Test fonctionnel

Méthodes de test fonctionnel

- Le test fonctionnel vise à examiner le comportement fonctionnel du logiciel et sa conformité avec la spécification du logiciel

  ⇒ Sélection des Données de Tests (DT)

- Méthodes de test fonctionnel
  - Analyse partitionnelle des domaines des données d'entrée et test aux limites -> test déterministe
  - Test combinatoire – Algorithmes Pairwise
  - Test aléatoire
  - Génération automatique de tests à partir d'une spécification

Analyse Partitionnelle

1ère idée :

- On effectue le produit cartésien des domaines des entrées du programme.
- On utilise ce résultat comme ensemble de toutes les DTs et on détermine les sorties attendues pour chacune de ces DTs.

Défaut ⇨ explosion combinatoire

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

Analyse Partitionnelle

2ème idée :

- *On partitionne le produit cartésien en classes d'équivalence.*
- *Chaque classe d'équivalence correspond aux valeurs qui font l'objet d'un même traitement dans la spécification.*

1 classe d'équivalence ⇨ 1 comportement du système

Exemples de comportements dans une spécification d'un ascenseur :

- *changer d'étage (monter ou descendre)*
- *rester au même étage*
- *éventuellement aller au rez-de-chaussée ou au dernier étage*

Partitionnement en Classes d'équivalences

Partition du domaine de données en classes d'équivalences selon la spécification

Idéalement les CEs devraient être

- telles que chaque donnée est dans une classe **disjointes**
- les éléments d'une même classe mis en correspondance avec leurs résultats de façon similaire

En pratique il est difficile de déterminer les CEs

- usage d'heuristiques

Exercice : encore des triangles

- Spécification : Le programme prend en entrée trois réels, interprétés comme étant les longueurs des côtés d'un triangle. Si ces longueurs forment un triangle, le programme retourne la propriété du triangle correspondant (scalène, isocèle ou équilatéral) ainsi que la propriété de son plus grand angle (aigu, droit ou obtus).

- Donner les classes d'équivalence sur les entrées de ce programme, ainsi qu'un cas de test pour chacune des classes.

Solution

- Classes d'équivalence et Données de Test

	Aigu	Obtus	Droit
Scalène	6,5,3	5,6,10	3,4,5
Isocèle	6,1,6	7,4,4	√2,2,√2
Equilatéral	4,4,4	impossible	impossible

+ non triangle - 1,2,8

Règles de partitionnement des domaines

- Si l'entrée e appartient à un intervalle [a,b] :
  - n classes valides découpant [a,b] (dépendant de l'algorithme)
  - une classe non valide pour les valeurs inférieures à a
  - une classe non valide pour les valeurs supérieures à b
  - Si l'entrée e est un ensemble de valeurs :
- n classes valides découpant l'ensemble des ensembles possibles (algo)
  - une classe pour e = ∅
  - une classe non valide pour les ensembles trop grands
- Si l'entrée e doit vérifier une contrainte/propriété P :
  - une classe pour e vérifiant P
  - une classe pour e ne vérifiant pas P

Heuristiques pour Identification de Classes d'équivalences

Pour chaque donnée:

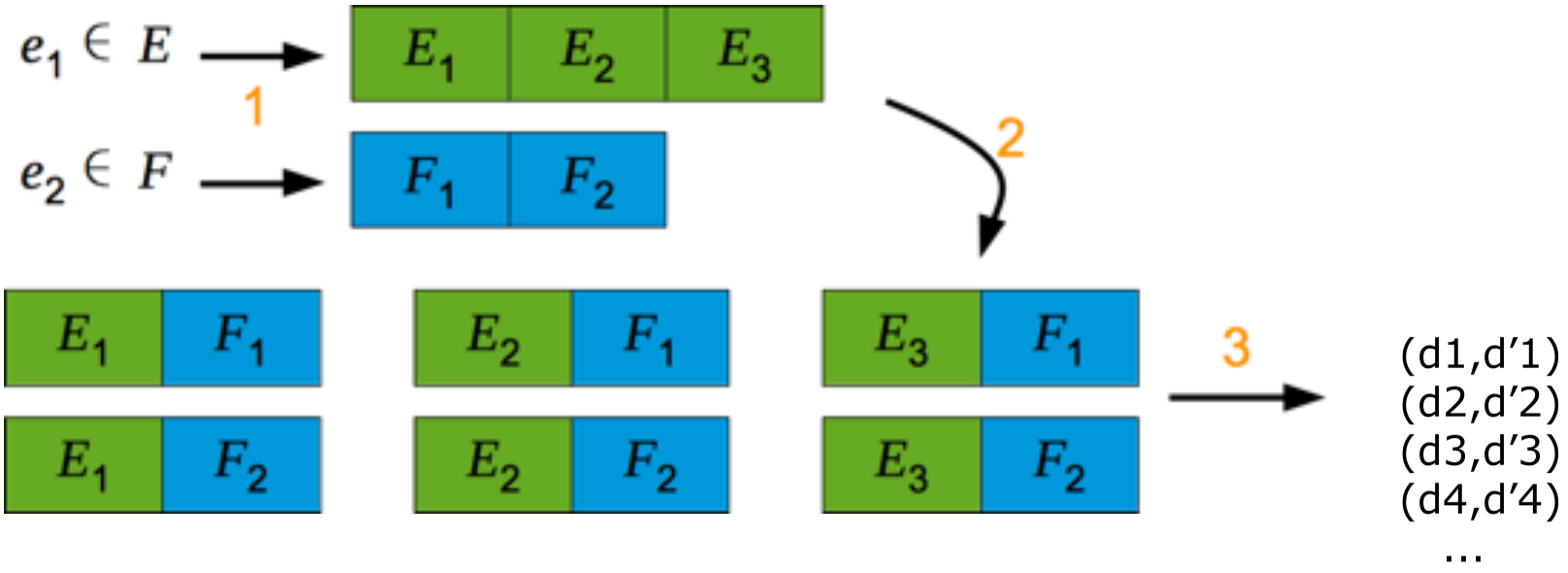
- si il y a une raison de croire que le programme traite les données valides différemment, définir
  - 1 CE valide par donnée valide

