TEST DE LOGICIEL

Introduction

Cout des fautes :

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* :

$$\text{Faute} \Rightarrow \text{erreur} \Rightarrow \text{anomalie} \Rightarrow \text{panne}$$

- Couts économiques : 64 milliards $/an rien qu'aux US (2002)
- Couts humains, environnementaux, etc.

Nécessite d'assurer la qualité des logiciels

- Domaines critiques : atteindre une très haute qualité imposée par les lois/normes/assurances/... (ex : DO-178B pour aviation).
- autres domaines : atteindre le rapport qualité/prix.

Validation et Vérification (V & V)

- Vérification : est-ce que le logiciel fonctionne correctement ?
- Validation : est-ce que le logiciel fait ce que le client veut ?

Méthodes de Validation et Vérification (V & V)

- Test statique : Review de code, de spécifications, de documents de design.
- Test dynamique : Exécuter le code pour s'assurer d'un fonctionnement correct.
- Vérification symbolique : Run-time checking, Execution symbolique, …
- Vérification formelle : Preuve ou model-checking d'un modèle formel, raffinement et génération de code.

Actuellement, le test dynamique est la méthode la plus diffusée et représente jusqu'à 60 % de l'effort complet de développement d'un produit logiciel.

Définitions du test

- «Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus »-IEEE (Standard Glossary of Software Engineering Terminology).
- «Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts »-G. Myers (The Art of Software testing).
- Tester c'est réaliser l'exécution du programme
- Notion d'Oracle : résultats attendus d'une exécution du logiciel
- Coût du test : 30 % à60 % du coût de développement total

Les tests ou vérifications dynamiques :

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.*

Les tests se font à partir de jeux de tests (jeux d'essais). 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*.

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.

Test boîte noire [black box testing] :

– évaluation de l'extérieur (sans regarder le code), uniquement en fonction des entrées et des sorties.

– sur le logiciel ou un de ses composants.

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.

Test boîte blanche [white/glass box testing]:

– exploite le code (→ besoin de la source/de l'architecture)

– tests de portions de code : bloc, branche, etc.

Remarque : Mise au point

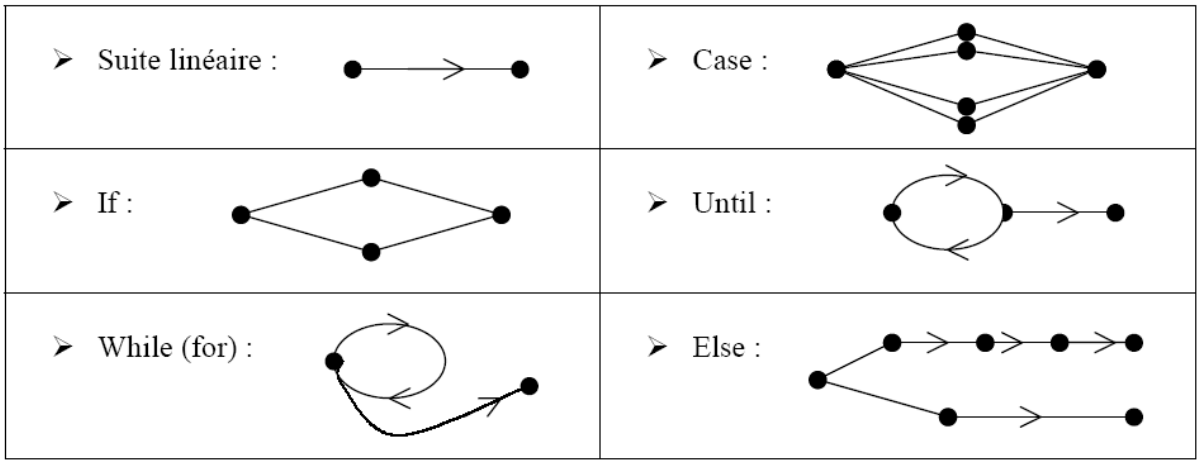
Si un test échoue :

– créer un *rapport d'anomalie*

– selon le niveau du test, revoir la phase d'analyse, de conception ou de codage

– générer les nouvelles versions de documents.

La structures de contrôle se présente sous la forme d'un graphe dit graphe de flot. On représente les instructions comme cela :

