Des modèles à la preuve de programme

Petit tour d'horizon des méthodes formelles

Nicolas Stouls

INSA Lyon, CITI
Nicolas.Stouls@inria.fr

Date : *trop tôt en 2009*

Introduction

Definition (Génie logiciel)

Art de la maîtrise des coûts, de la complexité et de la qualité des développements.

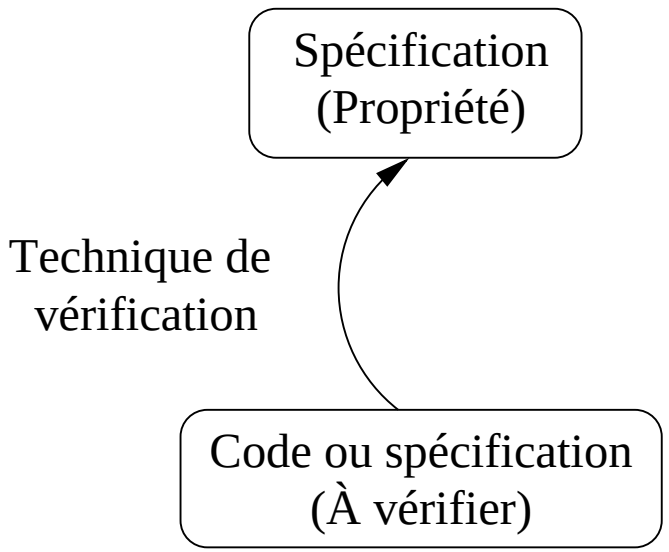
Méthodes formelles

- Méthodes de génie logiciel
- Haut niveau de confiance
- Doit réunir 3 éléments :
  - Langage

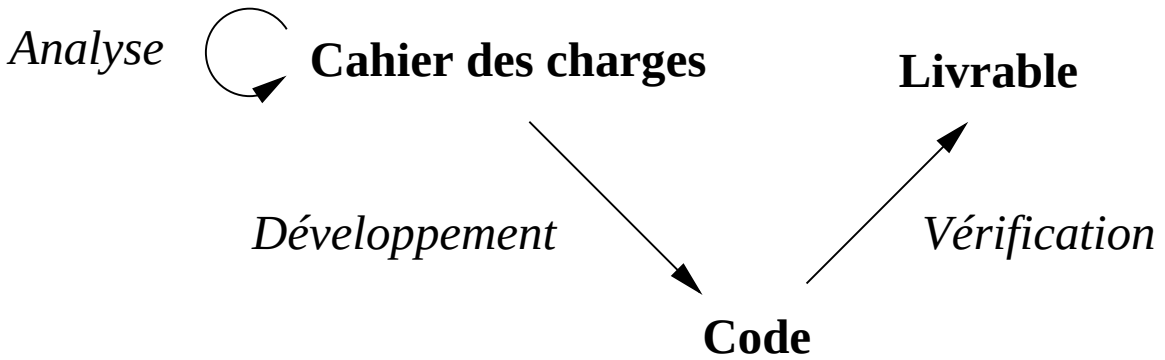
(expressif et à la sémantique bien définie)

  - Technique(s) de vérification
  - Outil

(pour une correcte utilisation des règles)



Cycle de développement du logiciel



Durant la phase d'analyse

Assistance à la définition des besoins

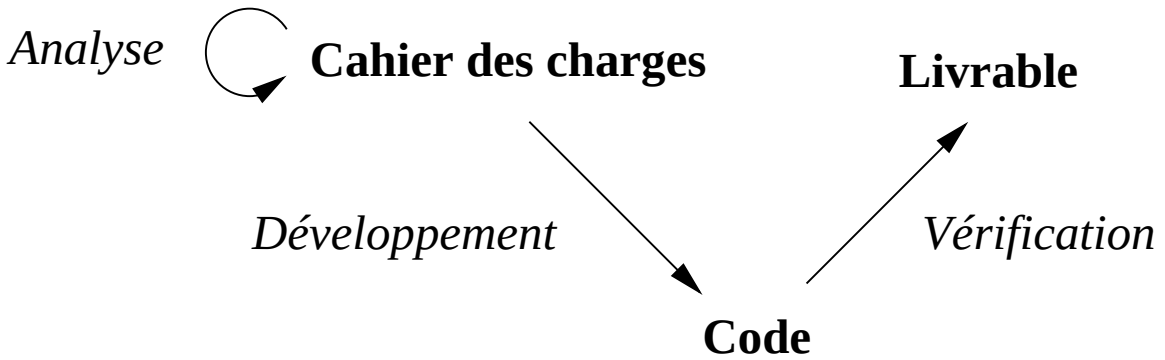
Prérequis Cahier des charges en langue naturelle

