Test et Validation du Logiciel
Test Fonctionnel

Sami Taktak

sami.taktak@cnam.fr

Centre d'Étude et De Recherche en Informatique et Communications
Conservatoire National des Arts et Métiers

le cnam

Test Fonctionnel

Objectif :

- Générer des cas de tests en utilisant des spécifications (non le code)

Données pouvant être utilisées :

- Type des paramètres d'une méthode
- Pré-condition 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
- Graphe causes – effets / tables de décision
- Diagramme états / transitions

Réduire la Combinatoire

Assez fréquemment :

- Petit nombre de paramètres
- Paramètres définis sur de petits ensembles dénombrables
- À priori possible d'énumérer tous les cas possibles : nombre de combinaisons égales au produit des cardinaux des ensembles

Mais énumérer tous les cas possibles a un coût et la réalisation d'un test exhaustif peut prendre beaucoup de temps

All Singles, All Pairs

$\Rightarrow$ essayer de **réduire la combinatoire** en gardant une bonne qualité de couverture

2 approches :

- jeux de tests *all singles* : jeux de tests utilisant au moins une fois chaque valeur de chaque paramètre
- jeux de tests *all pairs* : jeux de tests utilisant au moins une fois chaque valeur de chaque pairs (dit aussi *pairwise*)

All Singles

- Jeux de tests couvrant au moins chaque possibilité pour chaque paramètre
- Nombre de jeux de tests : cardinal du plus grand ensemble de valeurs
- Les valeurs des autres ensembles sont énumérées dans les jeux de tests construits

- Permet de réduire grandement le nombre de jeux de tests
- Garantie une certaine qualité : toute valeur d'un paramètre est testée au moins une fois
- Oubli de réalisation ou grosse erreur seront détectés

Mais ne détecte pas efficacement les erreurs liées à une combinaison de paramètres

Exemple *All Singles*

On veut tester un logiciel qui génère des scripts pour valider le comportement de serveurs web.

Ce logiciel prend 4 paramètres en entrée :

- Le type de système d'exploitation (OS) du client : Ubuntu, OpenSuse, Windows, Mac OS X, Chrome OS
- Le type de navigateur web : Firefox, Opera, Chrome
- Le type de données téléchargées : jpeg, avi, ogg, mp3
- Le fait d'utiliser une connexion sécurisée ou non

Test exhaustif : $5 \times 3 \times 4 \times 2 = 120$ tests différents

Exemple *All Singles*

Jeux de tests *All Singles* : 5 tests seulement (cardinal du plus grand ensemble : 5 OS)

	OS	**Navigateur**	**Données**	**Sécurisé**	
(	Ubuntu,	Firefox,	jpeg,	oui	)
(	OpenSuse,	Opera,	avi,	non	)
(	Windows,	Chrome,	ogg,	oui	)
(	Mac OS X,	Firefox,	mp3,	non	)
(	Chrome OS,	Chrome,	jpeg,	oui	)

Chaque valeur de chaque paramètre est testée au moins une fois

All Pairs

Objectif :

- Tester les erreurs dues à une combinaison de 2 paramètres

Nécessite :

- Construire des jeux de tests avec toutes les paires possibles

Complexité :

- produit du cardinal des 2 plus grands ensembles de valeurs

Méthodologie

- Ordonner en ordre décroissant les ensembles de valeurs en fonction de leur cardinal
- Prendre les 2 ensembles de valeurs E_1 et E_2 ayant les 2 plus grands cardinaux notés V_1 et V_2
- Construire toutes les paires à valeurs dans E_1 et E_2 ($V_1 \times V_2$ paires)
- Pour les autres ensembles de valeurs, alterner les différentes valeurs afin de construire des paires avec les autres ensembles de valeurs
- Toutes les paires possibles sont ainsi couvertes par un jeux de $V_1 \times V_2$ tests

$\Rightarrow$ Rajouter un paramètre avec un ensemble de valeurs plus petit que le cardinal des 2 plus grands ensembles de valeurs ne rajoute pas de jeux de test

Exemple *All Pairs*

OS	**Navigateur**	**Données**	**Sécurisé**
Ubuntu,	Firefox,	jpeg,	oui
Ubuntu,	Opera,	avi,	non
Ubuntu,	Chrome,	ogg,	oui
Ubuntu,	Firefox,	mp3,	non
Chrome OS,	Opera,	jpeg,	non
Chrome OS,	Chrome,	avi,	oui
Chrome OS,	Firefox,	ogg,	non
Chrome OS,	Opera,	mp3,	oui
⋮ ,	⋮ ,	⋮ ,	⋮