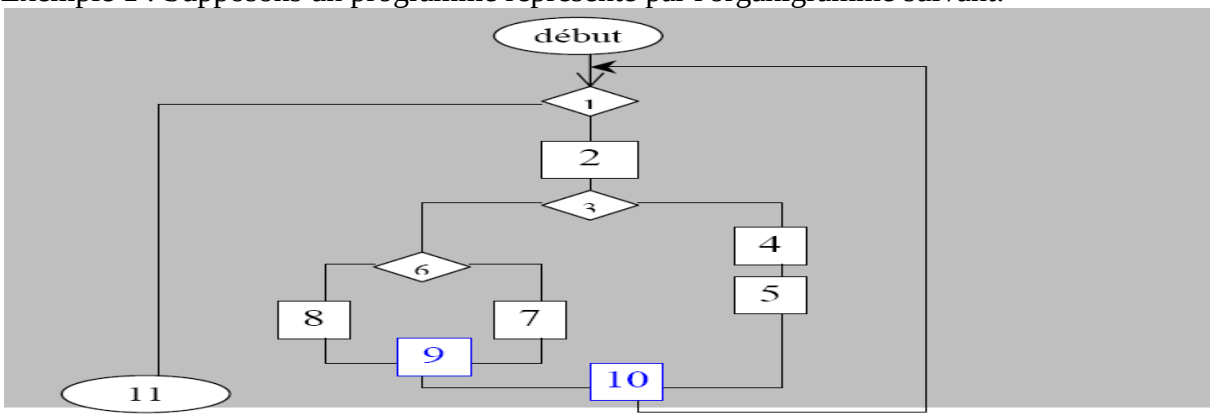


Mesure de complexité de Mac Cabr :

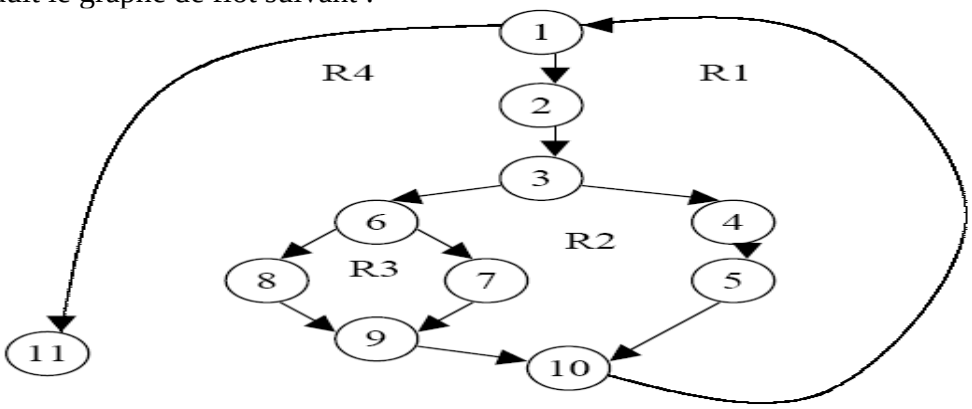
Cette mesure donne le nombre de chemins minimaux. Elle est donnée par la formule suivante qui correspond au nombre de régions du graphe de flot :

Nb.Arcs – Nb.Noeuds + 2 *Nombre cyclomatique*

Exemple 1 : Supposons un programme représenté par l'organigramme suivant:



On en déduit le graphe de flot suivant :



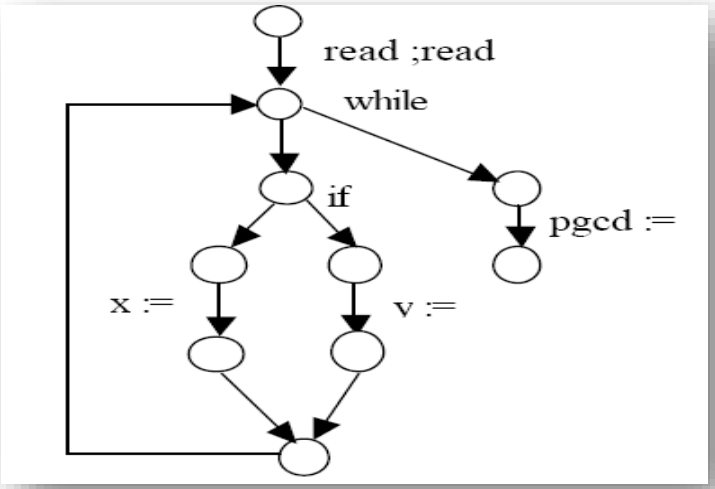
Donc le nombre cyclomatique est : Nb.Arcs – Nb.Noeuds + 2 = 13 – 11 + 2 = **4**

Pour vérifier, on regarde les chemins minimaux (un test par chemin pour tester toutes les possibilités du programme) :

1. 1.11
2. 1.2.3.4.5.10.1.11
3. 1.2.3.6.7.9.10.1.11
4. 1.2.3.6.8.9.10.1.11

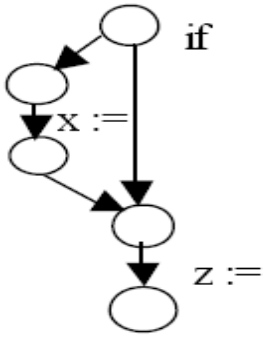
Exemple 2 : soit l'algorithme d'Euclide qui calcule le pgcd de 2 nombres (plus grand commun diviseur) et son graphe de contrôle.

```
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 ;
```



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 2, C = 11 – 10 + 2 = 3 ; c'est aussi la taille de notre jeu de tests.
Exemple 3 : soit le programme suivant :

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



Graphe de contrôle d'un si sans else

Dans l'exemple 3, C = 5 – 5 + 2 = 2 ; c'est aussi la taille nécessaire pour couvrir tous les arcs.