- si spécifie une situation *doit être*, définir
  - 1 CE valide (satisfaisant le *doit être*)
  - 1 CE invalide (ne satisfaisant pas le *doit être*)

- si il y a une raison de croire que les éléments d'une CE ne sont pas traités de façon identiques par le programme
  - subdiviser la CE en plus petites CEs.

Méthode générale

1. Pour chaque paramètre d'entrée, calculer les classes d'équivalence sur les domaines de valeurs de ce paramètre
2. S'il y a plusieurs paramètres, faire le produit cartésien des classes obtenues
3. Choisir des données de test



Partitionnement en Classes d'équivalences - Problèmes

- Spécification ne définie pas toujours les résultats attendus pour les cas de tests *invalid*es
- Langages fortement typés éliminent le besoin de la considération de certaines données invalides
- L´approche force-brute de définition de cas de test pour chaque combinaison de Ces
  - donne une bonne couverture, mais
  - non pratique lorsque le nombre d'entrées et classes associé est grand

Les valeurs hors champ semblent injustifiées puisque le programme n'a pas été spécifié pour ces valeurs. Mais la pratique montre que ces valeurs révèlent souvent des erreurs de programmation.

On peut également classer ces DTs hors limites dans les tests de robustesse

Analyse des Valeurs aux Bornes (AVB) ou Test aux limites

Erreurs ont tendance à survenir vers les valeurs extrêmes (*bornes*)

AVB améliore le Partitionnement en Classes d'équivalences en

- sélectionnant les éléments juste et autour des bornes de chacune des CEs
- dérivant des cas de tests en considérant des CEs des résultats également

Conditions de bornes

- Situation à la bordure des limites opérationnelles envisagées

- Types de données ayant des conditions de bornes
  - Numérique, Caractère, Position, Quantité, Vitesse, Location, Taille

- Caractéristiques de conditions de bornes
  - Premier/Dernier, Début/Fin, Minimum/Maximum, Au-dessus/ Au-dessous, Vide/Plein, Plus-lent/Plus-rapide, Plus-grand/Plus-petit, Plus-proche-de/Plus-loin-de, Plus-court/Plus-long, Plus-tôt/Plus-tard, Plus-haut/Plus-bas

Analyse des Valeurs aux Bornes - Directives

Pour chaque condition de borne

- ajouter la valeur de borne correspondante dans au moins un cas de test valide
- ajouter la valeur juste au delà (ou en deçà) de la valeur de borne correspondante dans au moins un cas de test invalide.

Appliquer les mêmes directives pour les résultats

- inclure des cas de tests avec des données, tels que des résultats aux bornes sont produits

Méthode générale

1. Pour chaque paramètre d'entrée, calculer les classes d'équivalence sur les domaines des valeurs de ce paramètre
2. S'il y a plusieurs paramètres, faire le produit cartésien des classes obtenues
3. Choisir des données de test
   - Une valeur pour chaque classe obtenue
   - Des valeurs aux limites

Analyse partitionnelle + test aux limites

- Analyse partitionnelle :
  - ✓ Réduction du nombre de cas de test
  - – Choix des classes délicat

- Test aux limites
  - ✓ Heuristique solide pour le choix des données au sein des classes
  - ✓ Production de tests de conformité et de tests de robustesse
  - – Relation d'ordre sur les entrées nécessaire
  - – Explosion combinatoire des données de test

Inconvénient majeur : Caractère intuitif ou subjectif de la partition en classe et de la notion de limite

Difficulté pour caractériser la couverture des tests

Exercice : analyseur syntaxique

- On veut tester l'analyseur syntaxique d'un compilateur sur l'instruction FOR en BASIC
- Spécification : L'instruction FOR n'accepte qu'un seul paramètre en tant que variable auxiliaire. Son nom ne doit pas dépasser deux caractères alphabétiques non blancs. Après le signe =, sont précisées une borne supérieure. Les bornes sont des entiers positifs et on place entre eux le mot-clé TO.
- Déterminer par analyse partitionnelle des domaines d'entreées un ensemble de tests pour l'intruction FOR.