All Singles, All Pairs

- Très efficace
- Simple à mettre en œuvre
- Peut servir conjointement à d'autres méthodes

Mais :

- Limité aux petits ensembles de valeurs
- Souvent les ensembles de valeurs ont 2^n valeurs (n : nombre de bits pour coder la variable)

Tests Aléatoires

- Génération de tests de manière probabiliste
- Fonction de distribution souvent uniforme sur les ensembles de valeurs des paramètres d'entrée
- Fiabilité du programme mesurée en fonction des résultats incorrects
- Mesure dépendant de la fonction de distribution aléatoire utilisée pour la génération des tests

Tests Aléatoires

Avantages :

- Jeux de tests objectifs
  - Jeu de tests des méthodes déterministes fourni par les testeurs
    $\Rightarrow$ influencé par la perception du système par le testeur
- Simplification de la génération des tests : génération automatique
- Génération automatique complète si analyse automatique des résultats possibles

$\Rightarrow$ impossible pour la plus par des application réelles

Tests Aléatoires

Méthodes d'analyse automatiques des résultats :

- Test dos à dos :
  - Deux variantes (ou plus) d'un système sont exécutées avec les mêmes entrées et les sorties sont comparées
  - Solution très coûteuse
  - Utilisé seulement pour les applications critiques
- Inversion de la fonction testée
  - Mais la plupart des fonctions ne sont pas inversibles (fonctions non injectives)

Tests Aléatoires

Principal inconvénient :

- Impossible de garantir le test de tous les comportements du système

Exemple :

```
If (a = 0) and (x = 4145)
then x = x / a ...
```

Très peu probable qu'un jeu de tests aléatoire effectue la division par zéro

Tests Aléatoires

- Très simple à mettre en œuvre surtout si l'analyse des résultats est automatique
- Très efficace lors des premières phases de tests
- Un grand nombre de défauts peuvent être détectés avec un minimum d'interventions
- Peut être complémentaire à d'autres méthodes de tests

Mais :

- Très coûteux d'augmenter le taux de couverture à partir d'une certaine limite

Analyse Partitionnelle

Impossible en général d'énumérer tous les cas possibles :

Produit cartésien des domaines de définition des entrées du programme

Explosion combinatoire ! !

Exemple : l'addition de 2 entiers de 32 bits génère 2^{64} vecteurs de tests !

Solution possible : génération de tests aléatoires

Mais se pose le problème de couverture :

Tous les comportements sont-ils couverts ?

Principe

Autre solution : Optimiser les jeux de tests

Partitionner le produit cartésien en classes d'équivalence :

- Analyse de la spécification du logiciel
- Identification des groupes de valeurs pour lesquelles le logiciel se comporte identiquement

$\Rightarrow$ Identification des *classes d'équivalence*

1 classe d'équivalence

$\Updownarrow$

1 comportement du programme

Exemple de Partitionnement

Exemple :

Fonction prenant en entrée un numéro de département (en métropole)

La donnée doit être comprise entre 1 et 95

Validité des Entrées	**Classe d'équivalence**	**Valeurs de Test**
Valides	$[1, 95]$	33
Invalides	$[minInt, 1[$	-40
Invalides	$]95, maxInt]$	10000

Classes d'Équivalence

Definition (Classe d'équivalence)

Dans le cadre des tests fonctionnels, une *classe d'équivalence* est un ensemble de valeurs pour lesquelles on ne peut destinguer le comportement du logiciel

Pour toute valeur choisie dans une même classe d'équivalence, le logiciel aura toujours le même comportement

Ce comportement peut être, soit correct, soit incorrect

$\Rightarrow$ **une seule valeur** à tester par classe d'équivalence

Stratégie de Test

- Identifier les classes d'équivalence valides et invalides d'après les spécifications
- Données en entrées indépendantes : partitionnement des domaines de valeurs en classes d'équivalence
- Données en entrées liées entre elles : certaines valeurs pourront appartenir à plusieurs classes d'équivalences
  $\Rightarrow$ partitionnement non exact

Génération de tests pour chacune des classes d'équivalence :

