Test du logiciel

Module NI557-2006fev

Valérie Ménissier-Morain
Philippe Ayrault

Université Paris 6

Second semestre 2005-2006

Première partie I

Généralités

Vie d'un dossier de test

données	comportement attendu	comportement observé	comparaison des deux comportements
			OK/KO
			OK/KO
			OK/KO
			OK/KO
			OK/KO

jeu d'entrée
travail sur (et dès) la spécification

établi par le testeur avec la documentation (spécfication plus détaillée) du client d'après les normes de SDF qui s'applique à l'application

code écrit
par le développeur

exécution des tests
réalisé par le testeur

analyse
réalisé par un autre membre de l'équipe de test ou franchement plus tard par le même testeur

synthèse

→ **envoi au développeur: fiches d'anomalie du testeur**

développeur émet une nouvelle release avec une fiche de version:
les fichiers modifiés sont identifiés (informations provenant du gestionnaire de version)
–> le testeur reteste ces fichiers–là et remet à jour son dossier

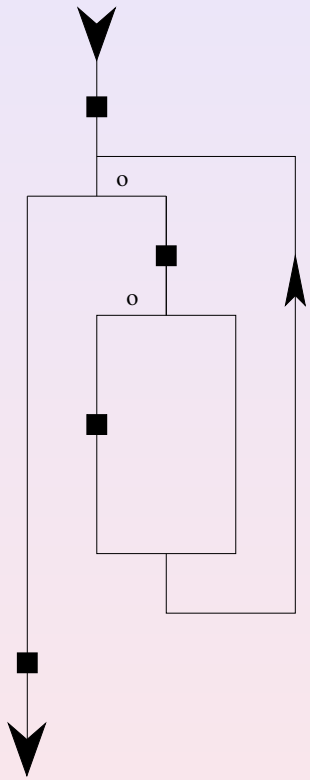
→ **envoi final au client (test fonctionnel, haut niveau, validation) et au vérificateur (logiciel de sécurité)**

+ PLAN DE TEST (définition de la politique de test et du banc de test), dès le début:
justification du choix de couverture, taux de couverture visé

Des jeux de tests : pour quoi faire ?

- Trace pour le contrôle (le vérificateur vérifie que le testeur a effectué les tests du jeu de test),
- Trace entre la conception et l'exécution du test (entre le moment de la spécification et celui où le code est écrit et peut être testé)
- Séparer l'exécution des tests de l'analyse des résultats (ce n'est pas au moment où on fait le test qu'il faut juger de la pertinence des sorties, ce n'est pas la même personne qui le fait la plupart du temps)

Flot de contrôle

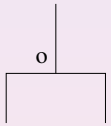



```
Trouve = False
Indice = -1

WHILE Trouve == False
     && Indice < Tab_Max {

   Indice = Indice + 1

   IF Tab[Indice] == Valeur {


       { Trouve = True }


   }

}

RETURN Indice
```

Légende :

: Sens de parcours du flot

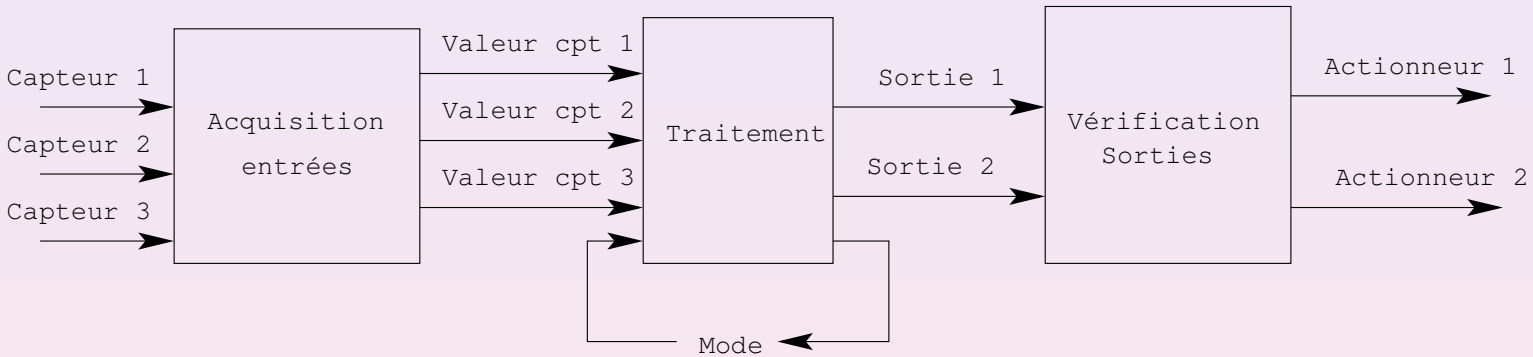
: Paquet d'instructions sans rupture de séquence



: Conditionnelle. La branche Vraie est marquée 'o'

L'analyse du flot de contrôle permet la construction des lignes du tableau de tests

Exemple de flot de données



L'analyse du flot de données permet la construction des colonnes du tableau de tests

Composant : exemple

```
vg1, vg2 : Int;

procedure proc (p1 : in Int;
                p2 : in out Int; p3 : out Int)
is
        l1, l2 : Int;
begin
        corps de la procédure
end proc;
```

Un composant (c'est-à-dire une procédure ou une fonction, *unit* en anglais) peut être caractérisé par :

- ses entrées :
  - paramètres d'entrée du composant
  - variables globales lues dans le composant
- ses sorties :
  - paramètres de sortie du composant
  - variables globales écrites dans le composant

identifiables soit syntaxiquement soit par une analyse du code

- les relations entre les entrées et les sorties de la procédure (corps de la procédure)
- les variables lues/écrites par les composantes appelées (causes d'indétermination si elles ne sont pas initialisées, voir bouchonnage à la fin de ce cours)

	Entrées du jeu			Sorties du jeu			
	P1	P2	VG1	P2	P3	VG1	VG2
Jeu 1							
Jeu 2							
Jeu 3							

En supposant que les variables globales `VG1` et `VG2` soient

- `VG1` lue et écrite
- `VG2` seulement écrite.

