Vérification interactive de propriétés à l'exécution d'un programme avec un débogueur

Raphaël Jakse, Yliès Falcone, Jean-François Méhaut, Kevin Pouget

Compas'2016 : Parallélisme/ Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Vérification interactive de propriétés à l'exécution d'un programme avec un débogueur

Raphaël Jakse, Yliès Falcone, Jean-François Méhaut, Kevin Pouget

Univ. Grenoble-Alpes, Inria, LIG - Prenom.Nom@imag.fr

Résumé
Le *monitoring* est l'étude d'un système pendant son exécution, en surveillant les évènements qui y entrent et qui en sortent, afin de découvrir, vérifier ou pour faire respecter des propriétés à l'exécution. Le *débogage* est l'étude d'un système pendant son exécution afin de trouver et comprendre ses dysfonctionnements dans le but de les corriger, en inspectant son état interne, de manière interactive.
Dans ce papier, nous combinons le monitoring et le débogage en définissant un moyen efficace et pratique de vérifier automatiquement des propriétés à l'exécution d'un programme à l'aide d'un débogueur afin d'aider à détecter des anomalies dans son code, en conservant le caractère interactif du débogage classique.

Mots-clés : monitoring, débogage, vérification, propriété, interactif

---

1. Introduction

Lors du développement d'un programme, corriger les anomalies le plus tôt possible est important. Ceci peut se révéler difficile. Une manière de faire est d'observer une anomalie et de commencer une session de débogage pour trouver sa cause. Le débogage peut être un processus fastidieux, avec beaucoup d'essais-erreurs. Une session de débogage consiste généralement à répéter les étapes suivantes : exécuter le programme dans le débogueur, poser des points d'arrêts avant l'apparition supposée du bogue, trouver l'endroit où l'exécution du programme devient erratique et inspecter l'état interne (pile d'appels, valeur des variables) pour déterminer la source du problème. Par ailleurs, une anomalie n'est pas toujours évidente à remarquer. Un bogue ne mène pas systématiquement à un crash, il peut rester non décelé pendant tout un cycle de développement.
Cet article présente une méthode pour faciliter la détection d'anomalies en rendant certaines vérifications systématiques et en automatisant certains aspects fastidieux d'une session de débogage. Notre approche s'inspire du *monitoring*. Dans ce domaine, le système étudié est considéré comme une boite noire, en s'appuyant sur la séquence d'évènements (trace) qu'il produit pendant son exécution. Considérer l'exécution sous cet angle peut faciliter l'expression de propriétés portant sur le comportement attendu du programme. Un débogueur peut fournir une trace lors de son exécution. Nous définissons un moyen d'évaluer des propriétés automatiquement à chaque exécution du programme dans le débogueur. Lorsqu'une propriété est violée, le développeur peut être averti et étudier à cet instant l'état interne du programme à l'aide du débogueur pour comprendre ce qu'il s'est passé.

Notre approche se veut être la moins intrusive possible. Elle s'intègre au débogueur, déjà utilisé par les développeurs, et ne bouleverse pas leur méthode de travail, ce qui pourra faciliter son adoption. Ceci lui permet également de tirer partie de l'infrastructure existante des débogueurs.
En Section 2, nous présentons notre approche de débogage interactif guidé par le monitoring de propriétés : nous décrivons comment i) récupérer la trace d'exécution d'un programme à l'aide du débogueur, ii) définir et iii) évaluer les propriétés. En Section 3, nous présentons *check-exec*, notre preuve de concept. En Section 4, nous évaluons notre l'approche en nous basant sur *check-exec*.