Objectif Lever les ambiguïtés et les incohérences

Exemple GL UML : *description graphique d'un système*

Exemple MF OCL : *propriétés logiques sur diagrammes UML*

Cycle de développement du logiciel



Durant la phase de développement

Logiciels corrects par construction

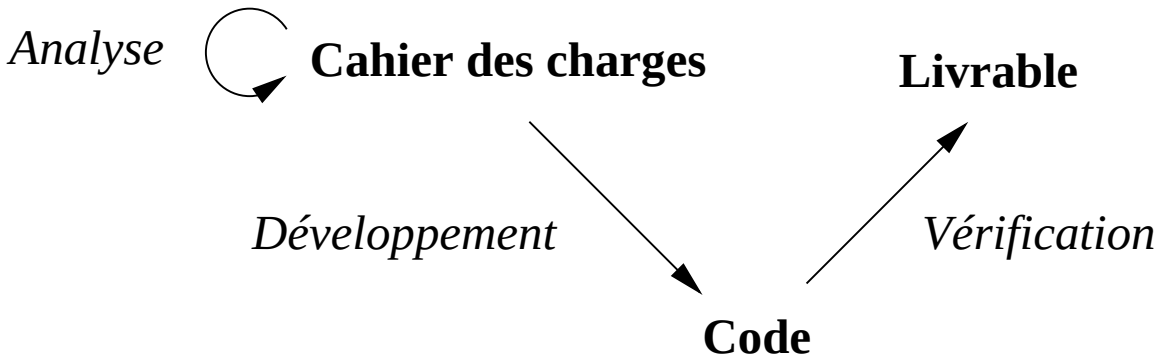
Prérequis Spécification formelle *(pas de code)*

Objectif produire un code correspondant à la spécification

Exemple GL IDM : *dérivation de code à partir d'un modèle*

Exemple MF Raffinement : *description incrémentale d'un système*

Cycle de développement du logiciel



Durant la phase de vérification

Vérification a posteriori

Prérequis Spécification formelle et code

Objectif vérifier la cohérence code/spec

Exemple GL Tests

Exemple MF Simulation symbolique

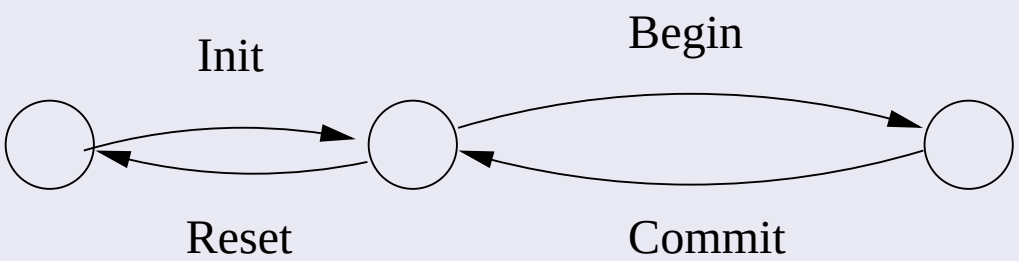
Langages de spécification

Graphique

- Comportemental *(comportements et espace d'états)*
- Organisationnel *(dépendance et répartition des données)*

Textuel

- Algèbre de processus *(description comportementale)*
- Fonctionnel *(description de la sortie en terme des entrées)*
- Transformationnel *(orienté données)*



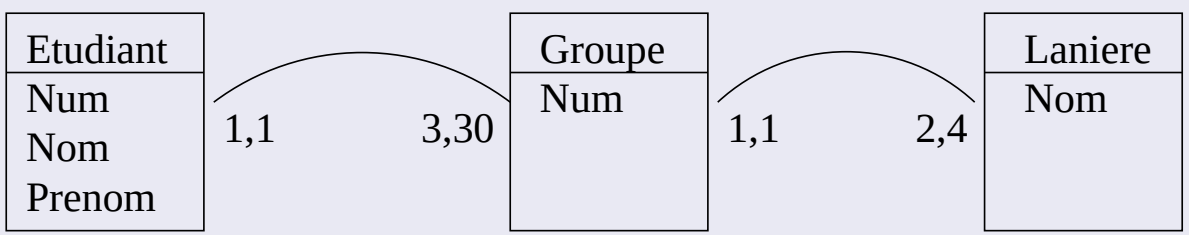
Langages de spécification

Graphique

- Comportemental *(comportements et espace d'états)*
- Organisationnel *(dépendance et répartition des données)*