- Retour au problème précédant
- Génération de tests avec la stratégie *all pairs*
- Génération de tests aléatoires

Stratégie de Test

Étape à suivre pour la construction de classe d'équivalence :

- Extraire de la spécification :
  - Le domaine des valeurs d'entrées
  - L'ensemble des fonctions du système
- Utiliser ces données pour :
  - Déterminer les entrées des différentes fonctions
  - Découper le domaine des entrées en classes d'équivalence
- Pour chaque classe d'équivalence :
  - Sélectionner un élément de la classe
  - Déterminer les valeurs de sortie pour l'élément sélectionné

Stratégie de Test

- Détermination des valeurs de sortie pas toujours possible :
  - spécification très complexe
  - certains éléments nécessaires au calcul des sorties non accessibles

  $\Rightarrow$ tests de propriétés sur les sorties
- Suivant la forme de la spécification, la détermination des classes d'équivalence est plus ou moins facile

Construction des Classes d'Équivalence

Règles de Construction des classes d'équivalence :

1. Condition d'entrée ou sortie définissant un intervalle de valeurs
   - 1 classe d'équivalence valide pour les valeurs dans l'intervalle)
   - 1 classe d'équivalence invalide pour les valeurs inférieurs
   - 1 classe d'équivalence invalide pour les valeurs supérieurs

Construction des Classes d'Équivalence

Règles de Construction des classes d'équivalence :

1. Condition d'entrée ou sortie définissant un intervalle de valeurs
2. Condition d'entrée ou de sortie définissant un tuple de N valeurs
   - 1 classe d'équivalence valide représentant les tuples de N valeurs
   - 2 classes d'équivalences invalides : une représentant les tuples de moins de N valeurs, une représentant les tuples de plus de N valeurs

Construction des Classes d'Équivalence

Règles de Construction des classes d'équivalence :

1. Condition d'entrée ou sortie définissant un intervalle de valeurs
2. Condition d'entrée ou de sortie définissant un tuple de N valeurs
3. Condition d'entrée ou de sortie définissant un ensemble de valeurs
   - 1 classe d'équivalence valide représentant les valeurs dans l'ensemble prédéfini ou bien une classe d'équivalence par valeur si le programme semble les différencier
   - 1 classe d'équivalence invalide représentant les valeurs hors ensemble

Construction des Classes d'Équivalence

Règles de Construction des classes d'équivalence :

1. Condition d'entrée ou sortie définissant un intervalle de valeurs
2. Condition d'entrée ou de sortie définissant un tuple de N valeurs
3. Condition d'entrée ou de sortie définissant un ensemble de valeurs
4. Condition d'entrée ou de sortie définissant une contrainte devant être vérifiée
   - 1 classe d'équivalence valide (la contrainte est vérifiée)
   - 1 classe d'équivalence invalide (la contrainte n'est pas vérifiée)

Construction des Classes d'Équivalence

Règles de Construction des classes d'équivalence :

1. Condition d'entrée ou sortie définissant un intervalle de valeurs
2. Condition d'entrée ou de sortie définissant un tuple de N valeurs
3. Condition d'entrée ou de sortie définissant un ensemble de valeurs
4. Condition d'entrée ou de sortie définissant une contrainte devant être vérifiée
5. Classe d'équivalence trop complexe ou les éléments d'une classe d'équivalence semble être traités différemment
   - Décomposer cette classe d'équivalence en plusieurs classes d'équivalence moins complexes

Exemple de Recherche de Classes d'Équivalence

- 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
  - $1 \leq$ `mois` ≤ 12
  - $1 \leq$ `jour` ≤ 31
  - $1000 \leq$ `an` ≤ 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

Exemple de Recherche de Classes d'Équivalence

Prise en compte des contraintes d'intervalle

CE Valides	CE Invalides
$1 \leq \texttt{mois} \leq 12 \wedge$ $1 \leq \texttt{jour} \leq 31 \wedge$ $1000 \leq \texttt{an} \leq 3000$	$\texttt{jour} < 1$
	$\texttt{jour} > 31$
	$\texttt{mois} < 1$
	$\texttt{mois} > 12$
	$\texttt{an} > 3000$
	$\texttt{an} < 1000$