2. Débogage interactif guidé par le monitoring de propriétés

2.1. Modèle d'exécution

Lors du débogage, l'exécution est vue comme une séquence d'états du programme, que l'on inspecte étape par étape à l'aide d'un débogueur afin de comprendre l'origine et la cause d'un dysfonctionnement. L'état du programme peut être modifié en cours d'exécution. Cela permet de tester une hypothèse sur le bogue sans devoir recompiler et exécuter le programme entièrement une nouvelle fois.
En monitoring, l'exécution est de manière similaire abstraite par une séquence d'évènements de mises à jour de l'état du programme. L'objectif principal du monitoring est de détecter les dysfonctionnements d'un système que l'on considère comme une boite noire : son état interne n'est pas accessible et ne peut pas être altéré. L'analyse de la trace d'exécution se fait parfois de façon *post-mortem*. Cette abstraction de l'exécution peut faciliter l'expression de propriétés sur le comportement d'un programme. Par exemple, une propriété possible dans un jeu vidéo peut prescrire qu'un personnage ne doit pas être tué juste après avoir apparu [11].

2.2. Méthodologie

Notre approche combine les idées du débogage et du monitoring comme suit. Le développeur décrit des propriétés à vérifier. Ces propriétés sont vérifiées en parallèle lors de l'exécution du programme en cours de développement, en suivant la trace en temps réel. Dès qu'une propriété n'est plus respectée, l'exécution peut être suspendue afin de laisser le développeur déboguer le programme de manière traditionnelle. Le programme est lancé dans un débogueur muni d'une extension dont le rôle est d'évaluer les propriétés que nous lui donnons et de suspendre l'exécution lorsqu'une propriété est violée.

2.3. Modèle de propriétés et instrumentation

Les propriétés peuvent être exprimées sur l'ensemble des évènements qu'un débogueur peut fournir. Des valeurs peuvent être liées aux évènements. L'appel de fonction, par exemple, est un évènement paramétré par les valeurs des arguments passés lors de cet appel, ainsi que par les valeurs accessibles à cet instant (les variables globales par exemple). Les évènements peuvent être surveillés en mettant en place des points d'arrêts. Ainsi, lorsque l'évaluation d'une propriété nécessite de surveiller les appels à une fonction particulière, un point d'arrêt est posé sur cette fonction. Lorsqu'il est atteint durant l'exécution, le moniteur concerné met à jour son évaluation et peut laisser l'exécution reprendre, ou laisser le développeur interagir avec le débogueur si la propriété a été violée durant l'exécution. Un exemple simple de propriété est donné dans la FIGURE 1. Lorsqu'un événement n'influence plus l'évaluation de la propriété, le point d'arrêt qui lui est associé n'est plus utile, et il est donc retiré : l'instrumentation est *dynamique*.

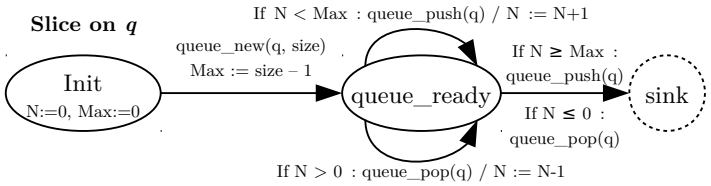

FIGURE 1 – Propriété portant sur le dépassement d'une file dans le programme producteur-consommateurs décrit dans la Section 4.1.

Nous décrivons les propriétés dans un modèle basé sur les automates à états finis. Il est composé d'états, de transitions et d'un environnement (une mémoire) et reconnait des séquences d'évènements. Les transitions sont munies de gardes exprimées en fonctions des paramètres des évènements et de l'environnement. Des actions liées aux états ou aux transitions peuvent modifier l'environnement, l'état du débogueur et interagir avec l'état interne du programme.

2.4. Traces paramétrées

Certaines propriétés ne portent pas sur des traces d'exécution entières, mais sur des parties de traces (*slice*) concernant des objets spécifiques. Par exemple, on peut vouloir exprimer une propriété sur chaque fichier manipulé par un programme : un fichier doit être fermé s'il a été ouvert. La propriété de la Figure 1 est, elle, paramétrée par la pile.
Nous nous sommes basés sur le travail de Chen et Roşu [7] pour pouvoir exprimer de telles propriétés. Un moniteur ne correspond plus à une seule machine à états finis : à chaque instance particulière d'un objet correspond une machine à états.
Dans la section suivante, nous présentons la preuve de concept mettant en œuvre ces principes.

