

Tests des logiciels

Jean-François Pradat-Peyre

2016

3 : Techniques de test dynamiques :

Tests boîte noire

■ Tests dynamiques

- On exécute le programme avec des valeurs en entrée et on observe le comportement

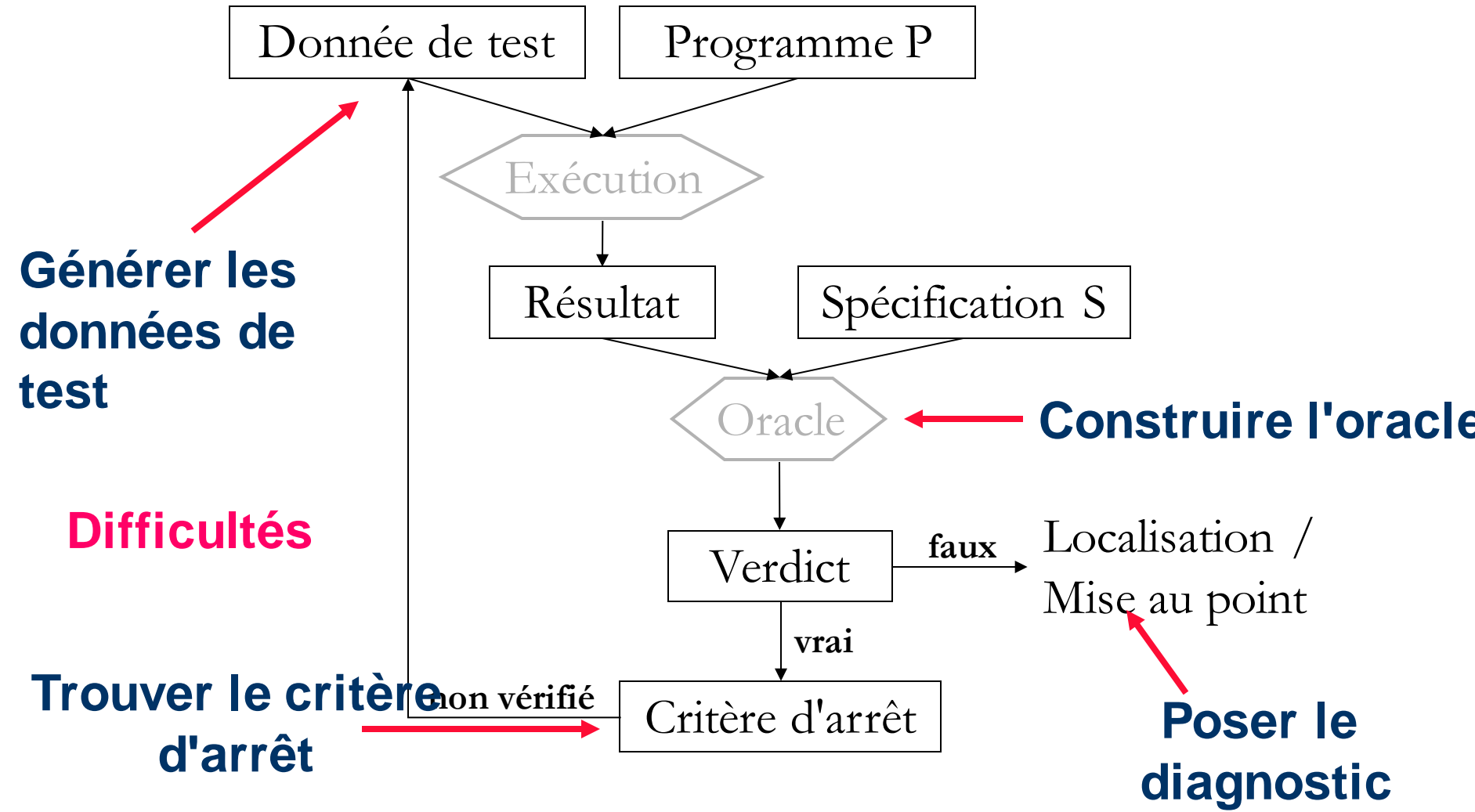
■ Tests statiques

- Revue (analyse sans exécuter le programme ou faire fonctionner le produit)
- Analyse automatique (vérification de propriétés, règles de codage...)

Techniques de tests – Techniques dynamiques

- ❖ Panorama et problématique
- ❖ Tests « boîte noire »
- ❖ Tests « boîte blanche » et couverture de code

Processus de test « dynamique »



La problématique du test dynamique

- Soit **D** le domaine d'entrée d'un programme **P** spécifié par **S** ;
on voudrait pouvoir garantir que

- $\forall x \in D, P(x) = S(x)$

i.e. pour toute donnée valide, le programme se comporte comme sa spécification

- Problème : le test exhaustif est **impossible** dans la plupart des cas

- Domaine D trop grand, voire infini
 - Trop long et trop coûteux

- Recherche d'un ensemble de données de test **TI** tel que
 - TI est inclus dans D, *fini et bien plus petit*
 - Si pour tout x dans **TI**, $P(x) = S(x)$ alors pour tout x dans D, $P(x) = S(x)$
- Le critère d'arrêt des tests est :
 - {données de test} = **TI**
- Problème :
 - Comment construire **TI** ?

Génération des données de tests :

Tests « boîte noire » vs « boîte blanche »

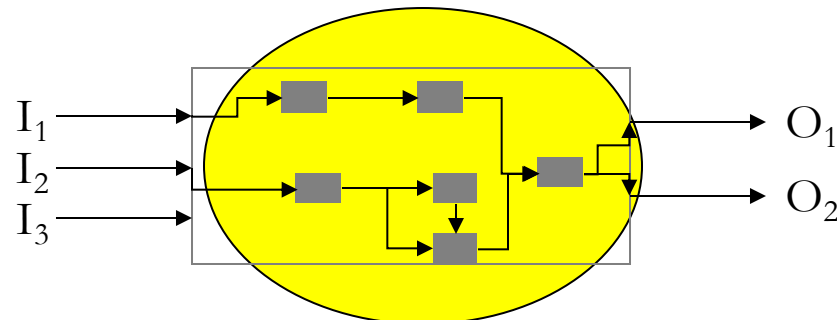
Test fonctionnel (test boîte noire)

Utilise la description des fonctionnalités du programme



Test structurel (test boîte blanche)

Utilise la structure interne du programme



- **Spécification formelle**
 - Modèle B, spécifications algébriques
 - Automate, système de transitions

- **Modèle UML (semi-formel)**
 - Use cases
 - Diagramme de classes (+ contrats)
 - Machines à états / diagramme de séquences

- **Description en langage naturel (informel)**

Données utiles pour les tests « boîte blanche »

- Données issues d'un modèle du code
 - modèle de contrôle (conditionnelles, boucles...)
 - modèle de données
 - modèle de flot de données (définition, utilisation...)

- Données issues d'un modèle de relation inter modules
 - modèle de dépendance

- ➔ Utilisation importante des parcours de graphes
 - critères basés sur la couverture du code

- Génération automatique aléatoire
- Génération automatique aléatoire contrainte
 - mutation
 - test statistique
- Génération déterministe « à la main » guidée par des méthodes
 - construction de classes d'équivalence
 - tests aux limites
 - tables de décision
 - diagramme d'états
 - critères de couverture
 - etc.
- Génération automatique guidée par les contraintes ou les spécifications

- Exécution de l'ensemble des cas de tests prévus
- Critères de couverture obtenus
- Atteinte d'un seuil (nombre ou taux d'erreurs trouvées)
- Ressources épuisées

Techniques de tests – Techniques dynamiques

- ❖ Panorama et problématique
- ❖ Tests « boîte noire »
- ❖ Tests « boîte blanche » et couverture de code

Tests « boîte noire »

- Objectif
 - Générer des cas de tests en utilisant des spécifications (**non le code**)
- Données pouvant être utilisées
 - type des paramètres d'une méthode
 - précondition sur une méthode
 - ensemble de commandes sur un système
 - cas d'utilisation
 - ...
- **On ne peut pas tout explorer** : il faut choisir de « bonnes » valeurs
 - Génération aléatoire (error guessing)
 - Partitionnement en classes d'équivalence
 - Test aux limites
 - Utilisation de graphes causes – effets / tables de décision
 - Utilisation de diagramme états / transitions

Trouver les bons cas de tests :
génération (semi) aléatoire

Générer des cas de tests (semi) aléatoirement (1/2)

