CNAM - CRA Nancy
2003

Jacques Lonchamp

CINQUIEME PARTIE

Les techniques de vérification.

1. Introduction

Tous les produits du cycle de vie devraient être vérifiés et pas seulement le code. Le résultat de ces vérifications n'est pas nécessairement binaire (acceptation ou rejet du produit). Des défauts peuvent être tolérés et de toute façon un certain pourcentage de défauts résiduels est inévitable dans tout gros programme. C'est pourquoi on peut lire souvent des phrases du type « cette nouvelle version corrige les défauts suivants de la précédente version ... ».

D'après la terminologie de l'IEEE (norme 729) la *faute* est à l'origine de *l'erreur* qui se manifeste par des *anomalies* dans le logiciel qui peuvent causer des *pannes* :

faute $\Rightarrow$ erreur $\Rightarrow$ anomalie $\Rightarrow$ panne

Il existe deux approches complémentaires de la vérification :

- *expérimenter* le comportement de l'application (la *tester*) avec un ensemble bien choisi de données; on parle aussi de vérification *dynamique* ;
- *analyser les propriétés du système*, sans exécution ; on parle aussi de vérification *statique*.

2. Les tests ou vérifications dynamiques

Ils se font à partir de jeux de tests (jeux d'essais) qui ne peuvent pas en général être exhaustifs. Le jeu de tests est sélectionné. Le programme est exécuté avec le jeu de tests. Les résultats obtenus sont comparés aux résultats attendus d'après les spécifications du problème.

Les tests ont pour but de mettre en évidence les erreurs. Les tests peuvent *prouver la présence d'erreurs mais ne peuvent pas prouver leur absence*.

2.1. La construction des jeux de tests

On distingue l'approche aléatoire, l'approche fonctionnelle ou 'boîte noire' et l'approche structurelle ou 'boîte blanche'.

a) *L'approche aléatoire*

Le jeu de tests est sélectionné *au hasard sur le domaine de définition des entrées* du programme. Le domaine de définition des entrées du programme est déterminé à l'aide des *interfaces* de la spécification ou du programme.

Cette méthode ne garantit pas une bonne couverture de l'ensemble des entrées du programme. En particulier, elle peut ne pas prendre en compte certains cas limites ou exceptionnels. Cette méthode a donc une *efficacité très variable*.

b) *L'approche fonctionnelle ou boîte noire :*

On considère seulement la spécification de ce que doit faire le programme, sans considérer sa structure interne.

On peut vérifier chaque fonctionnalité décrite dans la spécification.

On s'appuie principalement sur les données et les résultats. Le danger est l'explosion combinatoire qu'entraîne un grand nombre d'entrées du programme.

Par contre, on peut écrire ces tests très tôt, dès qu'on connaît la spécification.

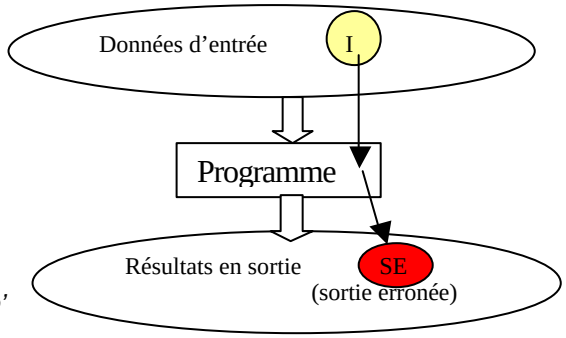

Figure 1 : tests 'boîte noire'

Une technique souvent utilisée est *l'analyse des valeurs frontières et le regroupement en classes d'équivalence*. Elle se base sur l'observation que les erreurs arrivent souvent aux limites des domaines de définition des variables du programme. Au lieu de tester toutes les valeurs, on partitionne les valeurs en classes d'équivalence et on teste *aux limites des classes*.
Exemple : si une donnée doit varier dans un intervalle [a,b], on peut définir 3 classes < a, dans l'intervalle et > b ; on peut tester par exemple 0, a-e, a, a+e, b-e, b, b+e où e est une petite valeur positive.

c) *L'approche structurelle ou boîte blanche :*
Dans l'approche par *boîte blanche* on tient compte de la structure interne du module. On peut s'appuyer sur différents critères pour conduire le test comme :
c1) Le *critère de couverture des instructions* : le jeu d'essai doit assurer que toute instruction élémentaire est exécutée au moins une fois.
c2) Le *critère de couverture des arcs du graphe de contrôle*; le graphe de contrôle est un graphe qui résume les structures de contrôle d'un programme (cf. Figure 2).

