

INTRODUCTION AUX ALGORITHMES ET PROGRAMMATION EN C++

V. Chabannes, P. Helluy

vincent.chabannes@math.unistra.fr

OUTLINE

1. 1. Vérification et Validation des algorithmes
2. 2. Vérification statique
3. 2.1 Techniques de base de la vérification déductive
4. 2.2 La logique de Hoare
5. 3. Vérification dynamique de programme en C++

1. VÉRIFICATION ET VALIDATION DES ALGORITHMES

Concevoir et programmer des algorithmes sans erreur est une tâche difficile : taille et complexité des applications, nombre de personnes impliquées, historique du code.

Pourtant cela est un enjeu fondamental pour les systèmes critiques (dans le domaine du médical, de l'aérospatial, des transports routiers et ferroviaires, du nucléaire...)

Objectif

Prouver qu'un programme s'exécute correctement dans toutes les situations et fait bien le travail demandé.

Vérification

Avons-nous bien construit le programme ?

Validation

Avons-nous construit le bon programme ?



Importance de la vérification et validation

Quelques bugs célèbres (aux conséquences catastrophiques) :

- Service d'ambulances de Londres LAS - 1992
- Institut National du Cancer (Panama) - 2001
- Panne de courant nord-américaine - 2003
- Automobile Toyota - 2009



- Après 37 secondes de vol, le lanceur dévie de son chemin, puis explose.
- Perte sèche pour Ariane Espace : 200 millions d'euros

Les raisons

- L'accélération maximale de la fusée a dépassé la valeur admissible par le logiciel, générant une défaillance dans le système de positionnement.
- Le code mis en cause provenait du programme Ariane 4, mais qui était moins puissant.
- La simulation avait été annulée afin d'économiser 800 000 francs.

L'analyse dynamique

Exécuter le code pour faire apparaître d'éventuels bugs (méthode de tests) : on compare le résultat obtenu au résultat attendu. Pour être efficace, il faut tester un sous-ensemble pertinent des données d'entrée possibles :

- Tests fonctionnels qui considèrent le programme comme une boîte noire et ne sont établis qu'à partir de la connaissance de la spécification du programme.
- Tests structurels qui, à partir de la connaissance du programme, cherchent à exécuter toutes les parties du code.

L'analyse statique

Parcourir le texte du code (algorithme sur papier), sans l'exécuter, pour prouver certaines propriétés. Différentes méthodes : model checking, l'interprétation abstraite, le raffinement, la vérification déductive (en particulier la logique de Hoare), ...

2. VÉRIFICATION STATIQUE

2.1 TECHNIQUES DE BASE DE LA VÉRIFICATION DÉDUCTIVE

L'idée clé est simple : un programme correct fait exactement ce que le programmeur attend de lui. Il suffit donc de rendre ces intentions explicites dans un langage mathématique, puis d'exprimer la correction du programme par des énoncés mathématiques — s'ils sont vrais, le programme est correct.

Ingrédients nécessaires

- Spécification formelle du programme : pré et post-conditions
- Modèle formel du langage de programmation : signification mathématique de chacune des expressions et instructions de ce langage

Résultat obtenu

Démonstration mathématique que le programme satisfait sa spécification.

Terminaison

L'exécution de l'algorithme produit-elle un résultat en un temps fini quelles que soient les données fournies ?

Correction partielle

Lorsque l'algorithme s'arrête, le résultat calculé est-il la solution cherchée quelles que soient les données fournies ?

Terminaison + correction partielle = correction totale

Quelles que soient les données fournies, l'algorithme s'arrête et donne une réponse correcte. Pour certains problèmes il n'existe que des algorithmes partiellement corrects !

Soit un problème instancié par une donnée D et dont la réponse est fournie par un résultat R .

Une spécification peut être donnée sous la forme :

- d'une propriété $P(D)$ qui dépend de la donnée D (précondition) ;
- d'une propriété $Q(D, R)$ qui dépend de la donnée D et du résultat R (postcondition).

Un programme satisfait cette spécification si

Pour toute donnée D qui vérifie la propriété $P(D)$, l'exécution du programme donne un résultat R qui vérifie $Q(D, R)$.

Le programme est alors dit correct par rapport à cette spécification.

Require: Deux entiers a et b

Ensure: Le quotient q et le reste r de la division euclidienne de a par b

```
procedure DIV(a, b)
  r := a
  q := 0
  while r >= b do
    r := r - b
    q := q + 1
  end while
  return q, r
end procedure
```

Données acceptables

- $b \neq 0$ sinon le problème n'a pas de sens (et la boucle non plus)
- $b > 0$
- $a \geq 0$

Résultat attendu

- $a = q \times b + r$
- $0 \leq r < b$

- Programme I : séquence d'instructions
- Pré-condition P : propriété à vérifier pour exécuter le programme
- Post-condition Q : propriété à vérifier après l'exécution du programme

Triplet de Hoare

Un triplet de Hoare est de la forme $\{P\}I\{Q\}$.

Signification : si le programme I est exécuté sur des données d'entrée satisfaisant la pré-condition P, alors si l'exécution termine, la post-condition Q est vérifiée.

Nous aurons besoin de la logique des prédicats du premier ordre :

- Connecteurs : $\vee, \wedge, \neg, \Rightarrow$
- Quantificateurs : $\forall x, \exists x$
- Opérations et prédicats usuels sur les données : $+, *, <, =, \dots$

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$
false	false	true	false	false	true
false	true	true	false	true	true
true	false	false	false	true	false
true	true	false	true	true	true

Require: Deux entiers a et b

Ensure: Le quotient q et le reste r de la
division euclidienne de a par b

```
procedure DIV(a, b)
  { a >= 0 ∧ b > 0 }                ▷ precondition
  r := a ; q := 0
  while r >= b do
    r := r - b
    q := q + 1
  end while
  { a = b × q + r ∧ 0 <= r < b }    ▷ postcondition
  return q, r
end procedure
```

Principe

On décompose l'algorithme en blocs d'une ou plusieurs instructions, chacun avec sa précondition et sa postcondition. De proche en proche, on établit que la postcondition est vraie chaque fois que la précondition l'est.

- Affectations, séquences d'instruction, structures conditionnelles : généralement simple à évaluer si la spécification est correctement écrite.
- Boucles itératives :
 - peut-être plus difficile à traiter car la valeur des données peut dépendre des itérations précédentes
 - invariant de boucle
- Procédures récursives :
 - preuves par induction
 - cas de base et cas inductif

Invariant de boucle

Un invariant de boucle est une propriété P vraie en début de boucle et conservée d'une itération à la suivante.

