Preuve, analyse statique et vérification *runtime*

Master CILS

2018–19

Besoin de sûreté

La fiabilité peut être parfois importante...
et parfois très importante !

- Centrales nucléaires, usines chimiques
- Commandes d'avions
- Pacemakers (voire cœur artificiel !)
- Réseaux ferroviaires, etc.

Systèmes auto-* ?

Des conséquences importantes

De petites erreurs peuvent avoir des conséquences désastreuses :
Coût astronomique, vies humaines !

- Vol inaugural d'Ariane 5
- Mars Climate Orbiter
- London Ambulance Service Computer-Aided Dispatch System
- Bug du Pentium d'Intel
- Debian/OpenSSL RNG

Solutions classiques d'ingénierie pour la fiabilité

Quelques stratégies bien connues issue du génie civil :

- Calcul précis/estimation des forces, tensions, etc.
- Redondance matérielle
- Conception robuste (une petite erreur n'est pas catastrophique)
- Séparation claire en sous-systèmes
- Utilisation des motifs dont on sait qu'ils fonctionnent

Pourquoi ça ne marche pas pour les logiciels ?

- Les systèmes logiciels calculent des fonction non continues
  Une simple inversion de bit peut changer le comportement complètement
- La redondance en répliquant ne protège pas des bugs
  La redondance du développement n'est viable que dans des cas extrêmes
- Pas de séparation claire en sous-système
  Une erreur locale affecte en général tout le système
- La conception logicielle a une très grande complexité logique
- La plupart des ingénieurs ne sont pas formés à la correction des logiciels
- Le coût supposé est souvent préféré à la fiabilité
- ...

Test

Une solution contre le manque de fiabilité : le test
Limites :

- Le test montre la présence d'erreurs, pas leur absence
  Exhaustivité possible uniquement pour des systèmes triviaux
- Problème de la représentativité des cas de test
  Comment tester l'inattendu ? Les cas rares ?
- Le test demande du travail, donc coûte cher
- Il n'est pas toujours réalisable avant le déploiement

Méthodes formelles

Méthodes rigoureuses utilisées lors de la conception du système et le développement

- Fondées sur les mathématiques et la logique $\rightarrow$ formelles
- Construire un modèle de l'implémentation et des exigences (spécification)
- Utiliser des outils pour prouver mécaniquement que le modèle d'exécution formel satisfait la spécification

Plusieurs approches de méthodes formelles

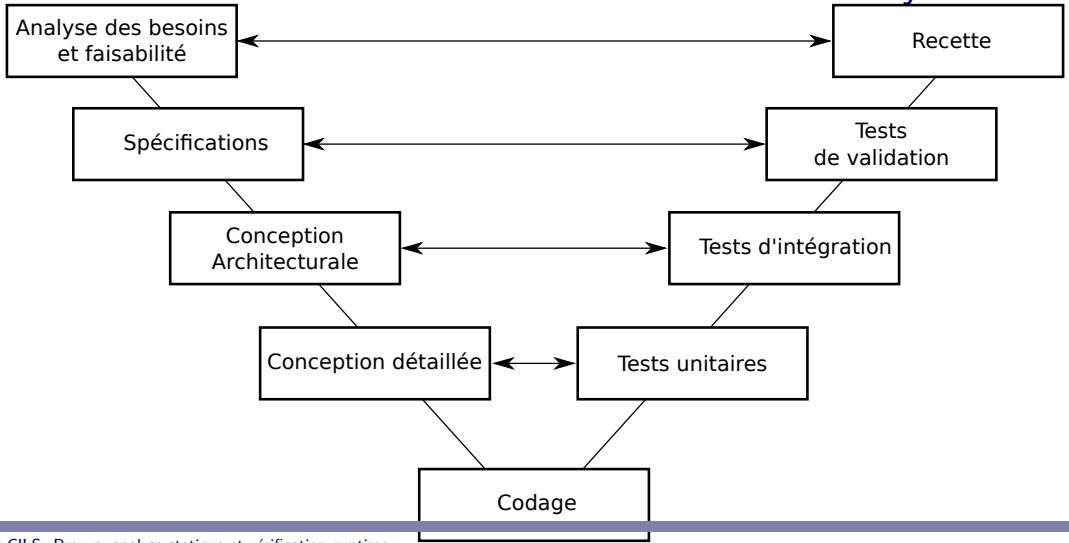
- Model checking
- Preuve
- Analyse statique
- Méthodes formelles pendant la conception (raffinement)

Runtime verification

Sur un système qui peut évoluer, on veut être sûr que le comportement reste celui prévu

- Production de traces
- Monitoring

Méthodes formelles et runtime verifications dans le cycle en V



Méthodes formelles : Pour qui ?

Applications critiques : vies humaines ou coût faramineux en jeux

- Aéronautique et espace (A380)
- Ferroviaire (Métro 14)
- Hardware (Intel)

Mais pas seulement :

- Gain de temps avant la découverte des bugs
- Au final, moins cher que de corriger les bugs après coup

Exemple : Facebook (Infer)

But de la vérification formelle

Etant donnée une spécification formelle S d'un programme P,

donner une preuve mathématique rigoureuse que toute exécution de P satisfait S

P est correct par rapport à S

Que faut-il pour vérifier formellement P ?

- une spécification formelle de P (rigoureuse, mathématique),
- une méthode de preuve de correction (logique de Hoare par exemple),
- des règles de preuve pour faire des preuves rigoureuses, mathématiques : un calcul.