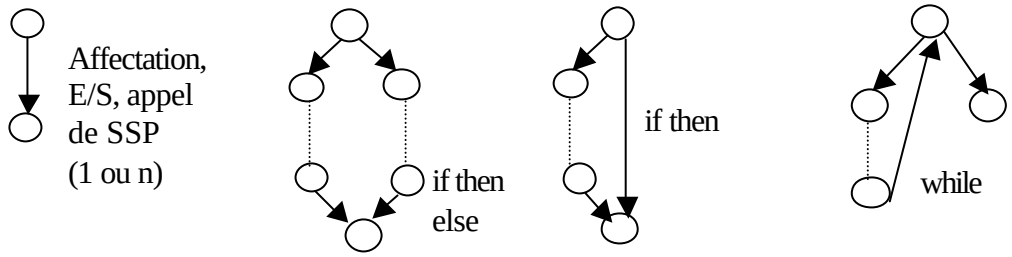

Figure 2 : éléments d'un graphe de contrôle

Exemple 1 : soit l'algorithme d'Euclide qui calcule le pgcd de 2 nombres (plus grand commun diviseur) et son graphe de contrôle (cf. Fig. 3).

```
begin
  read(x) ; read(y) ;
  while (not (x = y) ) loop
    if x > y then
      x := x - y ;
    else
      y := y - x ;
    end if ;
  end loop ;
  pgcd := x ;
end ;
```

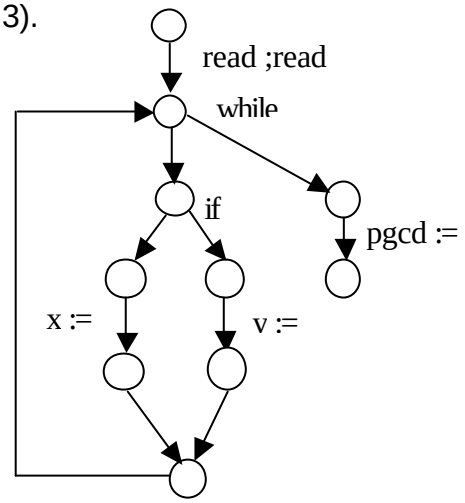

Figure 3 : graphe de contrôle de l'algorithme d'Euclide

Le jeu d'essai (x=3,y=3), (x=4,y=3), (x=3,y=4) satisfait les 2 critères de couverture.

Exemple 2 : soit le programme suivant (cf. Figure 3).

```
if x < 0 then
   x := -x ;
end if ;
z := x ;
```

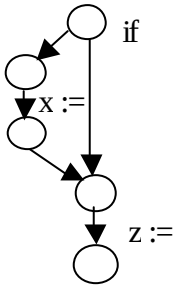

Figure 3 : graphe de contrôle d'un si sans else

(x = -2) satisfait le critère de couverture des instructions mais pas celui des arcs. Il faudrait aussi tester ce qui se passe quand x est positif.

Remarque : la *complexité cyclomatique* C vaut A – N + 2, où A est le nombre d'arcs et N est le nombre de noeuds ; c'est une mesure de la complexité d'un programme ; on peut prouver que C est aussi *la borne supérieure du nombre de tests à effectuer pour que tous les arcs soient couverts au moins une fois*.

Dans l'exemple 1, C = 11 – 10 + 2 = 3 ; c'est aussi la taille de notre jeu de tests.