3. Check-exec : une extension à GDB

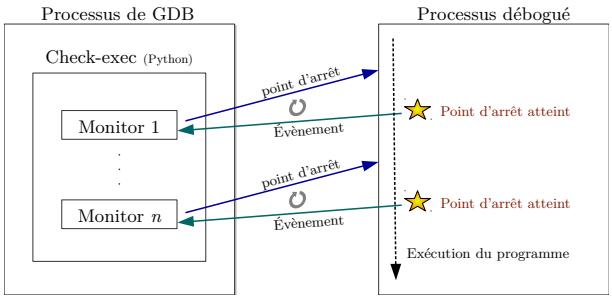

FIGURE 2 – Architecture de *check-exec*

Afin d'évaluer et de concrétiser l'approche définie en Section 2, nous avons développé une preuve de concept nommée *check-exec* [1] qui étend GDB en utilisant son interface Python. *Check-exec* propose une vue graphique et animée de l'automate de la propriété actuellement testée. Elle est optionnelle mais peut faciliter la compréhension de l'état actuel de la propriété et, par conséquent, du programme. L'outil permet également de contrôler les moniteurs et d'accéder à leur état interne (instances des propriétés, états courants, environnements).
Dans ce travail, nous nous sommes concentrés sur les programmes écrits en C et C++. Cependant, gérer d'autres langages de programmation pris en charge par GDB est possible.
Check-exec est écrit en Python et il est exécuté dans le processus de GDB. Il peut faire fonctionner un ou plusieurs moniteurs, chacun évaluant les instances d'une propriété de manière

1. *Check-exec* est téléchargeable à l'adresse `http://check-exec.forge.imag.fr/`.

indépendante. Chaque moniteur, selon les évènements observés, place et supprime des points d'arrêts (internes) dans le processus surveillé. Quand un point d'arrêt est atteint, l'état de la propriété est mis à jour et l'exécution reprend.

```
1  slice on queue
2  initialization {
3    N   = 0
4    max = 0
5  }
6  state init accepting {
7    transition {
8      event queue_new(queue, size : int)
9      success {
10       max = size - 1
11     } queue_ready
12   }
13 }
14 state queue_ready accepting {
15   transition {
16     event queue_push(queue, prod_id) {
17       return N < max
18     }
19     success {
20       N = N + 1
21       print("nb elem: "+str(N));
22     } queue_ready
23     failure {
24         print("%d made %d overflow!"
25             % (prod_id, queue))
26     } sink
27   }
28   transition {
29     event queue_pop(queue, prod_id) {
30       return N > 0
31     }
32     success {
33       N = N - 1
34       print("nb elem: "+str(N));
35     } queue_ready
36     failure sink
37   }
38 }
39 state sink non-accepting sink_reached()
```

FIGURE 3 – Description dans le format de *check-exec* de la propriété représentée dans la FIGURE 1

Un schéma de l'exécution d'un programme avec l'outil est présentée dans la FIGURE 2. Un exemple de propriété écrite pour *check-exec* est donné en FIGURE 3.

4. Évaluation

Nous avons évalué notre approche à l'aide de *check-exec* dans cinq scénarios afin de mesurer son utilité, son efficacité et ses limitations en termes de performance [2].

4.1. Producteur-consommateurs multi-threadé

Dans cette section, nous évaluons l'approche sur ce scénario : un développeur travaille sur une application multi-threadée dans laquelle une file est remplie et vidée par différents threads. Une erreur de segmentation se produit dans certains cas. Nous avons écrit un programme sur ce modèle en y introduisant volontairement une erreur de synchronisation, ainsi qu'une propriété (voir FIGURE 1) portant sur le nombre d'ajouts dans la file afin de détecter un dépassement. La taille de la file est un paramètre de l'évènement `queue_new`. La fonction `queue_push` ajoute un élément dans la file, dont l'appel est attendu par la transition définie à la ligne 15 de la FIGURE 3. Le programme est lancé avec *check-exec*. L'exécution s'arrête dans l'état `sink` (défini à la ligne 39 de la FIGURE 3). Dans le débogueur, nous avons accès à la ligne précise dans le code source où la fonction est appelée, ainsi qu'à la pile d'appels complète. Dans certaines conditions, un mutex n'est pas verrouillé. Après correction, le programme fonctionne correctement.