Textuel

- Algèbre de processus *(description comportementale)*
- Fonctionnel *(description de la sortie en terme des entrées)*
- Transformationnel *(orienté données)*



Langages de spécification

Graphique

- Comportemental *(comportements et espace d'états)*
- Organisationnel *(dépendance et répartition des données)*

Textuel

- Algèbre de processus *(description comportementale)*
- Fonctionnel *(description de la sortie en terme des entrées)*
- Transformationnel *(orienté données)*

$$\mathcal{X} \triangleq x' = x + 1 \ \wedge \ y' = y$$
$$\mathcal{Y} \triangleq y' = y + 1 \ \wedge \ x' = x$$
$$\phi \triangleq (x = 0) \ \wedge \ (y = 0) \ \wedge \ \Box[\mathcal{X} \ \vee \ \mathcal{Y}]$$

Langages de spécification

Graphique

- Comportemental *(comportements et espace d'états)*
- Organisationnel *(dépendance et répartition des données)*

Textuel

- Algèbre de processus *(description comportementale)*
- Fonctionnel *(description de la sortie en terme des entrées)*
- Transformationnel *(orienté données)*

$$\backslash\textbf{ensures}\ \forall i \in 1..9 \Rightarrow \backslash result[i] \leq \backslash result[i+1]$$

Langages de spécification

Graphique

- Comportemental *(comportements et espace d'états)*
- Organisationnel *(dépendance et répartition des données)*

Textuel

- Algèbre de processus *(description comportementale)*
- Fonctionnel *(description de la sortie en terme des entrées)*
- Transformationnel *(orienté données)*

$res \longleftarrow sort(tab) \mathrel{\hat{=}}$ **PRE** $tab \subseteq \mathbb{N} \rightarrow \mathbb{N}$ **THEN**
ANY T **WHERE** $T \in \mathrm{dom}(tab) \rightarrow \mathrm{ran}(tab)$
$\wedge\ (\forall i, j \in \mathrm{dom}(T)\ \wedge\ i \leq j \Rightarrow T[i] \leq T[j])$
THEN $res := T$
END END

Langages de spécification

Finalement ?

- Sert à décrire un ensemble de propriétés
- Tout langage peut être utilisé
- Seules contraintes :
  - Suffisamment expressif
  - Non ambiguë *($\neq$ non-déterministe)*
- Un sous ensemble de C pourrait être un langage formel
- Le français aussi ....

A quoi ça sert ?

- 1 modèle = 1 aspect du système complet *(très compact)*
- Objectif : vérifier que le système respecte les propriétés

Langages de spécification

Finalement ?

- Sert à décrire un ensemble de propriétés
- Tout langage peut être utilisé
- Seules contraintes :
  - Suffisamment expressif
  - Non ambiguë *($\neq$ non-déterministe)*
- Un sous ensemble de C pourrait être un langage formel
- Le français aussi ....

A quoi ça sert ?

- 1 modèle = 1 aspect du système complet *(très compact)*
- Objectif : vérifier que le système respecte les propriétés

Principales techniques de vérification statique

Globalement : 4 familles

- Simulation (Test statique)
- Model checking
- Interprétation abstraite
- Preuve de programme

Simulation (Test statique fonctionnel)

Principe

Question Est-ce que ça semble correct ?

1. Construction du flot de contrôle de la propriété
2. Choix d'un critère de couverture
3. Énumération de l'ensemble des chemins
4. Caractérisation des inputs de chaque chemin
5. Simulation d'un test au moins par chemin

Bilan

Difficultés Complétude, choix de données pertinentes, lourdeur de l'interface (bouchons, prémisses, ...)

Garanties Faisabilité et correction **pour les entrées testées**

Model checking

Principe

Question Existe-t-il un contre exemple ?

Exemple d'une propriété LTL :

1. Construction de l'automate du programme
2. Construction du négatif de l'automate de la propriété
3. Faire produit synchrone des 2
4. Résultat : ensemble des traces violant la propriété

Bilan

Difficultés Construction des automates, passage à l'échelle
*(1 état = 1 valuation de **chaque** variable)*

Garanties Absence de trace = preuve de correction
Sinon : exemple violant la propriété

Interprétation abstraite

Principe

Question La même en plus simple ?

1. Construction d'une abstraction du programme
2. Vérification de la propriété sur l'abstraction

Exemple : Calcul d'intervalles de valeurs

Bilan

Difficultés Choix de **bonnes** fonctions d'abstraction et de concrétisation (connexions de Galois), nécessite une seconde technique de vérification

Garanties Correction du programme par rapport à la propriété

Preuve de programme

Principe

Question La propriété est elle une abstraction du programme ?