Dans l'exemple 2, C = 5 – 5 + 2 = 2 ; c'est aussi la taille nécessaire pour couvrir tous les arcs.

c3) Le *critère de couverture des chemins du graphe de contrôle*.

Exemple 3 : soit le programme de la figure 4.

```
if (not ( x=0)) then
   y := 5 ;
else
   z := z - x ;
end if ;
if (z > 1) then
   z := z / x ;
else
  z := 0 ;
end if ;
```

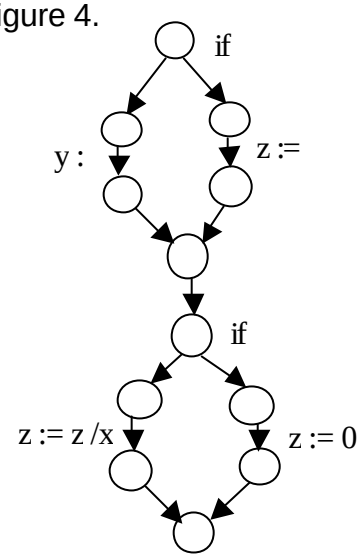

Figure 4 : graphe de contrôle d'une double conditionnelle

Le jeu d'essai (x=0,z=1), (x=1,z=3) vérifie la couverture des arcs mais pas celle des chemins et ne peut donc détecter une division par zéro.

(x=0,z=3), (x=1,z=1), (x=0,z=1), (x=1,z=3) vérifie la couverture des chemins et détecte donc la division par zéro.

Malheureusement *le nombre de chemins peut devenir très grand dans des programmes réels* ce qui rend ce dernier critère souvent inexploitable en pratique. Par exemple, un programme de quelques lignes comprenant 2 boucles imbriquées de 20 pas chacune, la boucle interne contenant une suite de 4 conditionnelles, donne naissance à 6400 chemins ! (20 x 20 x 2 x 2 x 2 x 2).

On se contente donc de couvertures incomplètes, comme la couverture des 'i-chemins' (cf. exercice 5.2).

c4) Le *critère de couverture des conditions* : le jeu de tests doit couvrir à vrai et à faux toutes les conditions élémentaires de toutes les conditionnelles.

Exemple 3 :

```
  if (a > 1 and b = 0)
    x := x / a ;
  end if ;
  if (a = 2 or x > 1) then
    x := x + 1 ;
  end if;
```

a > 1	False	True
b = 0	False	True
a = 2	False	True
x > 1	False	True

(a=1,b=0,x=3), (a=2,b=1,x=1) couvre toutes les conditions mais pas toutes les instructions.

En général il est conseillé de *mélanger différents critères*. Ce type de test structurel ne peut être réutilisé tel quel en cas de modification du code.

2.2 Les différents types de tests

a) Les test unitaires de programmes ou de modules
Dans ce qui précède nous avons fait l'hypothèse du test d'un programme isolé. Le test d'un module ressemble au test d'un programme isolé si ce n'est que le module ne fonctionne pas seul mais utilise d'autres modules et est appelé par d'autres modules. Pour tester un module, il faut simuler le comportement des modules appelés (relation 'utilise') et simuler les appels du module.

b) Les tests d'intégration
Après avoir testé unitairement les modules il faut tester leur intégration progressive jusqu'à obtenir le système complet.
Test alpha : l'application est mise dans des conditions réelles d'utilisation, au sein de l'équipe de développement (simulation de l'utilisateur final).

c) Les test de réception
Test, généralement effectué par l'acquéreur dans ses locaux après installation d'un système, avec la participation du fournisseur, pour vérifier que les dispositions contractuelles sont bien respectées.
Test bêta : distribution du produit sur un groupe de clients avant la version définitive.

d) Les tests de non régression
A la suite de la modification d'un logiciel (ou d'un de ses constituants), un test de non régression a pour but de montrer que les autres parties du logiciel n'ont pas été affectées par cette modification.

3. Les vérifications statiques

3.1 Les techniques informelles

Il s'agit d'activités réalisées par un groupe d'inspecteurs qui examinent un document à la recherche d'erreurs.

