

Validation Formelle

Plan de l'exposé

- 1 Problématique
- 2 Logique de Hoare
 - Langage
 - Correction partielle et totale
 - Axiomes
 - Exemple
 - Résultats
- 3 Spécification Formelle
 - Langage de spécification
 - Design by Contract: Principe
- 4 JML un BSL pour Java
 - Les bases
 - Effets de bords
 - Constructions Avancées

I have a dream

Mon programme est correct

I have another dream

La preuve de correction est automatique

Nécessité de formaliser:

- spécifications
- sémantique du langage
- calcul de correction

Spécification formelle (voir cours précédent):

- Décrire formellement le comportement
- Vérifier que l'exécution est conforme
 - De manière dynamique à l'exécution du programme
 - De manière statique par examen du code

Validation

- Approchée
 - Typage: un programme bien typé ne déclenche pas d'erreur
Typage fort, système de type cohérent,
 - Analyse statique: vérification d'un modèle abstrait, vérification de certaines propriétés, analyse correcte mais incomplète.
- Exacte
 - Dérivation du programme à partir de spécifications formelles (CoQ, B, ...)
 - Vérification a posteriori du programme dans une logique adaptée (Hoare, ...)

Validation par annotations

- Assertions dans le code (formalisme logique adapté au langage C, Java, ...)
- Evaluation des annotations à l'exécution (JMLRAC, ...) ou statique (ESC/Java2, ...)
- Principe de développement (Design by Contract)

Difficultés:

- Problèmes difficiles (indécidable!)
- Passage à l'échelle
- Formation des développeurs

Résultats:

- Théoriques: pointeurs, mémoire, SD évoluées
- Pratiques: langages, prouveurs génériques, dédiés

Plan de l'exposé

- 1 Problématique
- 2 **Logique de Hoare**
 - Langage
 - Correction partielle et totale
 - Axiomes
 - Exemple
 - Résultats
- 3 Spécification Formelle
 - Langage de spécification
 - Design by Contract: Principe
- 4 JML un BSL pour Java
 - Les bases
 - Effets de bords
 - Constructions Avancées

Objectif: une logique de preuve de programme

Un petit langage impératif

Exp ::= $n \mid e \text{ op } e$ $op \in OP = \{+, -, *, /\}, n \in Z$
 Cond ::= $T \mid F \mid e \text{ comp } e$

Ins ::= $x=e;$ $comp \in \{==, >, <\}$
 | if C then Ins (affectation)
 | if C then Ins else Ins (conditionnelle simple)
 | Ins Ins (conditionnelle)
 | while C do Ins (séquence)
 | (itération)

Programme = élément généré par *Ins*

Propriétés

- Turing complet
 - ① Permet de tout calculer
 - ② Indécidabilité (arrêt, valeur retournée,...)
- Pas de fonctions, pas de classes,... expressivité **pratique** limitée.

Formalisme

- I programme utilisant les variables x_1, \dots, x_n .
- Etat s = affectation de valeurs aux variables.
- Condition C (pour I): formule logique avec variables libres de C incluses dans $\{x_1, \dots, x_n\}$ (Logique du 1er ordre, logique d'ordre supérieure,...)
- Etat s valide P noté $s \models P$ si l'affectation s rend vraie P .

Triplet de Hoare

Formule de la logique de Hoare:

$$\{P\}I\{Q\}$$

avec P, Q conditions pour I

Exemple:

$$\{y > 0\}x = y + 1\{x > 1\}$$

Correction partielle

$$\vdash \{P\}I\{Q\}$$

s modèle de P , si l'exécution de I à partir de s termine et renvoie un état s' alors s' est un modèle de Q .

(si P est vraie et si I termine alors Q est vraie)

Correction

$$\vdash [P]I[Q]$$

s modèle de P , alors l'exécution de I à partir de s termine et renvoie un état s' qui est un modèle de Q .

(si P est vraie alors I termine et Q est vraie)

Correction=Correction partielle + preuve de terminaison.

Exemple:

est ce que $\{y > 0\}x = y + 1; \{x > 1\}$?

est-ce que $[x > 0]while\ T\ do\ x = x + 1; [x > 1]$?

est-ce que $[x > 0]while\ T\ do\ x = x + 1; [x = -1]$?