1. Décomposer en formules logiques (CV/OP)
2. Chaque CV est elle satisfaisable ?

Bilan

Difficultés Indécidabilité de la logique du premier ordre (SAT)

Garanties Correction du programme par rapport à la propriété

Preuve de programme : 2 parties

Notion de correction partielle

Correction du programme par rapport à sa spécification sous réserve que les instructions terminent

Notion de terminaison

- Associer un compteur aux **boucles** et **appels récursifs**
  - *Compteur doit être borné inférieurement*
  - *Valeur initialement supérieure à cette borne*
  - *Compteur décroît strictement à chaque itération*
- Tout appel respecte la spécification de l'opération appelée

Finalement

Preuve de programme = correction partielle + terminaison

Notions de preuve

Différentes méthodes / notations

- Déduction naturelle
- Calcul de séquents
- Système à la Hilbert

Toujours le même principe

Axiome Vérité jamais remise en question

$$x = x \qquad p \Rightarrow (q \Rightarrow p)$$

Règle d'inférence Règle de transformation logique

$$\frac{p \Rightarrow q \ , \ p}{q} \qquad \frac{(\forall x \in E \Rightarrow P(x)) \ , \ E \neq \emptyset}{\exists x \in E \ \wedge \ P(E)}$$

(Pas toujours réversible)

Logique de Hoare (1)

{P} Instructions {Q}

- P et Q : 2 prédicats logique
- Si P est vrai en entrée et que les instructions terminent alors Q est vérifié en sortie.
- P : précondition / Q : post-condition / (P, Q) : spécification
- Exemples

$$\{x = 0\}\ x := x + 2\ \{x > 0\}$$
$$\{x \in \mathbb{N}\}\ x := x * 2\ \{x\%2 = 0\}$$

Principe général

- instructions du langage $\rightsquigarrow$ axiomes / règles d'inférences
- la correction partielle est prouvée par résolution des triplets
- la terminaison est prouvée en considérant le code outillé

Logique de Hoare (2)

Règles simples

- Renforcement de la précondition

$$\frac{P' \Rightarrow P \quad , \quad \{P\}\ I\ \{Q\}}{\{P'\}\ I\ \{Q\}}$$

- Affaiblissement de la post-condition

$$\frac{Q \Rightarrow Q' \quad , \quad \{P\}\ I\ \{Q\}}{\{P\}\ I\ \{Q'\}}$$

WP/SP

WP Plus faible précondition (Weakest Precondition)

SP Plus forte post-condition (Strongest Postcondition)

$$\frac{WP(I,Q),\ \{P\}\ I\ \{Q\}}{P \Rightarrow WP(I,Q)} \qquad \frac{SP(P,I),\ \{P\}\ I\ \{Q\}}{SP(P,I) \Rightarrow Q}$$

Logique de Hoare (2)

Règles simples

- Renforcement de la précondition

$$\frac{P' \Rightarrow P \quad , \quad \{P\}\ I\ \{Q\}}{\{P'\}\ I\ \{Q\}}$$

- Affaiblissement de la post-condition

$$\frac{Q \Rightarrow Q' \quad , \quad \{P\}\ I\ \{Q\}}{\{P\}\ I\ \{Q'\}}$$

WP/SP

WP Plus faible précondition (Weakest Precondition)

SP Plus forte post-condition (Strongest Postcondition)

$$\frac{WP(I,Q),\ \{P\}\ I\ \{Q\}}{P \Rightarrow WP(I,Q)} \qquad \frac{SP(P,I),\ \{P\}\ I\ \{Q\}}{SP(P,I) \Rightarrow Q}$$

De la logique de hoare à la preuve de programmes

Étant donné un programme spécifié $\{P\}\ I\ \{Q\}$...

Si l'on sait calculer *WP(I,Q)*

Il faut vérifier que :

- P est une précondition

$$P \Rightarrow WP(I,Q)$$

Si l'on sait calculer *SP(P,I)*

Il faut vérifier que :

- Q est une post-condition

$$SP(P,I) \Rightarrow Q$$

Dans la pratique

le calcul le plus efficace est celui de la précondition

Calcul de la plus faible précondition (Dijkstra)

Aussi appelé calcul des substitutions *WP(I,Q)* $\hat{=} [I]Q$

- **de droite à gauche**
- 1 règle de substitution par instruction

$$[x := E]Q \;\hat{=}\; \textit{Remplacement de x par E dans Q}$$

$$[\textbf{IF } c \textbf{ THEN } I_1 \textbf{ ELSE } I_2 \textbf{ END}]Q \;\hat{=}\; (c \Rightarrow [I_1]Q) \wedge (\neg c \Rightarrow [I_2]Q)$$