Dans le cas de code, les participants peuvent 'jouer à la machine' (on parle de 'code walk-through'). Dans le cas de code ou de documents de conception, les participants

peuvent aussi chercher des défauts (on parle de 'revue' ou 'd'inspection') en s'aidant souvent d'une check-list des défauts les plus courants.
Exemple : parmi les erreurs classiques en programmation on peut citer

- l'utilisation de variables non initialisées,
- les sauts dans des boucles,
- les affectations incompatibles,
- les boucles infinies,
- les débordements de tableaux,
- les allocations et libérations impropres de zones mémoire,
- les mauvaises correspondances entre paramètres formels et effectifs,
- les tests d'égalité entre valeurs flottantes, etc.

Des procédés structurés plus ou moins complexes (phase de préparation, phase privée, phase de corrélation, phase publique) pour réaliser ces activités ont été proposés avec des rôles bien codifiés pour les différents acteurs (ex : présentateur, inspecteur, modérateur, secrétaire, producteur). Des conseils tirés de l'expérience sont proposés :

- de 3 à 5 participants,
- une durée raisonnable (quelques heures),
- se concentrer *sur la découverte des défauts pas sur leur correction*, etc.

Ces techniques sont lourdes et donc chères, mais efficaces et complémentaires des tests. Elles sont très utilisées avec des procédures bien définies pour les logiciels critiques (NASA, MOTOROLA, etc.)

3.2 Les techniques formelles

Il s'agit de prouver formellement la correction d'un programme. Le programme est caractérisé par sa précondition (condition éventuelle à respecter par les données du programme) et sa postcondition (condition vraie à la fin du programme qui définit donc son objectif). Pour réaliser la preuve, la méthode de Hoare définit des assertions logiques intermédiaires; on part de la post condition du programme et à chaque instruction on regarde *quelle est l'assertion qui doit être vraie avant l'appel de l'instruction pour que l'assertion après l'appel de l'instruction soit vraie*. Si on peut remonter de la sorte ('backward substitution') jusqu'à la pré condition du programme on prouve ainsi sa correction (si la pré condition est vraie alors la post condition est vraie).

Considérons d'abord uniquement des affectations et des entrées et sorties. Une instruction d'écriture write(x) équivaut à une affectation output := x. Une instruction de lecture read(a) équivaut à une affectation a := $input_i$.

L'axiome de substitution arrière :

{Q[x\exp]} x := exp {Q}

dit que si Q est vraie pour x après l'affectation alors Q est vraie pour exp avant l'affectation. Il suffit de réécrire l'expression Q avec exp à la place de x.

Exemple :
Prouvons le programme élémentaire suivant :

{true}
begin
read(a) ;
read(b) ;
x := a + b ;
write(x) ;
end
{output = $input_1$ + $input_2$ }

Si {output = $input_1$ + $input_2$ } est vrai après write(x), c'est à dire output = x, { x = $input_1$ + $input_2$ } doit être vrai après x := a + b. Donc {a + b = $input_1$ + $input_2$} doit être vrai après read(a), c'est à dire a := $input_1$. Donc {$input_1$ + b = $input_1$ + $input_2$} doit être vrai après read(b), c'est à dire b := $input_2$. Donc {$input_1$ + $input_2$ = $input_1$ + $input_2$} doit être vrai initialement. Comme cela est toujours vrai on prouve bien que la pré condition du programme {true} implique la post condition. Le programme est correct.

Pour les structures de contrôle, on donne des règles générales de preuve sous la forme :

$$\frac{\text{assertion1, assertion2}}{\text{assertion3}}$$

signifiant que si les assertions 1 et 2 sont vraies alors l'assertion 3 est vraie (et réciproquement).

Soit une conditionnelle **if** cond **then** S1 **else** S2 **endif** dont la pré condition est Pre et la post condition est Post. La règle de preuve est :

$$\frac{\{\text{Pre } \textbf{and} \text{ cond}\}\ S1\ \{\text{Post}\}\ ,\ \{\text{Pre } \textbf{and not} \text{ cond}\}\ S2\ \{\text{Post}\}}{\{\text{Pre}\}\ \textbf{if}\ \text{cond}\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{endif}\ \{\text{Post}\}}$$

Elle se lit en général du bas en haut pour décomposer la preuve d'une conditionnelle en deux preuves élémentaires des blocs S1 et S2.

