Fiabilité Logicielle

Méthodes formelles (preuves de programmes)

Arnaud Labourel

2025-26

amU Faculté des sciences Aix Marseille Université

Section 1

preuves de programmes

Preuve de programmes - vérification déductive

Idée :

- raisonner par déduction sur le programme de manière automatique
- prend en compte les spécifications formelles (notamment comme objectif des preuves)
- extraire de la spécification et du programme les propriétés à prouver, des **Obligations de Preuves** (OP)
- utilisation de prouveurs automatiques et/ou semi-automatiques afin de prouver ces propriétés

But : si toutes les OP sont toutes vérifiées, une méthode déductive garantit la correction du programme vis-à-vis de sa spécification.

Triplets de Hoare

Un **triplet de Hoare** est un triplet noté $\{P\}i\{Q\}$ où P et Q sont des propriétés (formules logiques) et i est une instruction du programme.

- P est appelé la **pré-condition** de i
- Q est appelé la **post-condition** de i

Un triplet de Hoare est valide si (sémantique opérationnelle de i) :

partant de n'importe quel état mémoire initial vérifiant la pré-condition P, l'instruction i termine et l'état mémoire final vérifie la post-condition Q.

Sémantique opérationnelle d'un langage de programmation infaisable en pratique.

$\Rightarrow$ on recourt à un système de déduction formel, la logique de (Floyd-)Hoare, qui permet de prouver la validité des triplets de Hoare à partir de règles d'inférence.

Exemples

Les triplets de Hoare suivants sont-ils valides ?

- $\{x = 10\}x = x + 1\{x = 11\}$
- $\{c = 0\}$if (c != 0) then x = c else x = $1\{x = 0 \wedge c = 0\}$
- $\{c = 2\}$if (c != 0) then x = c else x = $1\{x = 2 \wedge c = 2\}$
- $\{faux\}x = 0; y = 1\{x = 0 \wedge y = 2\}$
- $\{i = 1\}$while (i <= 10) do i = i + $1\{i = 10\}$
- $\{i = 1\}$while (! (i <= 0)) do i = i - $1\{i = 0\}$

Exemples : solutions

Les triplets de Hoare suivants sont-ils valides ?

- $\{x = 10\} x = x + 1 \{x = 11\}$ $\oplus$
- $\{c = 0\}$ if (c != 0) then x = c else x = 1 $\{x = 0 \wedge c = 0\}$ $\ominus$
- $\{c = 2\}$ if (c != 0) then x = c else x = 1 $\{x = 2 \wedge c = 2\}$ $\oplus$
- $\{faux\} x = 0; y = 1 \{x = 0 \wedge y = 2\}$ $\oplus$
- $\{i = 1\}$ while (i <= 10) do i = i + 1 $\{i = 10\}$ $\ominus$
- $\{i = 1\}$ while (! (i <= 0)) do i = i - 1 $\{i = 0\}$ $\oplus$

Comment prouver les triplets de Hoare ?

Comment savoir si un triplet de Hoare est valide ? On utilise un système de déduction formel.

Une règle est constituée :

- d'hypothèses ou prémisses $H_1, \dots, H_n$
- d'une conclusion C

$$\frac{H_1, \dots, H_n}{C}$$

Exemple : Conjonction en déduction naturelle

$$\frac{A \ , \ B}{A \vee B}$$

Système de déduction = ensemble fini, fixé de règles

Preuve dans un système formel

Démontrer A dans le système choisi, c'est exhiber un arbre complet dont la conclusion est A.

Un arbre de preuve est formé de plusieurs règles, les conclusions des unes formant les prémisses des autres :

$$\dfrac{\dfrac{}{H_1} \quad \cdots \quad \dfrac{}{H_n}}{A}$$ prouve A sous les hypothèses $H_1, \ldots, H_n$

- Un axiome est une règle sans prémisse.
- Un arbre complet est un arbre de preuve dont toutes les feuilles sont des axiomes.