Limitations de la vérification formelle

- pas de notion absolue de correction
  La correction est toujours relative à une spécification donnée
- Difficile et coûteux d'écrire des spécifications formelles
  En pratique, on ne spécifie pas formellement toutes les fonctionnalités mais les propriétés de sûreté comme la bonne formation des données (accès hors des bornes, déréférencement du pointeur null, etc.), l'absence d'exceptions non détectées, les parties critiques du logiciel etc.
- Coûteux également de faire des preuves (mais on gagne sur le temps de test)

Limitations de la vérification formelle

- Il existe des propriétés difficiles ou impossibles à spécifier
  ressources comme le temps et la mémoire (possible) le comportement de l'utilisateur, l'environnement en général.
- Des programmes vérifiés formellement peuvent planter à l'exécution
  - bugs dans le compilateur ($\rightarrow$ vérification du compilateur C Compcert)
  - bugs dans l'environnement d'exécution ($\rightarrow$ vérification NICTA d'un micro-kernel SEL4)
  - bugs dans le matériel ($\rightarrow$ preuve de hardware Intel par ex.)

Le test et la preuve sont complémentaires

Spécifications

Conception par contrat

Idée : transférer la notion de contrat entre partenaires commerciaux aux ingénieurs logiciel

- Qu'est-ce qu'un contrat

  Un accord contraignant qui précise explicitement les obligations et les bénéfices
- En ingénierie logicielle ? (une interface)

Fournisseur (implémenteur), en C, une procédure, en Java, une classe ou une méthode

Client Une procédure ou un objet appelant, voire un utilisateur humain (e.g. pour `main()`)

Contrat Un ensemble de clauses ensures/requires qui définissent les obligations mutuelles du client et de l'implémenteur

Exemple

La fonction racine carrée sur les entiers

```
/*@ requires a est un entier positif
 *  ensures l'entier retourne est la racine carree de a */
int root (int a) {
  int i = 0;
  int k = 1;
  int sum = 1;
  while (sum <= a) {
    k = k+2;
    i = i+1;
    sum = sum+k;
  }
  return i;
}
```

Formellement

Langage de spécification ACSL

```
/*@ requires a >= 0;
 *  ensures \result*\result<=a && (\result+1)*(\result+1)>a; *
int root (int a) {
  int i = 0;
  int k = 1;
  int sum = 1;
  while (sum <= a) {
    k = k+2;
    i = i+1;
    sum = sum+k;
  }
  return i;
}
```

Exercice

Maximum de deux entiers

```
/*@ requires ?
 *  ensures ? */
int max(int a, int b) {
  if (a <= b)
    return b;
  else
    return a;
}
```

Plus d'expressivité

Indice maximum dans un tableau

```
/*@ requires sizeof(a) > 0;
    ensures
       \forall int i;
          (0 <= i && i < sizeof(a))
          ==> a[i] <= a[\result];   */
int indice_max(int a[]) {
  int i;
  int r = 0;
  for (i = 1; i < sizeof(a); i++)
    if (a[i] > a[r]) r = i;
  return r;
}
```

Logique du premier ordre

Notation math.	ACSL	Sémantique
$\top$	`\true`	toujours vrai
$\bot$	`\false`	toujours faux
$\neg A$	`!A`	non A
$A \wedge B$	`A && B`	A et B
$A \vee B$	`A || B`	A ou B
$A \Rightarrow B$	`A ==> B`	A implique B
$\forall x : i.\ A$	`\forall i x; A`	pour tout x de type i, on a A
$\exists x : i.\ A$	`\exists i x; A`	il existe un x de type i tel que A

Exercices

Donner des spécifications aux fonctions suivantes :

```
int remainder(int a, int b);
/* reste de la division euclidienne
   de a par b */

int maxarray(int a[]);
/* retourne la valeur maximum du tableau a */
```

Tri

```
int[] sort(int a[]);
/* tri du tableau a */
```

Tri

```
int[] sort(int a[]);
/* tri du tableau a */
```

Tableau résultant trié :

```
ensures \forall int i, j;
  0 <= i < j < sizeof(a) ==> \result[i]<= result[j];
```

Tri

```
int[] sort(int a[]);
/* tri du tableau a */
```

Tableau résultant trié :

```
ensures \forall int i, j;
  0 <= i < j < sizeof(a) ==> \result[i]<= result[j];
```

Tableau résultant permutation du tableau initial :

```
ensures sizeof(\result) == \sizeof(a) &&
          \forall int i; 0 <= i < sizeof(a) ==>
          \exists int j; 0 <= j < sizeof(a) &&
            a[i] == \result[j];
```

Est-ce assez ?

Tri (cont.)

Non, `[2; 1; 1]` et `[1; 2; 2]`.
Il faut que le nombre d'occurrences soient les mêmes : nouveau prédicat `nbocc` à définir, puis

```
ensures \forall int v;
            nbocc(a, v) == nbocc(\result, v);
```

Tri en place

```
void sort(int a[]);
/* tri du tableau a en place */
```

Utilisation de `\old` pour accéder à la valeur avant la fonction.

```
ensures \forall int i, j;
  0 <= i < j < sizeof(a) ==> a[i]<= a[j];
ensures \forall int v;
  nbocc(\old(a), v) == nbocc(a, v);
```

Logique de Hoare, Weakest Precondition

Comment démontrer les spécifications ?