Exemple de Recherche de Classes d'Équivalence

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

CE Valides	**CE Invalides**
$\texttt{mois} \in \{2,4,6,9,11\} \Rightarrow 1 \leq \texttt{jour} \leq 30$	$\texttt{mois} \in \{2,4,6,9,11\} \wedge (\texttt{jour} \leq 1 \vee \texttt{jour} > 30)$
$\texttt{mois} = 2 \wedge$ (année non bissextile) $\Rightarrow (\texttt{jour} \leq 28)$	$\texttt{mois} = 2 \wedge$ (année non bissextile) $\wedge (\texttt{jour} > 28)$
$\texttt{mois} = 2 \wedge$ (année bissextile) $\Rightarrow (\texttt{jour} \leq 29)$	$\texttt{mois} = 2 \wedge$ (année bissextile) $\wedge (\texttt{jour} > 29)$

Exemple : Génération de jeux de tests

- Couverture des CE valides
  - `(jour = 15, mois = 2, année = 2000)`
  - `(jour = 29, mois = 2, année = 2004)`
- Couverture des CE invalides
  - (jour = -10, mois = 2, année = 2000)
  - (jour = 40, mois = 2, année = 2000)
  - (jour = 20, mois = -2, année = 2000)
  - (jour = 15, mois = 15, année = 2000)
  - À compléter...

Conclusion

- Approche systématique qui donne une bonne couverture
- Stratégie permettant de rendre praticable les méthodes de test de logiciels réels
- Nécessite un effort d'abstraction et d'analyse des spécifications
- Permet de détecter des incohérences au niveau des spécifications

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

Test aux Limites

Test aux limites $\Rightarrow$ Vérifier le comportement du logiciel aux frontières de ses domaines de fonctionnement

- La plupart des erreurs sont dues à une mauvaise gestion des valeurs aux bornes des domaines des variables
- S'appuie sur le partitionnement des domaines des valeurs en entrée
- Utilisé en complément du test par partitionnement

Test aux Limites

Dans le cas du test fonctionnel les frontières se définissent grâce aux domaines obtenus lors du calcul des classes d'équivalence :

- Bornes des intervalles pour les valeurs numériques
- Valeurs des types énumérés
- Collections d'éléments :
  - Tableau, liste : taille (vide, plein,. . .)
- Valeurs alphanumériques : choix de valeurs proches syntaxiquement

Test aux Limites

Utilisation de la technique du test aux limites :

1. Trouver les classes d'équivalence
2. Pour chaque classe d'équivalence :
   - Générer une valeur médiane
   - Générer une ou plusieurs valeurs aux bornes

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 des Entrées	**Classe d'équivalence**	**Valeurs de Test**
Valides	$[1, 95]$	1, 50, 95
Invalides	$[minInt, 1[$	-10000, 0
Invalides	$]95, maxInt]$	96, 10000

Test aux Limites

Exemple avec valeurs non numériques :

Fonction prenant en paramètre la chaîne de caractère `"oui"` ou la chaîne de caractère `"non"`

Validité	**Classe d'équivalence**	**Valeurs de Test**
Valide	{`"oui"`}	`"oui"`
Valide	{`"non"`}	`"non"`
Invalide	Autres chaînes	"abcd", "", `"ouii"`,`"nom"`

Apport du Test aux Limites

- 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 $(x + y <= 0)$
  - Boucles mal réglées, mauvaise gestion des indices de tableau

Test aux Limites

- 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

Tester avec des Tables de Décisions

- Le comportement du logiciel dépend de conditions d'entrée : chaque action dépend (normalement) de conditions d'entrée
- Table de décisions $\Rightarrow$ représenter de façon concise les actions effectuées en fonction des conditions d'entrée
- 1 Cas de test par colonne (variant) conforme aux conditions / valeurs définies par le variants

C_1			V_{C_1}					
C_i			V_{C_i}					
Action_n			$\times$					

Exemple de Construction d'une Table de Décision

- Soient 3 conditions / variables C_1, C_2, C_3 telles que :
  - C_1 peut prendre les valeurs 0,1,2
  - C_2 peut prendre les valeurs 0,1
  - C_3 peut prendre les valeurs 0,1,2,3
- Soient 4 actions A_1, A_2, A_3, A_4 à effectuer en fonction des combinaisons des valeurs des 3 conditions
- Le nombre de ligne de la table sera 7 : 3 conditions + 4 actions
- Le nombre de colonnes sera $3 \times 2 \times 4$ (3 valeurs pour C_1, 2 valeurs pour C_2 et 4 valeurs pour C_4)