Principe de la logique de Hoare:

- I un programme,
- P une formule qui exprime les préconditions sur les valeurs initiales des variables,
- Q une formule qui spécifie le comportement de I sur les valeurs initiales

Montrer **par un calcul d'un système formel** que $\{P\}I\{Q\}$

Les axiomes de la logique

Renforcement précondition
$$\frac{P \implies P' \quad \{P'\}I\{Q\}}{\{P\}I\{Q\}}$$

Affaiblissement postcondition
$$\frac{\{P\}I\{Q\} \quad Q \implies Q'}{\{P\}I\{Q'\}}$$

Sequence
$$\frac{\{P\}I\{Q\} \quad \{Q\}I\{R\}}{\{P\}I \quad I'\{R\}}$$

$$\text{Conditionnelle} \frac{\{P \wedge \text{cond}\} I \{Q\} \quad \neg \text{cond} \implies Q}{\{P\} \text{if } \text{cond} \text{ then } I \{Q\}}$$

$$\text{Conditionnelle double} \frac{\{P \wedge \text{cond}\} I \{Q\} \quad \{P \wedge \neg \text{cond}\} J \{Q\}}{\{P\} \text{if } \text{cond} \text{ then } I \text{ else } J \{Q\}}$$

$$\text{Itération} \frac{\{P \wedge C\} I \{P\} \quad P \wedge \neg C \implies Q}{\{P\} \text{while } C \text{ do } I \{Q\}}$$

Affectation $\{P\}x = e\{P_{x \leftarrow e}\}$

$\{x == 0\} x = x + 1 \{x + 1 == 0\}$ i.e. $\{x == 0\} x = x + 1 \{x == -1\}!!!$

\Rightarrow Faux

Affectation $\{P\}x = e\{P_{x \leftarrow e}\}$

$\{x == 0\} x = x + 1 \{x + 1 == 0\}$ i.e. $\{x == 0\} x = x + 1 \{x == -1\}!!!$

\Rightarrow Faux

La bonne règle

Affectation $\{Q_{x \leftarrow e}\}x = e\{Q\}$



raisonner à l'envers de la conclusion vers l'hypothèse

$\{??\}x = x + 1\{x > 1\}$

$\{x > 1_{x \leftarrow x+1}\}x = x + 1\{x > 1\}$

\equiv

$\{x + 1 > 1\}x = x + 1\{x > 1\}$

\equiv

$\{x > 0\}x = x + 1\{x > 1\}$

Exemple 1: montrer

$$\vdash \{x = a \wedge y = b\} r = x; x = y; y = r \{x = b \wedge y = a\}$$

Exemple 2: montrer

$$\begin{aligned} &\vdash \{n \geq 0\} \\ &\quad x = 0 \\ &\quad i = 1 \\ &\quad \text{while } i \neq n + 1 \text{ do} \\ &\quad \quad x = x + i \\ &\quad \quad i = i + 1 \\ &\quad \{x = n(n + 1)/2\} \end{aligned}$$

Notation:

$\vdash \{P\}I\{Q\}$ ssi on peut déduire $\{P\}I\{Q\}$ avec les axiomes du calcul.

Questions.

Correction: Si $\vdash \{P\}I\{Q\}$ est-ce que $\{P\}I\{Q\}$ est un triplet de Hoare sémantiquement valide.

Complétude: Si $\{P\}I\{Q\}$ est un triplet de Hoare sémantiquement valide est-ce que $\vdash \{P\}I\{Q\}$?

Paramétré par la logique utilisée pour écrire les conditions (supposée correcte et complète par exemple).

I est le programme

```
q=0;  
r=a;  
while a > b do  
    r=r-b;  
    q=q+1;
```

Montrer

$$\{a = a_0 \wedge b = b_0 \wedge a_0 \geq 0 \wedge b > 0\} I \{0 \leq r < b_0 \wedge a_0 = b_0 q + r\}$$

Idée trouver un invariant de boucle.

Résultats

La logique de Hoare établit des triplets valides:

Theorem (Correction)

Le système de preuve est correct.

Réciproquement, si la logique d'écriture des conditions est suffisamment expressive on a:

Theorem (Complétude)

Le système de preuve est complet.

Suffisamment expressive signifie en particulier qu'on peut écrire des assertions correspondant aux points fixes et qu'on dispose d'un oracle permettant de décider cette logique

Extensions

- Effet de bords
- Constructions supplémentaires (for, switch,...)
- Interruptions, exceptions
- Appel de procédures
- SD évoluées, pointeurs, listes, tableaux
- Système de preuve de terminaison

Plan de l'exposé

- 1 Problématique
- 2 Logique de Hoare
 - Langage
 - Correction partielle et totale
 - Axiomes
 - Exemple
 - Résultats
- 3 Spécification Formelle**
 - Langage de spécification
 - Design by Contract: Principe
- 4 JML un BSL pour Java
 - Les bases
 - Effets de bords
 - Constructions Avancées

Spécifier?

Décrire **formellement** le comportement du système

- 1 Etat avant l'exécution
- 2 Etat après l'exécution

Limitations?

Spécifier?

Décrire **formellement** le comportement du système

- 1 Etat avant l'exécution
- 2 Etat après l'exécution

Limitations?

système d'exploitation? contrôleur d'un ascenseur,...

modules, classes: spécification des interfaces, methodes,...