Procédure qui permet de prédire la valeur des sorties par rapport aux valeurs d'entrée.

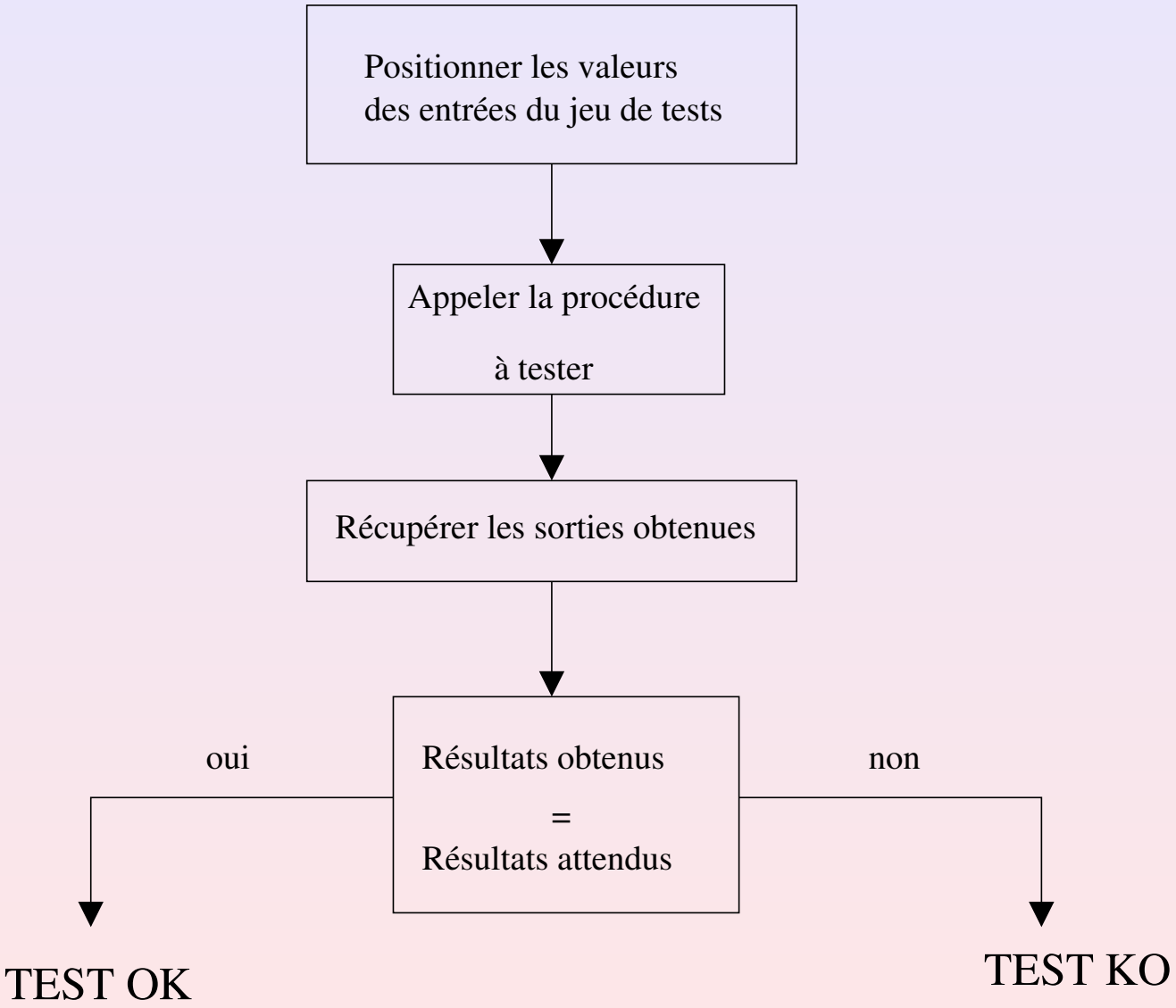
- Bas niveau : valeur précise
- Haut niveau : de la valeur précise à une simple propriété à vérifier (intervalle, propriété mathématique, propriété de sûreté, etc.)

Difficultés du test d'un logiciel

Toute la difficulté du test d'un logiciel provient de la détermination :

- d'un jeu d'entrées par rapport à une couverture de tests
- des valeurs de sortie d'une procédure par rapport à un jeu d'entrées déterminé (problème de l'oracle)
- traitement des composants appelés (voir le bouchonnage à la fin de ce cours)

Exécution d'un test



Importance du jeu de tests

On voit ici l'importance du jeu de tests, car tout repose sur lui :

- Le jeu de tests est une représentation pour des valeurs particulières de la spécification d'une procédure.
- Si l'on veut tester complètement un logiciel, il faut élaborer tous les jeux de tests permettant de couvrir la spécification.
- La combinatoire de n'importe quel problème même de très petite taille est trop importante pour faire un test exhaustif. Par exemple, pour vérifier l'addition sur les entiers 32 bits, cela nécessiterait 2^{64} jeux de tests.

Équilibre d'arbitrage OK/KO

- trop de OK : le client n'est pas content
- trop de KO : le développeur est mécontent (30 à 50 % des KO sont liés à des erreurs du testeur)

Il faut séparer (au moins temporellement) l'arbitrage et l'exécution des tests.

Couverture de tests, critère d'arrêt

Couverture de tests :

- niveau de confiance dans le logiciel pour le client,
- le contrat entre le client et le testeur, jugé par le vérificateur,
- pour le testeur, critère de mesure

Critère d'arrêt : quand s'arrêter de tester ?