On donne au programme une sémantique

- L'état du système est abstrait par les propriétés logiques qu'il satisfait
- L'exécution des pas du programme modifie la validité de ses propriétés

On vérifie ensuite que les postconditions (`ensures`) sont bien impliquées par les préconditions (`requires`) via l'exécution du programme.

Logique de Hoare

Définie par Hoare (inventeur de QuickSort) en 1969

Pour les langages impératifs. Pour ce cours, C restreint à :

- Seulement des variables entières, pas de pointeurs
- Affectation `x = e;`
- Séquence `{` $\texttt{i}_1$ `...` $\texttt{i}_n$ `}`
- Conditionnelle **`if`** `(c)` $\texttt{i}_1$ **`else`** $\texttt{i}_2$
- Boucle **`while`** `(c) i`
- Instruction vide `;` (permet de ne pas avoir de **`else`**)

Langages des assertions

L'état du système est décrit par des propositions de la logique du premier ordre avec arithmétique

- $P ::= b \mid P \wedge P \mid P \vee P \mid \neg P \mid P \Rightarrow P \mid \forall x.\ P \mid \exists x.\ P$
- b peut être n'importe quelle expression arithmétique du langage, mais sans appel de fonction
- les variables du programme et celles de la logique sont identifiées
- implicitement, les variables prennent des valeurs entières

Exemple :

$$\forall z.\ x \leq z \Rightarrow y + 2 = z$$

Triplet de Hoare

$\{P\}$`c`$\{Q\}$ avec P et Q des assertions logiques et `c` un programme

- P : précondition
- Q : postcondition
- si `x` est une variable du programme, dans P et Q
  x représente la valeur de `x` à ce point du programme

Lire : si la propriété P est vraie avant l'exécution de `c` et si l'exécution termine, alors la propriété Q est vraie après l'exécution de `c`
(Correction partielle, pas de preuve de terminaison)

Exemple

$\{x = n \wedge n > 0\}$ `y = 1; while (x != 1) { y = y * x; x = x - 1; }`
$\{y = n! \wedge n > 0\}$

Ici, n est une variable logique, elle permet de se référer à la valeur de x au début du programme

Validité des triplets de Hoare

Tous les triplets de Hoare ne sont pas valides.

Intuitivement, on veut que $\{P\}\texttt{c}\{Q\}$ soit valide si quand P est vrai avant d'exécuter le programme, alors Q est vrai après l'exécution.

Exemple de triplets valides :

- $\{\bot\}\texttt{x = 5;}\{\top\}$
- $\{x = y\}\texttt{x = x + 3;}\{x = y + 3\}$
- $\{x > 0\}\texttt{x = 2 * x;}\{x > -2\}$
- $\{x = a\}\texttt{if (x < 0) x = -x; else ;}\{x = |a|\}$

Mais $\{x < 0\}\texttt{x = x - 3}\{x > 0\}$ n'est pas valide

Logique de Hoare : les règles d'inférence

La validité des jugements est définie par induction sur le programme
Ensemble de règles de déduction (règles d'inférence) sur les triplets

- Les règles s'utilisent comme des règles d'inférence de logique.

$$\frac{F_1 \ \ldots \ F_k}{F_0}$$

  Ici les F_i sont des triplets de Hoare.
- Une règle exprime coment déduire un triplet (dénominateur) à partir d'autres triplets (ceux du numérateur)

Logique de Hoare : les règles d'inférence

- Pour montrer qu'un triplet F_0 est dérivable, on doit construire un arbre de dérivation de racine F_0 en appliquant les règles d'inférence.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{F_1} \quad \dfrac{}{F_2}}{F_3} \quad \dfrac{\dfrac{}{F_4}}{F_5}}{F_6}}{F_0}$$

- 1 règle par construction du langage + 2 règles logiques

Skip

Pour toute proposition P :

$$\text{skip}\ \frac{}{\{P\}\,;\{P\}}$$

; ne change pas l'état, si P est vraie avant, P est vraie après

Séquence

$$\text{seq}\ \frac{\{P\}\texttt{c}_1\{Q\} \qquad \{Q\}\texttt{c}_2\{R\}}{\{P\}\texttt{\{c}_1\ \texttt{c}_2\texttt{\}}\{R\}}$$

`{c1 c2}` exécute `c1` puis `c2`

Exemple :

$$\text{seq}\ \frac{\{x > 2\}\texttt{x = x + 1;}\{x > 3\} \qquad \{x > 3\}\texttt{x} = \texttt{x} + 2;\{x > 5\}}{\{x > 2\}\texttt{\{ x = x + 1; x = x + 2; \}}\{x > 5\}}$$

Affectation

$$\text{aff}\ \frac{}{\{[e/x]P\}\texttt{x = e;}\{P\}}$$

Rappel : $[e/x]P$ est la formule P où toutes les occurrences libres de x ont été substituées par e, par exemple $[x+1/x](x>1)$ vaut $x+1>1$

- L'affectation change l'état, il faut donc que la logique de Hoare reflète ce changement
- De droite à gauche :
  Pour que P soit vrai après l'exécution, il fallait que, pour un x remplacé par sa nouvelle valeur, il soit vrai avant