- Simple à mettre en œuvre mais relativement inefficace (la probabilité de rejouer les mêmes cas tend très rapidement vers 1)

- Peut être amélioré en utilisant l'expérience du testeur
 1. Construire une liste d'erreurs possibles ou situations conduisant à des erreurs
 2. se baser sur des modèles d'erreurs
 3. Développer des cas de tests pour couvrir le modèle d'erreurs
 4. Maintenir les modèles d'erreurs

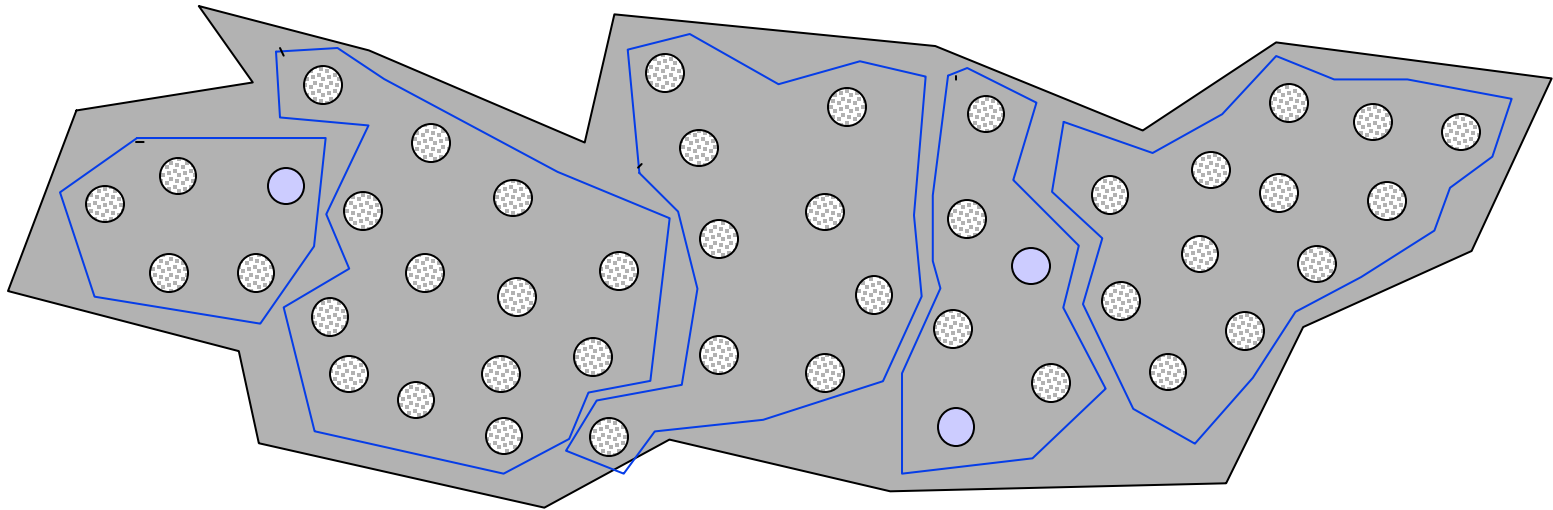
Générer des cas de tests (semi) aléatoirement (2/2)

- Exemple : fonction de tri de tableau
- Modèle d'erreur :
 - Tableau vide
 - tableau trié
 - tableau trié à l'envers
 - grand tableau non trié
 - ...
- On génère des cas de tests correspondant

**Trouver les bons cas de tests :
partitionnement et classes d'équivalence**

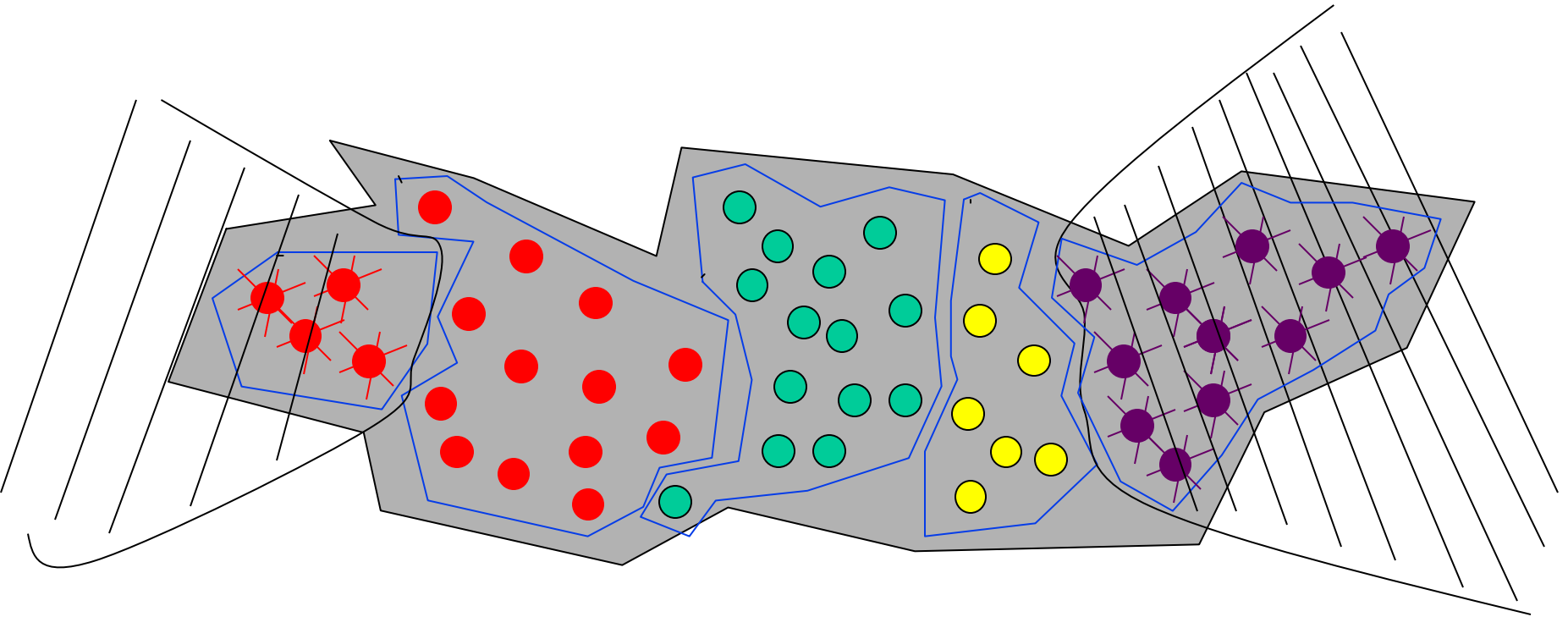
Partitionnement et classes d'équivalence : Principe (1/4)

- Partitionner le domaine des données en Classes d'Equivalence selon la spécification vis à vis d'une condition externe à l'élément testé (par exemple une propriété sur les valeurs d'entrée ou de sortie)



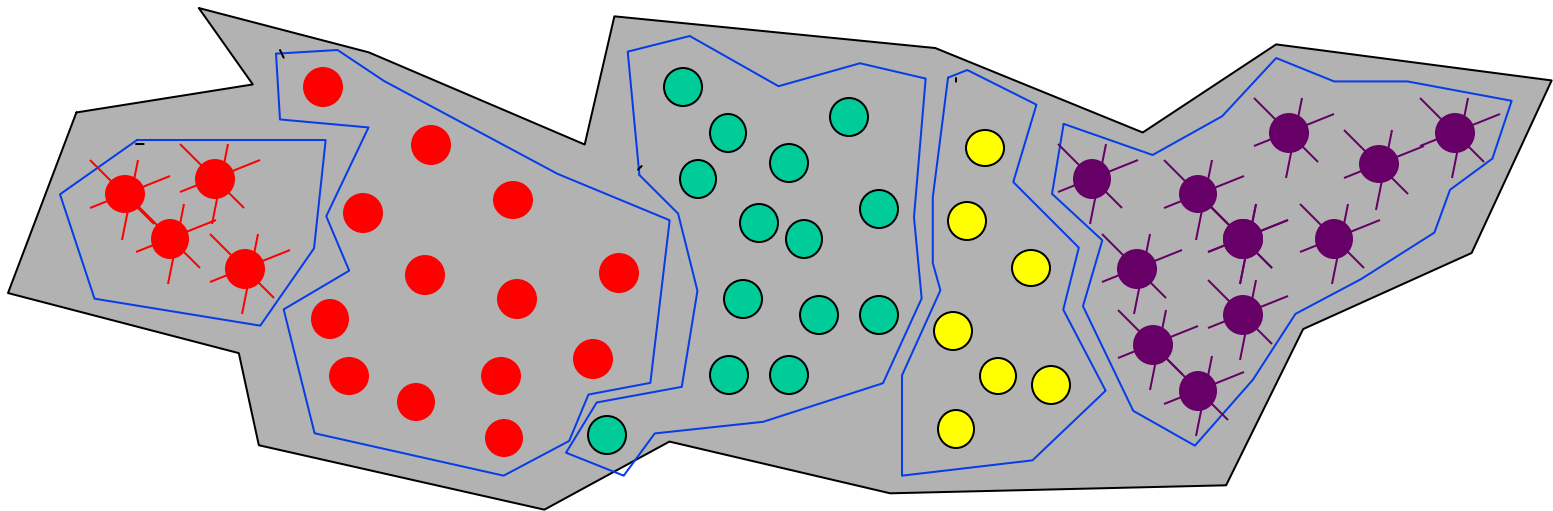
Partitionnement et classes d'équivalence : Principe (2/4)

