Vérification d'un composant java: le vérificateur de bytecode

Julien Charles

EVEREST INRIA

Introduction

- But: Vérifier un composant de système embarqué en utilisant Java Modelling Language (JML)
- Un vérificateur de bytecode (bcv) pour Java:
  - On annote le bcv en JML.
  - On effectue les preuves à l'aide d'un prouveur interactif; le prouveur automatique ne suffisant probablement pas.

Plan

1. Le vérificateur de bytecode Java
2. Un plugin pour Jack
3. Premiers résultats
4. Conclusion

Le vérificateur Java

1. Principe
2. Propriétés à démontrer
3. Limitations
4. Architecture
5. Mise en oeuvre

Principe

- On implémente l'algorithme défini dans *The Java Language Specification* 2nde édition
- C'est un algorithme de Kildall:
  - En itérant sur les instructions,
    - On exécute l'instruction sur son état
    - On fusionne l'état obtenu avec les états de ses successeurs
    - Quand les états ne changent plus l'algorithme se termine

Propriétés à démontrer

- Propriétés simples (démontrable automatiquement):
  - Les renvois d'exceptions
  - Les débordements arithmétiques
- Propriétés non-triviales (en général démontrable seulement interactivement):
  - Correction
  - Complétude

Limitations

- Travail sur un sous-ensemble de JML mais problèmes:
  - Ordre partiel sur les états
    - On ne peut pas avoir de prédicats à spécification récursive
  - Prise en compte des structures récursives:
    - Terminaison de la boucle dure à exprimer en JML
- L'algorithme de Kildall est souvent exprimé avec un design pattern visiteur (Bcel, Ovm)
  - On en implémente un à base de tableaux

Architecture

- On implémente un bcv minimal auquel on rajoutera des fonctionnalités au fur et à mesure:
  - On utilise des classes abstraites
    - Une classe pour représenter les états
    - Une classe pour représenter les instructions
    - Une classe pour représenter le vérificateur
  - On raffinera la classe état et la classe instruction pour correspondre à Java, la classe vérificateur (contenant l'algorithme de Kildall) ne devrait plus être modifiée

Mise en oeuvre

- Après l'implémentation du bcv et ses annotations à peu près terminées, j'ai essayé de générer des obligations de preuves avec Esc/Java 2; Krakatoa et Jack:
  - Esc/Java 2: prouveur automatique (Simplify)
  - Krakatoa: ne fonctionne pas avec les classes abstraites, plutôt orienté preuve interactive
  - Jack: classes abstraites, approche principalement automatique.

Un plugin pour Jack

1. Jack
2. Preuves
3. Un plugin Coq
4. Automatisations

Jack (1)

- Il a une approche mixte avec plugins
- Il est intégré dans Eclipse; ce qui facilite son utilisation
- Il ne gère pas toutes les constructions JML mais les principales

Jack (2)

- Plugins:
  - AtelierB: Jack a été conçu au départ pour être utilisé avec AtelierB
  - PVS: permet de résoudre de manière interactive
  - Simplify: permet de résoudre de manière automatique
  - haRVey: Le plugin est encore en beta
  - Coq ???

Preuves

- Les preuves générées avec Jack ont une forme semblables à celles générées avec Krakatoa
- Une obligation de preuve Jack se compose de 3 parties:
  - Un prélude contenant des définitions générales ainsi que des définitions spécifiques à la classe traitée
  - Une série d'hypothèses
  - Un but à prouver

Un plugin Coq

- Nous avons retranscrit cette structure dans les obligations de preuves générées par le plugin Coq:
  - un prélude statique d'un côté
  - une section contenant les variables et hypothèses et un lemme contenant le but

Automatisations

- Pourquoi?
  - Le nombre d'obligations de preuves
  - Proof Carrying Code
- Simplify vs. Coq
- On aimerait bien prouver autant automatiquement que Simplify, même si c'est plus lent

Mise en oeuvre

- On utilise coqtop comme prouveur automatique:
  - une seule instance
  - on utilise les tactiques sur un mode essai/erreur
  - chaque série de lemmes à prouver correspond à une section

Les tactiques (1)

- Certaines obligations de preuves avec des formes analogues revenaient souvent
  - Nous avons utilisé le langage Ltac pour les résoudre
- Dans le prélude: un nouveau fichier, localTactics.v:
  - elimAnd
  - absurdJack
  - overriding
  - tryApply

Les tactiques (2)

- Les tactiques plus générales
  - startJack
  - autoJack
- Une tactique servant principalement à la preuve automatique:
  - startJackAuto: startJack; do 2 (tryApply; autoJack; intuition autoJack).

Problème

- Avec startJackAuto, sur certaines preuves coqtop bouclent
- On a résolu en faisant 2 modes automatiques pour Coq:
  - un mode léger utilisant uniquement startJack
  - un mode fort utilisant startJackAuto
  - en rajoutant un temps de grâce

Premiers résultats

- Premiers résultats encourageant:
  - Avec les tactiques de base on arrive un peu en dessous de Simplify (sur un ex. Simplify faisait 60%, et Coq fait 57%)
  - Avec les tactiques plus performantes Coq est meilleur que Simplify (sur le même ex., Coq fait alors 70%)
- Maintenant dans Jack on peut faire des preuves automatiques et des preuves interactives uniquement avec Coq!

Retour au vérificateur

- Pour l'exemple cité du vérificateur abstrait annoté (partiellement), j'arrive aux résultats suivants pour le moment:
  - La classe Etat est prouvée à 100% automatiquement
  - La classe Instruction est prouvée à 70%
  - La classe VerifierException est prouvée à 100%
  - La classe Verifier est prouvée à 50%
- Ce qui reste est la partie réellement non automatique

Conclusion

- J'ai commencé à annoter un bcv en JML
- Grâce à Jack et au plugin Coq, je peux résoudre la plupart des obligations de preuves rapidement pour me consacrer à celles réellement importantes
- Un problème subsiste: les prédicats à spécification récursive dans Jack; il faudrait ajouter une nouvelle construction permettant d'accéder à des prédicats définis en Coq