Exemples

$[x := E](x > 3) \Leftrightarrow E > 3$

$[\textbf{IF } x > 0 \textbf{ THEN } x := x - 2 \textbf{ ELSE } x := x + 1 \textbf{ END}]x > 0$

$\Leftrightarrow (x > 0 \Rightarrow [x := x - 2]x > 0) \wedge (x \leq 0 \Rightarrow [x := x + 1]x > 0)$

$\Leftrightarrow (x > 0 \Rightarrow x - 2 > 0) \wedge (x \leq 0 \Rightarrow x + 1 > 0)$

$\Leftrightarrow (x \leq 0 \vee x - 2 > 0) \wedge (x > 0 \vee x = 0)$

$\Leftrightarrow x = 0 \vee x > 2$

Calcul de la plus faible précondition (Dijkstra)

Aussi appelé calcul des substitutions *WP(I,Q)* $\hat{=} [I]Q$

- **de droite à gauche**
- 1 règle de substitution par instruction

$$[x := E]Q \;\hat{=}\; \textit{Remplacement de x par E dans Q}$$

$$[\textbf{IF } c \textbf{ THEN } I_1 \textbf{ ELSE } I_2 \textbf{ END}]Q \;\hat{=}\; (c \Rightarrow [I_1]Q) \wedge (\neg c \Rightarrow [I_2]Q)$$

Exemples

$[x := E](x > 3) \Leftrightarrow E > 3$

$[\textbf{IF } x > 0 \textbf{ THEN } x := x - 2 \textbf{ ELSE } x := x + 1 \textbf{ END}]x > 0$

$\Leftrightarrow (x > 0 \Rightarrow [x := x - 2]x > 0) \wedge (x \leq 0 \Rightarrow [x := x + 1]x > 0)$

$\Leftrightarrow (x > 0 \Rightarrow x - 2 > 0) \wedge (x \leq 0 \Rightarrow x + 1 > 0)$

$\Leftrightarrow (x \leq 0 \vee x - 2 > 0) \wedge (x > 0 \vee x = 0)$

$\Leftrightarrow x = 0 \vee x > 2$

Calcul de la plus faible précondition (Dijkstra)

Aussi appelé calcul des substitutions *WP(I,Q)* $\hat{=} [I]Q$

- **de droite à gauche**
- 1 règle de substitution par instruction

$$[x := E]Q \;\hat{=}\; \textit{Remplacement de x par E dans Q}$$

$$[\textbf{IF } c \textbf{ THEN } I_1 \textbf{ ELSE } I_2 \textbf{ END}]Q \;\hat{=}\; (c \Rightarrow [I_1]Q) \wedge (\neg c \Rightarrow [I_2]Q)$$

Exemples

$[x := E](x > 3) \Leftrightarrow E > 3$

$[\textbf{IF } x > 0 \textbf{ THEN } x := x - 2 \textbf{ ELSE } x := x + 1 \textbf{ END}]x > 0$

$\Leftrightarrow (x > 0 \Rightarrow [x := x - 2]x > 0) \wedge (x \leq 0 \Rightarrow [x := x + 1]x > 0)$

$\Leftrightarrow (x > 0 \Rightarrow x - 2 > 0) \wedge (x \leq 0 \Rightarrow x + 1 > 0)$

$\Leftrightarrow (x \leq 0 \vee x - 2 > 0) \wedge (x > 0 \vee x = 0)$

$\Leftrightarrow x = 0 \vee x > 2$

Calcul de la plus faible précondition (Dijkstra)

Aussi appelé calcul des substitutions *WP(I,Q)* $\hat{=} [I]Q$

- **de droite à gauche**
- 1 règle de substitution par instruction

$$[x := E]Q \;\hat{=}\; \textit{Remplacement de x par E dans Q}$$

$$[\textbf{IF } c \textbf{ THEN } I_1 \textbf{ ELSE } I_2 \textbf{ END}]Q \;\hat{=}\; (c \Rightarrow [I_1]Q) \wedge (\neg c \Rightarrow [I_2]Q)$$

Exemples

$[x := E](x > 3) \Leftrightarrow E > 3$

$[\textbf{IF } x > 0 \textbf{ THEN } x := x - 2 \textbf{ ELSE } x := x + 1 \textbf{ END}]x > 0$

$\Leftrightarrow (x > 0 \Rightarrow [x := x - 2]x > 0) \wedge (x \leq 0 \Rightarrow [x := x + 1]x > 0)$