- négatif : bugs bloquants, on n'est pas en mesure de tester la suite
- positif (taux de couverture)

On ne cherche pas 100% de la couverture à chaque fois
($>$ 90% bien fait, sauf normes de sûreté très spécifiques)

Choix de la couverture de tests

3 critères de choix :

- criticité du logiciel (normes de sûreté, imposé par le vérficateur)
- contraintes imposées au logiciel (facteurs qualité imposés par le client : temporelles, portabilité, etc.)
- type de logiciel (domaine : BD, réseaux, embarqué, etc.)

Notion de taux de couverture : mesure de la couverture (justification)
Chaque jeu de test doit augmenter la couverture de tests

Tests structurels

Couvertures basées sur la structure du code à tester (flot de contrôle et flot de données)

Tests fonctionnels

Couvertures basées sur la fonctionnalité réalisée par le code à tester

Il existe une centaine de couvertures de tests différentes, voir le site

`http://www.kaner.com/coverage.htm`

pour en avoir une liste. Nous ne verrons que les principales.

Ces deux familles de tests sont complémentaires et doivent obligatoirement être effectuées si l'on veut avoir une certaine confiance dans le logiciel produit.

- La première permet de valider l'implémentation d'un algorithme. Elle est automatisable.
- La seconde permet de vérifier que l'algorithme mis en œuvre permet de résoudre le problème posé. Elle relève du testeur humain essentiellement.

Deuxième partie II

Test structurel

But des tests structurels

- détecter les fautes d'implémentation.
- vérifier que le logiciel n'en fait pas plus que sa spécification et qu'il n'existe pas de cas de plantage (overflow, non initialisation, ...)
- Critère d'arrêt : lié à la structure du code et non à la fonctionnalité du logiciel

Couvertures des Tests Structurels

- Couverture du flot de contrôle (des instructions, des branches, des chemins, LCSAJ, etc.)
- Couverture de chaque condition logique
- Couverture des itérations
- Couverture de chaque donnée

Couverture des instructions

```
procedure p is
begin
if c1 then
s1;
endif;
if c2 then
s2;
endif;
end p;
```

test 1
vrai
s1
vrai
s2

Couverture des branches

`procedure p is`	test 1	test 2
`begin`		
`if c1 then`	vrai	faux
`s1;`	s1	
`endif;`		
`if c2 then`	faux	vrai
`s2;`		s2
`endif;`		
`end p;`		

Couverture des chemins

`procedure p is`
`begin`
`if c1 then`
`s1;`
`endif;`
`if c2 then`
`s2;`
`endif;`
`end p;`
	test 1
	
	vrai
	s1
	
	vrai
	s2
		test 2
		
		vrai
		s1
		
		faux
			test 3
			
			faux
			
			vrai
			s2
				test 4
				
				faux
				
				faux

Couverture du code source : couverture LCSAJ

Linear Code Subpath And Jump (LCSAJ) en anglais
Portion Linéaire de Code Suivie d'un Saut (PLCS) en français

Un peu de vocabulaire

Dans le graphe de contrôle de la procédure, on distingue :

- Les noeuds spéciaux (grisés sur le dessin) : l'entrée, la sortie, l'arrivée d'un branchement (saut)
- Les arcs spéciaux (traits épais sur le dessin) : les sauts ; l'arc de sortie en fait partie par défaut.

On appelle *LCSAJ* un chemin allant d'un noeud spécial à un autre comprenant exactement un saut entre l'avant-dernier et le dernier noeud.

Un exemple de PLCS en Basic

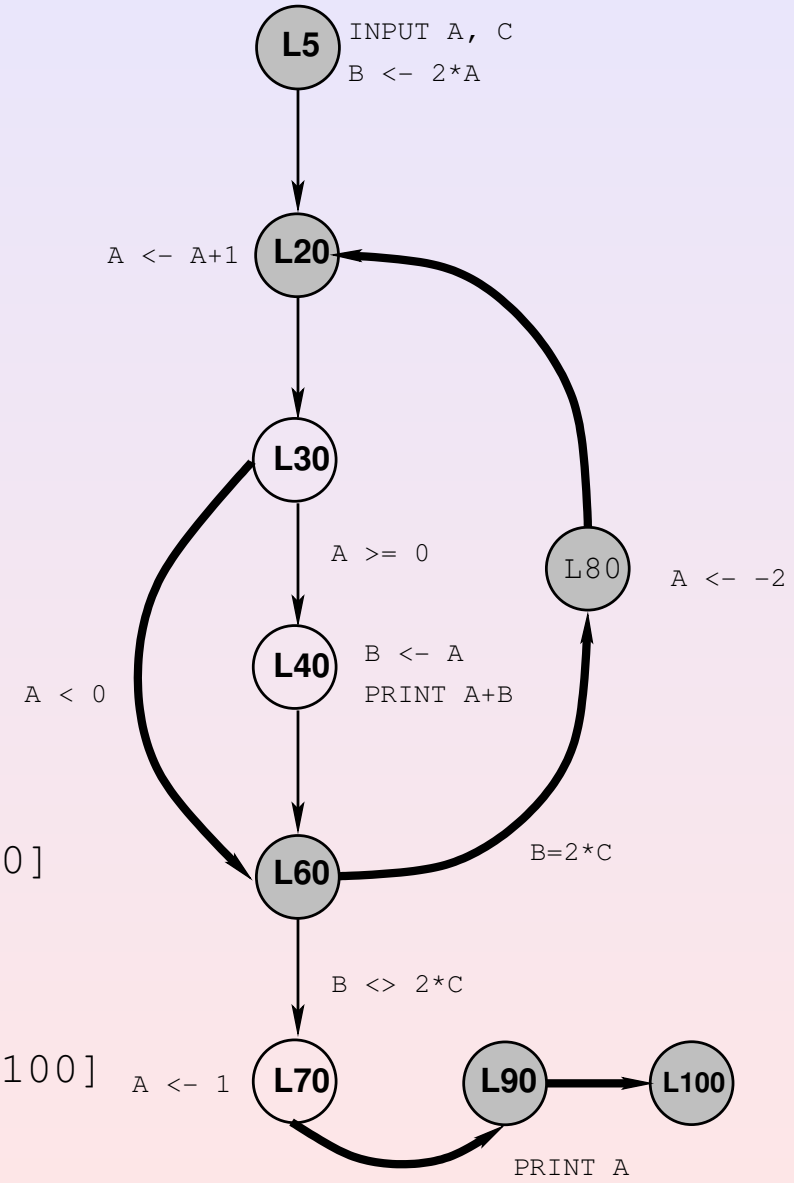
```
005 INPUT A,C
010 B=2*A
020 A=A+1
030 IF A < 0 THEN GOTO 60
040 B=-A
050 PRINT A+B
060 IF B=2*C THEN GOTO 80
070 A = 1; GOTO 90
080 A = -2; GOTO 20
090 PRINT A
100 END
```