1. Choisir et exprimer un invariant judicieux. $\Rightarrow$ Il n'y a pas de méthode systématique
2. Démontrer qu'il est vérifié avant d'entrer dans la boucle. $\Rightarrow$ Utiliser les préconditions
3. Démontrer que s'il est vérifié au début d'une itération quelconque, il l'est aussi au début de l'itération suivante. $\Rightarrow$ Analyser le corps de la boucle : on notera x' la valeur d'une donnée x en fin de boucle et on montrera que les données vérifient toujours l'invariant.
4. Instancier l'invariant en sortie de boucle et en déduire une postcondition. $\Rightarrow$ Utiliser la condition d'arrêt de la boucle itérative

```
procedure DIV(a, b)
{ a >= 0 ∧ b > 0 }
r := a
q := 0
{ invariant : a = b × q + r }
while r >= b do
  r := r - b
  q := q + 1
  { b × q' + r' = b × (q + 1) + (r - b)
    = b × q + b - b + r
    = b × q + r = a }
end while
{ a = b × q + r ∧ r < b }
return q, r
end procedure
```

Remarque

Avec ce choix d'invariant, on ne peut pas encore démontrer la postcondition de l'algorithme, $a = b \times q + r \wedge 0 \leq r < b$: il manque encore à prouver $0 \leq r$.

Quel(s) algorithme(s) calcule(nt) la factorielle de n dans la variable F ?

Algorithme A

```
i := 0 ; F := 1
while i <= n do
  i := i + 1
  F := F * i
end while
```

Algorithme B

```
i := 0 ; F := 1
while i < n do
  i := i + 1
  F := F * i
end while
```

Algorithme C

```
i := 1 ; F := 1
while i <= n do
  F := F * i
  i := i + 1
end while
```

Algorithme D

```
i := 0 ; F := 1
while i <= n do
  F := F * i
  i := i + 1
end while
```

```
i := 0
F := 1      { F = 1 = 0! = i! }
while i <= n do
  { F = i! }
  i := i + 1
  F := F * i
  { F' = F × i' = F × (i + 1) = i! × (i + 1) = (i + 1)!
    donc F' = i'! }
end while
```

Invariant

$$F = i!$$

Mais en sortie de boucle, on a $i = n + 1$ d'où $F = (n + 1)!$

⇒ Algorithme incorrect !

```
i := 0
F := 1          { F = 1 = 0! = i! }
while i < n do
  { F = i! }
  i := i + 1
  F := F * i
  { F' = F × i' = F × (i + 1) = i! × (i + 1) = (i + 1)!
    donc F' = i'! }
end while
```

Invariant

$$F = i!$$

En sortie de boucle, nous avons $i = n$ d'où $F = n!$

⇒ Algorithme correct !

```
i := 1
F := 1          { F = 1 = 0! = (i - 1)! }
while i <= n do
  { F = (i - 1)! }
  F := F * i
  i := i + 1
  { F' = F × i = (i - 1)! × i = i!
    donc F' = (i' - 1)! }
end while
```

Invariant

$$F = (i - 1)!$$

En sortie de boucle, nous avons $i = n + 1$ d'où $F = n!$

⇒ Algorithme correct !

Remarque

Si l'on choisit $F = i!$ comme invariant alors en fin d'itération $F' = F \times i = i! \times i \neq i'! = (i + 1)!$ donc ce n'est pas un invariant.

```
i := 0
F := 1
while i <= n do
  F := F * i
  i := i + 1
end while
```

Invariant

$F = (i - 1)!$ correctement propagé par la boucle (voir algo C) mais cet invariant est faux à l'entrée de la boucle.

En réalité on a toujours $F = 0$ après la 1ère itération.

⇒ Algorithme incorrect !

Correction partielle

Pour toute donnée D qui vérifie la précondition P , si le programme se termine, alors son exécution donne un résultat R qui vérifie $Q(D, R)$.

Correction totale

Pour toute donnée D qui vérifie la précondition P , l'exécution du programme se termine et donne un résultat R qui vérifie $Q(D, R)$.

Jusqu'ici nous avons prouvé la correction partielle. Reste à démontrer que l'algorithme se termine toujours : c'est la Terminaison. Pour cela, on enrichit les annotations avec la notion de variant de boucle.

$\text{Correction partielle} \wedge \text{Terminaison} \implies \text{Correction totale}$

Variant de boucle

Un variant de boucle est une expression :

- qui est un entier positif tout au long de la boucle
- qui décroît strictement à chaque itération

- Logique : une suite d'entiers $(u_i)_{i \geq 0}$ strictement décroissante devient négative à partir d'un certain rang α . Or dans la boucle $u_i > 0$: l'algorithme termine donc nécessairement.
- Variants usuels :
 - i pour une boucle du type “for” $i = n$ “to” 1
 - $n - i$ pour une boucle du type “for” $i = 1$ “to” n
 - $j - i$ pour deux variables i croissante et j décroissante
 - ...
 - mais pas de technique “systématique”

```
procedure DIV(a, b)
  { a >= 0 ∧ b > 0 }                ▷ precondition
  r := a ; q := 0
  while r >= b do
    r := r - b
    q := q + 1
  end while
  { a = b × q + r ∧ 0 <= r < b }    ▷ postcondition
  return q, r
end procedure
```

La quantité r est un variant correct

Notons r_k la valeur de r à la fin de la $k^{\text{ième}}$ itération.

- La suite $(r_k)_{k \geq 0}$ est strictement décroissante car $\forall k > 0, r_k - r_{k-1} = -b < 0 \Leftrightarrow r_k < r_{k-1}$
- Par récurrence, on montre que $\forall k > 0, r_k \geq 0$:
 - $r_0 = a$ est un entier positif
 - $r_k = r_{k-1} - b \geq b - b = 0$ car on se trouve dans la boucle while

Précondition et postcondition

- généralement faciles à écrire si le problème est correctement spécifié
- éventuellement nécessaire de renforcer la précondition si la preuve de l'algorithme n'aboutit pas
- attention aux postconditions trop faibles

Invariants

- souvent difficiles à exprimer
- demandent une compréhension fine de l'algorithme
- les trouver peut même précéder l'écriture de l'algorithme

Variants

- souvent immédiats, mais des cas particuliers très difficiles
- nécessitant parfois de s'appuyer sur les autres assertions

Proposé en 1976 par Edsger Wybe Dijkstra (1930-2002)

