

# Vérification logicielle

CY Tech ING2 GSI

# Contexte

**Historique** : les erreurs logiciels (bugs) jalonnent l'histoire de l'informatique et sont autant de d'arguments en faveur du test et de la vérification logiciel.

- calculateurs des voitures
- logiciels bancaires (2006, 2010)
- logiciels critiques (Ariane 5, Mars Climate Orbiter,...)
- logiciels très critiques (Therac-25, missiles Patriot,...)

**Chiffres** :

- Coût des bugs : 64 milliards \$/an aux US
- impacts humains, environnementaux,...
- Coût du test logiciel : 10 milliards \$/an aux US
- Mais les tests ne suffisent pas :

**E.W. Dijkstra (Notes on Structured Programming, 1972)**

**« Testing can only reveal the presence of errors but never their absence. »**

# Logique de Hoare

Objectif : la logique de Hoare sert à formaliser la preuve de la correction des programmes informatiques :

`<precondition> programme <postcondition>`

Signification : si la precondition est vraie alors, après l'exécution du programme, la postcondition est vraie.

Méthode: utilisation de règles de déductions logiques.

# Rappel : logique des prédicats

Langage d'alphabet  $\{V, C, F, P, L\}$

- V : variables,
- C : constantes
- F : fonctions de  $\{C \times V\} \rightarrow \{C \times V\}$  d'arité quelconque
- P : prédicats de  $\{C \times V\} \rightarrow \text{Bool}$  d'arité quelconque
- L : connecteurs entre prédicats :
  - ! : négation
  - || : ou logique non exclusif
  - && : et logique
  - $\Rightarrow$  : implication logique
  - $\forall$  : quantificateur universel
  - $\exists$  : quantificateur existentiel

# Triplets de Hoare

Forme générale :

$$\{p\}S\{q\}$$

avec p et q des formules de la logique des prédicats et S un programme

Exemples :

$$\{x \geq 0\} \quad x = x + 1; \quad \{x > 0\}$$

$$\{(x > 0) \&\& (y \geq 0)\} \quad z = y/x; \quad \{(x > 0) \&\& (y \geq 0) \&\& (z \geq 0)\}$$

# Règles de déductions

Pour démontrer la validité d'un triplet de Hoare nous allons utiliser des règles de déductions qui auront toutes la même forme :

$$\begin{array}{c} \text{premisses} \\ | \\ \text{conclusions} \end{array}$$

Signification: si toutes les prémisses sont vraies, la conclusion est vraie

Résultat: un arbre de déduction dont les feuilles sont des axiomes (formule toujours vraie dans la logique de Hoare) et la racine le triplet de Hoare à démontrer.

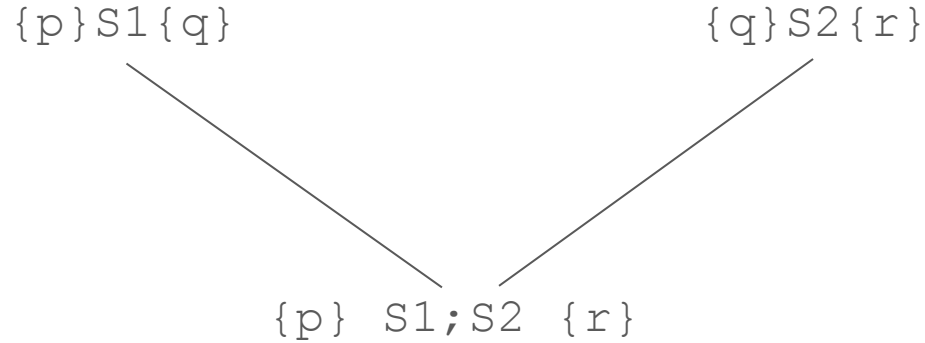
# Axiome

$$\{p(t)\} \quad x=t; \quad \{p(x)\}$$

Les axiomes sont les feuilles de l'arbre de démonstration.