On va utiliser un système de règles de déduction : logique de (Floyd-)Hoare

Axiome de l'affectation : logique de (Floyd-)Hoare

La propriété $P[x \leftarrow e]$ signifie la propriété P dans laquelle la variable x a été substituée par l'expression e.

Exemple : $P = (x = 3)$ alors $P[x \leftarrow y + 2] = (y + 2 = 3)$

Instruction $x := E$, associant à la variable x la valeur de l'expression E. Règle :

$$\overline{\{P[x \leftarrow e]\} x := e \{P\}}$$

Exemples :

- $\{x + 1 = 43\}\ y := x + 1\ \{y = 43\}$
- $\{x + 1 \leq N\}\ x := x + 1\ \{x \leq N\}$
- $\{y = 3\}\ x := 2\ \{y = 3\}$ valide si x et y ne sont pas des alias (variable $\neq$)

Règle de composition : logique de (Floyd-)Hoare

Programmes S et T exécutés séquentiellement donnant le programme $S;T$.

$$\frac{\{P\}\ S\ \{Q\}\ ,\ \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}$$

Soient les deux triplets de Hoare suivants :

- $\{x+1=43\}\ y := x+1\ \{y=43\}$
- $\{y=43\}\ z := y\ \{z=43\}$

La règle de composition nous permet de conclure :

$$\frac{\{x+1=43\}\ y := x+1\ \{y=43\}\ ,\{y=43\}\ z := y\ \{z=43\}}{\{x+1=43\}\ y := x+1; z := y\ \{z=43\}}$$

Règle de conditionnelle : logique de (Floyd-)Hoare

Combiner deux programmes **S** et **T** dans un bloc **si C alors S sinon T**, lorsque les conditions le permettent.

$$\frac{\{P \wedge B\}\ S\ \{Q\}\ ,\ \{P \wedge \neg B\}\ T\ \{Q\}}{\{P\}\ \textbf{si}\ B\ \textbf{alors}\ S\ \textbf{sinon}\ T\ \textbf{fin si}\ \{Q\}}$$

Soient les deux triplets de Hoare suivants :

- $\{true \wedge y = 0\}\ x := y\ \{x = 0\}$
- $\{true \wedge \neg(y = 0)\}\ x := 0\ \{x = 0\}$

$$\frac{\{true \wedge y = 0\}\ x := y\ \{x = 0\}\ ,\ \{true \wedge \neg(y = 0)\}\ x := 0\ \{x = 0\}}{\{true\}\ \textbf{si}\ y = 0\ \textbf{alors}\ x := y\ \textbf{sinon}\ x := 0\ \textbf{fin si}\ \{x = 0\}}$$

Règle de la conséquence (ou de l'affaiblissement)

Souvent, les règles d'inférence peuvent syntaxiquement différentes mais équivalentes ou conséquences des propriétés déduites.

Exemple : $x = 3$ et $x + 1 = 4$ syntaxiquement $\neq$ et seulement équivalentes.

La règle de la conséquence permet ainsi d'affaiblir les pré-conditions et post-conditions en les remplaçant par des conséquences logiques.

$$\frac{P \rightarrow P' \ , \ \{P'\}\ S\ \{Q'\} \ , \ Q' \rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Ces implications sont à prouver :

- à la main
- par d'autres règles d'inférence (déduction naturelle)

Règle d'itération : logique de (Floyd-)Hoare

Combiner un programme **S** dans une boucle **tant que C faire S**, lorsque les conditions le permettent.

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \textbf{tant que}\ B\ \textbf{faire}\ S\ \textbf{fait}\ \{\neg B \wedge I\}}$$

où I est l'invariant. On le qualifie alors d'invariant de boucle.

À noter que cette règle là n'assure pas la terminaison de la boucle, mais assure seulement que la post-condition soit vérifiée une fois la boucle terminée (si elle termine).