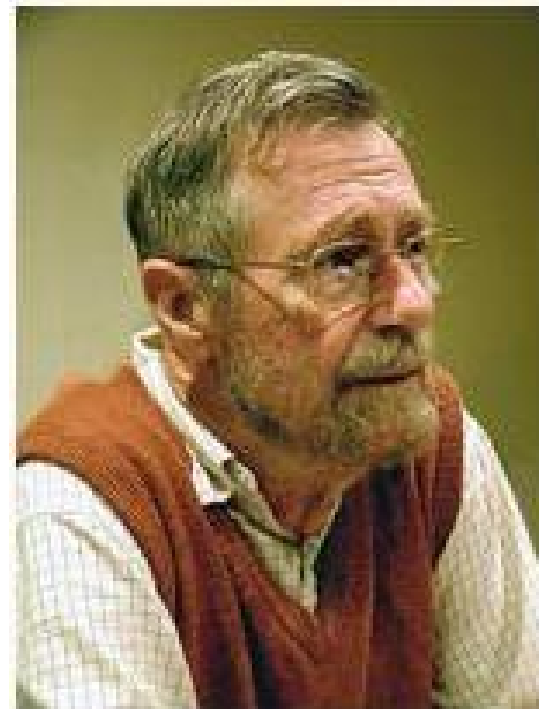
- Un des fondateurs de la science informatique
- Algorithme de recherche de plus court chemin
- Développement de langages tels l'ALGOL
- Turing Award 1972

Le problème du drapeau hollandais

Tableau de n éléments, chacun coloré en bleu, blanc ou rouge.

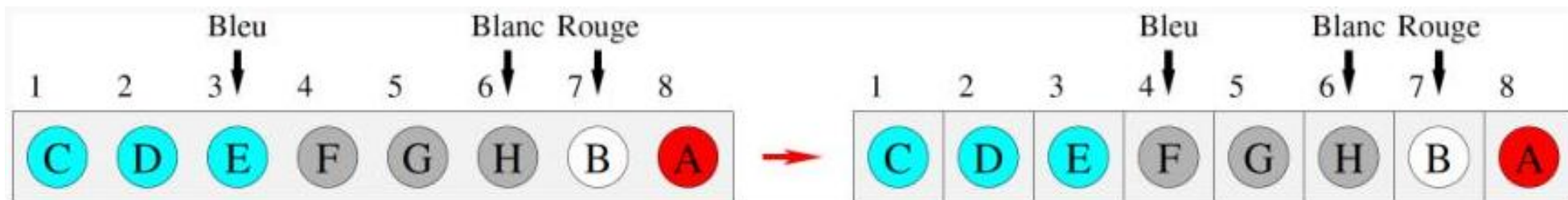
Objectif : réorganiser le tableau pour que :

- les éléments bleus soient sur la partie gauche
- les éléments blancs au centre
- les rouges en fin de tableau

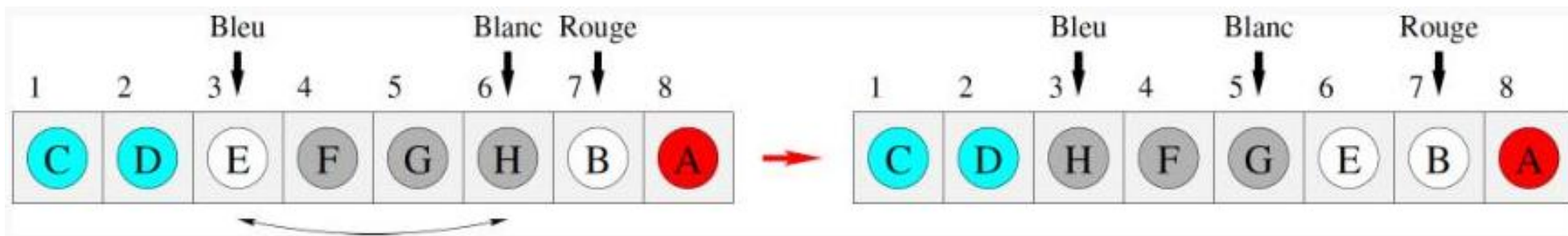




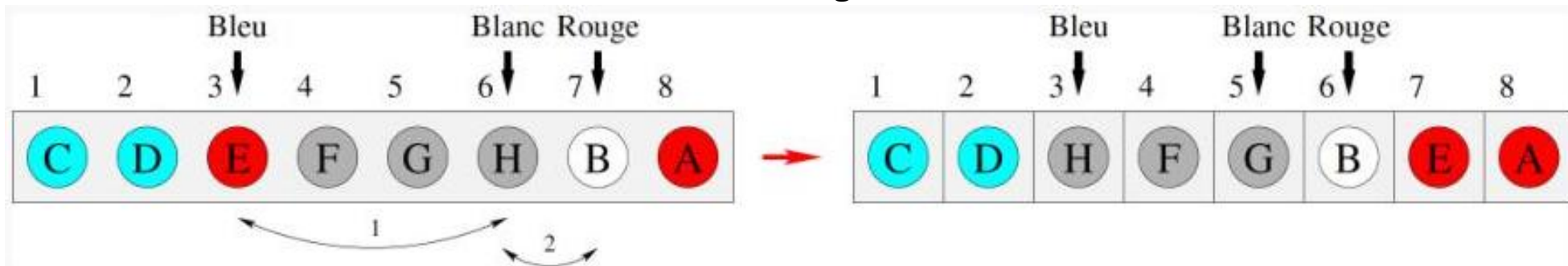
Cas bleu



Cas blanc



Cas rouge



```
Require: Un tableau T de n éléments colorés (couleur C1, C2 et C3)
Ensure: T contient les mêmes éléments rangés par couleur croissante
i1 := 1 ; i2 := n ; i3 := n      ▷ ik est la place du prochain élément de couleur Ck
while i1 <= i2 do
  switch COULEUR(T[i1]) do
    case C1
      i1 := i1 + 1                      ▷ l'élément est à la bonne place
    case C2
      ÉCHANGE(T, i1, i2) ; i2 := i2 - 1      ▷ on le place à la bonne position
    case C3
      ÉCHANGE(T, i1, i2)
      ÉCHANGE(T, i2, i3) ; i3 := i3 - 1 ; i2 := i2 - 1  ▷ permutation circulaire
  end switch
end while
```

Complexité de la fonction COULEUR

Renvoie la couleur d'un élément du tableau
 $\Rightarrow$ accès à un tableau $\Rightarrow T^{C(n)} \in \Theta(1)$

Complexité de la fonction ÉCHANGE

Échange de deux couleurs dans le tableau
4 accès à un tableau, 3 affectations
 $\Rightarrow T^{E(n)} \in \Theta(1)$

```
procedure ÉCHANGE(T, i1, i2)  
  tmp := T[i1]  
  T[i1] := T[i2]  
  T[i2] := tmp  
end procedure
```

