GECCOO

Génération de code certifié pour des applications orientées objet

Spécification, raffinement, preuve et détection d'erreurs

Rapport à mi-parcours
http://geccoo.lri.fr

Avril 2005

Résumé

Ce document présente le rapport à mi-parcours de l'ACI sécurité GECCOO. Il met en évidence les activités de collaboration ayant pris place au sein du projet, en particulier dans le développement de synergies entre les outils, l'étude des mêmes applications suivant les différentes approches. Il résume les principales avancées et conclut en indiquant les perspectives pour la fin du projet.

1 Introduction

Le projet GECCOO a officiellement débuté en juillet 2003 pour une durée de 3 ans.

1.1 Problématique

L'objectif du projet est de proposer des méthodes et des outils pour le développement de programmes orientés objets offrant des garanties fortes de sécurité. Le projet s'intéresse plus spécifiquement à des programmes embarqués sur des cartes à puce ou dans des terminaux qui sont décrits dans des sous-ensembles de Java (par exemple JavaCard).

Notre approche s'apuie sur la spécification des programmes Java à l'aide de formules exprimées dans le langage JML et la possibilité de construire des outils de vérification qui transforment un programme annoté en un ensemble d'obligations de preuve qu'il faut ensuite démontrer.

Le projet s'intéresse à plusieurs problèmes dans cette approche. Tout d'abord, les contraintes de correction et de sécurité des applications doivent être exprimées dès la phase de conception du système. Il convient de les spécifier dans un langage de haut niveau, le système étant ensuite raffiné jusqu'à des programmes exécutables et efficaces. Les étapes de raffinement peuvent nécessiter la résolution d'obligations de preuves. Les obligations de preuves engendrées pour la correction de programmes objets nécessitent de raisonner sur les états de la mémoire, ces preuves sont laborieuses mais assez spécifiques, il est nécessaire de développer des procédures automatiques de preuves adaptées à ces cas. Enfin il est rare qu'une spécification et un programme soient corrects du premier coup. Le système doit donc détecter le plus tôt possible les

erreurs et permettre d'utiliser les obligations de preuve pour engendrer des tests ou produire un code défensif dans le cas où certaines propriétés ne peuvent être garanties de manière statique.

Les méthodes développées dans ce projet ont un champ large d'application dans les environnements destinés à produire du code objet garanti correct par rapport à des spécifications formelles.

Le projet met en relation des équipes dont l'expertise est diversifiée (preuves automatiques, générateur d'obligations de preuves, modèles objets, raffinement, sécurité des applications JavaCard, génération de tests).

1.2 Composition au 1er avril 2005

1.2.1 équipe TFC, LIFC, Besançon

Une partie de l'équipe TFC participe au projet INRIA CASSIS commun avec le LORIA à Nancy.

Bellegarde Françoise	Professeur
Bouquet Fabrice	MCF
Dadeau Frédéric	Doctorant (2ème année)
	Allocation ministère liée à l'ACI
Giorgetti Alain	MCF
Groslambert Julien	Doctorant (1ère année)
	Allocation ministère
Julliand Jacques	Professeur
Kouchnarenko Olga	MCF (HDR)
Legeard Bruno	Professeur
Masson Pierre-Alain	MCF

1.2.2 Projet CASSIS, LORIA, Nancy

Le projet CASSIS est commun avec le Laboratoire d'informatique de Franche-Comté.

Silvio Ranise	CR INRIA
Calogero Zarba	Post-doc INRIA lié à l'ACI

1.2.3 Projet Everest, INRIA Sophia-Antipolis

Le projet Everest est issu du projet Lemme qui apparait dans la proposition GECCOO.

Gilles Barthe	DR INRIA
Julien Charles	Stagiaire M2 Recherche 2005
Thibault Dupont	Stagiaire M2 Recherche 2005
Marieke Huisman	CR INRIA

1.2.4 Projet ProVal INRIA Futurs, LRI, Orsay

Le projet ProVal est issu du projet LogiCal qui apparait dans la proposition GECCOO.

Nicolas Ayache	Stagiaire M2 Recherche 2005
Sylvain Conchon	MCF
Evelyne Contejean	CR CNRS
Jean-Christophe Filliâtre	CR CNRS
Claude Marché	MCF
Christine Paulin	Professeur

Stagiaires : Vikrant Chaudhary (IIT Dehli, mai-juillet 2004) Julien Roussel (Epita 3A, janvier-juin 2005)

1.2.5 Équipe VASCO, LSR, Grenoble

2 Résumé des principales avancées

2.1 Développement d'outils

Les participants du projet développent depuis plusieurs années des outils de modélisation et de preuve. Il s'agit de :

- Jack et Krakatoa, deux générateurs d'obligations de preuve pour des programmes Java spécifiés en JML.
- BZ-Testing-tools, un animateur et générateur de test pour des spécifications écrites en B ou en Z.
- HaRVey, un outil pour décider de la validité de formules du premier ordre modulo certaines théories equationnelles utiles à la preuve de programmes.
- GénéSyst, une représentation de systèmes B événementiels par un système de transitions. Basé sur la BoB (Boîte à outils B) qui permet de manipuler les substitutions et d'exécuter des calculs de plus faible précondition.