Exemple : $\{x+1>5\}\texttt{x = x + 1;}\{x>5\}$

Substitutions

$$[y/x]x =$$

Substitutions

$$[y/x]x = y$$

Substitutions

$[y/x]x = y$

$[y/x]z =$

Substitutions

$$[y/x]x = y$$

$$[y/x]z = z$$

Substitutions

$$[y/x]x = y$$

$$[y/x]z = z$$

$$[y/x]x > y =$$

Substitutions

$$[y/x]x = y$$

$$[y/x]z = z$$

$$[y/x]x > y = y > y$$

Substitutions

$$[y/x]x = y$$

$$[y/x]z = z$$

$$[y/x]x > y = y > y$$

$$[y/x](\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z) =$$

Substitutions

$$[y/x]x = y$$

$$[y/x]z = z$$

$$[y/x]x > y = y > y$$

$$[y/x](\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z) =$$
$$\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z$$

Substitutions

$$[y/x]x = y$$

$$[y/x]z = z$$

$$[y/x]x > y = y > y$$

$$[y/x](\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z) = \\ \forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z$$

$$[y/x](\forall z, z \ mod \ 2 = 0 \Rightarrow x * x = z) =$$

Substitutions

$$[y/x]x = y$$

$$[y/x]z = z$$

$$[y/x]x > y = y > y$$

$$[y/x](\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z) = \\ \forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z$$

$$[y/x](\forall z, z \ mod \ 2 = 0 \Rightarrow x * x = z) = \\ \forall z, z \ mod \ 2 = 0 \Rightarrow y * y = z$$

Substitutions

$$[y/x]x = y$$

$$[y/x]z = z$$

$$[y/x]x > y = y > y$$

$$[y/x](\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z) = \\ \forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z$$

$$[y/x](\forall z, z \ mod \ 2 = 0 \Rightarrow x * x = z) = \\ \forall z, z \ mod \ 2 = 0 \Rightarrow y * y = z$$

$$[y/x](\forall y, x \ mod \ 2 = 0 \Rightarrow x * y = z) =$$

Substitutions

$[y/x]x = y$

$[y/x]z = z$

$[y/x]x > y = y > y$

$[y/x](\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z) =$
$\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z$

$[y/x](\forall z, z \ mod \ 2 = 0 \Rightarrow x * x = z) =$
$\forall z, z \ mod \ 2 = 0 \Rightarrow y * y = z$

$[y/x](\forall y, x \ mod \ 2 = 0 \Rightarrow x * y = z) =$
renommage du y lié pour éviter la capture de y
$[y/x](\forall t, x \ mod \ 2 = 0 \Rightarrow x * t = z) =$

Substitutions

$[y/x]x = y$

$[y/x]z = z$

$[y/x]x > y = y > y$

$[y/x](\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z) =$
$\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z$

$[y/x](\forall z, z \ mod \ 2 = 0 \Rightarrow x * x = z) =$
$\forall z, z \ mod \ 2 = 0 \Rightarrow y * y = z$

$[y/x](\forall y, x \ mod \ 2 = 0 \Rightarrow x * y = z) =$
renommage du y lié pour éviter la capture de y
$[y/x](\forall t, x \ mod \ 2 = 0 \Rightarrow x * t = z) =$
$(\forall t, y \ mod \ 2 = 0 \Rightarrow y * t = z)$

Exemple

Soit le programme c suivant :

```
{ tmp = x;
  x = y;
  y = tmp; }
```

Montrer que $\{x = a \wedge y = b\}\texttt{c}\{y = a \wedge x = b\}$ est valide

Conditionnelle

$$\text{if}\ \frac{\{P \wedge b\}\texttt{c1}\{Q\} \qquad \{P \wedge \neg b\}\texttt{c2}\{Q\}}{\{P\}\texttt{if (b) c1 else c2}\{Q\}}$$

- Quand la conditionnelle est exécutée, soit `c1` soit `c2` est exécuté
- Pour établir que Q est une post-condition, il faut donc que ce soit une post-condition des deux branches
- De la même façon, P est une pré-condition des deux branches
- Dans la branche `c1`, on sait que la condition b est vraie, on peut donc l'ajouter en pré-condition
- Similairement, on ajoute $\neg b$ comme pré-condition de `c2`

Utilisation

Prenons $\{x > 2\}$`if (x>2) y = 1; else y = -1;`$\{y > 0\}$

En utilisant la règle if, il faut donc montrer
$\{x > 2 \wedge x > 2\}$`y = 1;`$\{y > 0\}$ et
$\{x > 2 \wedge \neg x > 2\}$`y = -1;`$\{y > 0\}$

ce que l'on peut simplifier en
$\{x > 2\}$`y = 1;`$\{y > 0\}$ et
$\{\bot\}$`y = -1;`$\{y > 0\}$

Or à partir de la règle aff, on ne peut dériver que :
$\{1 > 0\}$`y = 1;`$\{y > 0\}$ et
$\{-1 > 0\}$`y = -1;`$\{y > 0\}$

Les règles structurelles ne suffisent pas, il faut utiliser des règles logiques

Conséquence gauche

Une condition P est dite plus forte qu'une condition Q si $P \Rightarrow Q$ est valide logiquement
Similairement, Q est dite plus faible que P.

$$\text{consG}\ \frac{P \Rightarrow P' \qquad \{P'\}\texttt{c}\{Q\}}{\{P\}\texttt{c}\{Q\}}$$