PLCS

```
[L5,L20,L30,L60],[L5,L20,L30,L40,L60,L80]
[L5,L20,L30,L40,L60,L70,L90],[L20,L30,L60],
[L20,L30,L40,L60,L80],[L20,L30,L40,L60,L70,L90]
[L60,L80],[L60,L70,L90],[L80,L20],[L90,L100]
```

Chemins couvrant toutes les PLCS

```
[L5,L20,L30,L60,L70,L90,L100]
[L5,L20,L30,L40,L60,L80,L20,L30,L60,L70,L90,L100]
[L5,L20,L30,L40,L60,L70,L90,L100]
```



Couverture du code source : couverture LCSAJ

Adapté à un certain type de programmation, de bas niveau (sinon cela dépend du compilateur).
Calculé automatiquement d'ordinaire, inutile de s'inquiéter !

Avantage La couverture est sensiblement meilleure que la couverture des branches sans être trop explosive (bon ratio coût/niveau de confiance, entre couverture des branches et couverture des chemins).

Inconvénient Elle ne couvre pas la totalité de la spécification. Elle dépend du code du programme plus que de la structure réelle du graphe de contrôle.

Limite de la couverture du flot de contrôle nombre de tests / couverture

Programmes qui font la même chose :

```
procedure p1 is
begin
s := faux
if c1 then
if c2 then
s := vrai;
endif;
endif;
end p1;

3 tests (couverture des chemins)
```

```
procedure p2 is
begin
s := c1 and c2
end p2;

1 test (couverture des chemins)
```

Est-il raisonnable que le niveau de confiance (la couverture) dépende de la façon dont le développeur a écrit son code ?
Les couvertures vues précédemment (du flot de contrôle) sont trop sensibles au flot de contrôle ! $\longrightarrow$ il faut les complèter par d'autres types de couvertures . . .

Couverture des conditions logiques

Dans les couvertures du flot de contrôle : on s'appuie sur le flot de contrôle, sans regarder le contenu de chacun des noeuds.

Dans la couverture des conditions logiques, au lieu de passer une fois dans le cas `true` et une fois dans le cas `false`, on cherche les différentes façons de rendre la condition logique vraie ou fausse ; on augmente ainsi la confiance obtenue dans le logiciel (couverture).

Couverture des conditions logiques : table de vérité

procedure p **is**	t1	t2	t3	t4	t5	t6	t7	t8
begin								
if c1 or c2 or c3	(V,V,V)	(V,V,F)	(V,F,V)	(V,F,F)	(F,V,V)	(F,V,F)	(F,F,V)	(F,F,F)
then								
s1;	s1	s1	s1	s1	s1	s1	s1	
else								
s2;								s2
endif;								
end p;								

Couverture des conditions logiques

Plusieurs façons de couvrir les conditions logiques.
Les deux extrêmes :

- évaluer les conditions une à une (n tests)
- la table de vérité de chaque condition (2^n tests par condition)

Stratégie pour limiter la combinatoire

- stratégie pour l'opérateur OU, effectuer
  - un test avec toutes les sous-conditions à FAUX
  - un test pour chaque sous-condition à VRAI unitairement (circulation d'un 1)
  - un test avec toutes les sous-conditions à VRAI

  Par conséquent, $n + 2$ tests par condition (sur l'exemple, on fait 5 tests au lieu de 8 : on évite les tests `t2`, `t3`, `t5`)
- stratégie pour l'opérateur ET, effectuer
  - un test avec toutes les sous-conditions à VRAI
  - un test pour chaque sous-condition à FAUX unitairement (circulation d'un 0).

  Par conséquent, $n + 1$ tests par condition

Cette stratégie pour limiter la combinatoire est appelée *couverture des conditions logiques*.

`procedure p is`
`begin`
`while c1`
`loop`
`s1;`
`end loop;`
`end p;`
	test1
	
	(F)
		test2
		
		(V,F)
		
		s1
			test 3
			
			(V,...,V,F)
			
			s1;...;s1

Couverture des itérations

Pour chaque boucle :

- 0 itération
- 1 itération
- max - 1 itérations
- max itérations
- max + 1 itérations.

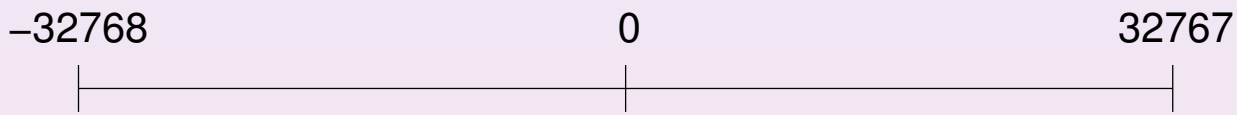
Dans les couvertures précédentes : on ne s'est intéressé ni au format des données manipulées ni aux *valeurs numériques* permettant de rendre une condition vraie ou fausse.