L'objet du projet était de développer des synergies entre ces outils pour les adapter à la spécification de propriétés de sécurité, la preuve et le test de programmes Java.

2.1.1 Générateurs d'obligations de preuve

Jack et Krakatoa Les outils Jack et Krakatoa partagent le même objectif de prouver des programmes Java annotés par des spécifications JML. Jack conçu initialement par Gemplus est maintenant développé par l'INRIA. Il intervient dans le projet européen Moebius. Jack est né dans un contexte industriel alors que Krakatoa dispose depuis ses premières versions d'une diffusion libre.

Krakatoa a expérimenté au début du projet GECCOO la génération d'obligations de preuve pour le prouveur automatique simplify [12], avec des résultats très encourageants. Une traduction analogue a été également mise en place dans Jack. La plateforme Jack a été adaptée pour effectuer des preuves de programmes écrits en byte-code Java, tandis que l'expérience Krakatoa a incité au développement de l'outil Caduceus de preuve de programmes C [9].

L'expérimentation de Krakatoa sur l'étude de cas Demoney [6] a amené à des modifications de la partie génération d'obligations de preuve (en pratique dans l'outil Why qui sert de base à Krakatoa) ceci afin de faire face à des problèmes d'efficacité.

2.1.2 Démonstrateurs automatiques

haRVey L'utilisation de méthodes de preuve automatique est essentielle dans l'architecture des outils de spécification et preuve de programmes. Le démonstrateur simplify développé par Compaq dans le cadre de l'outil d'analyse de programmes ESC/Java donne de très bons résultats, mais on peut espérer faire mieux en terme d'efficacité et de traçabilité des preuves.

haRVey [8] est développé dans le projet CASSIS en collaboration avec David Déharbe (DIMAp/UFRN, Natal, Brazil). L'outil prend comme entrée une théorie T et des formules ϕ_i et vérifie que les formules ϕ_i sont vérifiées dans la théorie T. Les principaux développements réalisés durant la première partie du projet pour prendre en compte une classe plus large d'obligations de preuve sont :

- L'ajout d'une procédure de décision pour l'arithmétique linéaire, la version de haRVey correspondante est en cours de validation.

– Un processus de simplification des théories utilisées pour ne retenir que les axiomes pertinents par rapport au but à prouver et augmenter l'efficacité.

Les outils Krakatoa et Jack ont été adaptés pour fournir des obligations de preuve à haRVey. Des expérimentations sont en cours.

2.1.3 Analyse de spécifications

Le projet regroupe des équipes spécialistes de la méthode B, méthode qui a fait ses preuves en milieu industriel dans le domaine de la spécification et du développement de systèmes sûrs. L'adaptation des techniques et outils spécifiques à la méthode B dans le cadre du développement de programmes objets est fructueuse.

JML-Testing-Tools L'outil JML-Testing-Tools est réalisé à Besançon. Il utilise des techniques développées dans le cadre de BZ-Testing-tools au cas des spécifications JML.

Cet outil utilise un système de résolution de contraintes ensemblistes et permet d'animer les spécifications afin de valider le modèle. Il s'appuie sur un format intermédiaire basé sur une notation ensembliste inspirée de B pour décrire les comportements des spécifications JML et ainsi permettre leur animation symbolique à contraintes, tel que présenté dans [4, 5].

Génésyst Génésyst est un outil développé au-dessus de la boite à outils BoB. Il permet de représenter un système B évenementiel par un système de transitions étiqueté. La description peut correspondre à une spécification ou un raffinement. L'utilisateur caractérise dans la description B, l'ensemble des états qu'il souhaite voir apparaître. Le système GénéSyst produit alors des obligations de preuve permettant de calculer les conditions de franchissement des transitions en terme de déclenchabilité et d'atteignabilité.

Le fonctionnement de Génésyst est décrit dans [16, 14, 3].

Cet outil a été expérimenté sur l'application Demoney.