Il est correct de rendre une précondition plus spécifique (plus forte). Cette règle permet de renforcer les préconditions.
Exemple :

$$\text{consG}\ \frac{x = 4 \Rightarrow x > 2 \qquad \{x > 2\}\texttt{x = x + 1;}\{x > 3\}}{\{x = 4\}\texttt{x = x + 1;}\{x > 3\}}$$

Conséquence droite

De même, il est correct d'affaiblir une post-condition :

$$\text{consD}\ \frac{Q' \Rightarrow Q \qquad \{P\}\texttt{c}\{Q\}'}{\{P\}\texttt{c}\{Q\}}$$

Exemple :

$$\text{consD}\ \frac{x > 3 \Rightarrow x > 1 \qquad \{x > 2\}\texttt{x = x + 1;}\{x > 3\}}{\{x > 2\}\texttt{x = x + 1;}\{x > 1\}}$$

Retour sur l'exemple

On a $x > 2 \Rightarrow 1 > 0$ donc à partir de $\{1 > 0\}$`y = 1;`$\{y > 0\}$ on déduit $\{x > 2\}$`y = 1;`$\{y > 0\}$

De même $\bot \Rightarrow -1 > 0$ donc à partir de $\{-1 > 0\}$`y` $= -1;\{y > 0\}$ on déduit $\{\bot\}$`y = -1;`$\{y > 0\}$

On peut donc finaliser la dérivation qui montre que $\{x > 2\}$`if (x>2) y = 1; else y = -1;`$\{y > 0\}$ est valide

Boucle : intuition

On ne connaît a priori pas le nombre d'itération de la boucle

Idée :

- on considère une formule I qui sera vraie à chaque pas de la boucle
- I est appelé l'invariant de boucle
- il faut vérifier
  - que I est bien valide en début de boucle
  - que I est préservé lors de l'exécution de la boucle

Boucle

$$\text{while}\ \frac{\{I \wedge b\}\texttt{c}\{I\}}{\{I\}\texttt{while (b) c}\{I \wedge \neg b\}}$$

- I est appelé l'invariant de la boucle
- I est vrai à chaque tour de boucle (mais pas nécessairement au milieu d'un tour)
- si la boucle s'arrête, la condition `b` est fausse et sa négation peut donc être ajoutée en post-condition
- le corps de la boucle n'est exécuté que si la condition `b` est vraie, donc pour vérifier que l'invariant est conservé après l'exécution de la boucle, on peut ajouter b en précondition

Exercice

Montrez que $\{x \leq 0\}$`while (x <= 5) x = x + 1;`$\{x = 6\}$ est valide.

Indication : vérifier que $x \leq 6$ est un invariant de la boucle.

Exercice

Soit c le programme :

```
{ i = 0;
  s = 0;
  while (i != n) {
    i = i + 1;
    s = s + 2 * i - 1; } }
```

Vérifier que $s = i * i$ est un invariant de la boucle.

Montrer que $\{\top\}\texttt{c}\{s = n * n\}$ est valide (la somme des n premiers nombres impairs est égale à n^2).

Exercice

1. Montrer que pour tout P et tout c, le triplet de Hoare $\{P\}\texttt{c}\{\top\}$ est valide.

2. Montrer que pour tout P, Q et tout c, le triplet de Hoare $\{P\}\texttt{while (1) c}\{Q\}$ est valide.

Automatisme

La logique de Hoare fournit un cadre général dans lequel on peut montrer la correction des programmes.

En pratique, comment l'utiliser ?

- Quelles règles appliquer quand ?
- Comment prouver les implications des règles de conséquence ?

Non-déterminisme

Étant donné un triplet de Hoare, pour essayer de montrer sa validité, on a plusieurs choix :

- Appliquer la règle correspondant à la syntaxe du programme
- Appliquer une règle de conséquence

Non-déterminisme

Étant donné un triplet de Hoare, pour essayer de montrer sa validité, on a plusieurs choix :

- Appliquer la règle correspondant à la syntaxe du programme
  - Comment trouver la propriété intermédiaire pour la séquence ?

$$\text{seq}\ \frac{\{P\}\texttt{c}_1\{Q\} \qquad \{Q\}\texttt{c}_2\{R\}}{\{P\}\{\texttt{c}_1\ \ \texttt{c}_2\}\{R\}}$$

  - Comment trouver l'invariant pour les boucles ?

$$\text{while}\ \frac{\{I \wedge b\}\texttt{c}\{I\}}{\{I\}\texttt{while (b) c}\{I \wedge \neg b\}}$$

- Appliquer une règle de conséquence

Non-déterminisme

Étant donné un triplet de Hoare, pour essayer de montrer sa validité, on a plusieurs choix :

- Appliquer la règle correspondant à la syntaxe du programme
- Appliquer une règle de conséquence
  - Gauche ou droite ?
  - Comment trouver la bonne précondition plus forte, resp. la bonne postcondition plus faible ?

$$\text{consG}\ \frac{P \Rightarrow P' \qquad \{P'\}\mathtt{c}\{Q\}}{\{P\}\mathtt{c}\{Q\}}$$

  - Comment prouver $P \Rightarrow P'$?