La couverture des données consiste à choisir :

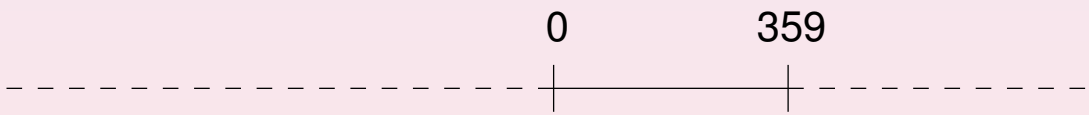
- les « *bonnes valeurs numériques* » pour rendre une condition vraie ou fausse (valeurs remarquables)
- les valeurs limites des données d'entrée
- les valeurs numériques permettant d'obtenir les valeurs ou les erreurs prévisibles (overflow, division par zéro, ...)

Couverture de chaque donnée : domaine d'une donnée

Domaine informatique



Domaine fonctionnel



Couverture d'une donnée

	test1	test2	test 3	test 4	test 5	test 6
procedure p **is**						
begin						
if e > 30	359	31	30	0	-32768	32767
then						
s1;	s1	s1				s1
endif;						
end p;						

Une procédure évalue les données pour des valeurs remarquables.
Par exemple l'instruction conditionnelle : $e > 30$ montre que la valeur 30 est une valeur remarquable pour la donnée e.

Il est possible de découper le domaine de chaque donnée en *classes d'équivalence*. Une classe définit un comportement propre à la donnée.
Par exemple, sur l'exemple précédent, la donnée peut être découpée en 2 classes d'équivalence $[val\_min, 30]$, $[31, val\_max]$.

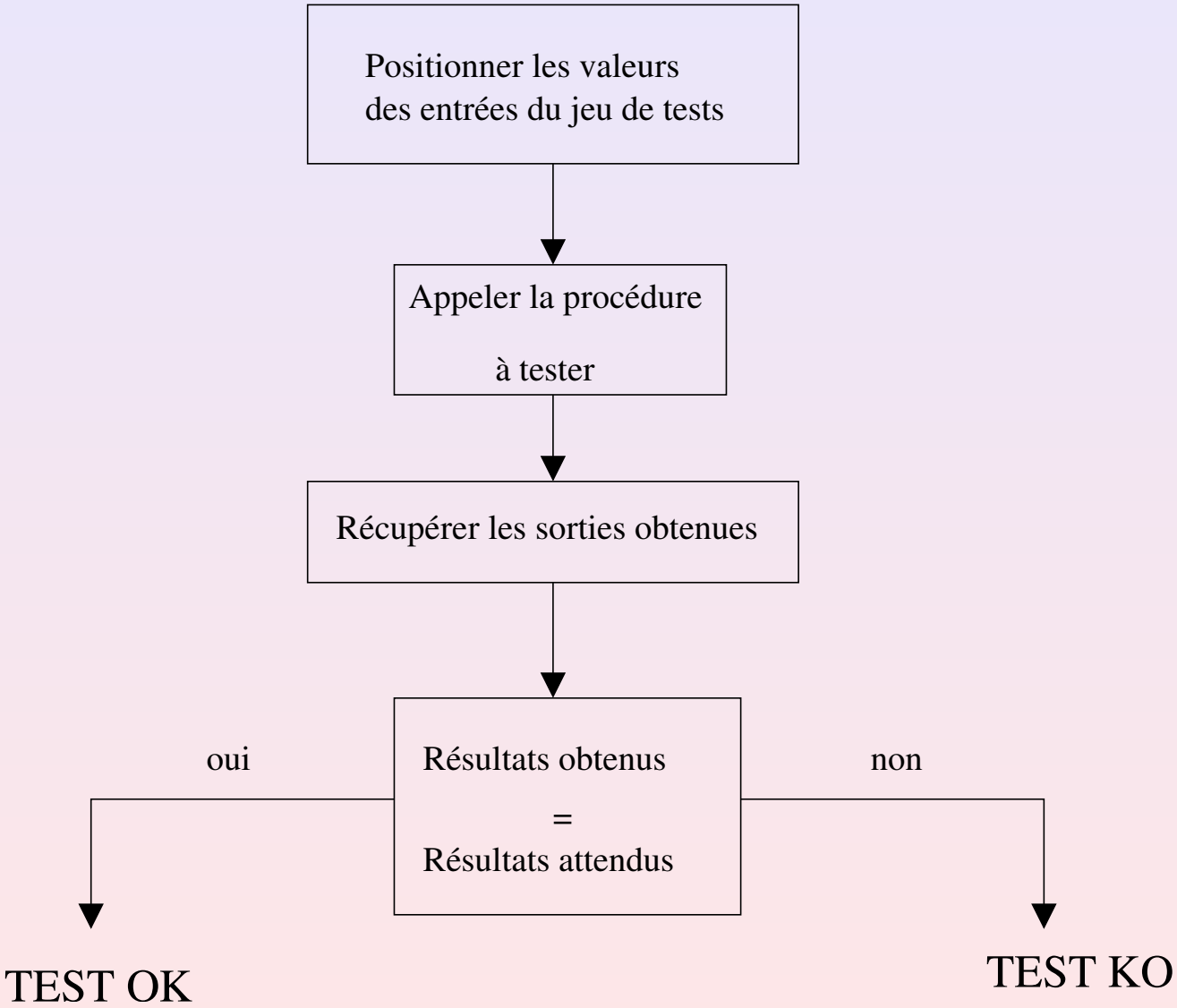
Du point de vue du testeur, cela nécessite deux types de test :

- les *tests aux limites* pour vérifier le comportement du logiciel pour chaque borne de chaque classe d'équivalence (les limites fonctionnelles des données et les valeurs remarquables)
- les *tests hors limites* pour vérifier le comportement du logiciel avec des valeurs en dehors de son domaine fonctionnel.

Pour les test structurels, il faut écrire des programmes de tests pour exécuter les tests.
Ces programmes de tests prennent en entrée les jeux de tests et sont « linkés » avec le composant à tester.

Ces programmes sont généralement écrits automatiquement par les logiciels d'exécution de tests (par exemple : RTRT de Rational, Cantata de IPL ?).

