# Approches formelles pour la vérification de programmes

Catherine Dubois et Guillaume Burel (ENSIIE)

Master 2 CNS

# Preuve de programmes impératifs

# 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 `{` $\mathtt{i}_1$ `...` $\mathtt{i}_n$ `}`
- Conditionnelle **`if`** `(c)` $\mathtt{i}_1$ **`else`** $\mathtt{i}_2$
- 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

$\{P\}\mathtt{c}\{Q\}$ 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 $\{P\}$`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 :

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

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

# Plus faible précondition (*Weakest precondition*)

## Définition

*Soit* c *un programme et* $Q$ *une formule, on note* $\mathsf{WP}(\mathsf{c}, Q)$ *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 $Q$ :
  $\{\mathsf{WP}(\mathsf{c}, Q)\}\mathsf{c}\{Q\}$ est valide
- c'est la plus faible de celles-ci :
  si $\{P\}\mathsf{c}\{Q\}$ est valide, alors $P \Rightarrow \mathsf{WP}(\mathsf{c}, Q)$

## Théorème (Correction partielle avec WP)

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

# Calcul de WP

Sans les boucles, WP peut être calculée automatiquement :

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

# Exercice

Calculer :

- WP(`{x = x + 1; y = y - 1;}`, $x > y$)
- WP(`if (x > y) x = x - y; else y = y - x;`, $(x > 0 \wedge y > 0)$)

```
if (x > y) x = x - y;
else y = y - x;
```

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

$\mathsf{WP}(\texttt{while (b) c}, Q) =$ pas de formule simple !

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

Donc $\mathsf{WP}(\texttt{while (b) c}, Q) =$
$$(b \Rightarrow \mathsf{WP}(\texttt{c}, \mathsf{WP}(\texttt{while (b) c}, Q))) \land (\neg b \Rightarrow Q)$$

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$(\mathtt{c}, Q)$ 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

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

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

où

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

(l'invariant est préservé par une itération)

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

(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 :

$\{x = n \land n > 0\}$

`{ y = 1; while (x != 1) { y = y * x; x = x - 1; } }`

$\{y = n! \land n > 0\}$

On prend comme invariant

$Inv = (y * x! = n! \land x > 0 \land n > 0)$

On a VC($\texttt{fact}, y = n! \land n > 0$) =

$1 * x! = n! \land x > 0 \land n > 0$

$\land\Big(\forall xy.\ (Inv \land x \neq 1) \Rightarrow (y * x * (x-1)! = n! \land x - 1 > 0 \land n > 0$

$\land\Big(\forall xy.\ (x = 1 \land Inv) \Rightarrow (y = n! \land n > 0)\Big)$

On peut "facilement" montrer que

$x = n \land n > 0 \Rightarrow$ VC($\texttt{fact}, y = n! \land n > 0$)

# 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 :

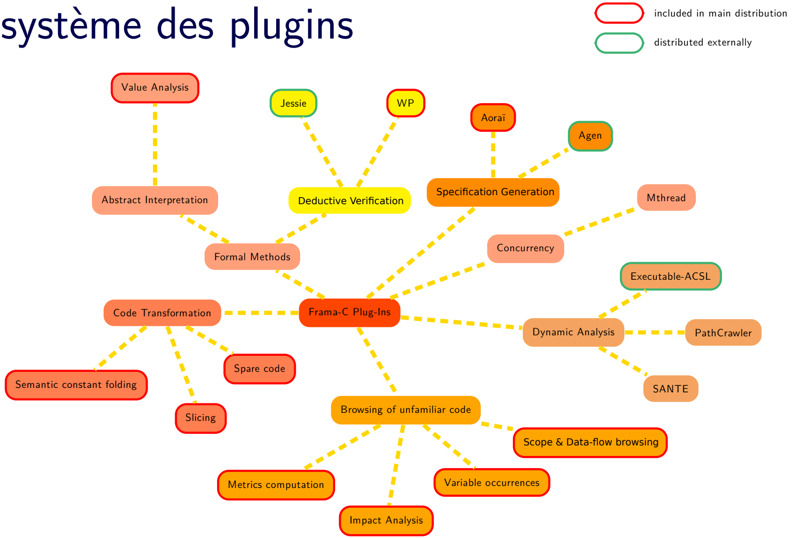
- Écrire un programme contenant les invariants des boucles et éventuellement des assertions à certains point du programme difficiles pour la preuve ;
- Transmettre ce code annoté à un outil qui engendre les conditions de vérification (exemple : plugin WP de Frama-C, cf. TP) ;
- Prouver ces conditions 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 (v21.1 Scandium en juin 2020)
- 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.

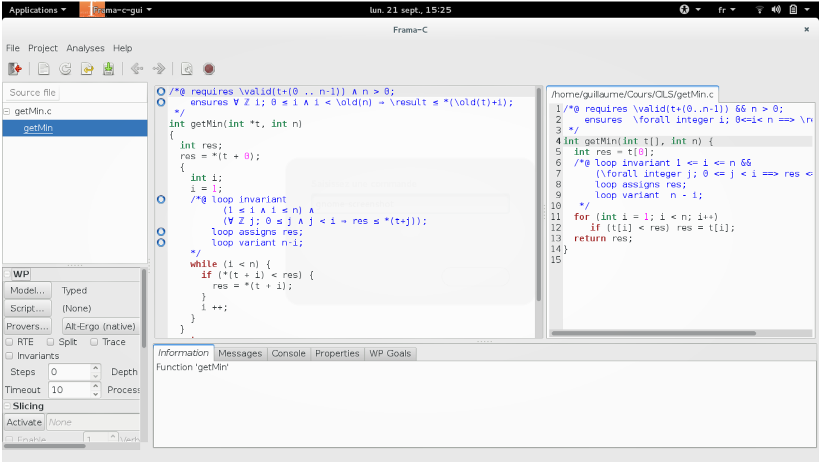
```
/*@ requires -100 <= x <= 100;
assigns \nothing;
behavior pos: assumes x >= 0;
              ensures \result == x;
behavior neg: assumes x < 0;
              ensures \result == -x; */
```

# 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 : Val
[wp] [Qed] Goal typed_find_loop_term_positive : Valid
[wp] [Cvc3] Goal typed_find_loop_inv_preserved : Valid (20m
[wp] [Cvc3] Goal typed_find_disjoint_not_found_found : Val
[wp] [Cvc3] Goal typed_find_found_post : Valid (Qed:4ms) (2
[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<=
 */
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).