4.2. Micro-benchmark

Lors de l'écriture d'une propriété, le développeur doit avoir conscience du surcoût de l'instrumentation en fonction de l'espacement temporel entre les évènements qu'elle traite.
Dans cette deuxième expérimentation, nous évaluons ce surcoût en fonction de la fréquence des évènements. Nous avons écrit un programme C qui appelle, dans une boucle, une fonction qui ne fait rien. Nous simulons l'espacement entre les évènements par une boucle d'une durée paramétrable. Les résultats de ce benchmark, en utilisant un Core i7-3770 à 3.40 GHz, sous Ubuntu 14.04 et le noyau Linux 3.13.0, sont présentés en FIGURE 4. Avec une période de 0.5 ms entre deux évènements, nous avons mesuré un ralentissement de 2. En dessous de 0.5 ms, le surcoût peut être important. À partir de 3 ms, le ralentissement est de moins de 20 % et à partir de 10 ms, l'exécution n'est plus ralentie que de 5 %. Nous avons observé que le surcoût est dominé par le traitement des points d'arrêts lorsqu'ils sont rencontrés à exécution. Le surcoût

2. Une vidéo et les codes sources nécessaires pour reproduire ces benchmarks sont disponibles à l'adresse `http://check-exec.forge.imag.fr/eval/`.

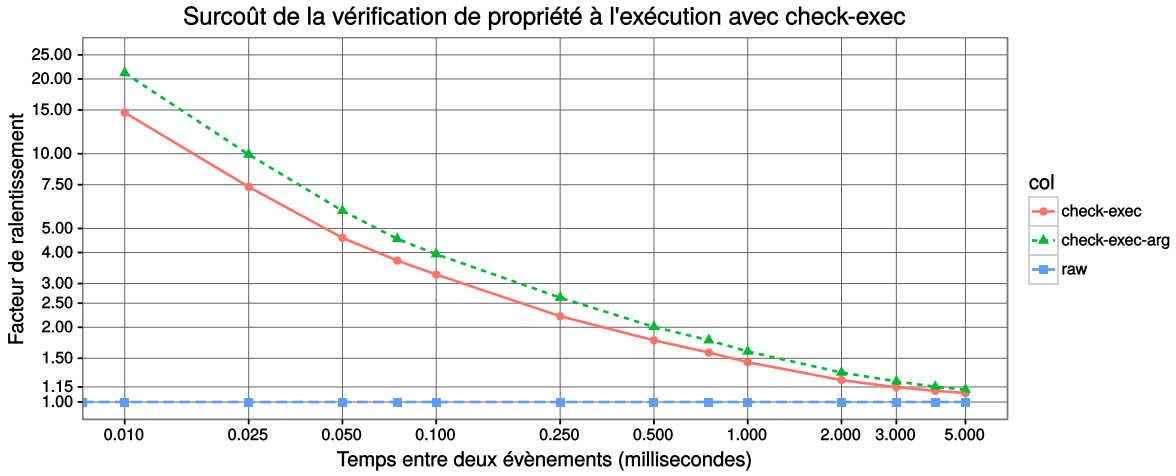

FIGURE 4 – Surcoût de l'instrumentation avec *check-exec*. La courbe `check-exec-arg` correspond à l'évaluation d'une propriété qui récupère un argument depuis les appels de la fonction surveillée.

absolu par évènement surveillé, à l'instar du surcoût de la récupération d'un paramètre de l'événement, est constant. Nous avons mesuré le coût moyen de la rencontre d'un point d'arrêt pendant l'exécution. Nous avons obtenu 95 µs sur la même machine et plutôt autour de 300 µs une machine moins rapide (i3-4030U CPU à 1.90 GHz).