Exemple de Construction d'une Table de Décision

Exemple de Table de décision :

C_1	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
C_2	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
C_3	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
A_1	×	×					×	×	×					×	×				×			×		
A_2			×								×		×					×			×			
A_3					×	×						×				×								×
A_4				×						×							×			×			×	

Construction des Tables de Décisions

- Une table de décision peut être construite de différentes manières
- Voici une façon de procéder :
  - Identifier les variables et les conditions de décisions
  - Identifier les actions résultantes
  - Identifier quelle action doit être produite en réponse à une combinaison de décision particulière
  - Vérifier la consistance et la complétude
- $3 + 4$ peuvent être remplacés par « Énumérer toutes les combinaisons de décisions (possibles) et associer l'action résultante »

Construction des Tables de Décisions

- Simplification des table de décisions lorsqu'une condition n'influence pas le résultat pour certains variants : on les regroupe et on donne la valeur « Don't Care » à cette condition au niveau de ce nouveau variant

C_1	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
C_2	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
C_3	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
A_1	×	×					×	×	×					×	×				×			×		
A_2			×								×		×					×			×			
A_3					×	×						×				×								×
A_4				×						×							×			×			×	

Construction des Tables de Décisions

- Simplification des table de décisions lorsqu'une condition n'influence pas le résultat pour certains variants : on les regroupe et on donne la valeur « Don't Care » à cette condition au niveau de ce nouveau variant

C_1	0	1	2	0	1	2	×	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
C_2	0	0	0	1	1	1	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
C_3	0	0	0	0	0	0	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3
A_1	×	×					×					×	×				×			×		
A_2			×						×		×					×			×			
A_3					×	×				×				×								×
A_4				×				×							×			×			×	

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)

Diagrammes d'États / Transitions

- Le comportement de certains systèmes dépendent de l'ordre dans lequel les paramètres lui sont passés
- L'histoire de son exécution influe sur son comportement
- Ce sont des *systèmes à états* ou encore des *machines à états*

Comportement différent des systèmes vus jusqu'à présent :

- Le comportement ne dépendait pas des valeurs précédentes des paramètres
- Toute exécution de ces systèmes avec les mêmes valeurs de paramètres produit le même comportement

Diagrammes d'États / Transitions

Pour un système à état :

- le système réagira en fonction de l'historique des valeurs des paramètres qui lui ont était passé
- Son comportement dépendra des valeurs des paramètres mais aussi de son état interne
- Le passage d'un état interne à un autre est appelé *transition*

Mais :

- Dans le cas des tests fonctionnels l'état interne du système n'est souvent pas observable