Exemple : on veut prouver

{true}
if x >= y **then**
max := x ;
else
max := y ;
endif
{(max = x **or** max = y) **and** (max >= x **and** max >= y)}

Si x >= y, on exécute max := x; pour que {(max = x **or** max = y) **and** (max >= x **and** max >= y)} soit vrai après max := x il faut que {(x = x **or** x = y) **and** (x >= x **and** x >= y)} soit vrai avant, c'est à dire x >= y. Ce qui est vrai d'après Cond.
Si x < y, on exécute max := y; pour que {(max = x **or** max = y) **and** (max >= x **and** max >= y)} soit vrai après max := y il faut que {(y = x **or** y = y) **and** (y >= x **and** y >= y)}, c'est à dire x <= y soit vrai avant. Or **not** Cond, c'est à dire x < y implique x <= y. C'est donc vrai.
Donc dans tous les cas ({true}) la post condition est vérifiée.

Pour l'itération **while** la règle de preuve est :

$$\frac{\{\text{I } \textbf{and} \text{ cond}\}\ S\ \{\text{I}\}}{\{\text{I}\}\ \textbf{while}\ \text{cond}\ \textbf{loop}\ S\ \textbf{end loop}\ \{\text{I } \textbf{and not} \text{ cond}\}}$$

où I est une assertion donnée, appelée invariant de boucle, qui est conservée vraie *à chaque itération* quand cond est vraie. Alors, quel que soit le nombre d'itérations, I sera vrai à la sortie du **while** ainsi que **not** cond.
Cet invariant de boucle, qui 'traverse' la boucle, n'est pas toujours facile à trouver.

Exemple :

```
{x >= 0}    pré condition
while (x > 0) loop
    x := x - 1 ;
end loop ;
{x = 0}     post condition
```

Il faut trouver un invariant qui avec la négation de la condition (I **and not** cond) donne la post condition. Ici, on peut prendre x >= 0 comme invariant de boucle, car x >= 0 **and not** x > 0, c'est à dire x >= 0 **and** x <= 0 donne bien la post condition x = 0.
Montrons que {I **and** cond} S {I} ; si x >= 0 est vrai à la fin de la boucle, x – 1 >= 0, c'est à dire x >= 1 doit être vrai avant x := x - 1; or I **and** cond qui vaut x >= 0 **and** x > 0, c'est à dire x > 0, implique bien que x >= 1. I traverse donc la boucle.
D'après la règle de preuve, I est donc vrai avant la boucle, ce qui vérifie la pré condition qui est égale à I.

Encore faut-il que la boucle se termine. On parle de *correction partielle* quand on peut prouver {pré condition} programme {post condition}. Il faut ajouter une preuve de terminaison du programme pour prouver la *correction totale*. Ces preuves sont plus compliquées que les précédentes.
Voici un exemple bien connu de programme dont on ne sait pas s'il termine ; x est un entier et **div** est la division entière :

```
while x > 1 loop
  if (impair(x)) then x := (3*x)+1 else  x := (x div 2) endif
end loop
```

Les preuves de programmes sont peu utilisées dans la pratique car leur complexité est grande et les erreurs y sont possibles au même titre que dans les programmes.

Mais la maîtrise des assertions, pré, post et invariants de boucle, doit aider à *concevoir et à documenter plus rigoureusement les programmes*. Par ailleurs, lors de la correction des défauts (déboguage), des assertions peuvent être placées dans le code *pour être testées lors de l'exécution*. Les langages modernes, y compris les dernières versions de Java, donnent des facilités pour gérer ces tests d'assertions au déboguage.