- On identifie au niveau des spécifications les entrées valides et les entrées invalides
- Construire des classes d'équivalence Valides / Invalides correspondant à ces entrées



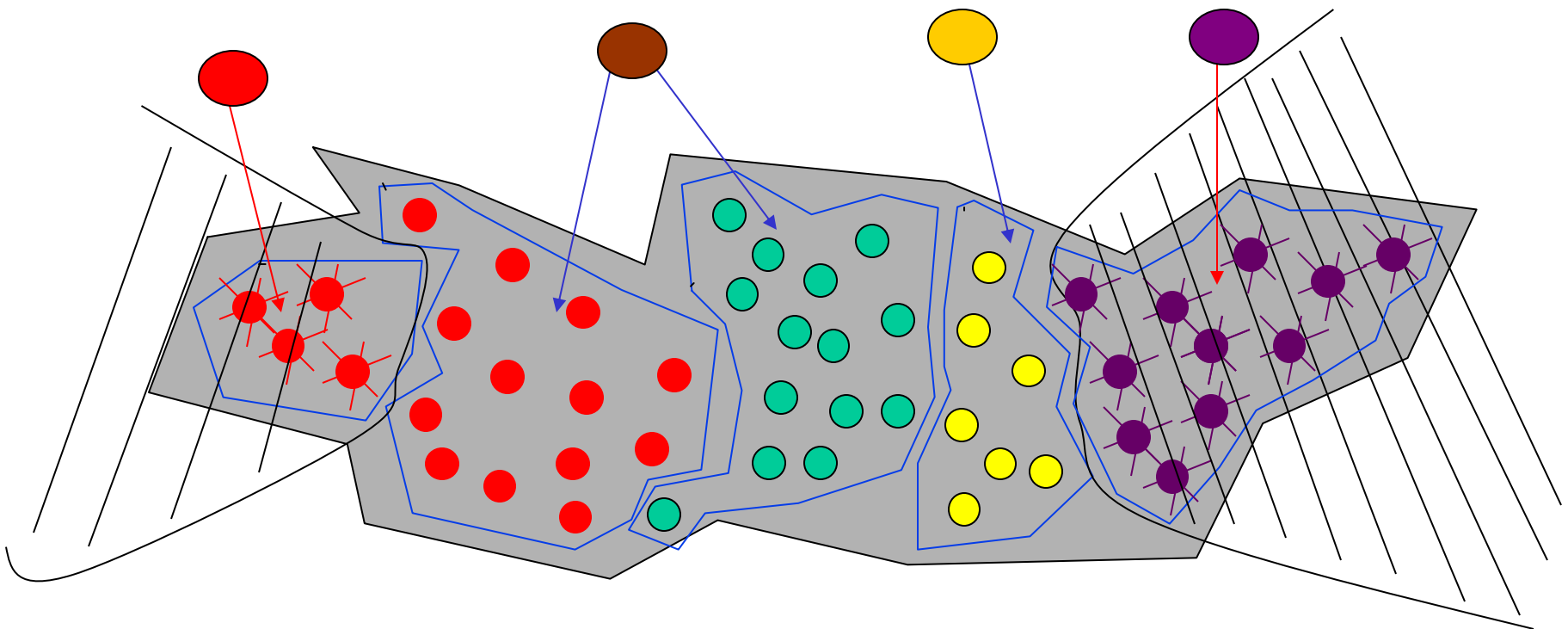
Partitionnement et classes d'équivalence : Principe (3/4)

- Les éléments d'une même classe doivent le même comportement vis à vis de la propriété à tester (la même probabilité de générer une erreur)
 - Si le résultat est **correct** avec **un** élément il l'est pour **tous** les éléments de la classe
 - Si le résultat est **incorrect** avec **un** élément il l'est pour **tous** les éléments de la classe



■ Générer des cas de test

1. Choisir des cas de test couvrant le plus de classes valides possibles (dans le cas où le partitionnement n'est pas parfait au sens mathématique)
2. Choisir un cas de test par classe invalide, ne couvrant que cette classe
3. Répéter jusqu'à ce que toutes les classes soient couvertes



Petit exemple

- Soit une fonction qui attend en entrée un numéro de département (en métropole)
- La donnée d'entrée doit être comprise entre 1 et 95 :

On a comme classes d'équivalence

Validité des entrées	Classes d'équivalence	Données de test
Entrées valides	[1 – 95]	11
Entrées invalides	[minInt – 1 [-30
Entrées invalides] 95 – MaxInt]	100

Trouver les classes d'équivalence (1/3)

En pratique, la construction des classes d'équivalence est uniquement basée sur des heuristiques :

1. Si une condition d'entrée ou de sortie définit un intervalle de valeurs (par exemple le numéro de département est compris entre 1 et 95) :
 - 1 CE valide (dans l'intervalle)
 - 2 CE invalides (une à chaque bout de l'intervalle)

2. Si une condition d'entrée ou de sortie définit N valeurs (par exemple un tableau) :
 - 1 CE valide
 - 2 CE invalides (vide et plus de N)

Trouver les classes d'équivalence (2/3)

3. Si une condition d'entrée ou de sortie définit un ensemble de valeurs (par exemple une énumération de valeurs)
 - ❑ 1 CE valide (dans l'ensemble) ou 1 CE par valeur si le programme semble les différencier
 - ❑ 1 CE invalide (hors de l'ensemble) si possible (langage fortement typé)
4. Si une condition d'entrée ou de sortie définit une contrainte devant être vérifiée (par exemple le premier caractère de l'identifiant doit être une lettre) :
 - ❑ 1 CE valide (la condition est vérifiée)
 - ❑ 1 CE invalide (la condition n'est pas vérifiée)
5. Si une CE semble trop complexe ou posséder des éléments traités différemment par le programme :
 - ❑ décomposer la CE en 2 ou plusieurs CE

Un exemple

- Quelles données de test pour une méthode *Lendemain* qui calcule le lendemain d'une date (*jour, mois, année*) passée en paramètre.
- Données: *mois, jour, an* représentant une *date*
 - $1 \leq \text{mois} \leq 12$
 - $1 \leq \text{jour} \leq 31$
 - $1000 \leq \text{an} \leq 3000$
- Résultat : date du jour suivant la date donnée
 - Doit considérer les années bissextiles : Année bissextile si divisible par 4 et pas siècle sauf si multiple de 400

Construction des classes d'équivalence (1/3)

- Prise en compte des contraintes d'intervalle

CE Valides	CE Invalides
1<= jour <=31 <i>ET</i> 1<= mois <= 12 <i>ET</i> 1000<= année <= 3000 <i>(a1)</i>	jour < 1 <i>OU</i> jour > 31 <i>OU</i> mois < 1 <i>OU</i> mois > 12 <i>OU</i> année > 3000 <i>OU</i> année < 1000 <i>(b1)</i>

Construction des classes d'équivalence (1/3)

- Décomposition de *b1* en plus petites CE (invalides)

CE Valides	CE Invalides
1<= jour <=31 ET 1<= mois <= 12 ET 1000 <= année <= 3000 (a1)	jour < 1 (b1)
	jour > 31 (b2)
	mois < 1 (b3)
	mois > 12 (b4)
	année > 3000 (b5)
	année < 1000 (b6)

Construction des classes d'équivalence (2/2)

- Prise en compte des contraintes liant les données d'entrées (nombre jour par mois et année bissextile) : peut être vu comme une décomposition de *a1* en plus petites CE

CE Valides	CE Invalides
mois ∈ {2,4,6,9,11} => 1<=jour <=30 (<i>a2</i>)	mois ∈ {2,4,6,9,11} ET (jour<1 OU jour>30) (<i>b7</i>)
(mois = 2) et (année est non bissextile) => (jours <= 28) (<i>a3</i>)	(mois = 2) et (année est non bissextile) ET (jours> 28) (<i>b8</i>)
(mois = 2) et (année est bissextile) => (jours <= 29) (<i>a4</i>)	(mois = 2) et (année est bissextile) ET (jours> 29) (<i>b9</i>)

Couverture des CE valides

- (jour = 15, mois = 2, année = 2000), (jour = 29, mois = 2, année = 2004)

Couverture des CE invalides

- b1: (jour = -10, mois = 2, année = 2000)
- b2: (jour = 40, mois = 2, année = 2000)
- b3: (jour = 20, mois = -2, année = 2000)
- b4: (jour = 15, mois = 15, année = 2000)
- ...