Meilleur des cas

Tableau rempli d'éléments bleus (C_1) : à chaque itération, un test de boucle (d_1), un appel à COULEUR ($T^C(n)$) et une affectation (d_2).

```
while i1 <= i2 do           ▷ d1
  switch COULEUR(T[i1]) do    ▷ Tc(n)
    case C1
      i1 := i1 + 1           ▷ d2
    end switch
  end while
```

$$T_{\min}(n) = n(d_1 + d_2 + T^C(n)) \implies T_{\min}(n) \in \Theta(n)$$

Pire des cas

Tableau rempli d'éléments rouges (C_3) : à chaque itération, un test de boucle, un appel à COULEUR et deux appels à ÉCHANGE.

```
while i1 <= i2 do
  switch COULEUR(T[i1]) do
    case C3
      ÉCHANGE(T, i1, i2)
      ÉCHANGE(T, i2, i3)
      i3 := i3 - 1 ; i2 := i2 - 1
    end switch
  end while
```

$$T_{\max}(n) = n(d_1 + d_2 + T^C(n) + 2T^E(n)) \implies T_{\max}(n) \in \Theta(n)$$

En moyenne

- Dépend de la distribution des données !
- Mais comme $T_{\min}(n) \in \Theta(n)$ et $T_{\max}(n) \in \Theta(n)$
 $\implies T_{\text{moy}}(n) \in \Theta(n)$

```
i1 := 1 ; i2 := n ; i3 := n
while i1 <= i2 do
  switch COULEUR(T[i1]) do
    case C1 : i1 := i1 + 1
    case C2 : ÉCHANGE(T, i1, i2) ; i2 := i2 - 1
    case C3 : ÉCHANGE(T, i1, i2) ; ÉCHANGE(T, i2, i3)
               i3 := i3 - 1 ; i2 := i2 - 1
  end switch
end while
```

Terminaison : $i_2 - i_1$ est un variant acceptable

- À chaque itération i_1 augmente (cas C_1) ou i_2 diminue (cas C_2 et C_3).
- La condition du while assure que $i_1 \leq i_2$.

```
i1 := 1 ; i2 := n ; i3 := n
while i1 <= i2 do
  switch COULEUR(T[i1]) do
    case C1
      i1 := i1 + 1
    case C2
      ÉCHANGE(T, i1, i2)
      i2 := i2 - 1
    case C3
      ÉCHANGE(T, i1, i2)
      ÉCHANGE(T, i2, i3)
      i3 := i3 - 1
      i2 := i2 - 1
  end switch
end while
```

▷ { 1,2,3,4 sont valides }

▷ { les éléments de 1 à $i_1' - 1$ sont C_1 }

▷ { permutation de T (2,3,4 non impactés) }

▷ { les éléments de $i_2' + 1$ à i_3 sont C_2 }

▷ { permutation de T (2,3,4 non impactés) }

▷ { permutation de T (2 non impacté) }

▷ { les éléments de $i_3' + 1$ à n sont C_3 }

▷ { les éléments de $i_2' + 1$ à i_3 sont C_2 }

▷ { 1,2,3,4 sont valides }

Invariant de boucle

1. T contient une permutation des éléments du tableau initial.
2. Si $i_1 > 1$, les éléments de 1 à $i_1 - 1$ sont de couleur C_1 .
3. Si $i_2 < n$, les éléments de $i_2 + 1$ à i_3 sont de couleur C_2 .
4. Si $i_3 < n$, les éléments de $i_3 + 1$ à n sont de couleur C_3 .

Postcondition

En sortie de boucle $i_2 = i_1 - 1$ donc :

- Les éléments de 1 à $i_1 - 1$ sont de couleur C_1 .
- Les éléments de i_1 à i_3 sont de couleur C_2 .
- Les éléments de $i_3 + 1$ à n sont de couleur C_3 .

2.2 LA LOGIQUE DE HOARE

La méthodologie vue précédemment fonctionne mais ne passe pas à l'échelle :

- L'invariant doit être « deviné »
- La précondition aussi pose parfois problème
- Et surtout la démonstration elle-même n'est pas automatisable

Solutions proposées

- Un système de déduction formel
- Annotation du programme en partant de la fin
- Progrès vers l'automatisation (mais cependant pas totale : la preuve d'algorithme est un problème indécidable)
- Possibilité d'utiliser des assistants de démonstration

Proposée par le britannique Tony Hoare (1934-)

- Tri rapide
- Compilateur pour ALGOL-60
- Spécification formelle de langages
- Turing Award 1980

Son principe

Cette méthode fournit un formalisme logique pour raisonner sur la correction des programmes : un ensemble de règles de déduction sur les triplets de Hoare.

Ces règles permettent de répondre à la question informelle suivante :

« Si je connais les évolutions de l'état sur des petits bouts de code, que puis-je dire sur les évolutions réalisées par des codes plus grands qui utilisent ces petits bouts de codes ? »



Un système formel de déduction est constitué d'un ensemble de règles permettant de déduire des faits depuis des axiomes et des hypothèses.

Règles de déduction

Elles sont constituées :

- d'hypothèses ou prémisses $H_1, \dots, H_n$
- d'une conclusion C
- éventuellement on étiquette une règle par son nom R

Notation : $\frac{H_1 \dots H_n}{C} R$

$\implies$ Si $H_1, \dots, H_n$ sont démontrées, alors la règle R permet de démontrer C .

Par définition, un axiome est aussi une règle mais sans hypothèses : $\overline{C} R$

La logique de Hoare se base sur ce système de déduction et utilisera la notation des triplets de Hoare pour définir les hypothèses, prémisses et conclusions.

Définition

Le triplet $\{P\}I\{Q\}$ exprime la propriété suivante : si P (précondition) est vérifiée avant l'exécution de I , alors Q (postcondition) est vérifiée après son exécution.

Exemples de triplets valides

- $\{x = 1\}x := x + 2\{x = 3\}$
- $\{y = 2\}x := y + 5\{x = 7\}$
- $\{y = 1\}x := y + 1; z := x - 1\{z = 1\}$
- $\{-4 \leq x \leq 4\}$ if $x \geq 0$ then $x := x - 2$ else $x := x + 2\{-2 \leq x \leq 2\}$
- $\{x \geq 0\}$ while $x > 0$ do $x := x - 1\{x > -1\}$

Exemples de triplets non valides

- $\{x > 0\}x := x - 1\{x < 2\}$
- $\{a > 0\}$ if $x < 0$ then $y := -a * x$ else $y := a * x\{y > 0\}$

Triplets de Hoare particuliers

- $\{\text{true}\} \text{ prog } \{Q\}$: le programme peut toujours être exécuté, aucune condition sur les entrées.

Exemple : $\{\text{true}\} x := 2x \{x \bmod 2 = 0\}$

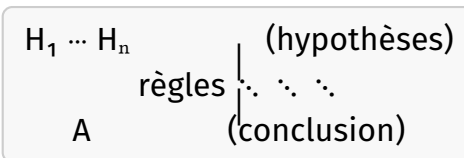
- $\{\text{false}\} \text{ prog } \{Q\}$: le programme ne peut jamais être exécuté, aucune valeur d'entrée n'est possible.
Le triplet est valide pour tout Q .