Logique de (Floyd-)Hoare : résumé des règles/axiomes

$$\frac{}{\{P\}skip\{P\}} \qquad \frac{\{P\}i_1\{R\} \quad \{R\}i_2\{Q\}}{\{P\}i_1; i_2\{Q\}}$$

$$\frac{}{\{P[x \leftarrow e]\}x := e\{P\}}$$

$$\frac{\{P \wedge e\}i_1\{Q\} \quad \{P \wedge \neg e\}i_2\{Q\}}{\{P\}if\ e\ then\ i_1\ else\ i_2\{Q\}}$$

$$\frac{\{I \wedge e\}i\{I\}}{\{I\}while\ e\ do\ i\{I \wedge \neg e\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}i\{Q'\} \quad Q' \Rightarrow Q}{\{P\}i\{Q\}}$$

Logique de (Floyd-)Hoare : exemple 1

$$\dfrac{\dfrac{}{\{\}\texttt{c} := 0; \{0 \leq c \leq 101\}}\,A \qquad \dfrac{}{\{0 \leq c \leq 101\}\texttt{while ... do ... done}\{c = 101\}}\,B}{\{\}\texttt{c} := 0; \texttt{while c <= 100 do c} := \texttt{c} + 1 \texttt{ done}\{c = 101\}}$$

$$\dfrac{\texttt{true} \implies 0 = 0 \qquad \dfrac{}{\{0 = 0\}\texttt{c} := 0\{c = 0\}} \qquad c = 0 \implies 0 \leq c \leq 101}{A}$$

$$\dfrac{\dfrac{\dfrac{}{\{0 \leq c + 1 \leq 101\}\texttt{c} := \texttt{c} + 1\{0 \leq c \leq 101\}} \qquad 0 \leq c \leq 101 \wedge \texttt{c <= 100} \neq 0 \implies 0 \leq c + 1 \leq 101}{\{0 \leq c \leq 101 \wedge \texttt{c <= 100} \neq 0\}\texttt{c} := \texttt{c} + 1\{0 \leq c \leq 101\}} \qquad 0 \leq c \leq 101 \implies \texttt{true}}{\{0 \leq c \leq 101\}\texttt{while ... do ... done}\{0 \leq c \leq 101 \wedge \texttt{c <= 100} = 0\}}}{B}$$

Logique de (Floyd-)Hoare : exemple 2

$$\dfrac{\top \Rightarrow 0 = 0 \wedge 1 = 1 \qquad \dfrac{\dfrac{}{\{0 = 0 \wedge 1 = 1\}\, x := 0\, \{x = 0 \wedge 1 = 1\}} \qquad \dfrac{}{\{x = 0 \wedge 1 = 1\}\, y := 1\, \{x = 0 \wedge y = 1\}}}{\{0 = 0 \wedge 1 = 1\}\, x := 0; y := 1\, \{x = 0 \wedge y = 1\}}}{\{\top\}\, x := 0; y := 1\, \{x = 0 \wedge y = 1\}}$$

Notation alternative avec propriétés intercalées entre les instructions

$$\begin{array}{l} \{\top\} \Rightarrow \\ \{0 = 0 \wedge 1 = 1\} \\ x := 0; \\ \{x = 0 \wedge 1 = 1\} \\ y := 1 \\ \{x = 0 \wedge y = 1\} \end{array}$$

Logique de (Floyd-)Hoare : exemple 3

$$\{0 \leq a\} \Rightarrow \{a = b \cdot 0 + a \ \wedge \ 0 \leq a\}$$

`r := a;`

$$\{a = b \cdot 0 + r \ \wedge \ 0 \leq r\}$$

`q := 0;`

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r\}$$

`while r ≥ b do`

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r \ \wedge \ r \geq b\} \Rightarrow$$
$$\{a = b \cdot (q + 1) + (r - b) \ \wedge \ 0 \leq r - b\}$$

`    r := r − b;`

$$\{a = b \cdot (q + 1) + r \ \wedge \ 0 \leq r\}$$

`    q := q + 1`

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r\}$$

`done`

$$\{a = b \cdot q + r \ \wedge \ 0 \leq r \ \wedge \ r < b\} \Rightarrow$$
$$\{q = a/b \ \wedge \ r = a \bmod b\}$$

Logique de (Floyd-)Hoare : discussion

La logique de Hoare ne donne pas directement un algorithme permettant de calculer les triplets de Hoare à prouver.