Solution

- DT obtenues par analyse partitionnelle pour l'instruction FOR :
  - FOR A=1 TO 10 cas nominal
  - FOR A=10 TO 10 égalité des bornes
  - FOR AA=2 TO 7 deux caractères pour la variable
  - FOR A, B=1 TO 8 Erreur - deux variables
  - FOR ABC=1 TO 10 Erreur - trois caractères pour la variable
  - FOR I=10 TO 5 Erreur - Borne sup < Borne inf
  - FOR =1 TO 5 Erreur - variable manquante
  - FOR I=0.5 TO 2 Erreur - Borne inf décimale
  - FOR I=1 TO 10.5 Erreur - Borne sup décimale
  - FOR I=7 10 Erreur - TO manquant

Exercice 2

Considérons la spécification suivante :

- Ecrire un programme statistique analysant un fichier comprenant les noms et les notes des étudiants d'une année universitaire. Ce fichier se compose au maximum de 100 champs. Chaque champ comprend le nom de chaque étudiant (20 caractères), son sexe (1 caractère) et ses notes dans 5 matières (entiers compris entre 0 et 20). Le but du programme est de :
  - calculer la moyenne pour chaque étudiant,
  - calculer la moyenne générale (par sexe et par matière),
  - calculer le nombre d'étudiants qui ont réussi (moyenne supérieure à 10)
- Déterminer par une approche aux limites les cas de test à produire pour cette spécification

Solution

- DT obtenues par test aux limites pour l'exemple Etudiants :
  - passage d'un fichier vide, puis comprenant 1 champ, 99, 100 et 101 (5 tests différents, la valeur -1 n'ayant pas de sens dans ce cas);
  - inclure un nom d'étudiant vide, un nom avec des caractère de contrôle, un nom avec 19, puis 20, puis 21 caractères;
  - inclure un code sexe vide, puis avec un caractère faux (C par exemple);
  - avoir un étudiant sans aucune note et un avec plus de 5 notes;
  - pour certains champs, les notes de doivent pas être des nombres entiers, mais des caractères, des réels avec plusieurs décimales, des nombres négatifs ou des nombres entiers supérieurs à 20.
- Les DT aux limites doivent être passées indépendamment : les erreurs peuvent se compenser.

Test combinatoire : approche Pairwise

- Constatation : Explosion combinatoire des valeurs d'entrées dans le cas de nombreux paramètres prenant des ensembles de valeurs
- Solution : Tester un sous-ensemble des combinaisons de valeurs tel que chaque combinaison de n variables est testée
  - Pairwise : n = 2
- Idée sous-jacente : Majorité des fautes détectées par combinaisons de deux valeurs de variables

Exemple

- On a trois variables x, y et z telles que x peut prendre les valeurs 1 et 2, y peut prendre les valeurs Q et R et z peut prendre les valeurs 5 et 6.
- Couvrir toutes les combinaisons (8) : 8 tests
- Couvrir toutes les paires (12) : 4 tests

- Pour chaque paire de variables, on énumère toutes les combinaisons de valeurs possibles : on obtient toutes les paires de valeurs

x	y
1	Q
1	R
2	Q
2	R

y	z
Q	5
Q	6
R	5
R	6

x	z
1	5
1	6
2	5
2	6

Exemple (suite)

- On construit ensuite un nombre minimun de tests qui permettent de couvrir toutes ces paires. On obtient ainsi 4 tests qui permettent de couvrir toutes les paires.