Exemple : $\{\text{false}\} x := 25 \{x = 0\}$

- $\{P\} \text{ prog } \{\text{false}\}$: le programme ne termine pas, aucune valeur de sortie n'est possible. Le triplet est valide pour tout P .

Exemple : $\{b = \text{true}\} \text{ while } b \text{ do } x := x - 1 \{\text{false}\}$

Un arbre de preuve est formé de plusieurs règles, les conclusions des unes formant les prémisses des autres :



prouve A sous les hypothèses $H_1, \dots, H_n$.

Définition

Un arbre complet est un arbre de preuve dont toutes les feuilles sont des axiomes.

Correction des règles de Hoare

Étant donné un triplet de Hoare t , s'il existe un arbre de déduction de Hoare complet ayant t à sa racine, alors le triplet est valide.

Afin de prouver la validité des triplets de Hoare, on peut utiliser un système de preuve compositionnel : la preuve de la spécification d'un code se ramène à la preuve de la spécification des parties de ce code.

Soit l'affectation $x := E$, où x est une variable.

Ce qu'on cherche

- Établir un lien logique entre la précondition et la postcondition
- Exploiter le changement de valeur de la variable x affectée

Première idée

- On dispose d'une propriété P en précondition
- On substitue x par sa nouvelle valeur dans P et on en déduit la postcondition $\{P\}x := E\{P[x \leftarrow E]\}$ mais cette règle est incorrecte !

Contre-exemple

- Soit l'instruction $x := x - 1$
- Soit la précondition $P : x \geq 0$

Alors cette règle donne :

$$\{x \geq 0\}x := x - 1\{x \geq 1\}$$

Il faut plutôt raisonner à l'envers : pour que Q soit vraie après l'affectation, il faut qu'elle soit déjà vraie pour la valeur que prend x .

- On dispose d'une propriété Q en postcondition
- On substitue x par sa nouvelle valeur dans Q et on en déduit la précondition

$$\{Q[x \leftarrow E]\} x := E \{Q\} \quad (\text{Affectation})$$

Remarquez que cette règle n'a pas d'hypothèse, c'est un axiome.

Exemple

- Soit l'instruction $x := x - 1$
 - Soit la postcondition $Q : x \geq 1$
- Alors cette règle donne : $\{x \geq 2\} x := x - 1 \{x \geq 1\}$

Soit une séquence de deux instructions $I_1; I_2$, de précondition P et de postcondition Q . Pour vérifier le triplet :

- On insère une assertion R décrivant l'état des variables entre I_1 et I_2
- On vérifie que les triplets $\{P\}I_1\{R\}$ et $\{R\}I_2\{Q\}$ sont corrects

$$\frac{\{P\}I_1\{R\} \quad \{R\}I_2\{Q\}}{\{P\}I_1; I_2\{Q\}} \quad (\text{Séquence})$$

On obtient alors la règle de la séquence (naturellement extensible pour une séquence de taille arbitraire) :

Exemple

```
{ P : ??? }  
x := x + 1  
y := 2 * x  
{ y > 2 }
```

$$\frac{\{P\}x:=x+1\{R\} \quad \{R\}y:=2x\{y>2\}}{\{P\}x:=x+1; y:=2x\{y>2\}}$$

Étape 1 : définir la postcondition Q

```
x := x + 1  
y := 2 * x  
{ y > 2 }
```

$Q \Leftrightarrow \{y > 2\}$

Objectif : remonter de la postcondition Q vers la précondition P , instruction par instruction, grâce à l'axiome d'affectation.

Étape 2 : insérer une assertion R avant la dernière instruction

```
x := x + 1  
{ x > 1 }  
y := 2 * x  
{ y > 2 }
```

$R \Leftrightarrow \{x > 1\}$

Raisonnement arrière sur y : $\{y > 2\}[y \leftarrow 2x] \Leftrightarrow \{2x > 2\} \Leftrightarrow \{x > 1\}$

donc $\{x > 1\}y := 2x\{y > 2\}$

Étape 3 : remonter à la précondition P

```
{ x > 0 }  
x := x + 1  
{ x > 1 }  
y := 2 * x  
{ y > 2 }
```

Raisonnement arrière sur x : $\{x > 1\}[x \leftarrow x + 1] \Leftrightarrow \{x + 1 > 1\} \Leftrightarrow \{x > 0\}$

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

Conclusion : règle de la séquence

$$\frac{\{x > 0\}x := x + 1\{x > 1\} \quad \{x > 1\}y := 2x\{y > 2\}}{\{x > 0\}x := x + 1 ; y := 2x\{y > 2\}}$$

Soit un branchement selon une condition C : une des deux instructions I_1 ou I_2 est exécutée. Pour que Q soit vraie après le branchement, elle doit être vraie dans les deux branches : la preuve se divise en deux sous-cas.

$$\frac{\{P \wedge C\} I_1 \{Q\} \quad \{P \wedge \neg C\} I_2 \{Q\}}{\{P\} \text{ if } C \text{ then } I_1 \text{ else } I_2 \{Q\}} \quad (\text{Conditionnelle})$$

Exemple

```
{ P }  
if C then  
  { P ∧ C } I1 { Q }  
else  
  { P ∧ ¬C } I2 { Q }  
end if  
{ Q }
```

```
{ P : ??? }  
if y >= 0 then  
  x := 2 - y  
else  
  x := y + 3  
end if  
{ x >= 0 }
```

$\{P\} \text{ if } y \geq 0 \text{ then } x := 2 - y \text{ else } x := y + 3 \{x \geq 0\}$

On remonte la preuve de la postcondition $Q \Leftrightarrow \{x \geq 0\}$ dans chaque branche :

Étape 1 : branche then

```
{ 2 - y >= 0 ∧ y >= 0 }  
x := 2 - y  
{ x >= 0 }
```

$$(2 - y \geq 0 \wedge y \geq 0) \Leftrightarrow \{y \leq 2 \wedge y \geq 0\}$$

Étape 2 : branche else

```
{ y + 3 >= 0 ∧ y < 0 }  
x := y + 3  
{ x >= 0 }
```

$$(y + 3 \geq 0 \wedge y < 0) \Leftrightarrow \{y \geq -3 \wedge y < 0\}$$

Étape 3 : union des deux branches = précondition P

- Branche then : $P \wedge y \geq 0 \Leftrightarrow \{y \leq 2 \wedge y \geq 0\}$
- Branche else : $P \wedge y < 0 \Leftrightarrow \{y \geq -3 \wedge y < 0\}$