Exercices

Exercice 5.1. *Test boîte blanche.*
Soit le programme suivant :

```
lire(x)
lire(y)
z = 0
signe = 1
si x < 0 alors
  signe = -1
  x = - x
finsi
si y < 0 alors
  signe = - signe
  y = - y
finsi
tant que x >= y faire
  x = x - y
  z = z + 1
fin
z = signe * z
```

a) Dessiner le graphe de contrôle associé à ce programme en numérotant ses nœuds.
b) Par quelle suite de nœuds faut-il passer pour satisfaire le critère de couverture des instructions ?
Donner un jeu d'essai minimum qui satisfasse ce critère.
c) Par quelle suite de nœuds faut-il passer pour satisfaire le critère de couverture des arcs ?
Donner un jeu d'essai minimum qui satisfasse ce critère.
d) on appelle ***critère de couverture des i-chemins***, le critère qui garantit que l'on passe sur tout les chemins possibles en répétant de 0 à i fois chaque boucle.
Par quelle suite de nœuds faut-il passer pour satisfaire le critère de couverture des 1-chemins?
Donner un jeu d'essai minimum qui satisfasse ce critère.

Exercice 5.2. *Blanc et noir*

a. Test de type "boîte blanche"

Donnez 3 jeux d'essai satisfaisant les critères de couverture des instructions, des arcs et des chemins pour l'extrait de code suivant :

```
if (x > 10) then a = a + 1; endif
if (x % 2 = 0) then b = b + 1; endif  (où x % 2 donne le reste de la division entière de x par 2).
```

b. Test de type "boîte noire"

On considère une procédure 'triangle' qui reçoit en paramètres 3 réels a, b et c qui sont les longueurs des côtés d'un triangle. La procédure retourne comme résultat un code 0 si le triangle défini par a, b et c est invalide, 1 si le triangle est équilatéral, 2 si le triangle est isocèle et 3 pour un triangle valide quelconque (ni isocèle, ni équilatéral).
Donnez un jeu d'essai *exhaustif* pour cette procédure testant tous les cas de figure en *distinguant* les 3 entrées a, b et c.

Exercice 5.3. *Tests boîte blanche*

Soit le programme de comparaison de chaînes de caractères suivant exprimé en pseudo code :

```
1. equal : un booléen
2. string1, string2 : deux chaînes de caractères
3. Lire(string1, string2)
4. si (string1.length = string2.length) alors      // sring1.length donne la longueur de la chaîne string1
5.      i <- 1                                     // la flèche correspond à une affectation  = teste une égalité
6.      tant que i <= string1.length  et  string1.character[i] = string2.character[i]
          // string1.character[i] donne le ième caractère de la chaîne string1
7.          i <- i + 1
8.      si i = string1.length + 1 alors            // le premier caractère a l'indice 1
9.        Afficher(« chaînes égales »)
```

Donner un jeu d'essai couvrant tous les chemins possibles ; on suppose que la plus petite chaîne à une taille <= 2 (0 ou 1 ou 2). Pour y parvenir, dessiner le graphe de contrôle en numérotant les noeuds du graphe puis lister tous les chemins avec la suite des numéros.

Exercice 5.4. *Test 'boîte blanche'*

Soit le code suivant ; table(i) dénote le ième élément du tableau table, les indices variant de 1 au nombre d'éléments du tableau :

```
found := false ;
if number_of_items ≠ 0 then counter := 1 ;
  while ( (not found ) and (counter < number_of_items) ) loop
    if table(counter) = desired_element then
      found := true ;
    end if ;
    counter := counter + 1 ;
  end loop ;
end if ;
if found then
  write(``the desired element exists'') ;
else
  write (``the desired element does not exist'') ;
end if ;
```

a) Ce programme est incorrect. Pourquoi ?
b) Construire le graphe de contrôle.
c) Montrer que le jeu d'essai suivant satisfait le critère de couverture des arcs du graphe de contrôle sans détecter pour autant l'erreur. {(number_of_items = 0), (number_of_items = 3, desired_element au rang 2 de table)}
d) Montrer que si l'on impose au jeu d'essai de tester toutes les combinaisons valides des termes des conditions composées (***couverture des conditions***) on peut trouver l'erreur.