4.3. Lecteur multimédia et jeu vidéo

Nous évaluons notre approche en l'appliquant à des applications multimédias. Une propriété demande au moniteur de surveiller la fonction qui dessine chaque image à l'écran dans les lecteurs vidéos VLC et MPlayer et dans le jeu de plateformes 2D SuperTux, respectivement *ThreadDisplayPicture*, *update_video* et *DrawingContext : :do_drawing*. Pour SuperTux, la fonction est appelée environ 60 fois par seconde. Pour les lecteurs vidéo, elle est appelée 24 fois par seconde. Dans tous les cas, le nombre d'images par seconde n'est pas affecté et le surcoût au niveau de l'utilisation du processeur reste modéré : nous obtenons une consommation de moins de 10 % supplémentaire par GDB. Ces résultats correspondent à nos mesures de la Section 4.2 : 16 ms séparent deux appels d'une fonction qui est exécutée 60 fois par seconde. Ainsi, notre approche n'engendre pas de surcoût notable pour les applications multimédia lorsque les évènements sont espacés par le temps qui sépare deux images.

4.4. Téléchargement d'un fichier

Nous mesurons le surcoût de l'instrumentation en présence d'entrées-sorties. Nous téléchargeons un fichier de 862 Mio avec l'application *wget* à partir d'un serveur HTTP lancé localement en utilisant la classe SimpleHTTPServer de l'interpréteur Python. On surveille tous les appels à la fonction `read`. Le téléchargement a été redirigé vers `/dev/null`, évitant les accès disque en écriture. Le téléchargement sans *check-exec* prend 6.0 secondes (133 Mio/s). Avec, le téléchargement prend 27 secondes (31.4 Mio/s), correspondant à un ralentissement de 4,5. Lors de ce téléchargement, `wget` fait 110294 appels à `read`, ce qui correspond à un écart moyen de 0.05 ms entre deux évènements. Les résultats sont similaires à ceux que nous obtenons en 4.2.

4.5. Ouverture et fermeture des fichiers, itérateurs

Nous évaluons le ressenti du surcoût avec des applications répandues. Nous vérifions que tous les fichiers ouverts sont bien fermés avec le gestionnaire de fichiers Dolphin, le navigateur Web NetSurf, l'éditeur de texte Kate et l'éditeur d'image Gimp. Malgré quelques ralentissements,

causés par des accès disques réguliers, ils restent utilisables.
De même, nous vérifions qu'aucun itérateur sur table de hachage du type (`GHashTableIter`) invalidé de la bibliothèque GLib n'est utilisé. Les applications les plus simples (comme la calculatrice de Gnome) restent utilisables mais de forts ralentissements se font sentir lors de l'évaluation de cette propriété. Ces itérateurs sont très souvent utilisés, même lors d'un simple mouvement de souris. Nous présentons en section 6 des pistes pour repousser les limites ici atteintes de notre approche.

4.6. Instrumentation dynamique sur une pile

Nous mesurons les effets de l'instrumentation dynamique sur la performance. Un programme ajoute et enlève, alternativement, les 100 premiers entiers naturels dans une pile et on vérifie que l'entier spécial 42 sort bien de la pile après avoir été ajouté. Une première version de cette propriété tire partie de l'instrumentation dynamique, pour laquelle l'appel à la fonction de retrait d'un élément de la file n'est surveillé que lorsque le moniteur sait que 42 est dans la file. Une deuxième surveille tous les évènements de façon inconditionnelle. Dans cet exemple, l'exécution est 2.2 fois plus rapide avec la première propriété.