Donc $P \Leftrightarrow \{y \leq 2 \wedge y \geq -3\}$

```
{ y ≤ 2 ∧ y ≥ -3 }  
if y ≥ 0 then  
  x := 2 - y  
else  
  x := y + 3  
end if  
{ x ≥ 0 }
```

Pour prouver une boucle, il faut un invariant P : vrai à l'entrée de la boucle, conservé par chaque itération. En sortie, l'invariant est encore vrai et la condition C est fausse :

$$\frac{\{P \wedge C\} I_1 \{P\}}{\{P\} \text{ while } C \text{ do } I_1 \{P \wedge \neg C\}} \quad (\text{Boucle})$$

```
{ P }  
while C do  
  { P ∧ C }  
  I1  
  { P }  
end while  
{ P ∧ ¬C } => { Q }
```

Étape 1 : énoncé

```
while x < b do  
  x := x + 1  
end while  
{ x = b }
```

$C \Leftrightarrow \{x < b\}$ $Q \Leftrightarrow \{x = b\}$

Objectif : trouver un invariant de boucle P tel que $P \wedge \neg C \Rightarrow Q$.

Étape 2 : choix de l'invariant et conservation

$$P \Leftrightarrow \{x \leq b\}$$

- Initialisation : on doit avoir $\{x \leq b\}$ en entrée de boucle.
- Conservation (règle d'affectation) : $\{x \leq b \wedge x < b\} x := x + 1 \{x \leq b\}$

En effet $\{x \leq b\} [x \leftarrow x + 1] \Leftrightarrow \{x + 1 \leq b\} \Leftrightarrow \{x < b\}$: l'hypothèse $x < b$ (condition de boucle) suffit à garantir l'invariant après l'affectation.

Étape 3 : sortie de boucle

$\{x \leq b\}$ while $x < b$ do $x := x + 1$ $\{x \leq b \wedge x \geq b\}$

En sortie de boucle, l'invariant et la négation de la condition donnent $x \leq b \wedge x \geq b$, donc $x = b$: la postcondition Q est vérifiée.

Conclusion

$$\frac{\{x \leq b \wedge x < b\} x := x + 1 \{x \leq b\}}{\{x \leq b\} \text{ while } x < b \text{ do } x := x + 1 \{x = b\}}$$

La logique de Hoare est un puissant outil pour prouver qu'un programme satisfait sa spécification. Elle reste cependant complexe à utiliser quand la taille et la complexité du programme augmentent :

- Des règles parfois lourdes à appliquer
- Des tailles importantes d'arbres de preuve
- Des difficultés pour trouver les invariants

Dans ce contexte, il est mieux adapté à une preuve automatisée.

On en retiendra cependant quelques idées pour les « petites » preuves

- Décomposer le programme par induction structurelle
- Annoter chaque étape du programme avec une propriété
- Remonter de la postcondition vers la précondition
- Exploiter $\neg C$ en sortie du while

L'algorithme suivant propose d'appliquer une réduction de prix en fonction d'un critère.

Objectif : déterminer quelle gamme de prix initial permet d'avoir un prix réduit ≤ 100 .

Require : a le prix initial

```
m := a
if m > 120 then
  m := m - 30
else
  q := 0.1 * m
  m := m - q
end if
return m
```

Étape 1 : définir une postcondition Q

```
m := a
if m > 120 then
  m := m - 30
else
  q := 0.1 * m
  m := m - q
end if
{ m <= 100 }
return m
```

Étape 2 : la structure conditionnelle

La postcondition Q doit être vérifiée quelle que soit la branche exécutée :

```
m := a
if m > 120 then
  m := m - 30
  { m <= 100 }
else
  q := 0.1 * m
  m := m - q
  { m <= 100 }
end if
{ m <= 100 }
return m
```

Étape 3 : déduction par règle d'affectation (branche else)

On remonte la postcondition $\{m \leq 100\}$ à travers l'affectation $m := m - q$:

```
else
  q := 0.1 * m
  { m - q <= 100 }
  m := m - q
  { m <= 100 }
```

Étape 4 : déduction par règle d'affectation (branche else)

On remonte à travers l'affectation $q := 0.1 * m$:

```
else
  {  $m - 0.1 * m \leq 100 \wedge m \leq 120$  }
   $q := 0.1 * m$ 
  {  $m - q \leq 100$  }
   $m := m - q$ 
  {  $m \leq 100$  }
end if
```

Étape 4 (suite) : simplification de la précondition

$$\{m - m \times 0.1 \leq 100 \wedge m \leq 120\}$$

$$\Leftrightarrow \{m \leq 111.11 \wedge m \leq 120\}$$

$$\Leftrightarrow \{m \leq 111.11\}$$

else

 { m <= 111.11 }

 q := 0.1 * m

 { m - q <= 100 }

 m := m - q

 { m <= 100 }

end if

Étape 5 : déduction par règle d'affectation (branche then)

On remonte la postcondition $\{m \leq 100\}$ à travers l'affectation $m := m - 30$:

```
if m > 120 then
  { m - 30 <= 100 ∧ m > 120 }
  m := m - 30
  { m <= 100 }
```

Étape 5 (suite) : simplification de la précondition

$$\begin{aligned} & \{m - 30 \leq 100 \wedge m > 120\} \\ & \Leftrightarrow \{m \leq 130 \wedge m > 120\} \\ & \Leftrightarrow \{120 < m \leq 130\} \end{aligned}$$

```
if m > 120 then
  { 120 < m <= 130 }
  m := m - 30
  { m <= 100 }
```

Étape 6 : union du branchement conditionnel

La précondition de l'instruction `if` est l'union des préconditions des deux branches :

```
{ 120 < m <= 130 v m <= 111.11 }  
if m > 120 then  
  { 120 < m <= 130 }  
  m := m - 30  
  { m <= 100 }  
else  
  { m <= 111.11 }  
  q := 0.1 * m  
  { m - q <= 100 }  
  m := m - q  
  { m <= 100 }  
end if  
{ m <= 100 }
```

Étape 7 : déduction par règle d'affectation

On remonte à travers l'affectation $m := a : \{120 < m \leq 130 \vee m \leq 111.11\}[m \leftarrow a] \Leftrightarrow \{120 < a \leq 130 \vee a \leq 111.11\}$

```
{ 120 < a <= 130 v a <= 111.11 }  
m := a  
{ 120 < m <= 130 v m <= 111.11 }  
if m > 120 then  
  { 120 < m <= 130 }  
  m := m - 30  
  { m <= 100 }  
else  
  { m <= 111.11 }  
  q := 0.1 * m  
  { m - q <= 100 }  
  m := m - q  
  { m <= 100 }  
end if  
{ m <= 100 }  
return m
```

Correction totale

Précondition : $120 < a \leq 130 \vee a \leq 111.11$

- Branche else : $\{m - m \times 0.1 \leq 100 \wedge m \leq 120\} \Leftrightarrow \{m \leq 111.11 \wedge m \leq 120\} \Leftrightarrow \{m \leq 111.11\}$
- Branche then : $\{m - 30 \leq 100 \wedge m > 120\} \Leftrightarrow \{m \leq 130 \wedge m > 120\} \Leftrightarrow \{120 < m \leq 130\}$

Le programme ne comporte pas de boucle : la terminaison est assurée, la correction est totale.

