Julien Charles Le 9 mai 2005

Vérification d'un composant Java

Le vérificateur de bytecode sert à vérifier le bon typage du bytecode Java. Cette propriété de typage assure une exécution sans erreur des programmes Java. La correction d'un vérificateur de bytecode est donc cruciale, en particulier dans le cadre de systémes embarqués (cartes à puces, téléphonie mobile...). Plus généralement la certification de composants système est un enjeu important à cause de la multiplication des petits objets portables sécurisé (POPs). Le but de mon stage est de développer de nouvelles technologies permettant de vérifier des composant systèmes écrit en Java. Dans un premier temps nous prenons comme cas d'étude le vérificateur de bytecode.

1 Spécification des programmes Java

Java Modelling Language (JML) est un langage de spécification permettant d'annoter et de vérifier les programmes Java. JML [6] est un langage dérivé d'Eiffel propre à Java. Ces spécifications peuvent être utilisées pour effectuer des tests lors de l'exécution ou peuvent être vu d'une manir̂e plus formelle comme des annotations dans une logique de Hoare.

Des outils ont ainsi été développés pour vérifier formellement ces spécifications [2] : ESC/Java, qui est très utilisé mais ne permet pas de résoudre des preuves interactivement, Krakatoa, qui permet de résoudre des obligations de preuves grâce à Coq (un prouveur interactif), mais qui gère mal l'héritage, ou Jack [3] qui gère correctement l'héritage, permet de faire des preuves automatiques, mais ne permet pas vraiment de faire des preuves interactives. Jack n'avait pas de plugin Coq, or pour prouver mon vérificateur de bytecode un prouveur automatique ne suffisait pas. Puisque Jack était développé dans l'équipe où je faisais mon stage, nous avons choisi de le rendre utilisable en écrivant un plugin Coq.

2 Génération des obligations de preuve

Coq [1] est un langage de preuve interactive fondé sur le calcul des constructions inductives, qui est très puissant (récemment G.Gontier a vérifié en Coq une preuve du théorème des 4 couleurs). Les preuves s'écrivent sous forme de scripts, chaque élément du script s'appelant 'tactique'. Une tactique est une commande qui peut être appliquée à une preuve. Le but de la plupart des tactiques est de modifier cette obligation de preuve afin de se raprocher de sa résolution. Coq comporte aussi un langage, Ltac, qui sert à créer des macros d'autre tactiques.

Jack est un outils qui, à partir d'un programme Java annoté avec JML, génère des obligations de preuves dans un langage intermédiaire. J'ai donc écrit un plugin qui traduit les obligations de preuves exprimées dans ce langage intermédiaire en des preuves résolvables avec Coq. Ce faisant, j'ai aidé à corriger des erreurs qui subsitaient dans Jack et j'y ai ajouté quelques fonctionnalités, notamment pour automatiser les preuves : dans Jack, certaines structures d'obligations de preuves se retrouvent fréquemment ; j'ai écrit des macros exprimées en Ltac pour les résoudre automatiquement et de manìre ciblée avec Coq.

3 Le vérificateur de bytecode

L'autre partie importante de mon stage était la preuve du vérificateur de bytecode [7]. Peu de choses semblables ont été effectuées avec JML (un des exemples d'annotations les plus connus étant celui de DeMoney). En revanche des travaux similaires ont été effectués au niveau de la preuve à l'aide d'un prouveur interactif mais en modélisant le programme dans le langage du prouveur (en Isabelle [5] et en Coq [4]). Ces preuves s'attachaient à vérifier la modélisation du vérificateur, je me suis attaché plutôt à vérifier l'implémentation.

Le vérificateur de bytecode est exécuté sur chacune des classes Java avant leur exécution. L'algorithme utilisé est celui de Kildall. Le programme pouvant facilement devenir important, j'ai choisi de le diviser en plusieurs parties indépendantes, afin de rendre mon implémentation plus abstraite : j'ai implémenté d'un côté l'itération principale qui, quelle que soit la spécification du langage Java choisi, restera pareille et de l'autre les instructions et les états mémoires qui dépendent du modèle choisi. Les instructions et les états sont ainsi implémentés avec des classes abstraites.

Références

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development ; Coq'Art : The Calculus of Inductive Constructions*. Series : Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[2] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005.

[3] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness : A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003 : Formal Methods : International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[4] G. Dufay. *Vérification formelle de la plate-forme Java Card*. PhD thesis, Université de Nice Sophia-Antipolis, December 2003.

[5] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 298(3) :583–626, 2003.

[6] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, and J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, Apr. 2003.

[7] X. Leroy. Java bytecode verification : an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.