique invariant linéaire inductif de la boucle $m+o+e+s+i = k$, où k est une constante.

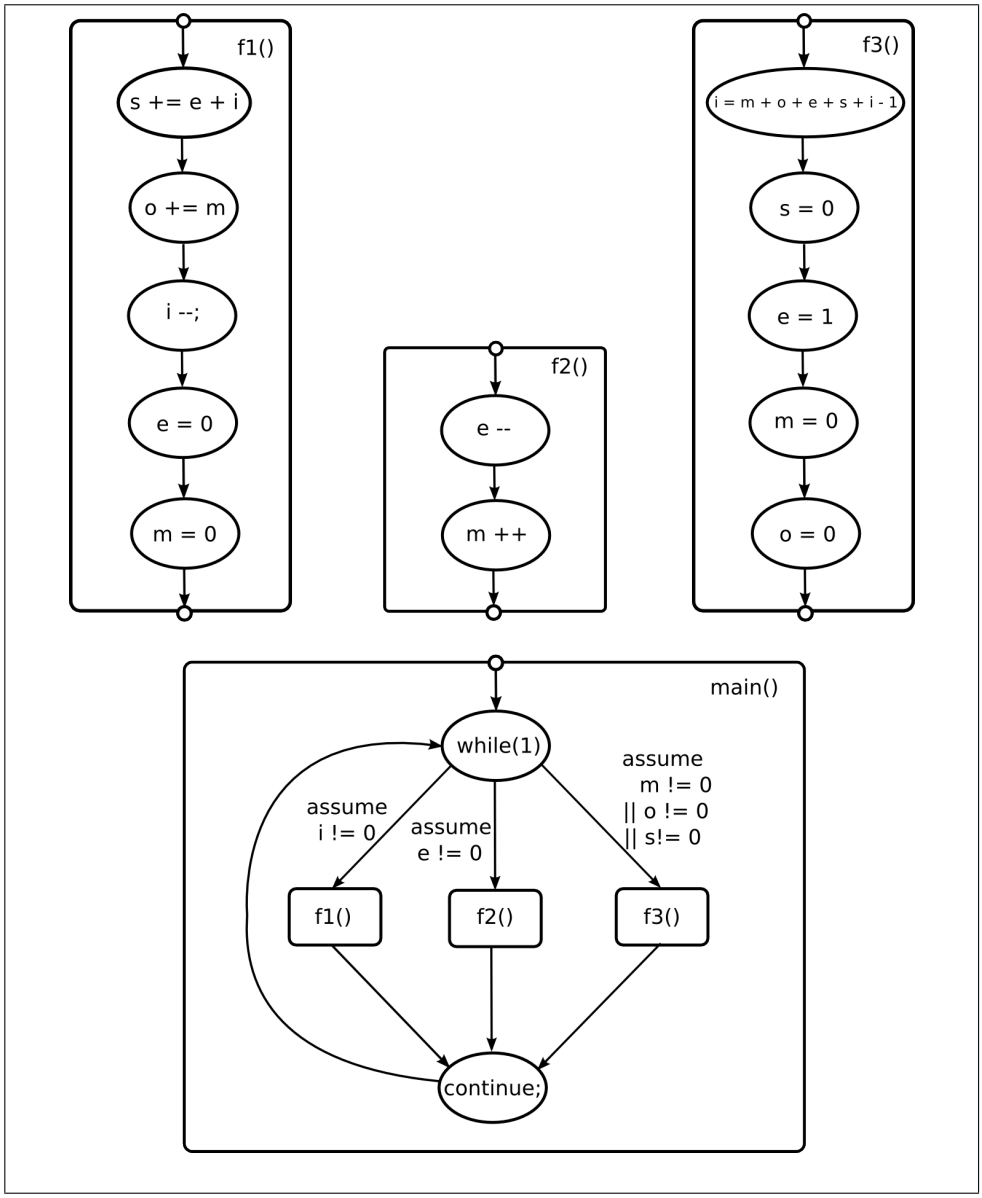

FIGURE 2: Machine à états récursifs représentant le protocole MOESI. *CaFE* effectue le produit de chaque automate par la négation de la spécification à vérifier.

2. *EVA* analyse l'ensemble du programme, prouvant en particulier que la boucle de la fonction *main* ne termine pas (le point de retour de la fonction est inatteignable) et ajoute à chaque instruction un invariant numérique dans un domaine choisi à l'avance (ici, le domaine des intervalles est utilisé).
3. Grâce à ces données, *CaFE* construit un automate généralisé récursif, puis supprime les états inaccessibles ou incohérents vis-à-vis des résultats des

deux greffons précédents. Il utilise également Z3 afin de préciser les résultats des deux autres outils.

4. Plusieurs états finaux sont encore présents. Un parcours en arrière du programme via le module dataflow et un appel à Z3 prouve rapidement que certains de ces états ne peuvent pas être atteints. Les états correspondants ainsi que les chemins possibles sont supprimés.