$\Leftrightarrow (x > 0 \Rightarrow x - 2 > 0) \wedge (x \leq 0 \Rightarrow x + 1 > 0)$

$\Leftrightarrow (x \leq 0 \vee x - 2 > 0) \wedge (x > 0 \vee x = 0)$

$\Leftrightarrow x = 0 \vee x > 2$

Calcul de la plus faible précondition (Dijkstra)

Aussi appelé calcul des substitutions *WP(I,Q)* $\hat{=} [I]Q$

- **de droite à gauche**
- 1 règle de substitution par instruction

$$[x := E]Q \;\hat{=}\; \textit{Remplacement de x par E dans Q}$$

$$[\textbf{IF } c \textbf{ THEN } I_1 \textbf{ ELSE } I_2 \textbf{ END}]Q \;\hat{=}\; (c \Rightarrow [I_1]Q) \wedge (\neg c \Rightarrow [I_2]Q)$$

Exemples

$[x := E](x > 3) \Leftrightarrow E > 3$

$[\textbf{IF } x > 0 \textbf{ THEN } x := x - 2 \textbf{ ELSE } x := x + 1 \textbf{ END}]x > 0$

$\Leftrightarrow (x > 0 \Rightarrow [x := x - 2]x > 0) \wedge (x \leq 0 \Rightarrow [x := x + 1]x > 0)$

$\Leftrightarrow (x > 0 \Rightarrow x - 2 > 0) \wedge (x \leq 0 \Rightarrow x + 1 > 0)$

$\Leftrightarrow (x \leq 0 \vee x - 2 > 0) \wedge (x > 0 \vee x = 0)$

$\Leftrightarrow x = 0 \vee x > 2$

Calcul de la plus faible précondition (Dijkstra)

Aussi appelé calcul des substitutions *WP(I,Q)* $\hat{=} [I]Q$

- **de droite à gauche**
- 1 règle de substitution par instruction

$$[x := E]Q \;\hat{=}\; \textit{Remplacement de x par E dans Q}$$

$$[\textbf{IF } c \textbf{ THEN } I_1 \textbf{ ELSE } I_2 \textbf{ END}]Q \;\hat{=}\; (c \Rightarrow [I_1]Q) \wedge (\neg c \Rightarrow [I_2]Q)$$

Exemples

$[x := E](x > 3) \Leftrightarrow E > 3$

$[\textbf{IF } x > 0 \textbf{ THEN } x := x - 2 \textbf{ ELSE } x := x + 1 \textbf{ END}]x > 0$

$\Leftrightarrow (x > 0 \Rightarrow [x := x - 2]x > 0) \wedge (x \leq 0 \Rightarrow [x := x + 1]x > 0)$

$\Leftrightarrow (x > 0 \Rightarrow x - 2 > 0) \wedge (x \leq 0 \Rightarrow x + 1 > 0)$

$\Leftrightarrow (x \leq 0 \vee x - 2 > 0) \wedge (x > 0 \vee x = 0)$

$\Leftrightarrow x = 0 \vee x > 2$

Vérification à posteriori (1)

Spécification de programme C « à la » Hoare

- Précondition + post-condition pour chaque opération
- Invariants *(S'ajoute à toute pré/post-condition)*

```
int cpt=3;
/*@ global invariant invcpt :
    0 ≤ cpt ≤ 3; */

int status=0;
/*@ global invariant invst :
    0 ≤ status ≤ 1; */

//@ ensures 0 ≤ \result ≤ 1;
int init() {...}

/*@ requires status == 1
                 && cpt > 0;*/
//@ ensures 0 ≤ \result ≤ 1;
int commit() {...}
```

```
//@ ensures 0 ≤ \result ≤ 1;
int main(){
  /*@ loop invariant i :
    @   0 ≤ status ≤ 1 ∧ 0 ≤ cpt ≤ 3
    @   ∧ (cpt = 0 ⇒ status = 0);
    @ loop variant v :
    @   cpt */
  while (cpt>0) {
    status=init();
    if (status && commit()) goto label_ok;
    cpt--;
  }
  return 0;
  label_ok : return 1;
}
```

Vérification à posteriori (1)