CE Valides	CE Invalides
1<= jour <=31 ET 1<= mois <= 12 ET 1000<= année <= 3000 (a1)	jour < 1 (b1)
	jour > 31 (b2)
	mois < 1 (b3)
	mois > 12 (b4)
	année > 3000 (b5)
	année < 1000 (b6)
mois ∈ {2,4,6,9,11} => 1<=jour <=30 (a2)	mois ∈ {2,4,6,9,11} ET (jour<1 OU jour>30) (b7)
(mois = 2) et (année est non bissextile) => (jours <= 29) (a3)	(mois = 2) et (année est non bissextile) ET (jours> 28) (b8)
(mois = 2) et (année est bissextile) => (jours <= 29) (a4)	(mois = 2) et (année est bissextile) ET (jours> 28) (b9)

Exemple à compléter : voir étude de cas

Approche systématique qui donne une bonne couverture

Mais

- La spécification ne définit pas toujours les résultats attendus pour les cas de tests *invalides*
- *La prise en compte de toutes les CE peut amener à générer un très grand nombre de cas de test*
- *La méthode peut être complexe à mettre en œuvre*

Trouver les bons cas de tests :
tester avec des tables de décisions

Tester avec des tables de décisions (1/2)

- Le comportement du logiciel dépend de conditions d'entrée : chaque action dépend (normalement) de conditions d'entrée
- Une table de décisions permet de représenter de façon concise les actions effectuées en fonction de combinaisons sur les conditions d'entrée

Conditions / Variables

Combinaison de valeurs (variants)

C1		V1					
C2		V2					
Action2		x					

Actions

Actions sélectionnées

Tester avec des tables de décisions (2/2)

Conditions / Variables

Combinaison de valeurs (variants)

C1		V1					
C2		V2					
Action2		X					

Actions

Actions sélectionnées

- On génère un cas de test par colonne (**variant**) en utilisant des valeurs conforme aux conditions / valeurs définies par le variants

Exemple de table de décisions (1/2)

- Supposons qu'il y ait 3 conditions / variables C1, C2, C3 telles que :
 - C1 peut prendre les valeurs 0,1,2
 - C2 peut prendre les valeurs 0,1
 - C3 peut prendre les valeurs 0,1,2,3

- Et 4 actions A1, A2, A3, A4 possibles dépendant des combinaisons des valeurs des 3 conditions.

- Le nombre de ligne de la table sera 7 : 3 conditions + 4 actions

- Sans autre informations, le nombre de colonnes sera 3x2x4 (3 valeurs pour C1, 2 valeurs pour C2 et 4 valeurs pour C4)

Exemple de table de décisions (2/2)

■ La table de décision ressemblera à

C	CR	E	D																								
C1	1	3	3	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
C2	3	2	6	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
C3	6	4	24	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
A1				X	X					X	X	X					X	X				X			X		
A2						X							X		X					X			X			X	
A3								X	X					X				X									
A4							X						X						X			X			X		

C.R : Coefficient de répétition
E : Étendue
D : Dimension

On observe A4 quand $C1=0$ et $C2=C3=1$

Les actions (A1, A2, A3, et A4) correspondant aux valeurs des conditions C1, C2 et C3 sont obtenues à l'aide des spécifications

Simplifier une table de décisions par la méthode « don't care » (1/4)

- La table de décisions peut être simplifiée en remarquant qu'une condition n'influence pas le résultat pour certains variants : on les regroupe et on donne la valeur « Dont Care » à cette condition au niveau de ce nouveau variant

Dont' Care (C1) Un seul variant

C	CR	E	D																								
C1	1	3	3	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
C2	3	2	6	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
C3	6	4	24	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
A1				X	X					X	X	X					X	X				X			X		
A2						X								X		X					X			X			X
A3								X	X						X				X								
A4							X						X						X			X			X		

Simplifier une table de décisions par la méthode « don't care » (2/4)

- La table précédente simplifiée par agrégation des don't care

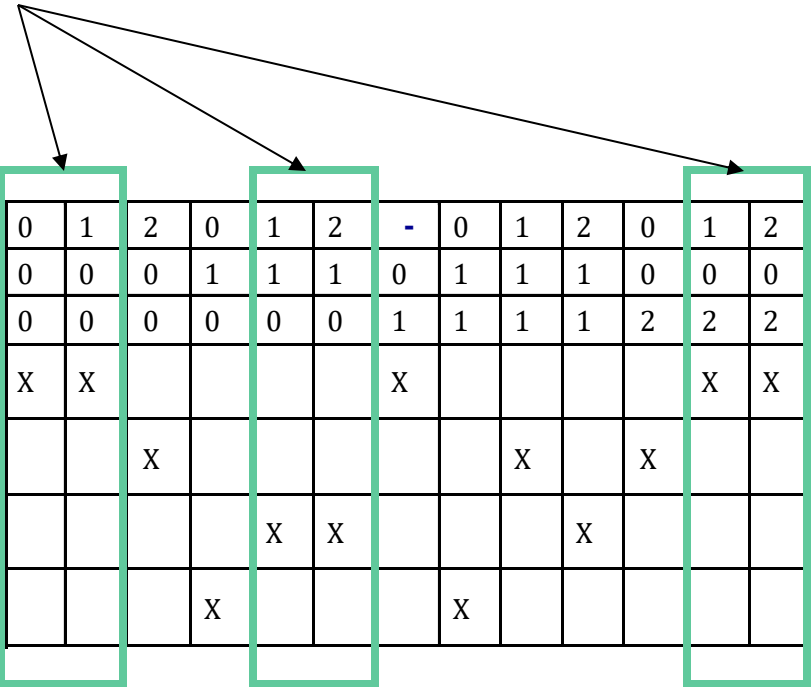
Dont' Care (C1) Un seul variant

C	CR	E	D																						
C1	1	3	3	0	1	2	0	1	2	-	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
C2	3	2	6	0	0	0	1	1	1	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
C3	6	4	24	0	0	0	0	0	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
A1				X	X					X					X	X				X			X		
A2						X						X		X					X			X			X
A3								X	X				X				X								
A4							X				X							X			X			X	

Simplifier une table de décisions par la méthode « don't care » (3/4)

- D' autres simplifications sont possibles (don't care partiels)

Dont' Care (partiel)



C	CR	E	D																						
C1	1	3	3	0	1	2	0	1	2	-	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
C2	3	2	6	0	0	0	1	1	1	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
C3	6	4	24	0	0	0	0	0	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
A1				X	X					X					X	X				X			X		
A2						X					X		X					X			X			X	
A3								X	X				X				X								
A4							X				X							X			X			X	

Simplifier une table de décisions par la méthode « don't care » (4/4)

- D' autres simplifications sont possibles (don't care partiels)

Dont' Care (partiel)

Diagram illustrating the relationship between variables C, CR, E, D and their corresponding values in a table. Three arrows point from the text "Dont" Care (partiel) to the cells C1D, C2D, and C3D, which are highlighted with green boxes.

