Compilation certifiée (avec optimisations) en ML/Java/C de réseaux de Petri colorés

Frédéric Gava & Franck Pommereau & Julien Tesson
frederic.gava@univ-paris-est.fr
ou franck.pommereau@ibisc.fr ou
julien.tesson@lacl.fr

Laboratoire d'Algorithme, Complexité, Logique (LACL)
Université de Paris-Est Créteil (UPEC)
61 avenue du Général de Gaulle
94010 Créteil cedex
Équipe : Vérification et Spécification des Systèmes
Directeur du laboratoire : Régine Laleau (laleau@u-pec.fr)
ou

IBISC
University of Évry
Tour Évry 2, 523 place des terrasses de l'Agora
91000 Évry, France

Thématiques : NECO, Compilation, Why, Coq, Vérification, Model-checking, Compilation.

1 Problématique

Pour trouver des **bugs** dans des systèmes (agents communicants, protocoles de sécurité [1, 7], critiques, *etc.*), une technique largement reconnue et utilisée est le **model-checking explicite** [2]. Le principe est de se limiter à des scénarios fixes et de calculer toutes les traces d'exécution possibles depuis un **état initial** tout en vérifiant que ces traces restent valides suivant une **formule logique** donnée par l'utilisateur (dans la logique **temporelle** intrinsèque du model-checker). L'**espace des états** est donc cet ensemble de traces. Par exemple, dans le cas d'un protocole de sécurité, il est possible de trouver une propriété de sécurité violée (attaque) par un agent ou prouver qu'il n'existe pas une telle trace. Le model-checking comporte deux sérieux problèmes :

1. Calculer cet espace des états peut s'avérer être **trop long** en temps de calculs.
2. L'utilisation de différentes techniques/algorithmes de réduction peut **rendre faux les résulats obtenus**.

Une solution pour le premier problème est la **compilation** du modèle. En effet, pour passer d'un état à un autre, il faut une fonction de successeur. Cette fonction est dépendante du modèle : automate, réseau de Petri, *etc.* Généralement, une interpretation du modèle est effectuée. Mais pour obtenir un outil efficace, il est préférable d'obtenir du code exécutable (C, Java, OCaml, *etc.*) c'.à-d. la compilation du modèle. Cependant pour être sûr de vérifier le modèle compilé (et non autre chose provenant d'un bug du compilateur), il faut pouvoir garantir les résultats de la compilation, ce qui est le coeur du stage.

Le second problème n'en n'est pas un dans le cas de la recherche d'erreurs (les traces entraînant une violation de la formule logique) car les traces peuvent être facilement et rapidement vérifiées par un outil externe (en Coq p. ex.). On parle alors **certificat**. Mais quand il n'y a pas une erreur de détectée, un model-checker ne produit que la réponse laconique "YES". On peut alors se poser la question de la validité d'une telle réponse. Et dans le cadre d'un model-checking explicite, la taille du certificat peut être proportionnelle à la taille de l'espace des états, c'-à-d. trop grande en pratique. Les model-checkers sont bien évidement des programmes non triviaux. Et il est légitime de se poser la question de leur **correction**/validité. Un model-checker bugué (comme ils le sont souvent

[8]) peut trouver des faux-négatifs ou pire rater une trace d'erreur (ce qui peut engendrer une perte de temps quand on recherche ensuite à prouver le système quelque soit le scénario).

On trouve très peu de travaux sur la **vérification** des outils de model-checking où seul la vitesse d'exécution semble compter. La principale raison est que ces outils servent essentiellement à trouver des erreurs et non pas à dire que le système est correct. Ainsi, de fausses erreurs peuvent être trouvées. Néanmoins, rien ne dit que les model-checker ne manquent pas des erreurs et il n'est pas rare de voir un modèle estampillé "formellement prouvé" même si on peut douter de la véracité de la réponse [9]. A notre connaissance, le premier développement (machine-checked avec l'assistant de preuves **Coq**, `http://coq.inria.fr/`) d'un model-checker a été fait dans [10]. On trouve par la suite, des développements de logiques temporelles en Coq dans [3] et [12]. On trouve aussi l'utilisation de model-checker pour la vérification des traces (les erreurs) et de la préservation de certaines propriétés (mais pas du résultat final) [11]. Des algorithmes séquentiels prouvés ont aussi été écrit en B dans [13]. [14] présente le développement d'un algorithme de vérification LTL en Maude (réécriture). Dans [4], les auteurs ont développé un model-checker séquentiel pour LTL dans le prouveur Isabelle avec une fonction de successeur de Maude certifiée en Isabelle (mais on peut douter de la certification de la fonction car l'auteur a utilisé sa propre sémantique de Maude qui, d'après l'auteur, est assez ambigue). Dans tout ces cas, ce sont des algorithmes purement fonctionnelles qui sont vérifiés et donc pas forcement efficace. Dans [6], nous avons travaillé sur la correction d'algorithmes séquentiels et distribués du calcul de l'espace des états (calculer le graphe mais pas les composantes fortement connexes).