Spécification de programme C « à la » Hoare

- Précondition + post-condition pour chaque opération
- Invariants *(S'ajoute à toute pré/post-condition)*

```
int cpt=3;
/*@ global invariant invcpt :
    0 ≤ cpt ≤ 3; */

int status=0;
/*@ global invariant invst :
    0 ≤ status ≤ 1; */

//@ ensures 0 ≤ \result ≤ 1;
int init() {...}

/*@ requires status == 1
                 && cpt > 0;*/
//@ ensures 0 ≤ \result ≤ 1;
int commit() {...}
```

```
//@ ensures 0 ≤ \result ≤ 1;
int main(){
  /*@ loop invariant i :
    @   0 ≤ status ≤ 1 ∧ 0 ≤ cpt ≤ 3
    @   ∧ (cpt = 0 ⇒ status = 0);
    @ loop variant v :
    @   cpt */
  while (cpt>0) {
    status=init();
    if (status && commit()) goto label_ok;
    cpt--;
  }
  return 0;
  label_ok : return 1;
}
```

Vérification à posteriori (1)

Spécification de programme C « à la » Hoare

- Précondition + post-condition pour chaque opération
- Invariants *(S'ajoute à toute pré/post-condition)*

```
int cpt=3;
/*@ global invariant invcpt :
    0 ≤ cpt ≤ 3; */

int status=0;
/*@ global invariant invst :
    0 ≤ status ≤ 1; */

//@ ensures 0 ≤ \result ≤ 1;
int init() {...}

/*@ requires status == 1
                && cpt > 0;*/
//@ ensures 0 ≤ \result ≤ 1;
int commit() {...}
```

```
//@ ensures 0 ≤ \result ≤ 1;
int main(){
  /*@ loop invariant i :
    @   0 ≤ status ≤ 1 ∧ 0 ≤ cpt ≤ 3
    @   ∧ (cpt = 0 ⇒ status = 0);
    @ loop variant v :
    @   cpt */
  while (cpt>0) {
    status=init();
    if (status && commit()) goto label_ok;
    cpt--;
  }
  return 0;
  label_ok : return 1;
}
```

Vérification à posteriori (1)

Spécification de programme C « à la » Hoare

- Précondition + post-condition pour chaque opération
- Invariants *(S'ajoute à toute pré/post-condition)*

```
int cpt=3;
/*@ global invariant invcpt :
    0 ≤ cpt ≤ 3; */

int status=0;
/*@ global invariant invst :
    0 ≤ status ≤ 1; */

//@ ensures 0 ≤ \result ≤ 1;
int init() {...}

/*@ requires status == 1
                 && cpt > 0;*/
//@ ensures 0 ≤ \result ≤ 1;
int commit() {...}
```

```
//@ ensures 0 ≤ \result ≤ 1;
int main(){
  /*@ loop invariant i :
    @   0 ≤ status ≤ 1 ∧ 0 ≤ cpt ≤ 3
    @    ∧ (cpt = 0 ⇒ status = 0);
    @ loop variant v :
    @   cpt */
  while (cpt>0) {
    status=init();
    if (status && commit()) goto label_ok;
    cpt--;
  }
  return 0;
  label_ok : return 1;
}
```

Vérification à posteriori (2)

Programme non spécifié : que vérifier ?

- Analyse d'intervalle de valeurs
- Non débordement de tableaux
- Non débordement d'entiers
- Pas de division par zéro
- Pas de déréférencement d'un pointeur null
- etc.

Bilan

- Quoi ? hypothèses liées à la sémantique du langage
  (On parle d'axiomatisation du langage)
- Toujours faisable ? Non
  (Plus simple avec une spécification utilisateur)

De Hoare à la construction correcte

Exemple de l'approche B

1. Description d'une spécification abstraite
2. Preuve de sa cohérence interne
3. Description d'un raffinement
4. Preuve de sa cohérence interne
5. Preuve de raffinement avec son abstraction
6. Retour en 2 jusqu'à avoir un programme

Principe de la preuve en B

- Opération = Précondition + modification d'état
- Système = Opérations + invariant
- Preuve = Respect de l'invariant par les opérations

$$I \wedge Pre \Rightarrow [Corps]I$$

De Hoare au raffinement (1)

Le raffinement : kesako ?

- Description de plus en plus fine du système
- Exemple :

Abstraction Opération *O* prend 1 tableau *T* et rend 1 tableau
Raffinement 1 Le tableau rendu par *O* est trié
Raffinement 2 Ce tableau contient les mêmes valeurs que *T*
Implantation Description d'un algorithme de tri

- **Remplacer une opération par un raffinement ne doit pas être perceptible de l'extérieur**
- Plus concrètement :
  - opération raffinée accepte toute entrée valide dans l'abstraction
    $(pre_{abstr} \Rightarrow pre_{raff})$
  - résultat d'opération raffinée doit être prévu dans l'abstraction
    $(post_{raff} \Rightarrow post_{abstr})$

De Hoare au raffinement (1)

Le raffinement : kesako ?