C	CR	E	D																			
C1	1	3	3	0/1	2	0	1/2	-	0	1	2	0	1/2	0	1	2	0	1	2	0	1	2
C2	3	2	6	0	0	1	1	0	1	1	1	0	0	1	1	1	0	0	0	1	1	1
C3	6	4	24	0	0	0	0	1	1	1	1	2	2	2	2	2	3	3	3	3	3	3
A1				X				X					X				X			X		
A2					X					X		X				X			X			X
A3							X				X			X								
A4						X			X						X			X			X	

Comment construire une table de décisions

- Une table de décision peut être construite de différentes manières
- Méthode par énumération complète des combinaisons des décisions
 1. Identifier les variables et les conditions de décisions
 2. Identifier les actions résultantes
 3. **Enumérer toutes les combinaisons possibles des décisions**
 4. **Simplifier par la méthode des don't care si nécessaire (et possible)**
- Méthode par abstraction initiale des combinaisons des décisions
 1. Identifier les variables et les conditions de décisions
 2. Identifier les actions résultantes
 3. **Identifier quelle action doit être produite en réponse à une combinaison de décision particulière en agrégeant les cas similaires (*partitionnement*)**
 4. **Vérifier la consistance et la complétude**

Exemple concret (1/3)

- Un programme lit trois valeurs entières inférieures à 20 et affiche à l'écran un message indiquant si le triangle est « scalène », « isocèle » ou « équilatéral »

- Spécification plus formelle : données en entrée
 - trois entiers (côtés du triangle: a , b , c)
 - chaque côté doit être un nombre positif inférieur ou égal à 20.

- Spécification plus formelle : résultat fourni (type du triangle) :
 - Équilatéral si $a = b = c$
 - Isocèles si 2 paires de côté sont égaux
 - Scalène si aucun côté n'est égal à l'autre ou
 - Pas Un Triangle si $a \geq b + c$, $b \geq a + c$, ou $c \geq a + b$

Exemple concret (2/3)

- Variables de Décision:

- côtés a, b, c

- Conditions (booléennes)

- C1: $a > 20$

- C2 : $b > 20$

- C3 : $c > 20$

- C4 : $a < b+c$

- C5 : $b < a+c$

- C6 : $c < a+b$

- C7 : $a = b$

- C8 : $a = c$

- C9 : $b = c$

9 conditions booléennes $\rightarrow 2^9=512$ combinaisons

\rightarrow Table de 16 lignes et 512 colonnes !!!!

\rightarrow Il faut abstraire a priori

- Actions : déterminer que les valeurs a, b, c définissent

- A1 : Une violation de contrainte

- A2 : Pas un triangle

- A3 : Un triangle scalène

- A4 : Un triangle isocèle

- A5 : Un triangle équilatéral

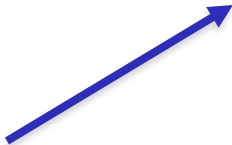
- A6 : *Un cas impossible*

Exemple concret (3/3)

Abstraction



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C1 : $a > 20$	Y	-	-	N	N	N	N	N	N	N	N	N	N	N
C2 : $b > 20$	-	Y	-	N	N	N	N	N	N	N	N	N	N	N
C3 : $c > 20$	-	-	Y	N	N	N	N	N	N	N	N	N	N	N
C4 : $a < \underline{b+c}$	-	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
C5 : $b < \underline{a+c}$	-	-	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
C6 : $b < \underline{a+c}$	-	-	-	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y
C7 : $a = b$	-	-	-	-	-	-	Y	N	Y	N	Y	N	Y	N
C8 : $a = c$	-	-	-	-	-	-	Y	Y	N	N	Y	Y	N	N
C9 : $b = c$	-	-	-	-	-	-	Y	Y	Y	Y	N	N	N	N
A1 : Violation de contrainte	X	X	X											
A2 : Pas un triangle				X	X	X								
A3 : Un triangle scalène													X	X
A4 : Un triangle isocèle										X		X		
A5 : Un triangle équilatéral							X							
A6 : <i>Un cas impossible</i>								X	X		X			



Enumération

Heuristique *Chaque condition* / *Toutes conditions*

- Pour chaque condition produire
 - un variant où la condition est mise à **vraie une fois** avec **toutes les autres** conditions à **fausses**
 - Un variant avec **toutes** les conditions à **faux**
 - Un variant avec **toutes** les conditions à **vrai**
- Pour l'exemple précédent supprime 3 variants (11 au lieu de 14)

Attention à ne pas supprimer des variants pertinents

Exemple de suppression de variants (1/2)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C1 : $a > 20$	Y	-	-	N	N	N	N	N	N	N	N	N	N	N
C2 : $b > 20$	-	Y	-	N	N	N	N	N	N	N	N	N	N	N
C3 : $c > 20$	-	-	Y	N	N	N	N	N	N	N	N	N	N	N
C4 : $a < b+c$	-	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
C5 : $b < a+c$	-	-	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
C6 : $b < a+c$	-	-	-	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y
C7 : $a = b$	-	-	-	-	-	-	Y	N	Y	N	Y	N	Y	N
C8 : $a = c$	-	-	-	-	-	-	Y	Y	N	N	Y	Y	N	N
C9 : $b = c$	-	-	-	-	-	-	Y	Y	Y	Y	N	N	N	N
A1 : Violation de contrainte	X	X	X											
A2 : Pas un triangle				X	X	X								
A3 : Un triangle scalène													X	X
A4 : Un triangle isocèle										X		X		
A5 : Un triangle équilatéral							X							
A6 : <i>Un cas impossible</i>								X	X		X			

Exemple de suppression de variants (1/2)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C1 : $a > 20$	Y	-	-	N	N	N	N	N	N	N	N	N	N	N
C2 : $b > 20$	-	Y	-	N	N	N	N	N	N	N	N	N	N	N
C3 : $c > 20$	-	-	Y	N	N	N	N	N	N	N	N	N	N	N
C4 : $a < b+c$	-	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
C5 : $b < a+c$	-	-	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
C6 : $b < a+c$	-	-	-	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y
C7 : $a = b$	-	-	-	-	-	-	Y	N	Y	N	Y	N	Y	N
C8 : $a = c$	-	-	-	-	-	-	Y	Y	N	N	Y	Y	N	N
C9 : $b = c$	-	-	-	-	-	-	Y	Y	Y	Y	N	N	N	N
A1 : Violation de contrainte	X	X	X											
A2 : Pas un triangle				X	X	X								
A3 : Un triangle scalène													X	X
A4 : Un triangle isocèle										X		X		
A5 : Un triangle équilatéral							X							
A6 : Un cas impossible								X	X		X			

Attention à ne pas supprimer des variants pertinents

Tester avec des tables de décisions : conclusion

- Construction systématique
- Exhaustivité des cas
- Permet de maîtriser efficacement la combinatoire
- Déduction aisée des cas de tests
- Peut également se faire avec des graphes de décisions
- Peut être simplifié par l'utilisation de BDD (Binary Decision Diagram)

Améliorer la pertinences des cas de tests :
tester aux limites

Tester aux limites (1/2)

- Les erreurs sont souvent aux frontières :
 - Boucle avec une itération de trop ou de moins
 - Indice de tableau trop grand ou trop petit
 - Oubli de cas particulier
 - ...

- Le test aux limites va améliorer le test par partitionnement en se concentrant sur les frontières qui sont potentiellement des zones « à problème »

- Comme pour le test basé sur les partitions d'équivalence le test aux limites peut s'appliquer à différents niveaux / technique de tests :
 - Tests fonctionnel, système (contraintes de performance)
 - Tests structurel (analyse de code)

- Dans le cas du test boîte noire ou fonctionnel les frontières se définissent grâce aux domaines obtenus lors du calcul des classes d'équivalence:
 - Par la spécification des variables d'entrée
 - Par la spécification des résultats
- Certaines données se prêtent naturellement à la définition de limites :
 - Valeurs numériques
 - Valeurs énumérées
- Pour certaines données il faut définir une « mesure » numérique :
 - Tableau, liste -> taille (vide, plein, grand,..)
 - Valeur alphanumérique (proche syntaxiquement « oui » -> « ouii », « non » -> « mon »)

Génération de cas de tests aux limites (1/6)

- Calculer les classes d'équivalence comme précédemment
- Pour chaque classe d'équivalence générer une valeur médiane et une ou plusieurs valeurs aux bornes (en utilisant la mesure associée à la donnée si nécessaire) :

Exemple : Soit une fonction qui attend en entrée un numéro de département (en métropole), la donnée d'entrée doit être comprise entre 1 et 95 :

Validité	Classes d'équivalence	Représentants avec limites	Large couverture
Entrées valides	[1 – 95]	1, 48, 95	1, 2, 48, 94, 95
Entrées invalides	[minInt – 1 [-3000, 0	-3000, -1, 0
Entrées invalides] 95 – MaxInt]	96, 1000	96, 97, 1000