Ils sont toujours vrais dans la logique de hoare, il faut juste trouver la bonne valeur de  $p$ , et sont utilisés pour chaque instruction d'affectation =

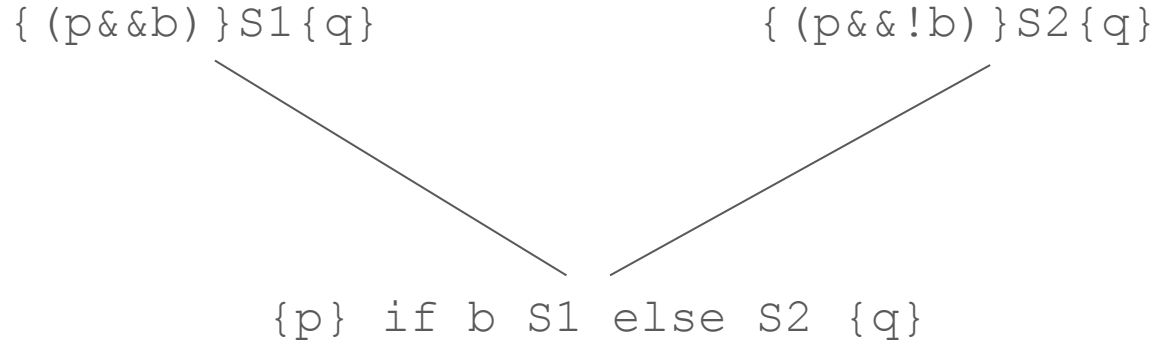
## Composition :



La composition permet de transformer une preuve en deux sous preuves plus petites, elle est utilisée lors de la séquence de deux instructions avec ;

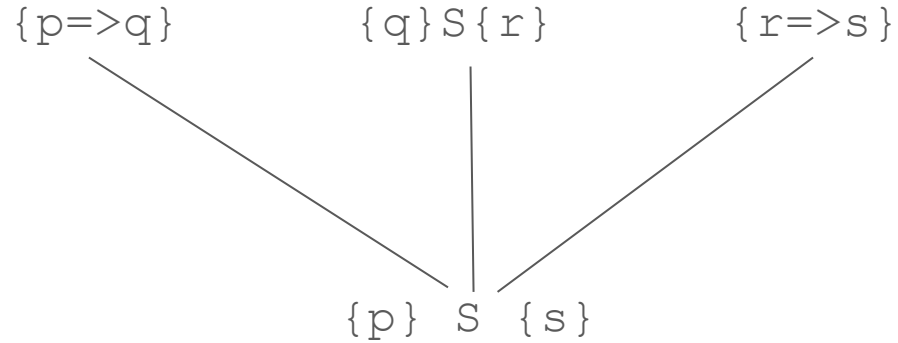


## Conditionnelle :




La conditionnelle permet de traiter les instructions de branchement en fonction d'un booléen  $b$ , en séparant également la preuve en deux preuves plus petites

# Conséquences :



La conséquence permet de modifier les preconditions et postconditions en formule équivalentes

## Boucle :

$$\{ (p \& \& b) \} S \{ p \}$$

$$\{ p \} \text{ while } b \text{ S } \{ p \& \& !b \}$$

La boucle permet de traiter les itérations en spécifiant la valeur de  $p$ , invariant de boucle.

## Exemple de démonstration :

```
{ (val ≥ 0) //pre_condition  
  
{cpt=val; res=0; while(cpt>0) {res=res+val; cpt=cpt-1}} //programme  
  
{ (res=val2) } //post-condition
```

Objectif : trouver un arbre de démonstration dont la racine est le triplet ci-dessus.

# Cohérence du système

Le système présenté est cohérent (tout ce qu'il prouve est vrai)

Il n'est pas complet : certaines formules vraies ne sont pas prouvables

Il permet uniquement d'établir la correction partielle d'un programme (si le programme termine, alors le résultat produit est correct).

Pour avoir la correction totale, il faut prouver sa terminaison.

Problème: question en générale indécidable (cf cours de décidabilité :-)

En pratique, il est souvent possible de prouver qu'un programme termine :

**variant de boucle**

# Variant de boucle

Idée : Trouver une variable  $v$  entière appelée variant de boucle telle que

- $v$  a une valeur positive avant la boucle et après chaque itération
- $v$  diminue à chaque itération

Exemples :

```
static void loop1(int n) {  
    //@ loop_variant = n ;  
    while (n > 0) n--; }
```

```
static void loop2(int n) {  
    //@ loop_variant = 100-n ;  
    while (n < 100) n++; }
```