Il y a de nombreuses questions à résoudre pour pouvoir calculer les triplets de Hoare à prouver :

- séquence : comment établir le prédicat R intermédiaire ?
- boucle : comment établir l'invariant de boucle I ?
- règle d'affaiblissement : quand et comment l'appliquer ?
- erreurs à l'exécution (évaluation des expressions e) : comment sont-elles prises en compte ?
- affectation : semble fonctionner en arrière (peu intuitif) mais base du calcul de l'algorithme de la plus faible précondition

Plus faible pré-condition - weakest precondition

Objectif

calculer de manière algorithme un triplet de Hoare $\{Pre\}p\{Post\}$ pour le programme p.

- Les instructions sont vues comme des transformateurs de prédicats (fonctions des propriétés dans les propriétés)
- On part de la post-condition $Post$ à la fin du programme et on raisonne en effectuant les étapes de calcul "à l'envers".
- à chaque étape de calcul, on connaît la post-condition Q à vérifier et l'instruction i concernée et on déduit la propriété P minimale (la plus faible) telle que $\{P\}i\{Q\}$ soit un triplet de Hoare valide
- Lorsque le début du programme est atteint, on doit prouver $Pre \Rightarrow P$.

Plus faible pré-condition - weakest precondition

La plus faible précondition (weakest precondition) d'une construction i avec postcondition Q est une assertion $WP(i, Q)$ telle que :

- c'est une précondition : $\{WP(i, Q)\}i\{Q\}$
- c'est la plus faible : si $\{P\}i\{Q\}$ alors $P \Rightarrow WP(i, Q)$.

Par conséquent : $\{P\}i\{Q\}$ si et seulement si $P \Rightarrow WP(i, Q)$

Intuition : les hypothèses nécessaires pour que l'instruction i calcule bien le résultat décrit par la postcondition Q.

Intuition originale (Dijkstra, 1975) : synthétiser le code i par raffinement à partir de sa postcondition Q.

Calcul de la plus faible pré-condition (WP)

$WP : Programme \times$ Propriété $\mapsto$ Propriété.

- **Règle *skip*** : $WP(skip, Q) = Q$
- **Règle composition** : $WP((i_1; i_2), Q) = WP(i_1, WP(i_2, Q))$
- **Règle affectation** : $WP(x := e, Q) = Q[x \leftarrow e]$ si e est évaluée sans erreur
- **Règle conditionnelle** :
  $WP(\text{if e then } i_1 \text{ else } i_2, Q) = ((e \Rightarrow WP(i_1, Q) \wedge \neg e \Rightarrow WP(i_2, Q))$
- **Règle itération** : $WP(\text{while e do } i, Q) = ?$ plus compliqué à définir, nécessite de faire intervenir des invariants de boucle.

Calcul de weakest liberal precondition (WLP) du while

Si on ne souhaite pas prouver la terminaison de la boucle, le WLP (*weakest liberal precondition*) du *while* est impliqué par tout invariant de boucle INV vérifiant les conditions suivantes :

$$WLP(\texttt{while}\ E\ \texttt{do}\ S\ \texttt{done}, R) \Leftarrow \mathit{INV} \text{ avec } \begin{array}{l}(E \wedge \mathit{INV} \Rightarrow wlp(S, \mathit{INV})) \\ \wedge\ (\neg E \wedge \mathit{INV} \Rightarrow R)\end{array}$$

Il faut rajouter la notion de variant de boucle pour prouver la terminaison de la boucle.