Diagrammes d'États / Transitions

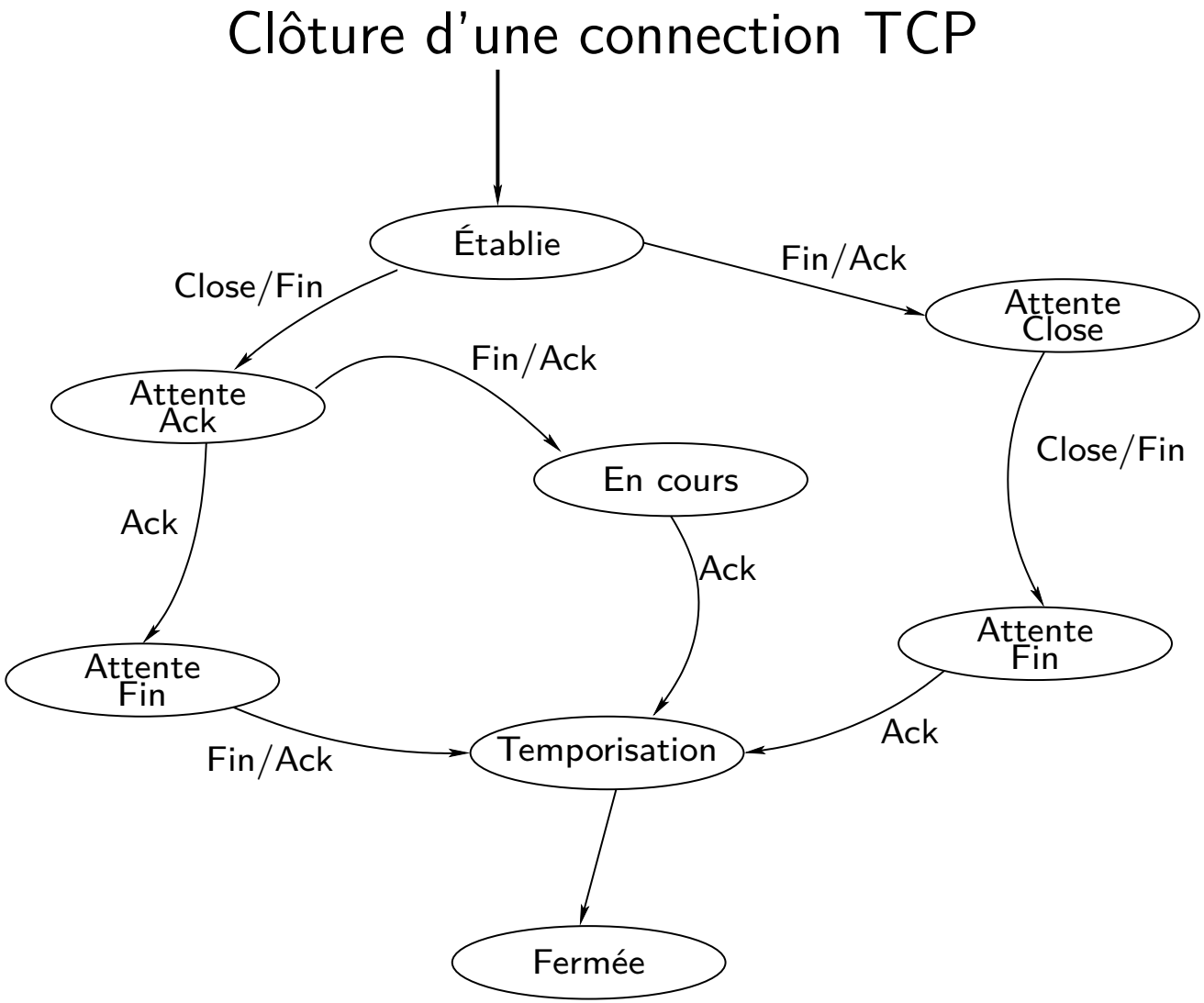
Description de ces systèmes sous forme d'automates ou diagrammes d'états / transitions :

- Représentation graphique
- Graphe dont les noeuds sont les états du systèmes et les arcs, les transitions
- Un arc est étiqueté avec une conditions de franchissement de la transition associée (passage d'un état à un autre)

Exemples :

- Protocole réseau TCP
- Serveur web
- Distributeur de billets

Diagrammes d'États / Transitions



Construction du Diagramme d'États

- Cas simple : le diagramme d'état fait parti de la spécification
- Sinon, construire le diagramme à partir de la spécification

$\Rightarrow$ Revient à réaliser une abstraction des comportements du système

- L'abstraction doit être suffisante pour que le modèle ne soit pas trop complexe
- L'abstraction doit être assez précise pour prendre en compte tous les comportements intéressants pour le test

Construction du Diagramme d'États

Étapes de construction du diagramme d'états :

- Définir les états observables du système et identifier les moyens d'observations de l'état (variable, appelle de fonction, réaction du système, ...)
- Définir l'environnement externe accessible (variable externes)
- Définir les variables internes qui interviennent dans le comportement
- Identifier les différents évènements auxquels le composant va réagir
- Définir pour chaque état et chaque évènement quelle action effectue le système (rien, changement d'état)

Construction des Séquences de Test

- Choix des objectifs en terme de parcours du diagramme d'États :
  - Atteindre au moins une fois chaque état
  - Exercer au moins une fois chaque transition
  - Exercer toutes les séquences d'une longueur donnée
  - Exercer toutes les séquences permettant d'atteindre un état donné
  - Évaluer les réactions vis-à-vis de tous les évènements dans un état donné
- Choix des séquences de tests permettant de réaliser ces objectifs :
  - Trouver les séquences de tests appropriées
  - Trouver les valeurs qui permettront d'exercer la séquence

Test fonctionnel : Conclusion

- Peu formalisé et donc peu automatisé
- Utilisation des tables de décisions pour des cas spécifiques
- Tests aléatoires efficaces pour les premières phases de test
- Le test partitionnel est la méthode la plus répandue
- Utilisation conjointe des tests nominaux, des tests aux limites et des tests de robustesse