Le **stage** a pour objectif de **générer des codes OCaml puis Java** ou C qui représentent l'exécution d'un réseau de Petri (les **marquages successeurs** d'un marquage). Principalement, on trouve des **boucles imbriquées** (une par place d'une **transition** d'un réseau de Petri) qui permettent de calculer tous les marquages (états successeurs) possibles d'une transition d'un réseau de Petri. Certaines boucles peuvent être **optimisées** en se basant sur la structure du réseau. Par exemple, si une transition ne prend que des jetons de type bool, alors il est inutile d'itérer sur le contenu de ces jetons, un simple test conditionel est suffisant. Le code généré est alors plus **efficace** car un "if-then-else" est toujours plus rapide à exécuter qu'une boucle (même triviale).

L'année dernière, un premier stagiaire avait travaillé sur les **invariants** de boucles en WHY nécessaires pour prouver la compilation de réseaux de Petri simples et les codes de compilation avait été donnés **manuellement**. Quelques optimisations avait été étudiées. Mais, le stage n'a pas donné lieu à un véritable compilateur (seul des exemples mains ont été traités) pour WHY et encore moins pour du code Java ou C. Néanmoins, la méthodologie a été partiellement étudiée, ce qui simplifiera le travail pour obtenir un véritable compilateur certifié de réseaux de Petri.

Dernier point important est l'idée d'une sortie (du compilateur) **multi-langage**. En effet, si nous avons un model-checker prouvé correct correct en Coq, alors l'extraction nous donnera du code OCaml. Mais, si nous voulons un model-checker plus efficace, il nous faudra vérifier soit du code Java soit directement du code C. Alors, il nous faudra une fonction de successeur en Java ou en C (les outils faisant les connexions entre ces différents langages ne sont pas dignes de confiance). Et si à l'avenir, un autre langage doit être utisé (du C#, du Scala, du C++, du ...), alors nous pourrons adapter facilement notre compilateur certifié pour le **brancher** sur le langage utilisé pour implémenter le model-checker.

2 Méthodes utilisées

2.1 Pour la preuve

Pour commencer, nous utiliserons l'outil **WHY** [1] ; Outil qui traite des pre-post conditions et des invariants de boucles dans un mini langage ML avec effets de bords ; Cet outil permet donc **la génération d'assertions de preuves** à partir **des annotations du code**. Si toutes les obligations sont prouvées par les théorèmes prouveurs (automatiques ou non) alors la post-condition finale est correcte. L'outil WHY génère les **assertions** dans différents prouveurs comme des **théorèmes prouveurs automatiques** (p. ex. Z3 [2]) et à des assistants de preuves comme Coq.

Ensuite, pour du code Java (a priori de préférence pour une première approche lors du stage) ou du code C (si possible et si assez de temps), nous utiliserons les outils connexes à WHY qui sont **krakatoa** (`http://krakatoa.lri.fr/`) et Frama-C (`http://www.frama-c.com/`) et qui permettent respectivement de vérifier du code Java et du code C avec les mêmes techniques que WHY (annotations des codes).

1. `http://why.lri.fr/index.fr.html`
2. `http://research.microsoft.com/projects/z3/`

2.2 Pour les réseaux de Petri

Pour ce qui est des réseaux de Petri, nous utiliserons l'outil **SNAKES** (`https://www.ibisc.univ-evry.fr/~fpommereau/SNAKES/index.html`) et qui permet la manipulation des réseaux en Python.

Pour la compilation des réseaux, nous nous aiderons de **NECO** [5] (`https://code.google.com/p/neco-net-compiler/`). L'auteur peut fournir une version (beta) modifiée de NECO et qui génère du code Java.