Arbre complet correspondant

$\{120 < a \leq 130 \vee a \leq 111.11\} \quad m := a \quad \{120 < m \leq 130 \vee m \leq 111.11\} \quad \{120 < m \leq 130\} \quad m := m - 30 \quad \{m \leq 100\} \quad \{m \leq 111.11\} \quad q := 0.1m; \quad m := m - q \quad \{m \leq 100\}$

(Séquence, Conditionnelle)

$\{ 120 < a \leq 130 \vee a \leq 111.11 \} \quad \text{prog} \quad \{ m \leq 100 \}$

Toutes les feuilles de l'arbre sont des axiomes (règle d'affectation) : l'arbre est complet, le triplet est valide.

Require: Deux entiers a et b

Ensure: Le quotient q et le reste r de la
division euclidienne de a par b

```
procedure DIV(a, b)
  r := a
  q := 0
  while r >= b do
    r := r - b
    q := q + 1
  end while
  return q, r
end procedure
```

Étape 1 : définir une postcondition Q

$$Q \Leftrightarrow a = b \times q + r \wedge 0 \leq r < b$$

```
procedure DIV(a, b)
  ...
  return q, r
  { Q }
end procedure
```

Étape 2 : annoter chaque instruction à déterminer

```
procedure DIV(a, b)
  { e1 } r := a { e2 } q := 0 { e3 }
  while r >= b do
    { e4 } r := r - b { e5 }
    q := q + 1 { e6 }
  end while
  { e7 }
  return q, r { Q }
end procedure
```

De proche en proche, on cherche à exprimer $e_1, \dots, e_7$ en remontant depuis Q grâce aux règles de Hoare.

Étape 3 : déterminer un invariant de boucle

```
procedure DIV(a, b)
  ...
  q := 0
  { I }
  while r >= b do
    { I ∧ r >= b }
    r := r - b
    q := q + 1
    { I }
  end while
  { Q } ⇔ { I ∧ r < b }
  ...
end procedure
```

- $Q \Leftrightarrow a = b \times q + r \wedge 0 \leq r < b$
- $I \Leftrightarrow a = b \times q + r \wedge 0 \leq r \wedge b > 0$

En sortie de boucle, I et la négation de la condition ($r < b$) donnent Q .

Étape 4 : déduction par règle d'affectation

- $I \Leftrightarrow a = b \times q + r \wedge 0 \leq r \wedge b > 0$
- $I[q \leftarrow q + 1]$
 $\Leftrightarrow a = b \times (q + 1) + r \wedge 0 \leq r \wedge b > 0$

Il faut maintenant vérifier que cette propriété, combinée à l'affectation $r := r - b$, implique bien I .

Étape 5 : vérification par règle d'affectation

- $I[q \leftarrow q + 1, r \leftarrow r - b]$
 $\Leftrightarrow a = b \times (q + 1) + r - b \wedge 0 \leq r - b \wedge b > 0$
 $\Leftrightarrow a = b \times q + r \wedge b \leq r \wedge b > 0$

(en utilisant que $0 < b \leq r \Rightarrow 0 < r \Rightarrow 0 \leq r$)

$\Rightarrow I \wedge r \geq b$

L'invariant est conservé par le corps de la boucle.

Étape 6 : déduction par règle d'affectation

On remonte l'invariant à travers l'affectation $q := q + 1$ (déjà fait), puis à travers $q := 0$:

- $I[q \leftarrow 0] \Leftrightarrow a = r \wedge r \geq 0 \wedge b > 0$

Étape 7 : déduction par règle d'affectation

On remonte enfin à travers l'affectation $r := a$:

- $I[q \leftarrow 0] \Leftrightarrow a = r \wedge r \geq 0 \wedge b > 0$
- $I[q \leftarrow 0, r \leftarrow a] \Leftrightarrow a = a \wedge a \geq 0 \wedge b > 0$

On retrouve la précondition : $a \geq 0 \wedge b > 0$

```
procedure DIV(a, b)
{ I[q ← 0, r ← a] }
r := a
{ I[q ← 0] }
q := 0
{ I }
while r >= b do
{ I ∧ r >= b }
r := r - b
{ I[q ← q+1] }
q := q + 1
{ I }
end while
{ Q } ⇔ { I ∧ r < b }
return q, r
end procedure
```

Correction partielle

Précondition : $a \geq 0 \wedge b > 0$

Terminaison

Précondition : $b > 0$
(variant de boucle : r décroît strictement)

Correction totale

Préconditions : $a \geq 0 \wedge b > 0$

3. VÉRIFICATION DYNAMIQUE DE PROGRAMME EN C++

Définition

Un test désigne une procédure de vérification partielle d'un système.

La mise en oeuvre d'un test comporte trois étapes successives :

1. définition d'un jeu d'entrées et des résultats attendus,
2. exécution du programme avec le jeu d'entrées défini,
3. comparaison du résultat obtenu avec le résultat attendu, appelé oracle, pour s'assurer que le résultat est correct ou non.

Le test d'un programme :

- a pour objectif de réduire les risques d'apparition d'anomalies avec des moyens manuels et informatiques.
- fait partie du processus de développement.
- n'a pas pour objectif :
 - de corriger le défaut détecté (débogage)
 - de prouver la bonne exécution d'un programme

Tests en boîte noire

On ne s'intéresse pas à la structure interne du code mais uniquement à la relation entrées/sorties.

- On lance des scénarios pour lesquels on connaît la bonne valeur de la sortie.
- Les scénarios sont choisis de telle sorte qu'ils soient révélateurs d'un comportement.

Tests en boîte blanche

On analyse la structure interne du code et on cherche à exécuter toutes les lignes.

- On choisit des scénarios permettant de traiter tous les cas possibles.
- On vérifie que l'exécution se termine et que le résultat est correct (si on le connaît).