2.2 Étude de cas : Demoney

Demoney est une applet JavaCard développée par la société Trusted Logic dans le cadre du projet européen SecSafe. Cette applet implante un porte-monnaie électronique, elle a été développée à des fins d'expérimentation d'outils de recherche. C'est donc une version simplifiée par rapport aux applets industrielles mais néanmoins représentative des principales difficultés.

Nous avons choisi dans le projet d'étudier plus précisément cette application. Nous avons analysé le document de spécification. À Grenoble, un travail a été réalisé autour de la spécification de la spécification de Demoney qui a été modélisée en B en se concentrant sur les messages d'erreur et les différents états des transactions et des canaux de communication. À Orsay, une spécification JML du code Java a été réalisée [6] et prouvée correcte à l'aide de l'outil Krakatoa.

2.3 Propriétés de sécurité en JML

Nous avons effectué un inventaire des propriétés de sécurité, basé sur des documents fournis par des industriels de la carte à puce. Nous distinguons deux niveaux de sécurité : le niveau le plus haut qui décrit des propriétés générales et le niveau inférieur qui identifie des règles concrètes que l'implémentation doit respecter. Le lien entre ces deux niveaux relève de l'analyse

de sécurité, faite par une équipe en charge de cet aspect; une partie de l'audit consiste à vérifier les règles imposées.

Jusqu'à présent, cet audit de sécurité est fait manuellement, nous avons développé un outil qui permet d'automatiser certaines phases de cet audit.

Notre outil prend une règle de sécurité et la traduit dans des annotations JML. Puis, nous vérifions si l'application respecte les annotations JML, en utilisant des outils comme Jack et Krakatoa. Si les annotations sont acceptées, nous pouvons conclure que l'application est correcte vis à vis de la règle traitée. Ce travail est décrit dans [15].

2.3.1 Propriétés temporelles

Ce travail est réalisé par l'équipe TFC à Besançon et le projet Everest à Sophia-Antipolis. Un langage de logique temporelle pour JML a été proposé par Huisman et Trentelman [20]. Nous avons étendu ce travail en proposant des obligations de preuves traduisibles en JML standard qui assure la vérification des propriétés de vivacité exprimables dans ce langage. La seconde phase du travail s'intéresse à vérifier la préservation de cette propriété de vivacité lorsque la classe est utilisée par un système. Cette seconde phase met en oeuvre des techniques de raffinement.

Ce travail a abouti à la rédaction d'un rapport technique INRIA [2], ainsi que d'un article soumis à une conférence. Un outil s'appuyant sur les travaux de [2] et [20] est en cours de développement. Les collaborations futures s'attacheront principalement à l'intégration du générateur d'assertions en cours de développement au LIFC à l'intérieur de l'outil Jack développé à l'INRIA, ainsi qu'à l'approfondissement du travail sur la sémantique des traces de JML et la définition des obligations de preuve de raffinement pour JML.

2.4 Techniques de démonstration

Le travail sur les techniques de démonstration automatique s'oriente dans plusieurs directions.

- La preuve de programmes nécessite de combiner différentes théories. On sait que c'est un problème difficile en général et les résultats connus imposent des restrictions sur les différentes sous-théories. Le projet CASSIS [19, 11, 17, 7, 21] a obtenu des résultats pour relacher certaines de ces conditions. Il s'intéresse plus particulièrement à la combinaison de théories sur les listes, les ensembles et les théories sous-jacentes sur les éléments.
- Il est possible de trouver des théories décidables pour certaines classes de propriétés à vérifier sur les programmes. Une expérience intéressante a été menée pour montrer des propriétés d'accessibilité de certaines parties de la mémoire d'un programme de GC [18, 13].
- Les démonstrateurs automatiques sont des programmes complexes dont les capacités sont limitées. Il est important de pouvoir combiner leur utilisation avec des outils de démonstration interactive. Une technique est de revérifier a posteriori des traces de preuve trouvées automatiquement.

L'intégration de la nouvelle version de HaRVey aux outils de preuve devrait permettre une expérimentation plus poussée afin de dégager les théories utiles aux preuves associées au modèle mémoire.

3 Conclusion

3.1 Apports du projet

Le projet a permis de favoriser des échanges entre des outils de même nature (Jack et Krakatoa) ou complémentaires haRVey. Il a permis également de transposer et adapter des technologies de la méthode B vers le formalisme Java/JML qui est plus adapté aux chaînes de développement de logiciel dans l'industrie (JML-Testing-Tools). Il contribue à prendre en compte dans les outils des spécifications de sécurité de haut niveau qui peuvent être mécaniquement vérifiées.

Le projet s'organise autour de recherches amont sur le raffinement pour des langages d'ordre supérieur avec effets de bord, ou la combinaison de procédures de décision; mais aussi de développement d'outils et d'expérimentation sur des études de cas.