5. Travaux connexes

Monitoring

Des travaux similaires existent au niveau de la vérification de propriétés pour le langage Java (le cadriciel jassda [4] qui utilise des spécifications de type CSP, LARVA [8] et JavaMOP [6]) et pour d'autres langages de programmation. Les évènements, dans ces travaux, sont capturés en utilisation les capacités d'introspection de ces machines virtuelles et la programmation par aspects. Chabot *et al.* ont présenté une approche pour vérifier des assertions temporelles à l'exécution dans des programmes C [5], avec deux techniques d'instrumentation : une transformation du code source avant sa compilation, et des points d'arrêts placés à l'aide de l'outil `ptrace`. Ces approches n'abordent pas le débogage interactif.

Débogage

Un débogueur a été écrit pour vérifier le typage de programmes écrits en C [10] en étiquetant les cellules mémoires avec des types. L'exécution est suspendue en cas d'inconsistance Ducassé et Jahier [9] se basent également sur la surveillance d'évènements. Le programme C débogué est lancé dans un autre processus que le moniteur, qui est connecté à un *traceur* qui produit des traces d'exécutions. Cependant, leurs travaux se concentrent sur l'analyse *post mortem* des traces : le débogage n'est pas interactif et l'exécution du programme n'est pas modifiée.

6. Travaux futurs

Pour l'instant, seuls les appels de fonction sont surveillées. Prendre en charge **d'autres types d'évènements** (par ex. les accès mémoire) pourrait être intéressant.
Le traitement des points d'arrêt est coûteux [5]. Une piste à explorer serait l'instrumentation avec de l'**injection de code** en utilisant le débogueur et l'intégration d'une partie du moniteur dans les fils d'exécutions des programmes surveillés. Les communications entre le débogueur et le programme seraient limitées au strict nécessaire (par exemple, lorsqu'une propriété est invalidée), en gardant la flexibilité de l'approche.
Intégrer la notion de **checkpoint** à notre approche ajouterait la possibilité de revenir en arrière dans l'exécution, modifier son état puis relancer le programme à partir de ce point en conser-

vant l'aspect vérification de propriété.

Bibliographie

1. Al-Sharif (Z.) et Jeffery (C. L.). – Language support for event-based debugging. – In *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), Boston, Massachusetts, USA, July 1-3, 2009*, pp. 392–399, 2009.
2. Al-Sharif (Z. A.), Jeffery (C. L.) et Said (M. H.). – Debugging with dynamic temporal assertions. – In *25th IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Naples, Italy, November 3-6, 2014*, pp. 257–262, 2014.
3. Auguston (M.), Jeffery (C.) et Underwood (S.). – A framework for automatic debugging. – In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*, pp. 217–222, 2002.
4. Brörkens (M.) et Möller (M.). – Dynamic event generation for runtime checking using the JDI. *Electr. Notes Theor. Comput. Sci.*, vol. 70, n4, 2002, pp. 21–35.
5. Chabot (M.), Mazet (K.) et Pierre (L.). – Automatic and configurable instrumentation of C programs with temporal assertion checkers. – In *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, pp. 208–217, 2015.
6. Chen (F.) et Rosu (G.). – Mop : an efficient and generic runtime verification framework. – In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pp. 569–588, 2007.
7. Chen (F.) et Rosu (G.). – Parametric trace slicing and monitoring. – In Kowalewski (S.) et Philippou (A.) (édité par), *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, *Lecture Notes in Computer Science*, volume 5505, pp. 246–261. Springer, 2009.
8. Colombo (C.), Pace (G. J.) et Schneider (G.). – LARVA — safer monitoring of real-time java programs (tool paper). – In *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pp. 33–37, 2009.
9. Ducassé (M.) et Jahier (E.). – Efficient automated trace analysis : Examples with morphine. *Electr. Notes Theor. Comput. Sci.*, vol. 55, n2, 2001, pp. 118–133.
10. Loginov (A.), Yong (S. H.), Horwitz (S.) et Reps (T. W.). – Debugging via run-time type checking. – In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pp. 217–232, 2001.
11. Varvaressos (S.), Lavoie (K.), Massé (A. B.), Gaboury (S.) et Hallé (S.). – Automated bug finding in video games : A case study for runtime monitoring. – In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pp. 143–152, 2014.