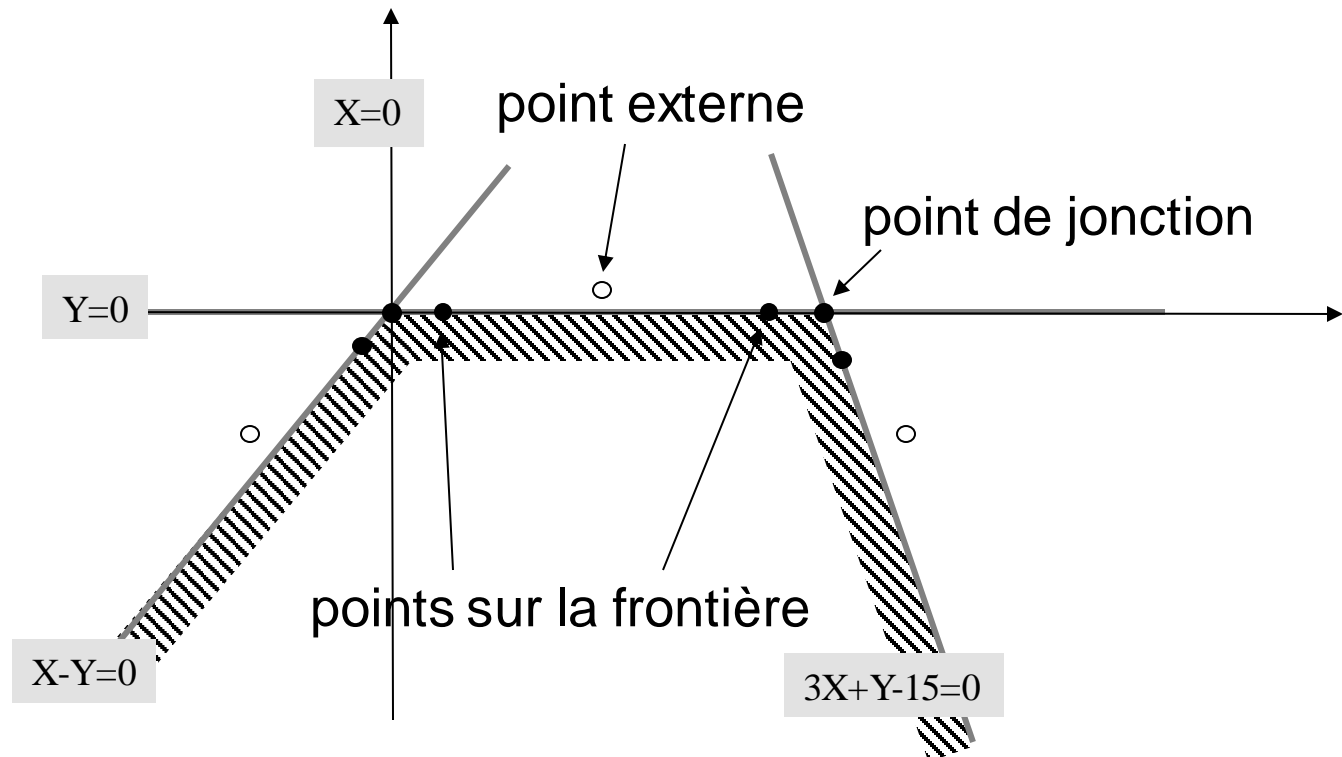
Génération de cas de tests aux limites (2/6)

Autre Exemple : Une fonction attend une valeur « oui » / « non »

Validité	Classes d'équivalence	Représentants avec limites
Entrées valides	{ « oui » }	« oui »
Entrées valides	{ « non » }	« non »
Entrées invalides	Autre chaîne	« hello word » « » « ouii » « mon »

Génération de cas de tests aux limites (3/6)

- Les données en entrées peuvent être liées ?
 - Exemple : X, Y tq $(Y < 0) \text{ AND } (X - Y > 0) \text{ AND } (3X + Y - 15 < 0)$
- Théoriquement, pour N entrées, 2^N valeurs aux limites



Génération de cas de tests aux limites (4/6)

- Que faire lorsque les données en entrées sont liées ?
- Traitement mathématique possible (transformation en polyèdre)
- En pratique, on considère 3 cas pour produire des valeurs aux limites :
 - Si la condition est une conjonction (AND) de M conditions booléennes
 - -> choisir un cas où **toutes** les conditions sont « juste » **vraies**
 - -> choisir M cas avec pour chaque **une seule** des conditions « juste » **fausse**
 - Si la condition est une disjonction (OR) de M conditions booléennes
 - -> choisir un cas où **toutes** les conditions sont « juste » fausses
 - -> choisir M cas avec pour chaque **une seule** des conditions « juste » **vraie**
 - Sinon, se ramener à un des deux cas précédents
- Produire également des valeurs « médianes »

Génération de cas de tests aux limites (5/6) : un exemple

X, Y tels que $(Y < 0) \text{ AND } (X - Y > 0) \text{ AND } (3X + Y - 15 < 0)$

Cas « tout vrai » :

Cas « $(Y < 0)$ faux les autres à vrai » :

Cas « $(X - Y > 0)$ faux les autres à vrai » :

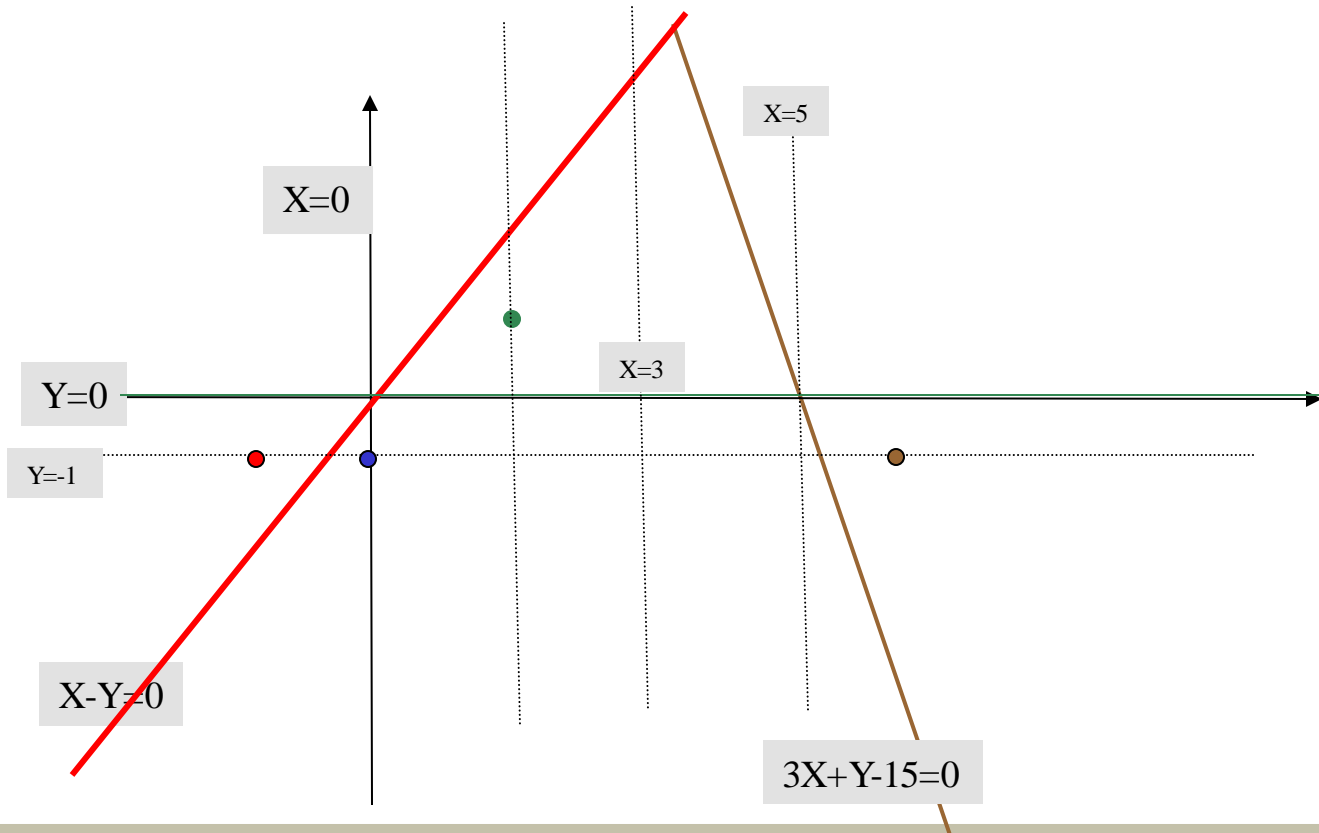
Cas « $(3X + Y - 15 < 0)$ faux les autres à vrai » :