3.2 Changements par rapport au projet initial

Le projet a réalisé un effort important de convergence des formalismes sur lesquels se basent les méthodes et les outils. Les outils proposés permettent déjà de développer des expérimentations intéressantes. Cependant, l'objectif initial de construire un environnement unique allant de la spécification de haut niveau jusqu'à la simulation et le test ne sera sans doute pas atteint à la fin du projet. Un tel environnement réclamerait un effort d'ingénierie trop important par rapport aux ressources du projet. Il est pour l'instant plus efficace de se concentrer sur un petit nombre d'outils pouvant communiquer entre eux.

Références

[1] B. Aichernig and B. Beckert, editors. IEEE Press, 2005.

[2] F. Bellegarde, J. Groslambert, M. Huisman, O. Kouchnarenko, and J. Julliand. Verification of liveness properties with jml. Technical report, INRIA, 2004.

[3] Didier Bert, Marie-Laure Potet, and Nicolas Stouls. Génésyst: a tool to reason about behavioral aspects of B Event Specifications. application to security properties. In *Proceedings of the International Conference on Formal Specification and Development in Z and B (ZB'05)*, volume 3455 of *LNCS*, pages 299–318, Guildford, United Kingdom, April 2005. Springer-Verlag.

[4] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-Testing-Tools: a Symbolic Animator for JML Specifications using CLP. In *Proceedings of 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 551–556, Edinburgh, United Kingdom, April 2005. Springer-Verlag.

[5] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic Animation of JML Specifications. In *Proceedings of the International Conference on Formal Methods (FM'2005)*, volume 3582 of *Lecture Notes in Computer Science*, pages 75–90, Newcastle Upon Tyne, United Kingdom, July 2005. Springer-Verlag.

[6] Vikrant Chaudhary. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML : A case study. Technical report, IIT internship report, July 2004.

[7] Jean-François Couchot, Alain Giorgetti, Silvio Ranise, and Calogero G. Zarba. Verification of set-based specifications. Draft, 2005.

[8] D. Déharbe and S. Ranise. Light-Weight Theorem Proving for Debugging and Verifying Units of Code. In *Proc. of the International Conference on Software Engineering and Formal Methods (SEFM03)*, Brisbane, Australia, September 2003. IEEE Computer Society Press.

[9] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Sixth International Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 15–29, Seattle, WA, USA, November 2004. Springer-Verlag.

[10] R. Foccardi, editor. *Proceedings of CSFW'04*, Pacific Grove,USA, June 2004. IEEE Press.

[11] Pascal Fontaine, Silvio Ranise, and Calogero G. Zarba. Combining lists with non-stably infinite theories. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2005.

[12] Claude Marché and Christine Paulin-Mohring. Reasoning on Java programs with aliasing and frame conditions. In *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, LNCS, August 2005. à paraitre.

[13] F. Mehta and S. Ranise. Automated provers doing (higher-order) proof search: A case study in the verification of pointer programs. In *Proc. of the 2nd Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'04)*, 2004.

[14] Xavier Morselli, Marie-Laure Potet, and Nicolas Stouls. Génésyst : Génération d'un système de transitions étiquetées à partir d'une spécification B événementiel. In *AFADL'2004*, pages 317–320, June 2004.

[15] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A.A. El Kalam, editors, *CARDIS'04*. Kluwer, 2004. An earlier version appeared as INRIA Technical Report, nr. RR-5061.

[16] Marie-Laure Potet and Nicolas Stouls. Explicitation du contrôle de développement B événementiel. In *AFADL'2004*, pages 13–27, June 2004.

[17] Silvio Ranise, Christophe Ringeissen, and Calogero G. Zarba. Combining data structures with non-stably infinite theories. Draft, 2005.

[18] Silvio Ranise and Calogero G. Zarba. Superposition-based of pointer programs. Draft, 2005.

[19] Cesare Tinelli and Calogero G. Zarba. Combining decision procedures for sorted theories. In José Júlio Alferes and João Alexandre Leite, editors, *Logics in Artificial Intelligence*, volume 3229 of *Lecture Notes in Computer Science*, pages 641–653. Springer, 2004.

[20] Kerry Trentelman and Marieke Huisman. Extending JML specifications with temporal logic. In *Algebraic Methodology And Software Technology (AMAST '02)*, volume 2422 of *LNCS*, pages 334–348. Springer-Verlag, 2002.

[21] Calogero G. Zarba, Domenico Cantone, and Jacob T. Schwartz. A decision procedure for a sublanguage of set theory involving monotone, additive, and multiplicative functions, I. The: two-level case. *Journal of Automated Reasoning*, 2005. To appear.