Spécifier?

Décrire **formellement** le comportement du système

- 1 Etat avant l'exécution
- 2 Etat après l'exécution

Limitations?

Etat défini par les **valeurs** des **variables** du système. \implies valeur **avant/après** l'exécution.

Spécification de méthode

Précondition: description des prérequis sur les variables d'états.

Postcondition: description des conséquences sur les variables d'états.

Langage:

- Formules logiques $\forall, \exists, \wedge, \vee, \implies, ..$
- Domaines interprétés (réels, entiers) et prédicats associés ($=, >, ..$)
- Constructions usuelles *listes, ensembles, tableau, ...*

P, Q : précondition, postcondition sur les variables d'état du programme I

Spécification du programme I :

$$\{P\}I\{Q\}$$

Si P est vraie, alors Q est vraie après l'exécution de I (si celle-ci termine)

P, Q : précondition, postcondition sur les variables d'état du programme I

Spécification du programme I :

$$\{P\}I\{Q\}$$

Si P est vraie, alors Q est vraie après l'exécution de I (si celle-ci termine)

STOP: quelle différence avec ce qui précède?

Spécification:

- P et Q donnés
- Ecrire $I!$

Preuve:

- I écrit, P, Q spécifiant le comportement.
- Montrer $\vdash \{P\}I\{Q\}$ dans le système formel donné.

Exemple

I : variables d'état entrée x, ϵ , sortie res

Précondition: $x \geq 0$

Postcondition: $|x - res * res| \leq \epsilon$

Convention: $\backslash old(x)$ valeur de x avant l'exécution

x non modifiée alors $x = \backslash old(x)$

Question: $I \equiv res = foo(x, \epsilon)$ quelle est la fonction foo spécifiée?

Exemple

I : variables d'état entrée x, ϵ , sortie res

Précondition: $x \geq 0$

Postcondition: $|x - res * res| \leq \epsilon$

Convention: $\backslash old(x)$ valeur de x avant l'exécution

x non modifiée alors $x = \backslash old(x)$

Question: $I \equiv res = foo(x, \epsilon)$ quelle est la fonction foo spécifiée?

Invariant

Formule logique dont la valeur est conservée

- 1 par l'évaluation du corps d'une boucle
- 2 par l'évaluation de la méthode
- 3 tout au long de l'existence d'un objet
- 4 ...

Utilité:

spécifier la permanence d'une propriété *le solde d'un compte en banque est toujours positif ou nul*

faciliter l'écriture du programme *la valeur cherchée est entre les éléments d'indice i et j ou n'est pas présente*

Le Paradigme du Design by Contract

Le comportement de chaque entité (méthode, objet, module,..) est donné par un contrat

$$\{P\}I\{Q\}$$

Si la précondition (mot clé **requires**) P est vraie

alors l'exécution de I ,

garantit (mot clé **ensures**) que la postcondition Q est vraie.

si l'exécution de I **termine**

Design by contract: écrire le code I tel que le contrat est assuré.
Le programmeur suppose donc que la précondition est assurée.
Le code est écrit pour assurer qu'alors la postcondition sera vraie.
Exemple:

$$\{a \geq 0 \wedge b > 0\} I \{a = bq + r \wedge 0 \leq r < b\}$$

\implies Le code n'a pas à vérifier $b \neq 0$

Qui vérifie la précondition? **c'est à la fonction appelante d'établir la précondition de l'appelée.**

Exercice: donner le contrat pour *depiler*

{pile non vide} depiler() {pile non pleine}

ou

{pile non vide} depiler() {pile non pleine && size = \ old(size)-1;}

Exercice: quel est le contrat le plus facile à assurer?

$\{\text{True}\} \mid \{ Q \}$

ou

$\{\text{False}\} \mid \{ Q \}$

Plan de l'exposé

- 1 Problématique
- 2 Logique de Hoare
 - Langage
 - Correction partielle et totale
 - Axiomes
 - Exemple
 - Résultats
- 3 Spécification Formelle
 - Langage de spécification
 - Design by Contract: Principe
- 4 JML un BSL pour Java
 - Les bases
 - Effets de bords
 - Constructions Avancées

JML un BISL pour Java

Behavior Interface Specification Language

- décrit le comportement des classes
- décrit le design et les choix d'implémentation

Moyens: annotation sur le code Java sous forme de commentaires.

Discipline: Design by Contract

Utilité?