Calcul de weakest precondition (WP) du while

$(wfs, <)$ well-founded set (ensemble bien fondé) : ensemble ordonné sans chaîne décroissante infinie (par exemple $(\mathbb{N}, <)$).

$WLP(\texttt{while}\ E\ \texttt{do}\ S\ \texttt{done}, R)$

$$\Leftarrow \mathit{INV} \text{ if } \begin{array}{l} (E \wedge \mathit{INV} \Rightarrow \mathit{vf} \in \mathit{wfs}) \\ \wedge\ (E \wedge \mathit{INV} \wedge v = \mathit{vf} \Rightarrow wp(S, \mathit{INV} \wedge v < \mathit{vf})) \\ \wedge\ (\neg E \wedge \mathit{INV} \Rightarrow R) \end{array}$$

Informellement, dans la conjonction des trois formules ci-dessus :

1. le variant doit faire partie de wfs avant d'entrer dans la boucle ;
2. S doit préserver l'invariant et diminuer le variant ;
3. lorsque la boucle termine, l'invariant doit impliquer la post-condition R.

Calcul de weakest precondition (WP) du while

$WP(\text{while e do } i, Q) = \text{lfp}(F \mapsto (e = 0 \wedge Q) \vee (e \neq 0 \wedge WP(i, F)))$

$\text{lfp}(F \mapsto (e = 0 \wedge Q) \vee (e \neq 0 \wedge WP(i, F)))$ est la plus petite propriété I (donc, minimale) telle que :

$$I \text{ égal à } (e = 0 \wedge Q) \vee (e \neq 0 \wedge WP(i, I))$$

Ceci "correspond" à I égal à

$$(e = 0 \wedge Q) \vee e \neq 0 \wedge WP(i, (e = 0 \wedge Q)) \vee \\ e \neq 0 \wedge WP(i, e \neq 0 \wedge WP(i, (e = 0 \wedge Q))) \vee ...$$

La plus petite propriété vérifiant ce point-fixe n'est malheureusement pas calculable en général.

Calcul de WP : exemple

$$WP(c := 0; \texttt{while (c <=100) do c:=c+1}, c = 101) =$$
$$WP(c := 0; WP(\texttt{while (c <=100) do c:=c+1}, c = 101))$$
$$WP(\texttt{while (c <=100) do c:=c+1}, c = 101) = ?$$

Invariant : $I = (0 \leq c \leq 101)$

- $I \wedge c > 100 \overset{?}{\Rightarrow} c = 101$ OK
- $I \wedge c \leq 100 \overset{?}{\Rightarrow} WP(c := c + 1, I)$
  $WP(c := c + 1, 0 \leq c \leq 101) = 0 \leq c + 1 \leq 101 = -1 \leq c \leq 100$
  $0 \leq c \leq 101 \wedge c \leq 100 \overset{?}{\Rightarrow} -1 \leq c \leq 100$ OK

$$WP(c := 0; I) = WP(c := 0; (0 \leq c \leq 101)) = (0 \leq 0 \leq 101) \quad OK$$

Calcul de WP : discussion

- boucle : comment établir l'invariant I ?
  - donnée manuellement par l'utilisateur
  - idéalement, déduit automatiquement (par interprétation abstraite, machine learning).
- WP définit un algorithme, modulo le problème des invariants de boucle

Corrections partielle et totale

- Correction partielle : garantie que si le programme p termine, alors p satisfait la spécification S
- Correction totale : garantie que le programme p termine et p satisfait la spécification S

$$\text{correction totale} = \text{correction partielle} + terminaison$$

Terminaison : prouver la terminaison nécessite la donnée d'une mesure décroissante sur un domaine sans chaine infinie décroissante pour chaque boucle (un **variant**), le plus souvent indiquée manuellement par l'utilisateur

Calcul de WP : exécution "machine"

Ce code C calculant la valeur absolue est-il correct ?

```
int z_abs_x(const int x){
    int z;
    if (x < 0)
        z = -x;
    else /* x >= 0 */
        z = x;
    return z;
}
```

Calcul de `z = -x` : si $x = -2^{31}$ alors $z = 2^{31}$ mais ce nombre n'est pas représentable sur 32 bits signé.