$(X=0, Y=-1)$

$(X=2, Y=1)$

$(X=-2, Y=-1)$

$(X=6, Y=-1)$



Génération de cas de tests aux limites (6/6) : l'exemple du lendemain

■ Limites sur (a3) :

- (mois = 2) et (année est non bissextile) => (1<=jours <= 28)
- peut se réécrire
 - (mois != 2) ou (année est bissextile) ou (1<=jours <= 28)

1. Tout à faux : (mois=2, année = 2007, jours=29)

2. Un vrai, les autres à faux :

- 1. (mois=3, année = 2007, jours=29)
- 2. (mois=2, année=2004, jours=29)
- 3. (mois=2, année=2007, jours=28)

CE Valides	CE Invalides
(a1)	(b1) -(b6)
mois ∈ {2,4,6,9,11} => 1<=jour <=30 (a2)	mois ∉ {2,4,6,9,11} ET (jour<1 OU jour>30) (b7)
(mois = 2) et (année est non bissextile) => (1<= jours <= 28)(a3)	(mois = 2) et (année est non bissextile) ET (jours> 28)(b8)
(mois = 2) et (année est bissextile) => (1<= jours <= 29) (a4)	(mois = 2) et (année est bissextile) ET (jours> 28) (b9)

- Types d'erreurs pour lesquelles le test aux limites est très efficace :
 - Mauvais opérateur de relation : $X > 2$ au lieu de $X \geq 2$
 - Erreur de borne : $X + 2 = y$ au lieu de $X + 3 = y$
 - Échange de paramètres : $2x + 3y > 4$ au lieu de $3x + 2y > 4$
 - Ajout d'un prédicat qui ferme un ensemble : $(2x > 4)$ **et** $(3x < 6)$
 - Frontière manquante : $(x + y > 0)$ **ou** $(x + y \leq 0)$
 - Boucles mal réglées, mauvaise gestion des indices de tableau

Test aux limites : conclusion

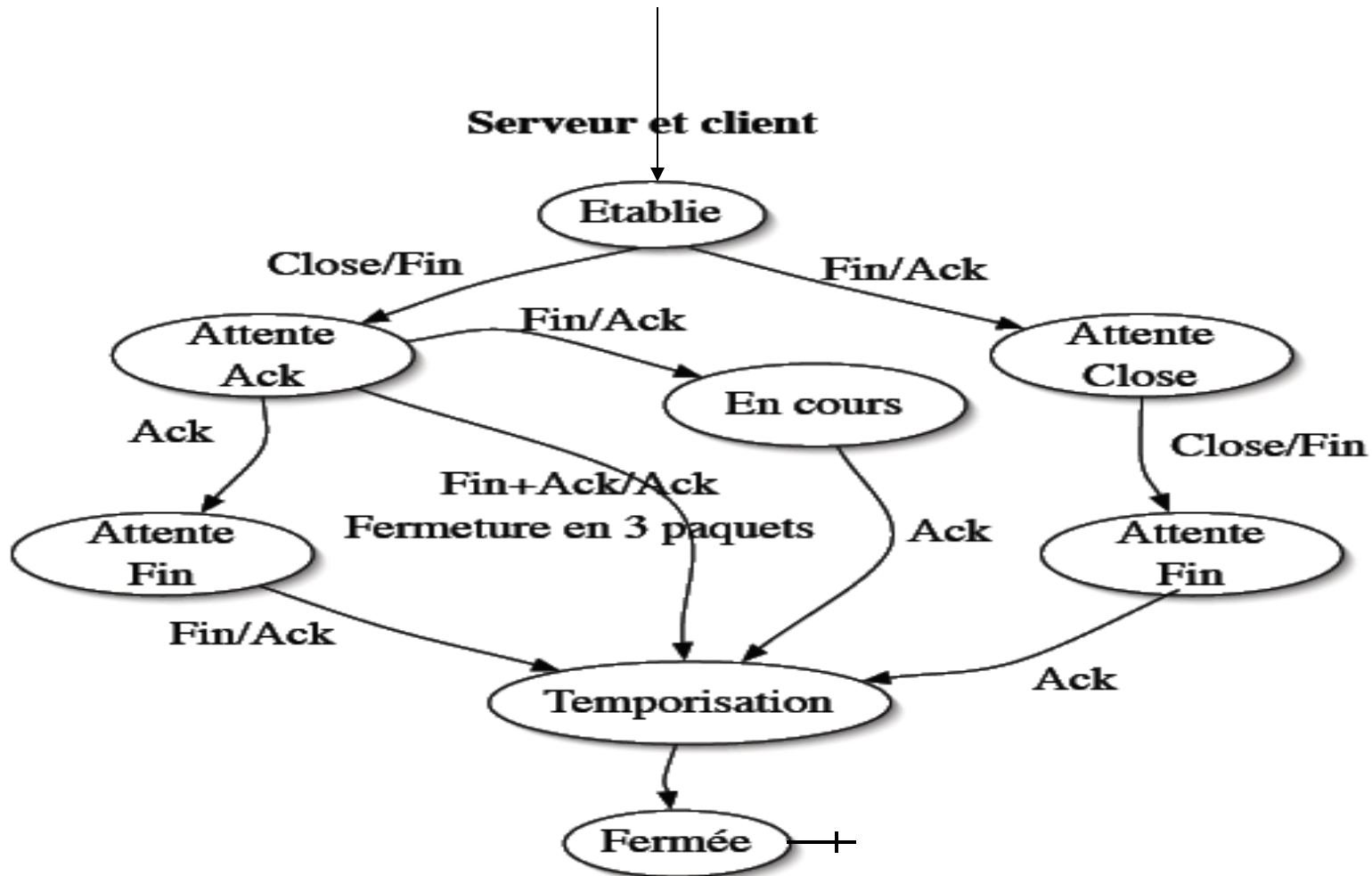
- Le test aux limites améliore le test par partitionnement mais ne le remplace pas
- Il est parfois complexe de construire les frontières
- La complexité de la construction des frontières peut être réduite par différents procédés d'approximation
- Une automatisation poussée peut être atteinte aussi bien dans le cadre de tests structurels que de tests fonctionnels
- ***Méthode utilisée implicitement dans de nombreuses méthodes formelles***

Trouver les bons cas de tests :
construire des diagrammes états/transitions

Tester avec des diagrammes états/transitions

- Une application présente différentes réponses en fonction de conditions actuelles et passées qui définissent son **état**
- Le passage d'un état à un autre se fait en fonction de conditions programmées (algorithmes) et d'événements externes ; on nomme ces changements d'états **transitions**
- La représentation sous forme de diagramme états / transitions (automates) permet d'appréhender de façon synthétique ces relations états / transitions
- Des cas de tests peuvent alors être produit pour
 - Couvrir toutes les transitions
 - Couvrir tous les états
 - Tester des séquences particulières ou invalides
- Il existe différentes sémantiques et représentation graphique

Le diagramme états/transitions fermeture connexion TCP



Construire un diagramme états/transitions

- Définir un état initial
- Recenser les événements qui peuvent se produire
- Définir les transitions d'états à partir des événements
- Définir les états de sortie (états finaux)

Exemple : vie d'un processus système (1/2)

■ États

- E0 : non existant (état à la fois initial et final)
- E1 : existant mais non actif
- E2 : en attente de CPU
- E3 : actif
- E4 : en attente d'un événement
- E99 : cas d'exception

■ Événements

- V1: création
- V2: destruction
- V3: suspension
- V4: demande de ressource
- V5: arrivée de l'événement attendu
- V6: processeur disponible
- V7: perte de CPU (slicing, priorité, ...)
- V8: activation

Exemple : vie d'un processus système (2/2)