3 But et déroulement envisagé du stage

Le but de ce stage est donc de **prouver formellement la compilation** de réseau de Petri en du code exécutable en OCaml puis Java. Ainsi, nous aurons une fonction de successeur prouvée (et efficace) qui pourra être utilisée dans un model-checker (à l'avenir, lui aussi certifié, si possible). Il faudra donc écrire des premiers codes en Why-ML (le langage d'entré de WHY) pour comprendre la méthodologie. Puis reprendre ce travail pour du Java et enfin implémenter le compilateur lui-même (le plus gros travail) et le tester. Il faudra alors générer les bons invariants de boucles et les faire vérifier par les théorèmes prouveurs.
Le stage (5 à 6 mois) sera organisé de la manière suivante :

1. Premier mois :
   (a) Découverte rapide de la programmation Why-ML, de l'outil Why/Krakato
   (b) connaître la compilation de réseaux de Petri
   (c) Apprentissage de Coq (si besoin)
   (d) Étude du précèdent stage
2. deuxième mois : Formalisation des algorithmes séquentiels en WHY-ML et en Java (krakatoa)
3. troisième/quatrième : implantation du compilateur
4. cinquième mois : étude et ajout d'optimisations
5. dernier 15 jours : Rédaction du mémoire et préparation de la soutenance.

Des connaissances sur les prouveurs formels (automatiques ou assistés) et les méthodes formelles seraient les vraiment bienvenues mais ne sont pas obligatoires (un goût pour le formalisme et la logique est, en revanche, fortement conseillé). Une publication de ces résultats dans une conférence internationale est envisagée. Possibilité de poursuivre par une thèse dans l'équipe "Vérification et Spécification" au LACL ou à l'IBISC si une une allocation doctorale est obtenue. Comme pour tout stage de M2, une **indemnité mensuelle** correspondant au 1/3 du SMIC (environ 500 euros) sera accordée.

Références

[1] David Basin, Cas Cremers, and Catherine Meadows. *Model Checking Security Protocols*, chapter 24. Springer, 2011.

[2] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[3] S. Coupet-Grimal. An Axiomatization of Linear Temporal Logic in the Calculus of Inductive Constructions. *Logic and Computation*, 13(6) :801–813, 2003.

[4] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable ltl model checker. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.

[5] Lukasz Fronc. *Compilation de réseaux de Petri : modèles haut niveau et symétries de processus*. PhD thesis, 2013.

[6] Frédéric Gava, Jean Fortin, and Michaël Guedj. Deductive verification of state-space algorithms. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods (IFM)*, volume 7940 of *LNCS*, pages 124–138. Springer, 2013.

[7] Frédéric Gava, Michaël Guedj, and Franck Pommereau. A bsp algorithm for on-the-fly checking ctl* formulas on security protocols. In Hong Shen, Yingpeng Sang, Yidong Li, Depei Qian, and Albert Y. Zomaya, editors, *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 79–84. IEEE, 2012.

[8] Fabrice Kordon, Alban Linard, Didier Buchs, Maximilien Colange, Sami Evangelista, Lukasz Fronc, Lom-Messan Hillah, Niels Lohmann, Emmanuel Paviot-Adet, Franck Pommereau, Christian Rohr, Yann Thierry-Mieg, Harro Wimmel, and Karsten Wolf. Raw report on the model checking contest at petri nets. *CoRR*, abs/1209.2382, 2012.

[9] Kedar S. Namjoshi. Certifying Model Checkers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 2–13. Springer, 2001.

[10] Christopher Sprenger. A Verified Model Checker for the Modal μ-calculus in Coq. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1384 of *LNCS*, pages 167–183. Springer-Verlag, 1998.

[11] Jun Sun, Yang Liu, and Bin Cheng. Model Checking a Model Checker : A Code Contract Combined Approach. In Jin Song Dong and Huibiao Zhu, editors, *Formal Engineering Methods (ICFEM)*, volume 6447 of *LNCS*, pages 518–533. Springer, 2010.

[12] Ming-Hsien Tsai and Bow-Yaw Wang. Formalization of CTL* in the Calculus of Inductive Constructions. In Mitsu Okada and Ichiro Satoh, editors, *Asian computing science conference on Advances in computer science : secure software and related issues (ASIAN)*, volume 4435 of *LNCS*, pages 316–330. Springer-Verlag, 2007.

[13] E. Turner, M. Butler, and M. Leuschel. A Refinement-based Correctness Proof of Symmetry Reduced Model-checking. In *Abstract State Machines, Alloy, B and Z*, LNCS, pages 231–244. Springer, 2010.

[14] Bow-Yaw Wang. Automatic verification of a model checker by reflection. In Pascal Van Hentenryck, editor, *Practical Aspects of Declarative Languages (PADL)*, volume 3819 of *LNCS*, pages 45–59. Springer, 2006.