Require: $n \geq 0$ un entier

```
procedure FactA(n)
  i := 0
  F := 1
  while i <= n do
    i := i + 1
    F := F * i
  end while
  return F
end procedure
```

Sortie (l'invariant $F = i!$ donne $F = (n + 1)!$ en sortie):

```
user $ ./factorielleA
0! = 1
1! = 2
2! = 6
3! = 24
4! = 120
5! = 720
6! = 5040
7! = 40320
```

```
int factA( int n )
{
  int i = 0;
  int F = 1;
  while (i <= n)
  {
    ++i;
    F *= i;
  }
  return F;
}

int main()
{
  for (int i=0; i<=7; ++i )
    std::cout << i << "! = "
      << factA(i) << "\n";
  return 0;
}
```

Require: $n \geq 0$ un entier

```
procedure FactB(n)
  i := 0
  F := 1
  while i < n do
    i := i + 1
    F := F * i
  end while
  return F
end procedure
```

Sortie (l'invariant $F = i!$ donne $F = n!$ en sortie):

```
user $ ./factorielleB
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
```

```
int factB( int n )
{
  int i = 0;
  int F = 1;
  while (i < n)
  {
    ++i;
    F *= i;
  }
  return F;
}

int main()
{
  for (int i=0; i<=7; ++i )
    std::cout << i << "! = "
      << factB(i) << "\n";
  return 0;
}
```

L'algorithme B est correct, mais avec le type `int` (32 bits) le calcul déborde à partir de $13!$:

```
user $ ./factorielleB
...
12! = 479001600
13! = 1932053504
14! = 1278945280
15! = 2004310016
16! = 2004189184
17! = -288522240
18! = -898433024
19! = 109641728
```

Valeurs de référence

n	$n!$	n	$n!$
8	40 320	9	362 880
10	3 628 800	11	39 916 800
12	479 001 600	16	20 922 789 888 000
13	6 227 020 800	17	355 687 428 096 000
14	87 178 291 200	18	6 402 373 705 728 000
15	1 307 674 368 000	19	121 645 100 408 832 000

Avec le type `long int`, le calcul reste exact jusqu'à 19! :

```
long int factB( long int n )
{
    int i = 0;
    long int F = 1;
    while (i < n)
    {
        ++i;
        F *= i;
    }
    return F;
}
```

```
user $ ./factorielleB
...
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
```

Jusqu'ici, la vérification s'est faite « à l'oeil » : on affiche le résultat et on le compare au résultat attendu. Ce procédé atteint vite ses limites dès que le résultat n'est plus un simple nombre (ex. : tri d'une séquence de taille 1 000 000).

Automatiser la vérification du test

Le but : un programme qui reçoit une série de données d'entrée et donne en sortie une seule réponse, réussi ou échec. Pour cela, on complète le programme par un algorithme de spécification :

- En entrée : la sortie du programme initial et la réponse attendue
- En sortie : la réponse de la vérification du test

Remarque : pour certains algorithmes longs et complexes, les résultats attendus sont difficiles à obtenir. On peut alors découper l'algorithme en sous-algorithmes, généralement plus faciles à vérifier.

```
void triParInsertion(int * tab, int size)
{
    int key, i;
    for ( int j = 1; j < size ; j++)
    {
        key = tab[j];
        i = j-1;
        while ( i >= 0 && tab[i] > key )
        {
            tab[i+1] = tab[i];
            i = i-1;
        }
        tab[i+1] = key;
    }
}
```

```
void verifieTri(int * tab, int size)
{
    bool isOk = true;
    for ( int i = 1; i < size ; ++i )
    {
        if ( tab[i-1]>tab[i] )
        {
            isOk = false;
            break;
        }
    }
    if ( isOk )
        std::cout << "valid tri\n";
    else
        std::cout << "invalid tri\n";
}
```

```
int main( int argc, char** argv )
{
    int size = std::stoi(argv[1]);
    // creation d'un tableau random [0,50]
    int * tab = new int[size];
    std::srand(std::time(nullptr)); // seed
    for ( int k=0;k<size;++k )
        tab[k] = std::rand()/(RAND_MAX/50);
    // tri du tableau
    triParInsertion( tab,size );
    // verifie tri
    verifieTri( tab,size );
    return 0;
}
```

```
user $ ./triParInsertion 100
valid tri
user $ ./triParInsertion 1000
valid tri
user $ ./triParInsertion 10000
valid tri
```

L'analyse formelle nous a permis d'annoter les programmes : des propriétés sur l'état des variables à une position précise de l'algorithme.

Ces propriétés peuvent aussi être vérifiées à l'exécution : on ajoute des instructions qui contrôlent que la théorie et la pratique concordent. Sinon, le programme s'arrête avec un message d'erreur.

Assertions en C++

```
{ i >= 0 & j > 0 }  
i := i + 3  
j := j - 2  
{ i >= 3 & j > -2 }
```

```
assert( i >= 0 && j > 0 );  
i = i + 3;  
j = j - 2;  
assert( i >= 3 && j > -2 );
```

$$\{P\} \Leftrightarrow \{a \geq 0 \wedge b > 0\}, \quad \{Q\} \Leftrightarrow \{a = b \times q + r \wedge 0 \leq r < b\}, \quad \{I\} \Leftrightarrow \{a = b \times q + r \wedge b > 0 \wedge r \geq 0\}$$

Require: Deux entiers a et b

```
procedure DIV(a, b)
{ P }
r := a
q := 0
{ I }
while r >= b do
{ I ∧ r >= b }
r := r - b
q := q + 1
{ I }
end while
{ Q }
end procedure
```

```
// divisionEuclidienne.cpp
#include <iostream>
#include <cassert>

void divEuclidienne(int a, int b)
{
    assert( a >= 0 && b > 0 );
    int r = a;
    int q = 0;
    assert( a == b*q+r && b > 0 && r >= 0 );
    while ( r >= b )
    {
        assert( a == b*q+r && b > 0
                && r >= 0 && r >= b );
        r = r - b;
        q = q + 1;
        assert( a == b*q+r && b > 0 && r >= 0 );
    }
    assert( a == b*q+r && 0 <= r && r < b );
    std::cout << a << " = " << b << "*"
               << q << "+" << r << "\n";
}
```

```
int main( int argc, char ** argv )
{
    int a = std::stoi(argv[1]);
    int b = std::stoi(argv[2]);
    divEuclidienne(a,b);
    return 0;
}
```

```
user $ ./divisionEuclidienne 8 3
8 = 3*2+2
user $ ./divisionEuclidienne 134 7
134 = 7*19+1
```

```
user $ ./divisionEuclidienne 8 -3
Assertion failed: (a>=0 && b>0),
function divEuclidienne,
file divisionEuclidienne.cpp, line 6.
```

Si on introduit un bug dans le code C++

Par exemple on remplace la ligne 14 $q = q + 1$ par $q = q + 2$.

```
user $ ./divisionEuclidienne 8 3
Assertion failed: (a == b*q+r
    && b > 0 && r >= 0),
function divEuclidienne,
file divisionEuclidienne.cpp,
line 15.
```

La théorie (les annotations) et la pratique (le code) ne concordent plus : l'assertion détecte le bug.