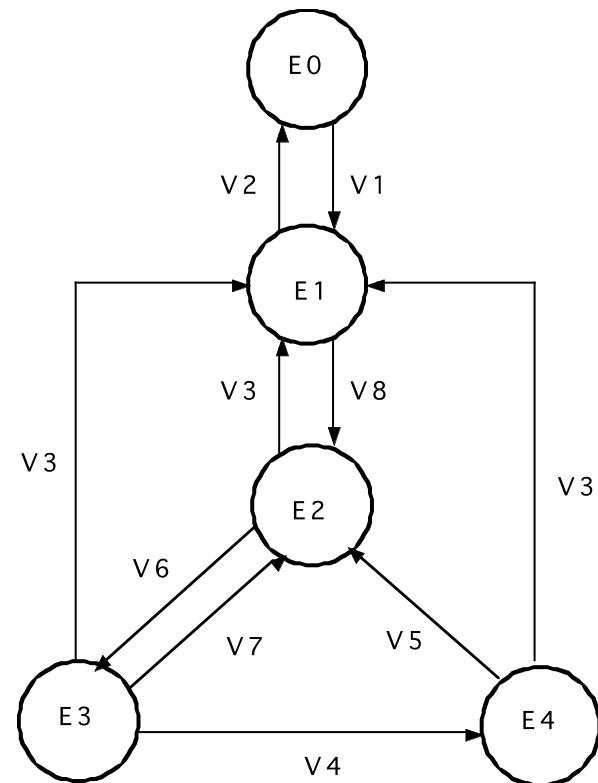
■ États

- E0 : non existant (état à la fois initial et final)
- E1 : existant mais non actif
- E2 : en attente de CPU
- E3 : actif
- E4 : en attente d'un événement
- E99 : cas d'exception

■ Événements

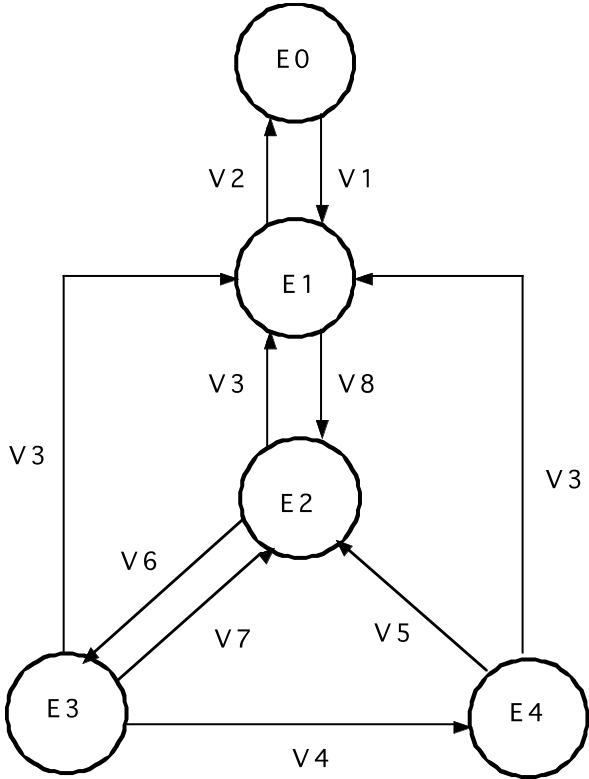
- V1: création
- V2: destruction
- V3: suspension
- V4: demande de ressource
- V5: arrivée de l'événement attendu
- V6: processeur disponible
- V7: perte de CPU (slicing, priorité, ...)
- V8: activation

Relations États / Transitions



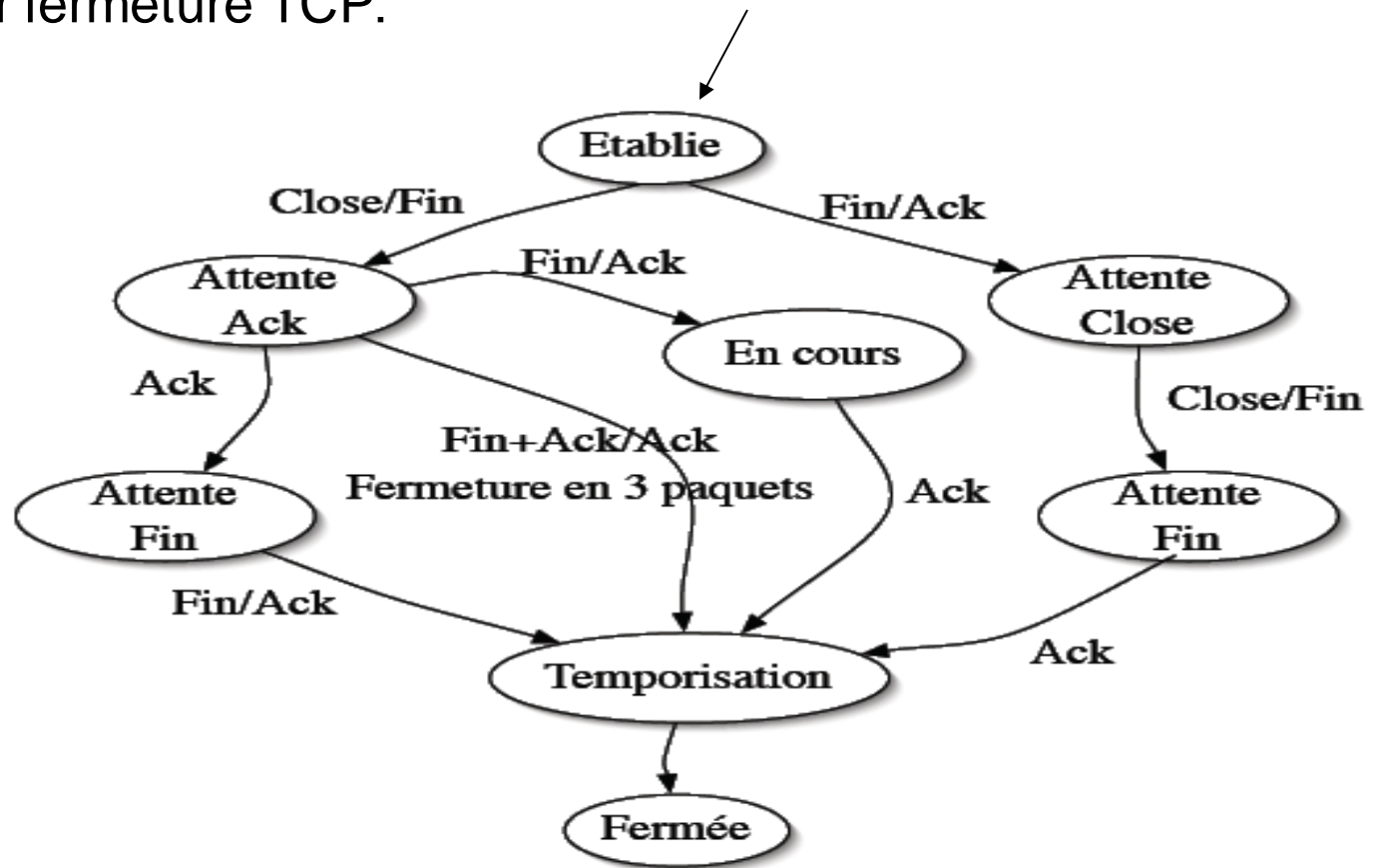
Le diagramme peut être mis sous forme de table

<div><div></div><div>V</div><div>E</div></div>	V1	V2	V3	V4	V5	V6	V7	V8
E0	E1	E99	E99	E99	E99	E99	E99	E99
E1	E99	E0	E99	E99	E99	E99	E99	E2
E2	E99	E99	E1	E99	E99	E3	E99	E99
E3	E99	E99	E1	E4	E99	E99	E2	E99
E4	E99	E99	E1	E99	E2	E99	E99	E99



Générer des cas de tests

- Couvrir tous les états, toutes les transitions
- Tester des séquences particulières, par exemple invalide
- Exemple sur fermeture TCP:



Tester avec les diagrammes états/transitions : conclusion

- Permet d'obtenir une description complète du comportement
- Permet de prendre en compte des cas « bizarres » auxquels on ne pense pas naturellement
- Utile pour tester des applications sensibles ou à forte interaction (par exemple IHM)
- S'automatise très bien
- Souffre du problème de l'explosion combinatoire

Tests boîte noire : conclusion

- Technique de test très efficace
- Se concentre sur les fonctionnalités du logiciel
- Met en avant des oublis de réalisation
- Utilise les spécifications pour définir les cas de tests
- La combinatoire peut être réduite par différentes techniques
- Mise en œuvre tôt, cette technique permet de préciser les spécifications (et éviter des erreurs de réalisation)
- Ne permet pas de détecter des erreurs de réalisation portant sur des cas d'utilisation très particuliers (ni nominaux ni aux limites)