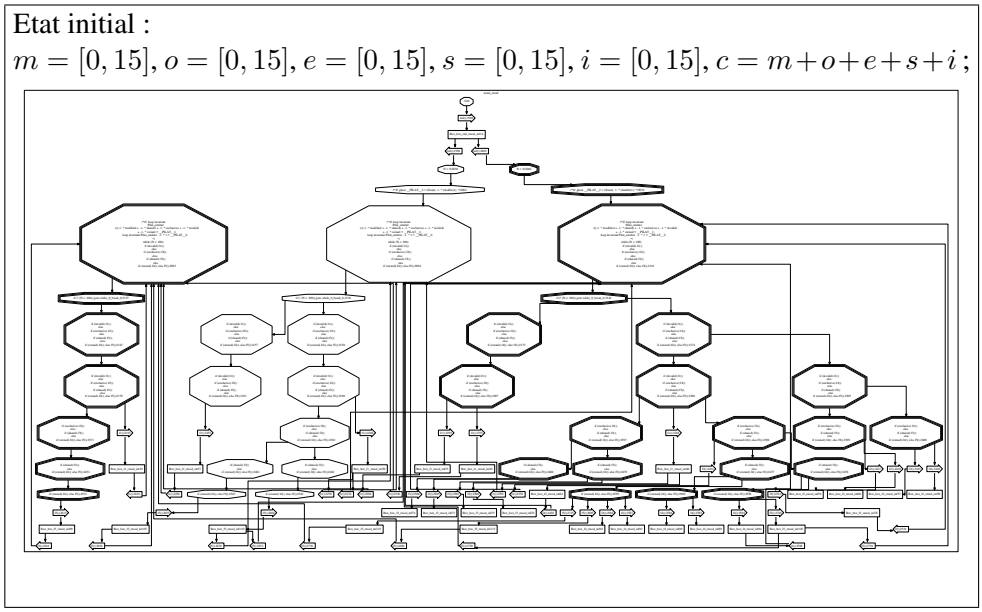

FIGURE 3: Résultat de *CaFE* : l'ensemble des contre-exemples ne respectant pas la formule $G^g(m + o + e + s + i = c)$. Il contient plusieurs états acceptant, la formule est donc infirmée. Dans le cas de la formule $G^a(m + o + e + s + i = c)$, aucun état ne reste à la fin de la simplification. Il n'existe aucun contre-exemple, et la formule est donc vérifiée.

Observations. Le résultat de la preuve de ces deux spécifications est visible en Figure 3. La spécification $G^g(m + o + e + s + i = c)$ est incorrecte puisque les fonctions n'étant pas considerées atomiques, des changements interviennent lors des mises à jour des différentes composantes de l'état du cache. *CaFE* arrive à générer plusieurs bons contre-exemples à cette spécification, qu'il retourne sous la forme d'un automate récursif contenant plusieurs états acceptants. Puisque chacun des états de cet automate correspond à une instruction, il est possible d'extraire un ensemble d'exécutions possibles menant à un état ne respectant pas la propriété. Inversement, l'invariant généré par *Pilat* prouve que l'ensemble des ressources est bel et bien préservé entre les appels et retours des fonctions modifiant l'état du cache. La spécification $G^a(m + o + e + s + i = c)$ spécifiant l'invariance du nombre de ressources entre le début et la fin de chaque étape de l'exécution du protocole est donc bien vérifiée.

5 Conclusion

Avec *CaFE*, la plate-forme *Frama-C* se dote d'un outil de model-cheking qui traite efficacement des spécifications dans une logique temporelle expressive. Il permet également d'apprécier les interactions possibles entre différents outils qui, décorrélés, ne permettent pas de vérifier la spécification souhaitée. La suite du développement de *CaFE* est principalement axée sur l'explosion combinatoire, toujours problématique lors du traitement d'exemples industriels, et sur une recherche de contre-exemple plus efficace. Un autre objectif est l'extraction des contre-exemples sous une forme adaptée aux autres greffons de la plateforme pour permettre leur vérification ou leur infirmation.

Références

[1] R. ALUR, M. BENEDIKT, K. ETESSAMI, P. GODEFROID, T. REPS et M. YANNAKAKIS : Analysis of recursive state machines. *ACM TOPLAS*, 27(4), 2005.

[2] R. ALUR, K. ETESSAMI et P. MADHUSUDAN : A temporal logic of nested calls and returns. *In TACAS 2004*, p. 467–481. Springer, 2004.

[3] P. BAUDIN, J.-C. FILLIÂTRE, C. MARCHÉ, B. MONATE, Y. MOY et V. PREVOSTO : ACSL : ANSI C Specification Language, version 1.12, 2016.

[4] S. BLAZY, D. BÜHLER et B. YAKOBOWSKI : Structuring abstract interpreters through state and value abstractions. *In VMCAI 2017, Proceedings*, p. 112–130, 2017.

[5] E. CARBONELL : Polynomial invariant generation. http://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/list.html.

[6] L. M. de MOURA et N. BJØRNER : Z3 : an efficient SMT solver. *In TACAS, Proceedings*, p. 337–340, 2008.

[7] S. de OLIVEIRA, S. BENSALEM et V. PREVOSTO : Polynomial invariants by linear algebra. *In ATVA 2016, Proceedings*, p. 479–494. Springer, 2016.

[8] S. de OLIVEIRA, V. PREVOSTO et S. BARDIN : Au temps en emporte le C. *In Actes des JFLA*, 2015.

[9] F. KIRCHNER, N. KOSMATOV, V. PREVOSTO, J. SIGNOLES et B. YAKOBOWSKI : Frama-C : A software analysis perspective. *Formal Aspects of Computing*, 27(3), 2015.

[10] A. PNUELI : The temporal logic of programs. *In 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, p. 46–57, 1977.