Rappel : exécution d'un test

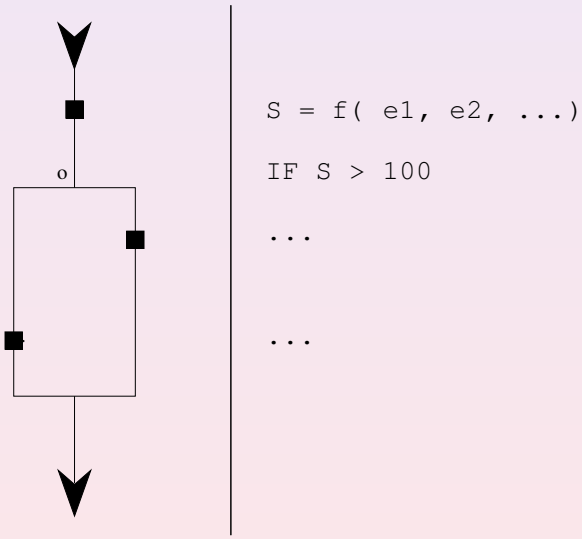


Les programmes de test sont architecturés comme suit :

- Initialisation des variables globales utilisées par le composant à tester à partir des valeurs d'entrée des jeux de tests
- Appel du composant à tester avec les paramètres initialisés à partir des valeurs d'entrée des jeux de tests
- Récupération des sorties produites par le composant (paramètres en sortie et variables globales)
- Comparaison des valeurs des sorties obtenues avec les valeurs des sorties décrites dans les jeux de tests
- Enregistrement et affichage du résultat de test.

Le bouchonnage

Le composant fait appel à une fonction externe `f`. Que l'on dispose ou non de cette fonction, on souhaite dans un premier temps tester le comportement du composant indépendamment de celui de `f`.
La technique du bouchonnage consiste donc à :



- ajouter une *entrée artificielle* dans les jeux de test. Cette entrée correspondra à la sortie de la fonction `f`
- écrire un *bouchon* pour remplacer le véritable code de la fonction `f` : ce code ne fait que retourner la valeur positionnée dans le jeu de test.

Ainsi le testeur peut maîtriser complétement les tests réalisés.

Troisième partie III

Test fonctionnel

Aussi appelé *tests boîte noire*

Son but

- Vérifier le comportement d'un logiciel / spécification (fonctions non conformes ou manquantes, erreurs d'initialisation ou de terminaison du logiciel)
- Vérifier le respect des contraintes (performances, espace mémoire, etc.) et des facteurs qualité associés au logiciel (portabilité, maintenabilité, etc.)

Couvertures de tests fonctionnels : « qualitatives ».
On ne peut connaître *a priori* le nombre de tests nécessaires
Se baser sur le seul élément dont on est sûr : *la spécification*.

Rappel : spécification du logiciel

Une spécification doit décrire **au minimum** :

- les fonctions à réaliser par le logiciel,
- les interfaces de ce logiciel
- les contraintes fixées au développeur.

Exemples de contraintes :

- performances temporelles
- performances spatiales
- contraintes matérielles
- critères de sécurité
- portabilité

Le test fonctionnel : quatre grandes catégories de couvertures

Que teste-t-on ? Comment ?

- Quoi ? : couvertures des tests
- Comment ? : analyse partitionnelle pour le test des fonctions de la spécification

Quatre grandes catégories

– Tests nominaux
– Tests aux limites fonctionnels
} test des fonctions du logiciel

– Tests de robustesse) facteurs qualité de robustesse
– Tests de conformité) autres facteurs qualité et contraintes

Les tests des fonctions du logiciel : tests nominaux, tests aux limites

- *Tests nominaux* : vérifier la conformité par rapport à la spécification pour un comportement normal du logiciel
- *Tests aux limites* : vérifier le comportement aux limites fonctionnelles du logiciel

Tous les tests permettant de valider la robustesse du logiciel vis-à-vis de son environnement.

Par exemple

- les tests hors limites fonctionnelles
- les tests en charge
- les pannes des équipements externes

Vérifier les contraintes associées au logiciel

Par exemple

- les tests de performance
- les tests d'intrusion
- les tests d'ergonomie (Interface Homme-Machine)
- les tests de portabilité (matériel, OS), d'interchangeabilité

L'analyse partitionnelle, une solution pour les jeux d'entrées

Première idée : force brutale (effectuer le produit cartésien des domaines des entrées du programme)
Défaut : nombre de tests à réaliser astronomique (exemple : addition de 2 entiers de 32 bits $\longrightarrow 2^{64}$ jeux de tests)
On se contenterait de valider chacun des comportements du logiciel pour une valeur particulière *représentative*.

$\downarrow$

Seconde idée : partitionner ce produit cartésien en classes d'équivalence des entrées (ensemble des entrées aboutissant au même comportement fonctionnel)
Sur notre exemple : 2 tests (1 sans débordement, 1 avec débordement)

Méthode couramment utilisée pour écrire les jeux de tests fonctionnels, appelée : *analyse partitionnelle*.

Analyse partitionnelle : la recette

Pour chaque fonction de la spécification à :

- déterminer les entrées de la fonction ainsi que leur domaine
- à partir de la partie contrôle de la spécification, découper le domaine des entrées en classes d'équivalence
- pour chaque classe d'équivalence :
  - sélectionner un élément dans la classe
  - à partir de la partie commande de la spécification, déterminer la valeur des sorties pour l'élément sélectionné.

Valeurs de sortie ou à défaut propriétés de ces valeurs

Problème de l'oracle

- algorithme trop complexe (régulation en automatisme)
- toutes les entrées nécessaires au calcul de la sortie ne sont pas accessibles au testeur (positionnées par le développeur, horloge système, etc).

