université
PARIS-SACLAY

UE14 · Vérification déductive de programmes

Conclusion & exercices additionels

Master Informatique · FIIL · 2017 / 2018

1. Conclusion

Que peut signifier un échec de preuve et que faire

- le code est faux
  - le corriger :-)
- la spécification est fausse
  - la corriger :-)
- il manque des annotations (invariant de boucle, etc)
  - vérifier vos `requires`, `assigns`, `loop invariants`, ...
- le prouveur n'est pas assez puissant
  - augmenter le *timeout*
  - essayer un autre prouveur
  - ajouter des `assert`, `lemma`, ...
  - modifier la façon dont les annotations sont écrites
  - finir la preuve par d'autres moyens (preuve interactive, test, revue de code manuelle, ...)

Conclusion

- spécifier formellement :
  - améliore la compréhension d'un programme
  - évite des ambiguïtés
  - permet de valider plus de propriétés (vérification à l'exécution)

Conclusion

- spécifier formellement :
  - améliore la compréhension d'un programme
  - évite des ambiguïtés
  - permet de valider plus de propriétés (vérification à l'exécution)
- prouver formellement :
  - requiert l'écriture d'annotations supplémentaires
  - activité coûteuse
  - mais économiquement viable dans des contextes fortement contraints (aéronautique, nucléaire)
  - attention aux hypothèses implicites des outils
  - activité ludique (par rapport au test)

Conclusion

- spécifier formellement :
  - améliore la compréhension d'un programme
  - évite des ambiguïtés
  - permet de valider plus de propriétés (vérification à l'exécution)
- prouver formellement :
  - requiert l'écriture d'annotations supplémentaires
  - activité coûteuse
  - mais économiquement viable dans des contextes fortement contraints (aéronautique, nucléaire)
  - attention aux hypothèses implicites des outils
  - activité ludique (par rapport au test)
- combiner avec d'autres techniques :
  - avec interprétation abstraite (absence d'erreurs à l'exécution)
  - avec du test/monitoring
  - permet de réduire les coûts de l'activité de vérification

Conclusion

- spécifier formellement :
  - améliore la compréhension d'un programme
  - évite des ambiguïtés
  - permet de valider plus de propriétés (vérification à l'exécution)
- prouver formellement :
  - requiert l'écriture d'annotations supplémentaires
  - activité coûteuse
  - mais économiquement viable dans des contextes fortement contraints (aéronautique, nucléaire)
  - attention aux hypothèses implicites des outils
  - activité ludique (par rapport au test)
- combiner avec d'autres techniques :
  - avec interprétation abstraite (absence d'erreurs à l'exécution)
  - avec du test/monitoring
  - permet de réduire les coûts de l'activité de vérification

que vérifie-t-on (vraiment) au final ?

Conclusion (2)

Même si on prouve un code correct par rapport à sa spécification,
sa correction n'est jamais garantie à 100%...

Les risques proviennent notamment de :

- même erreur dans le code et la spécification
- non respect des hypothèses implicites des analyseurs
- bugs des analyseurs (analyseur certifié : Verasco)
- bugs du compilateur (compilateur certifié : CompCert)
- bugs matériel
- contraintes physiques mal modélisées
- non respect des contraintes d'utilisation par l'opérateur
- ...

L'activité de preuve doit s'inscrire dans un processus de développement et de validation rigoureux pour réduire les risques

2. Exercices additionels

Recherche dichotomique

1. Spécifier et prouver la fonction suivante (fichier `binary_search.c`) :

```
/* takes as input a sorted array a, its length,
   and a value key to search,
   returns the index of a cell which contains key,
   returns -1 iff key is not present in the array
*/
int binary_search(int* a, int length, int key) {
  int low = 0, high = length - 1;
  while (low<=high) {
    int mid = (low + high) / 2;
    if (a[mid] == key) return mid;
    if (a[mid] < key) { low = mid+1; }
    else { high = mid - 1; }
  }
  return -1;
}
```

2. Qu'en est-il de l'absence d'erreur à l'exécution ?

Différence

Spécifier et prouver la fonction suivante (fichier `mismatch.c`) :

```
/* Retourne un indice auquel les tableaux [a] et [b]
   ne contiennent pas la même valeur.
   Si les deux tableaux sont égaux, retourne [len]. */
int mismatch(int *a, int *b, int len) {
  for(int i = 0; i < len; i++)
    if (a[i] != b[i])
      return  i;
  return len;
}
```

Maximum d'un tableau

Spécifier et prouver la fonction suivante (fichier `max_seq.c`) :

```
int max_elt(int *t, int len) {
  int max = 0;
  if (len == 0) return 0;
  for(int i = 0; i < len; i++)
    if (a[max] < a[i])
      max = i;
  return max;
}

/* Retourne la valeur maximale contenue dans
   le tableau [t] de longueur [len].
   Retourne 0 si le tableau est vide. */
int max_seq(int *t, int len) {
  return t[max_elt(t, len)];
}
```

Copie ou remplace

Spécifier et prouver la fonction suivante (fichier `replace_copy.c`) :

```
/* copie les [len] premières cases du tableau [src]
   dans [dst] en remplaçant les valeurs [old] par [new]. */
int replace_copy
  (int *src, int *dst, int len, int old, int new)
{
  for(int i = 0; i < len; i++)
    dst[i] = (src[i] == old ? new : src[i]);
}
```