Plus faible précondition (*Weakest precondition*)

Définition

Soit c *un programme et* Q *une formule, on note* $\mathsf{WP}(\mathtt{c}, Q)$ *la plus faible précondition telle que si* c *se termine, alors* c *se termine dans un état satisfaisant* Q*.*

Autrement dit :

- C'est une précondition qui mène à Q :
  $\{\mathsf{WP}(\mathtt{c}, Q)\}\mathtt{c}\{Q\}$ est valide
- c'est la plus faible de celles-ci :
  si $\{P\}\mathtt{c}\{Q\}$ est valide, alors $P \Rightarrow \mathsf{WP}(\mathtt{c}, Q)$

Théorème (Correction partielle avec WP)

Pour montrer $\{P\}\mathtt{c}\{Q\}$*, il suffit de montrer* $P \Rightarrow \mathsf{WP}(\mathtt{c}, Q)$*.*

Calcul de WP

Sans les boucles, WP peut être calculée automatiquement :

- $\mathsf{WP}(\texttt{;}, Q) = Q$
- $\mathsf{WP}(\texttt{x = a;}, Q) = [a/x]Q$
- $\mathsf{WP}(\texttt{\{c1 c2\}}, Q) = \mathsf{WP}(\texttt{c1}, \mathsf{WP}(\texttt{c2}, Q))$
- $\mathsf{WP}(\texttt{if (b) c1 else c2}, Q) = (b \Rightarrow \mathsf{WP}(\texttt{c1}, Q)) \wedge (\neg b \Rightarrow \mathsf{WP}(\texttt{c2}, Q))$

Exercice

Calculer :

- WP($\texttt{\{x = x + 1; y = y - 1;\}}, x > y$)
- WP$\left(\begin{array}{l}\texttt{if (x > y) x = x - y;}\\ \texttt{else y = y - x;}\end{array}, (x > 0 \wedge y > 0)\right)$

Montrer en utilisant WP que

$\{x = n\}$ `if (x % 2 == 0) x = x + 2;` `else x = x + 1;` $\{x > n \wedge pair(x)\}$

Cas de la boucle

WP(`while (b) c`, Q) = pas de formule simple !

En effet `while (b) c` est (sémantiquement) équivalent à
`if (b) { c; while (b) c } else {}`

Donc WP(`while (b) c`, Q) =
$$(b \Rightarrow \text{WP}(\texttt{c}, \text{WP}(\texttt{while (b) c}, Q))) \wedge (\neg b \Rightarrow Q)$$

Équation récursive pas toujours possible à calculer

On approxime WP par un prédicat plus simple à calculer et utilisable automatiquement :
la condition de vérification

Condition de vérification

On part d'un programme annoté :

- invariant de boucle
- spécification des fonctions

On note VC(c, Q) la condition de vérification de c pour l'assertion Q
La différence avec WP est qu'on utilise les annotations pour calculer VC

Calcul de VC

- $\mathsf{VC}(\texttt{;}, Q) = Q$
- $\mathsf{VC}(\texttt{x = a;}, Q) = [a/x]Q$
- $\mathsf{VC}(\texttt{\{c1 c2\}}, Q) = \mathsf{VC}(\texttt{c1}, \mathsf{VC}(\texttt{c2}, Q))$
- $\mathsf{VC}(\texttt{if (b) c1 else c2}, Q) = (b \Rightarrow \mathsf{VC}(\texttt{c1}, Q)) \wedge (\neg b \Rightarrow \mathsf{VC}(\texttt{c2}, Q))$
- $\mathsf{VC}(\texttt{while (b) c //@invariant } I, Q) =$

$I \wedge \mathit{Preserv} \wedge \mathit{Final}$ (l'invariant est vérifié en début de boucle)

où

$\mathit{Preserv} = \forall x_1, \ldots, x_m.\ I \wedge b \Rightarrow \mathsf{VC}(\texttt{c}, I)$

(l'invariant est préservé par une itération)

$\mathit{Final} = \forall x_1, \ldots, x_m.\ \neg b \wedge I \Rightarrow Q$

(la postcondition est satisfaite lorsque la boucle se termine)

$x_1 \ldots x_m$ sont les variables modifiées dans c

Exemple

On reprend la factorielle :
$\{x = n \wedge n > 0\}$`{ y = 1; while (x != 1) { y = y * x; x = x - 1; } }`
$\{y = n! \wedge n > 0\}$

On prend comme invariant $Inv = (y * x! = n! \wedge x > 0 \wedge n > 0)$

On a VC(`fact`, $y = n! \wedge n > 0$) =
$1 * x! = n! \wedge x > 0 \wedge n > 0$
$\wedge\Big(\forall xy.\ (Inv \wedge x \neq 1) \Rightarrow (y * x * (x-1)! = n! \wedge x - 1 > 0 \wedge n > 0)\Big)$
$\wedge\Big(\forall xy.\ (x = 1 \wedge Inv) \Rightarrow (y = n! \wedge n > 0)\Big)$

On peut “facilement” montrer que $x = n \wedge n > 0 \Rightarrow$ VC(`fact`, $y = n! \wedge n > 0$)

Correction des conditions de vérification

Lemme

Si $\{P\}\mathtt{c}\{Q\}$ *est valide et* R *ne contient pas de variable libre modifiée par* c*, alors* $\{P \wedge R\}\mathtt{c}\{Q \wedge R\}$ *est valide.*

Théorème

$\{\mathsf{VC}(\mathtt{c}, Q)\}\mathtt{c}\{Q\}$ *est valide.*

Preuve : par induction sur c

Théorème

Si $P \Rightarrow \mathsf{VC}(\mathtt{c}, Q)$ *alors* $\{P\}\mathtt{c}\{Q\}$ *est valide.*

Preuve : condG appliqué au théorème précédent.