Rappel : partition, classes d'équivalence

Partition

Soit D un domaine.
Les ensembles $C_1 \ldots C_n$ forment une partition de classes d'équivalence sur D si :

$$\forall i, j, C_i \cap C_j = \emptyset \qquad \textbf{Règle 1 (exclusion mutuelle)}$$

$$\bigcup_{i=1}^{n} C_i = D \qquad \textbf{Règle 2 (recouvrement)}$$

Première règle violée : spéc pas déterministe
Deuxième règle violée : spéc non complète

Programme calculant : $\sqrt[2]{\frac{1}{x}}$ sur les entiers

3 classes d'équivalence :

$x < 0$

$x = 0$

$x > 0$

Sur les 3 classes d'équivalence, une seule est valide.

Détermination des classes d'équivalence

Langage formalisé	Détermination des chemins de la spécification
Automate, Réseau de Petri, . . .	Parcours de l'automate, . . .
Matrice causes/effets	Parcours de la matrice
Langage naturel	Remodélisation de la spécification en langage formalisé ou automate

Dans ce cas, la spécification n'est pas testable en l'état. Il faut donc soit la refuser, soit :

- réaliser un modèle de cette spécification dans le formalisme le mieux adapté
- faire valider ce modèle par l'équipe de développement et le client (est-ce bien cela que vous vouliez construire ?)
- déterminer les classes d'équivalence sur le modèle.

Ce processus de remodélisation permet très souvent de trouver des anomalies dès la spécification :

- incohérence entre différentes parties de la spécification
- incomplétude des cas traités

- Sélection d'une valeur « intelligente » dans la classe d'équivalence
- Varier les valeurs à l'intérieur d'un même intervalle (entropie)

Exemple de partition en classes d'équivalence

Fonction : `Produit_valeurs_absolues`
Entrées : `E1`, `E2`
Sorties : `S`
Traitement :
Cette fonction calcule la valeur absolue du produit des entrées `E1` et `E2`.

Classes d'équivalence pour chaque entrée

E1	E2
[Min_Int,-1]	[Min_Int,-1]
[0, Max_Int]	[0, Max_Int]

Choix « intelligent » des valeurs dans les classes d'équivalence

Classes d'équivalence pour toutes les entrées

E1		E2	
[Min_Int,-1]	-734	[Min_Int,-1]	-525
[Min_Int,-1]	-7445	[0, Max_Int]	3765
[0, Max_Int]	7643	[Min_Int,-1]	-765
[0, Max_Int]	9864	[0, Max_Int]	3783

Tests aux et hors limites fonctionnelles

- Tests aux limites fonctionnelles : sélection de valeurs aux bornes de chaque classe d'équivalence fonctionnelles
- Tests hors limites fonctionnelles : sélection de valeurs hors bornes de chaque classe d'équivalence fonctionnelles

Tests aux et hors limites fonctionnelles pour l'exemple précédent

Si les entrées `E1` et `E2` ont un domaine fonctionnel de :
`[-100, 100]`

Tests aux limites fonctionnelles

E1		E2	
[-100,-1]	-100	[-100,-1]	-57
[-100,-1]	-1	[0, +100]	64
[0, +100]	0	[-100,-1]	-5
[0, +100]	100	[0, +100]	98
[-100,-1]	-59	[-100,-1]	-1
[0, +100]	48	[-100,-1]	-100
[-100,-1]	-63	[0, +100]	0
[0, +100]	75	[0, +100]	100

Tests hors limites fonctionnelles

E1		E2	
[-100,-1]	-234	[-100,-1]	-42
[0, +100]	174	[0, +100]	39
[-100,-1]	-84	[Min_Int, -1]	-115
[0, +100]	48	[0, +100]	120

Vérifier le comportement du logiciel face à des événements non spécifiés ou dans des situations dégradées.

- Tests en charge
- Tests des pannes des équipements externes
- etc.

Vérifier le comportement du logiciel en cas de stress du logiciel tel que :

- avalanche d'alarmes
- saturation des réseaux
- saturation des requêtes
- ...

Exemple : la saturation de Yahoo fin 1999.

Tests de pannes des équipements externes

Simuler des pannes sur les équipements en interface avec le logiciel afin de vérifier son comportement.
Par exemple :

- arrêt inopiné de l'équipement
- débranchement brutal de l'équipement
- changement brusque de valeurs
- . . .

Tests de pannes des équipements externes, connaissances requises

Ces tests nécessitent une bonne connaissance du hardware afin de spécifier les bons modes de défaillance des équipements.
Par exemple, connaître les cas de défaillance d'un interrupteur :

- collage à 1 ou à 0
- bagottements intempestifs
- parasitage à différentes fréquences.

Le but des tests des interfaces est double :

- vérifier les interfaces logicielles entre les composants un sous-système logiciel
- vérifier les interfaces physiques entre le logiciel et la machine cible (carte sur laquelle tourne le logiciel)