- Description de plus en plus fine du système
- Exemple :

Abstraction Opération O prend 1 tableau T et rend 1 tableau
Raffinement 1 Le tableau rendu par O est trié
Raffinement 2 Ce tableau contient les mêmes valeurs que T
Implantation Description d'un algorithme de tri

- **Remplacer une opération par un raffinement ne doit pas être perceptible de l'extérieur**
- Plus concrètement :

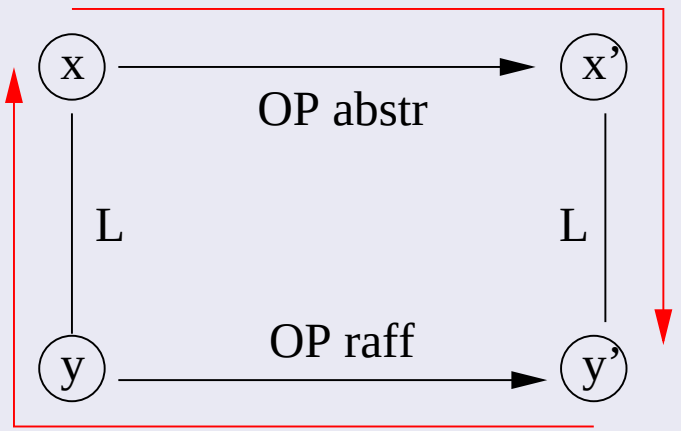
  opération raffinée accepte toute entrée valide dans l'abstraction
  $(pre_{abstr} \Rightarrow pre_{raff})$
  résultat d'opération raffinée doit être prévu dans l'abstraction
  $(post_{raff} \Rightarrow post_{abstr})$

De Hoare au raffinement (2)

Principe de la preuve de raffinement en B

- Preuve = Respect de l'abstraction par chaque opération

$$L \wedge Pre_{abstr} \Rightarrow [Corps_{raff}]\neg[Corps_{abstr}]\neg L$$



$$\forall x, y, y' \cdot \left(\begin{array}{l} (x,y) \in L \quad \wedge \quad (y,y') \in op_{raff} \\ \Rightarrow \exists x' \cdot ((x,x') \in op_{abstr} \quad \wedge \quad (x',y') \in L) \end{array} \right)$$

Note : $L \mathrel{\hat{=}} l_{raff} \wedge l_{abstr}$

Bilan

- MF = Langage de spec + technique de vérif + outil
- Vérif = Correction du code par rapport à la spec
- Langage de spec doit être très expressif
  $\Rightarrow$ indécidabilité des techniques de vérification
- Techniques de vérif différent par :
  - Objectif de vérification
  - Garanties obtenues
  - Emplacement de l'indécidabilité

Simulation	indécidabilité de la correction (incomplétude)
Model checking	indécidabilité de la correction (incomplétude)
Interprétation Abstraite	indécidabilité du choix de l'abstraction (ou de la vérif)
Preuve	indécidabilité de la preuve en logique du 1er ordre

Exemple de problèmes ouverts

- Définition de langages de spec ad-hoc
- Meilleure axiomatisation des langages de programmation
- Aide à la prise en main des MF
- Travail de fond sur les techniques de vérification
  - Simulation
    - Génération **automatique** de cas de test **pertinents**
  - Model checking
    - Maîtrise de l'explosion combinatoire
  - Interprétation Abstraite
    - Choix **automatique** d'abstractions favorisant la décidabilité de la vérification
  - Preuve
    - Génération d'OP plus précises
    - Heuristiques de vérification de preuve

Mes contributions

Calcul des comportements d'un modèle B raffiné

Objectif Aide au développement / vérification de propriétés

- Représentation sous la forme d'automates symboliques
- Construction par résolution d'obligations de preuve

Heuristiques de sélection d'hypothèses

Objectif Amélioration de l'automaticité des preuves

- Proposition de critères quantifiables de pertinence d'hypothèses

Vérification de propriétés LTL sur des programmes C

Objectif Ramener ce problème à de la logique de Hoare

- Proposition de critères de réécriture
- Accent mis sur l'automaticité des preuve

Et le CITI dans tout ça ?

Analyse statique pour le middleware

- Proposition de langage(s) de spécification
  - Automates ?
  - Invariants+Pré/post-conditions ?
  - Raffinement ?
- Technique de vérification orientée preuve

Adaptation à la vérification dynamique

- Définir un contexte où la vérif est décidable
- Complétude VS correction VS décidable

Analyse de réseau de capteurs

- 1 capteur = 1 automate
- 10000 automates = 1 réseau
- Mon problème : conformité du programme vis à vis de l'automate

Merci de votre attention