Génération de conditions de vérification

Grâce aux programmes annotés et à VC, on obtient une méthode réaliste pour faire de la preuve de programmes impératifs :

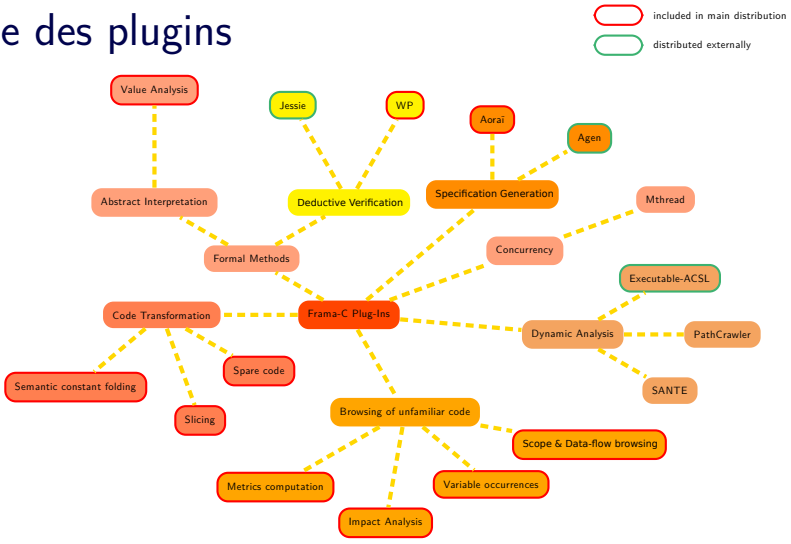
- Écrire un programme contenant les invariants des boucles et éventuellement des assertions à certains point du programme difficiles pour la preuve ;
- Transmettre ce code annoté à un outil qui engendre les conditions de vérification (exemple : plugin WP de Frama-C, cf. TP) ;
- Prouver ces conditions de vérification, de préférence avec un prouveur automatique (alt-ergo, Z3, Vampire, ...) ou interactif (Coq, Isabelle, ...)

Frama-C et WP

Frama-C

- A **fra**mework for **m**odular **a**nalysis of **C** code
- Frama-C : plateforme développé par le CEA et INRIA pour la vérification de programmes C (analyse statique, preuve déductive ...)
- `http://frama-c.com`, distribué sous LGPL (Fluorine v3 en juin 2013)
- Noyau construit sur CIL (Necula et al., Berkeley)
- ACSL langage d'annotations
- Plateforme extensible :
  - Collaboration d'analyseurs sur un même code
  - Communication entre les plugins via les annotations ACSL
  - Ajout facile de nouveaux plugins

Ecosystème des plugins



Crédits : Kosmatov, Signoles

Main plugins of Frama-C ?

- Value analysis Static verification of C code using Abstract Interpretation techniques
- WP
  Static verification of C code using Weakest Precondition calculus
  Jessie, a similar tool
- A lot of other plugins useful in specific cases
  InOut (computation of outputs from inputs), Metrics (analyze code complexity), Aorai (temporal verification), PathCrawler (test generation), Spare code (remove spare code), ...

Contrat de fonctions : les principes

Objectif : spécification de programmes/fonctions impératifs

Approche : donner des propriétés sur les fonctions

- la pré-condition est supposée vraie à l'entrée dans la fonctions (assurée par l'appelant de la fonction)
- la post-condition doit être vraie à la sortie de la fonction (assurée par la fonction si elle termines)
- rien n'est garanti si la pré-condition n'est pas satisfaite
- terminaison peut être garantie ou non (correction totale vs correction partielle)

Annotations en ACSL

`requires`	introduit une pré-condition
	`requires n>=0;`
`\result`	représente le résultat de la fonction
`ensures`	introduit une post-condition
	`ensures \result>=0;`
`assigns`	précise les (locations mémoire des) variables
	que la fonction a le droit de modifier
	en dehors de ses variables locales
	`assigns \nothing ;`
`loop invariant`	invariant de boucle
`loop variant`	variant
`loop assigns`	spécifie les variables modifiées par la boucle

Validité des emplacements mémoire

`\valid` précise la validité des emplacements-mémoire
`\valid(p)` : validité de `*p`
`\valid(p+(x..y))` : validité de `*(p+x) .. *(p+y)`

Ce prédicat peut s'utiliser dans un contrat ou toute assertion (invariant de boucle)

Spécification par cas, behaviors

Pour spécifier plusieurs cas possibles (behaviors)

Dans chaque *behavior* : la clause `assumes` détermine dans quel cas un *behavior* s'applique. La clause `ensures` spécise le comportement dans ce cas.

On peut aussi avoir une post-condition qui s'applique à tous les cas.