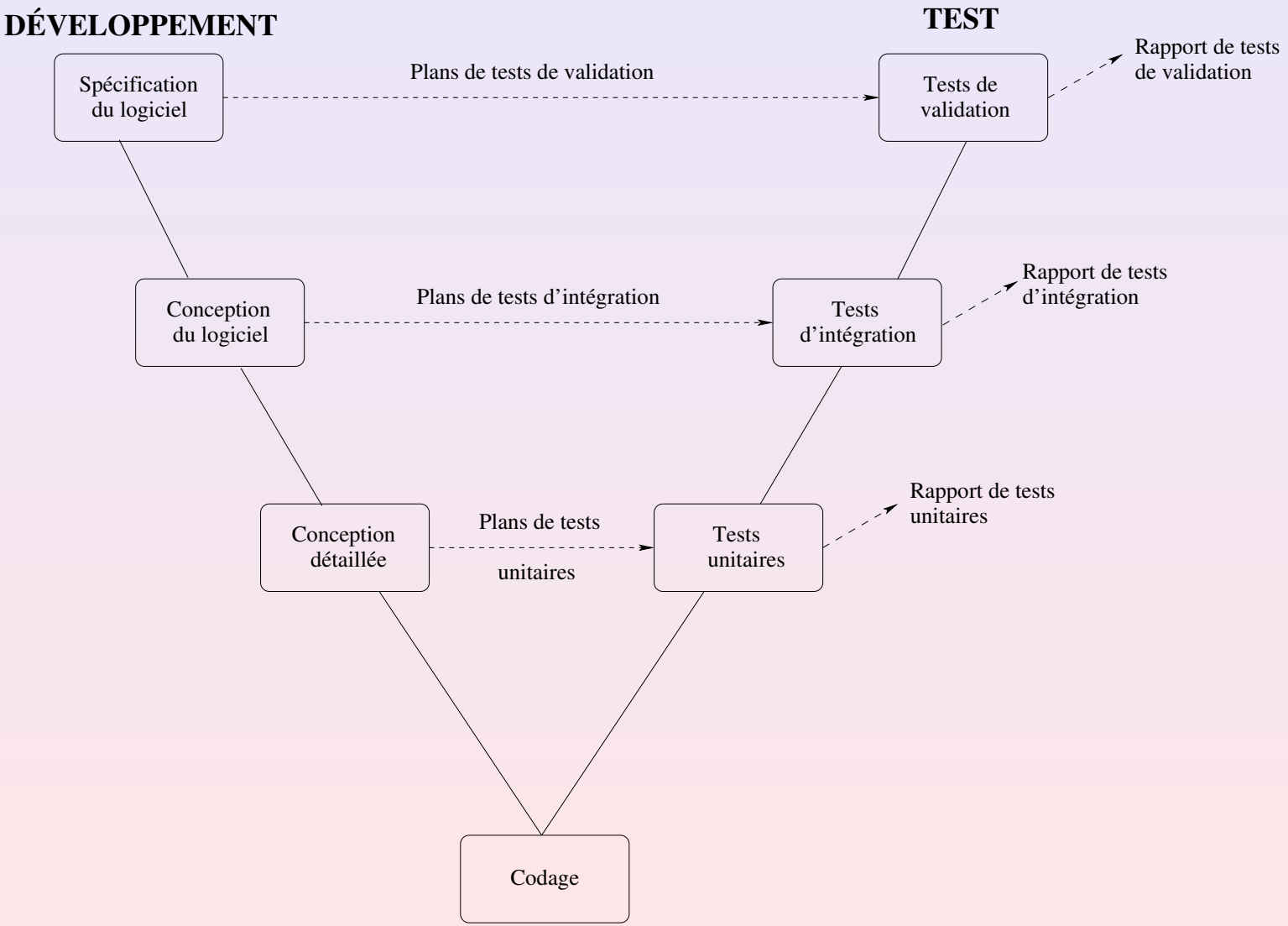
Conclusion pour les tests fonctionnels

Ce ne sont que des exemples pour le test de robustesse et le tests de pannes des équipements externes, cela dépend énormément du métier pour lequel le logiciel est développé.

Quatrième partie IV

Les phases de test

Les phases de tests



- Le travail du testeur ne commence pas après la phase de codage. Elle débute au même moment que pour le développement et se poursuit en parallèle avec celui-ci.
- Afin d'éviter l'intégration « big bang » de l'ensemble du logiciel, les phases de tests valident le logiciel depuis le module jusqu'au logiciel complet.

- Chaque phase de tests a un but précis dans la découverte des anomalies et dans la construction de la confiance attendue dans le logiciel. Elle représente une brique sur laquelle repose les phases suivantes.
- Un « trou » dans la couverture d'une phase entraîne automatiquement, soit le masquage de problème qui pourront entraîner des incidents graves lors de l'exploitation, soit une difficulté de diagnostic lors de la mise en évidence d'une anomalie durant les phases suivantes.
- Leitmotiv : **Plus un défaut est découvert tardivement, plus il coûte cher à corriger**.

Travail durant les phases de descente du cycle

Durant les phases de descente du cycle, le testeur élabore les *Plans de Tests du Logiciel* et fabrique les bancs de tests.

Les plans de tests décrivent essentiellement :

- la stratégie de tests mise en place
- les moyens mis en oeuvre (matériel, logiciel et humain)
- l'ensemble des fiches de tests.

Travail durant les phases de remontée du cycle

Durant les phases de remontée du cycle, le testeur exécute les fiches de tests décrites dans les plans et produit les rapport de tests associés. Ces rapports contiennent essentiellement :

- la synthèse des résultats de tests
- les résultats de tests détaillés
- la trace d'exécution des tests.

But Validation de la conformité de chaque composant logiciel pris unitairement par rapport à sa spécification détaillée.

Quand ? Dès qu'une pièce de code a été codée et compilée correctement

Type de tests structurels

Comment ? sur machine hôte, généralement sans banc de tests

Qui ? Pour les logiciels de faible criticité, elle peut être réalisée par l'équipe de développement (mais pas par le développeur ayant codé la pièce de code).

Les Tests d'Intégration (TI)

But Validation des sous-systèmes logiciels entre eux
Tests d'Intégration Logiciel/Logiciel (interface entre composants logiciels)
Tests d'Intégration Logiciel/Matériel (interface entre le logiciel et le matériel)

Quand ? Dès qu'un sous-système fonctionnel (module, objet) est entièrement testé unitairement

Type de tests des interfaces

Comment ? logiciel/logiciel généralement sur machine hôte
logiciel/matériel sur machine cible avec un banc de test minimal (simulation des entrées, acquisition des sorties).

Qui ? toujours par une équipe de tests indépendante de l'équipe de développement.

Les Tests de Validation (TV)

But Vérifier la conformité du logiciel à la spécification du logiciel

Quand ? Dès que l'ensemble des sous-systèmes fonctionnels ont été testé et intégré

Type de tests fonctionnels et de robustesse

Comment ? toujours réalisés sur machine cible et nécessitent généralement la fabrication d'un banc de tests élaboré

- Ces tests sont toujours réalisés par une équipe de tests indépendante de l'équipe de développement.