$\Rightarrow$ obligation de preuves **pour la gestion des erreurs**

Obligations de preuve

Elles sont générées pour

- pour garantir la sûreté du programme, dès qu'une opération peut entraîner une erreur à l'exécution
  - divisions par zéro
  - déréférencement des pointeurs
  - *overflow* des opérations arithmétiques
- pour vérifier les spécifications utilisateurs
  - post-conditions
  - assertions
  - invariants et variants de boucles

Comment prouver les obligations de preuve

- à la main
  - impossible : sur un vrai programme, plusieurs milliers d'obligation de preuves générées
- à l'aide d'un assistant de preuves (COQ, PVS, Isabelle) : outil pour assister un humain dans l'activité de preuves en le guidant, en résolvant les parties triviales ou calculatoires, et en vérifiant la validité des preuves fournies par l'utilisateur
  - relativement lourd et fastidieux
- avec des prouveurs automatiques : outil "presse-bouton" effectuant les preuves de manière automatique (Z3, Alt-Ergo, CVC)

Guider les preuves

Vérifier les obligations de preuve en pratique

- utilisation de différents prouveurs automatiques en parallèle
- spécifications ad-hoc pour aider les prouveurs

Spécifications additionnelles

- annotations de boucle : clauses `loop invariant` et `loop variant`

```
int a = 0, b = 10;
/* loop invariant 0 <= a && b <= 10;
  loop variant b - a; */
while (a <= b) { a++; b--; }
```

- insertion d'assertions (joue le rôle de lemme intermédiaire)
- insertion de lemmes additionnels

SMT Solveurs : prouveurs automatiques

Satisfiability Modulo Theories (SMT) solver

- Satisfiability : vérifier qu'une formule peut être satisfaite
- Modulo Theory : présupposés sur l'arithmétique, les tableaux, les inégalités, les types de données algébriques, ...
- En d'autres termes : vérifier que l'on peut trouver une solution à une formule booléenne

Outils entièrement automatiques

- Vérifie ou échoue (erreur ou *timeout*)
- les prouveurs automatiques ne savent pas tout faire (différents prouveurs différentes capacités)

Exemples : Alt-Ergo, Z3, CVC3, Yice, ...

Frama-C : Programmation par contrat

Frama-C : framework pour l'analyse et la preuve de code C

- Construction d'un arbre de syntaxe abstrait à partir de fichiers C
- Applications de diverses analyses : plug-ins WP et Jessie : preuve en logique de Hoare
- Développé par le CEA et l'INRIA : Libre frama-c.com
- Utilisations : Airbus, Dassault Aviation, IAE Brasil, ...

Contrat sur chaque fonction

- Pré-condition : conditions vérifiées par l'appelant
- Post-condition : conditions garanties par l'appelé

abs() en Frama-C

Fonction prouvée correcte dans tous les cas légaux d'utilisation

```
frama-c-gui -wp abs-proved.c
```

```
/*  requires x >= -2147483647;
    ensures \result >= 0;
    ensures x < 0 ==> \result == -x;
    ensures x >= 0 ==> \result == x;
*/
int abs(int x){
    if (x  < 0)
        return -x;
    else
        return x;
}
```

Bilan

- domaine d'application très large
- garanties très fortes (correction totale)
- niveau d'expertise utilisateur : moyen à élevé
- limitation venant plutôt des prouveurs
- pas de preuve possible pour certains programmes (ex : programmes avec des boucles dont on ne peut pas trouver d'invariant de boucle)