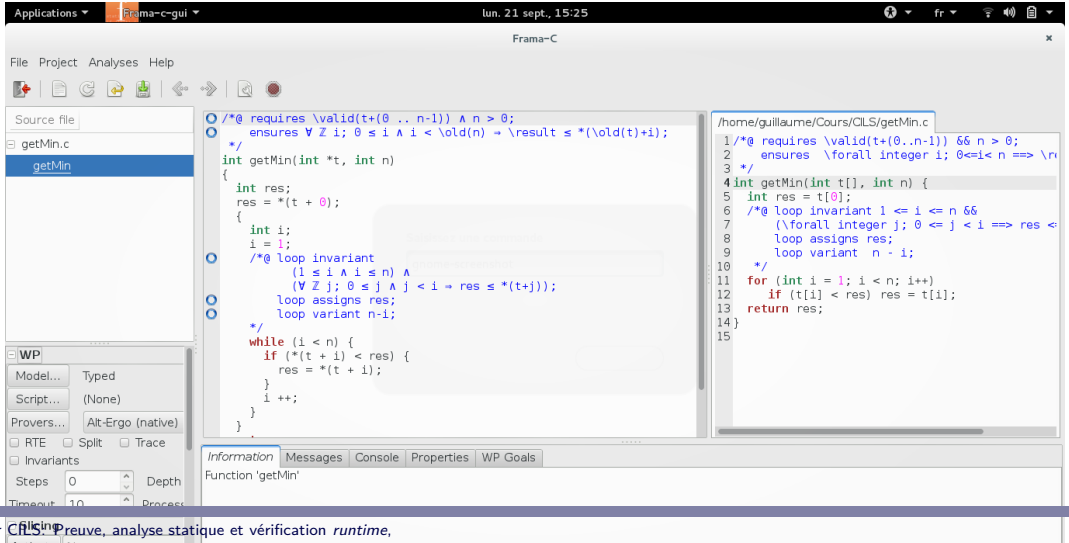
```
/*@ requires -100 <= x <= 100;
assigns \nothing;
behavior pos: assumes x >= 0;
              ensures \result == x;
behavior neg: assumes x < 0;
              ensures \result == -x; */
```

Behaviors (cont.)

- `complete behaviors` : les comportements décrits couvrent tous les cas
- `disjoint behaviors` : les comportements ne se recouvrent pas.

Ces deux clauses génèrent des obligations de preuve.

Interface graphique



Interface textuelle

```
frama-c -wp getMin.c
```

ou

```
frama-c -wp getMin.c -wp-prover XX
```

avec XX $\in$ {z3, cvc3, simplify, coq ..} = ensemble des prouveurs disponibles.

```
frama-c -wp find.c -wp-prover cvc3

[kernel] preprocessing with "gcc -C -E -I.  find.c"
[wp] Running WP plugin...
[wp] Collecting axiomatic usage
[wp] warning: Missing RTE guards
[wp] 9 goals scheduled
[wp] [Qed] Goal typed_find_loop_inv_established : Valid
[wp] [Qed] Goal typed_find_loop_assign : Valid
[wp] [Qed] Goal typed_find_loop_term_decrease : Valid
[wp] [Cvc3] Goal typed_find_complete_not_found_found : Valid (20ms)
[wp] [Qed] Goal typed_find_loop_term_positive : Valid
[wp] [Cvc3] Goal typed_find_loop_inv_preserved : Valid (20ms)
[wp] [Cvc3] Goal typed_find_disjoint_not_found_found : Valid (20ms)
[wp] [Cvc3] Goal typed_find_found_post : Valid (Qed:4ms) (20ms)
[wp] [Cvc3] Goal typed_find_not_found_post : Valid (720ms)
```

Un exemple : getMin

Spec informelle : Calculer le plus petit élément d'un tableau non vide

```
/*@ requires \valid(t+(0..n-1)) && n > 0;
    ensures  \forall integer i; 0<=i< n ==> \result<=t[i];
 */
int getMin(int t[], int n) {
  int res = t[0];
  /*@ loop invariant 1 <= i <= n &&
      (\forall integer j; 0 <= j < i ==> res <= t[j]);
      loop assigns res;
      loop variant  n - i;
   */
  for (int i = 1; i < n; i++)
     if (t[i] < res) res = t[i];
  return res;
}
```

WP : OK

Mais la spécification n'est pas complète ...

```
/*@ requires \valid(t+(0..n-1)) && n > 0;
    ensures  \forall integer i; 0<=i< n ==> \result<=t[i];
 */
int getMin_wrong(int t[], int n) {
  int res = t[0];
  /*@ loop invariant 1 <= i <= n &&
      (\forall integer j; 0 <= j < i ==> res <= t[j]);
      loop assigns res;
      loop variant  n - i;
   */
  for (int i = 1; i < n; i++)
     if (t[i] < res) res = t[i];
  return (res-1);
}
```

WP : OK

```
/*@  requires \valid(t+(0..n-1)) && n > 0;
     ensures \forall integer i; 0<=i< n ==> \result<=t[i];
---> ensures \exists integer k; 0<=k<n && \result == t[k];
 */
int getMin(int t[], int n) {
  int res = t[0];
  /*@ loop invariant 1 <= i <= n &&
      (\forall integer j; 0 <= j < i ==> res <= t[j]);
-->   loop invariant
      \exists integer k; 0<=k<i && res == t[k];
      loop assigns res;
      loop variant  n - i;
   */
  for (int i = 1; i < n; i++)
     if (t[i] < res) res = t[i];
  return res;
}
```

WP : OK

Attention à ce que l'on prouve ...

Les entiers : C et ACSL

ACSL utilise des entiers mathématiques (integer) alors qu'à l'exécution C utilise des entiers machine (int)

On peut utiliser les entiers bornés si besoin.

On peut aussi vérifier les overflows (RTE - runtime error).