- 1 Ecrire la spécification facilite la programmation
- 2 Documentation du code
- 3 Vérification dynamique (outil JMLRAC et ...)
- 4 Vérification statique (outil ESC/Java2 et ...)
- 5 Génération de schémas de tests à partir des annotations

Le Langage

- ligne préfixée par @
- instruction avec syntaxe Java
- mots clés spécifiques
- emplacements imposés (ou pas)

- **requires** (précondition)
- **ensures** (postcondition)
- **invariant** (de classe, de boucle)
- **assert** (assertion)

Pre-postcondition précèdent la méthode

Invariant de classe: pas d'emplacement spécifique (usuel: début ou fin de classe), peut être éclaté.

Exemple de la pile

Méthode pop

- Précondition: pile non vide **requires !empty();**
- Postcondition: pile non pleine **ensures !full();**

Ecrit comme commentaire avant la méthode, préfixé par @

```
//@requires !empty();  
//@ensures !full();  
public void pop(){  
    lg--;  
}
```

ou

```
/*@  
  @requires !empty();  
  @ensures !full();  
  @*/  
public void pop(){  
    lg--;  
}
```

Et les autres méthodes?

Préconditions et Postconditions pour:

- top retourne le sommet de pile
- push empile un élément e
- empty retourne vrai ssi la Pile est vide
- full retourne vrai ssi la Pile est pleine
- Pile constructeur qui fabrique une Pile vide

Ecrire les annotations JML correspondantes.

Assert

Mettre une assertion dans le programme

```
if (i <=0 || j<0) {  
  ...}  
else if (j<5) { //@assert i>0&& 0<j&&j<5;  
  ...}  
else {//@assert i>0&& j>5;  
  ...}
```

Effets de Bords

requires pile[top++]==x;

inacceptable: comportement différents selon que les annotations sont exécutées ou non.

→ Pas de constructions Java avec effet de bord.

→ Uniquement des méthodes **pires**

Méthode pure: observateur, ne modifie pas l'objet.

Mot Clé: **pure**

```
public /*@ pure @*/ empty(){  
return top==0;  
}
```

Spécifier les attributs qui sont modifiés par une méthode.

```
/*@requires !empty();  
   @assignable top;  
   @ensures !full();  
*/  
public void pop() {  
  top--;  
}
```

Mot clé **assignable**

Par défaut tout est assignable

assignable everything;

Possible:

assignable nothing;

Constructions Avancées

Java:

- Visibilité
- Exceptions
- Héritage

Visibilité

Problème: spécification peut utiliser des noms privés.

```
private int lg;  
/*@  
...  
@requires lg >= 0;  
@*/
```

File "PileP.java", line 14, character 24 error: Field "lg" (private visibility) can not be referenced in a specification context of "package" visibility [JML]

Visibilité

→ différencier visibilité spécification et implémentation.

Mot clé **spec_public**

```
private /*@ spec_public @*/int lg;  
/*@  
    ...  
    @requires lg >= 0;  
@*/
```

Discipline: utiliser des accesseurs pour lire les attributs.

```
public int getLg() {  
    return lg;}  
}
```

Héritage

C hérite de C' : tout objet de type C est un objet de type C' (avec des propriétés supplémentaires).

\implies

héritage des spécifications.

Principe de substitution (Liskov): partout où un objet de type C' est demandé, on peut fournir un objet de type C (substituer $o' : C'$ par $o : C$) sans causer d'erreur.

\implies

le contrat établi pour C' doit rester valide pour C

Héritage

foo méthode de C' redéfinie dans C . Précondition de *foo* dans C ?

- 1 x déclaré de type C'
 $x.foo(\dots)$;
i.e. le contrat $\{P'\}foo(\dots)\{Q'\}$ de *foo*() dans C' est assuré.
- 2 x instancié par un objet o de type C $o.foo(\dots)$ invoque la precondition P de *foo*() dans C (liaison dynamique)
si P est plus forte que P' alors elle peut ne pas être vraie!
violation du principe de substitution.
- 3 Conclusion: la precondition de *foo*(...) dans C doit être **plus faible**!

Postcondition de *foo* dans C ?

Symétrique: la postcondition de *foo*(...) dans C doit être **plus forte**!

Exceptions

Spécification doit prendre en compte les exceptions:

→ mot clef `signals`

```
/*@ requires x>=0;
   @ signals (ISOException e)
   @ x> mont && mont==\old mont &&
   @     e.getReason()==MONTANT_INSUFFISANT;
   @*/
public int retire(int x){...}
```

L'invariant doit être préservé!!

Exceptions

Défaut: signal (Exception) true;
Pas d'exceptions: signals (Exception) false;
ou

```
/*@ normal_behavior  
   @requires  
   @ensures  
  @*/
```

Utilisation JML?

- vérification de code Java 1.4 possible (outils existant sous Java 1.5/eclipse)
- Travail en cours pour annotation de versions 1.5+ avec JM4rac
- Génération automatique de schémas de test à partir des annotations JML possible avec certains outils.