Tests	x	y	z	
Test 1	1	Q	5	couvre (1,Q), (Q,5) et (1,5)
Test 2	1	R	6	couvre (1,R), (R,6) et (1,6)
Test 3	2	Q	6	couvre (2,Q), (Q,6) et (2,6)
Test 4	2	R	5	couvre (2,R), (R,5) et (2,5)

A faire : un exemple plus réaliste

- On veut tester l'impression depuis plusieurs applications sur des OS et via des réseaux différents

OS	Réseau	Imprimante	Application
Windows7	IP	HP35	Word
Linux	WIFI	Canon900	Excel
Mac OS X	Bluetooth	Canon-EX	Powerpoint

Test Fonctionnel (suite)

- Les spécifications étant rarement formelles, il y a peu d'approches "systématiques":
  - "Tables de décision" ("matrices de test") : une matrice avec les conditions en entrée et les effets possibles, et on coche ce qui va ensemble…
  - "Parcours structurels" de spécifications semi-formelles (diagrammes SADT, scénarios UML, statecharts)
  - Graphes "causes-effet"
- Spécifications fonctionnelles formelles avec OCL

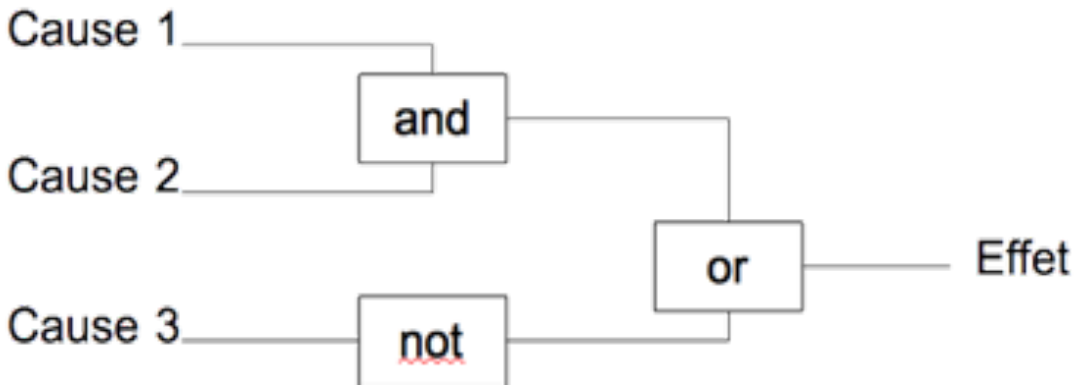
Graphes Causes-Effets

En résumé

- La technique de test basée sur les graphes cause-effet peut se résumer par les étapes suivantes :
- Si l'objet a de nombreuses fonctionnalités, le décomposer en entités plus simples
- Identifier les causes
- Identifier les effets
- Établir le graphe des relations de causes à effets
- Compléter le graphe par les contraintes entre causes ou entre effets
- Convertir le graphe en table de décision
- Produire les DTs associées aux effets à tester en tenant compte des règles de simplification données.

Graphe Cause-Effet

- Représentation graphique des relations logiques entre les entrées et les sorties d'un programme



- Principe: Couvrir les liens causes-effet

➢Un test pour chaque combinaison de causes ayant un effet

Méthode de construction des tests

- Identifier les causes et les effets
- Etablir le graphe des relations entre causes et effets
- Convertir le graphe en table de décision
- Réduire la table de décision en éliminant les causes inutiles à un effet
- Construire un test pour chaque façon de produire un effet

Graphe cause-effet

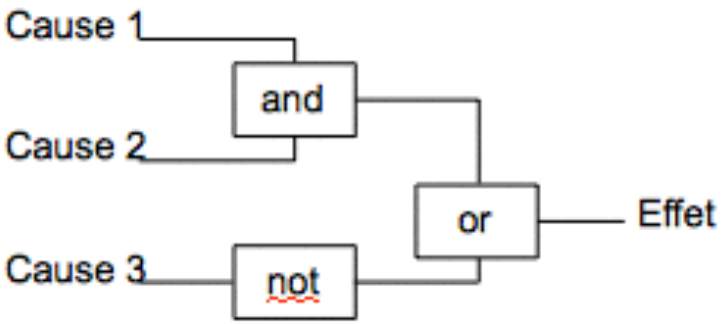


Table de décision

Cause 1	0	0	0	0	1	1	1	**1**
Cause 2	0	0	1	1	0	0	1	**1**
Cause 3	**0**	1	**0**	1	**0**	1	**0**	1
Effet	x		x		x		x	x

Méthode de construction des tests

- Identifier les causes et les effets
- Etablir le graphe des relations entre causes et effets
- Convertir le graphe en table de décision
- Réduire la table de décision en éliminant les causes inutiles à un effet
- Construire un test pour chaque façon de produire un effet

Graphe cause-effet

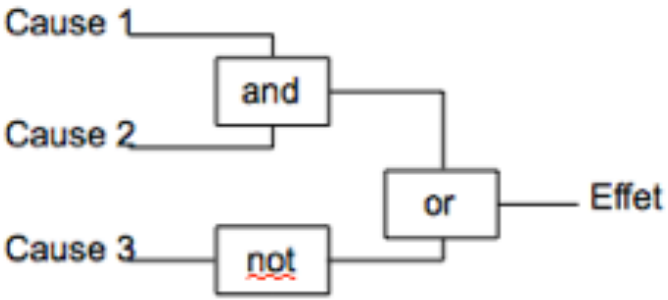
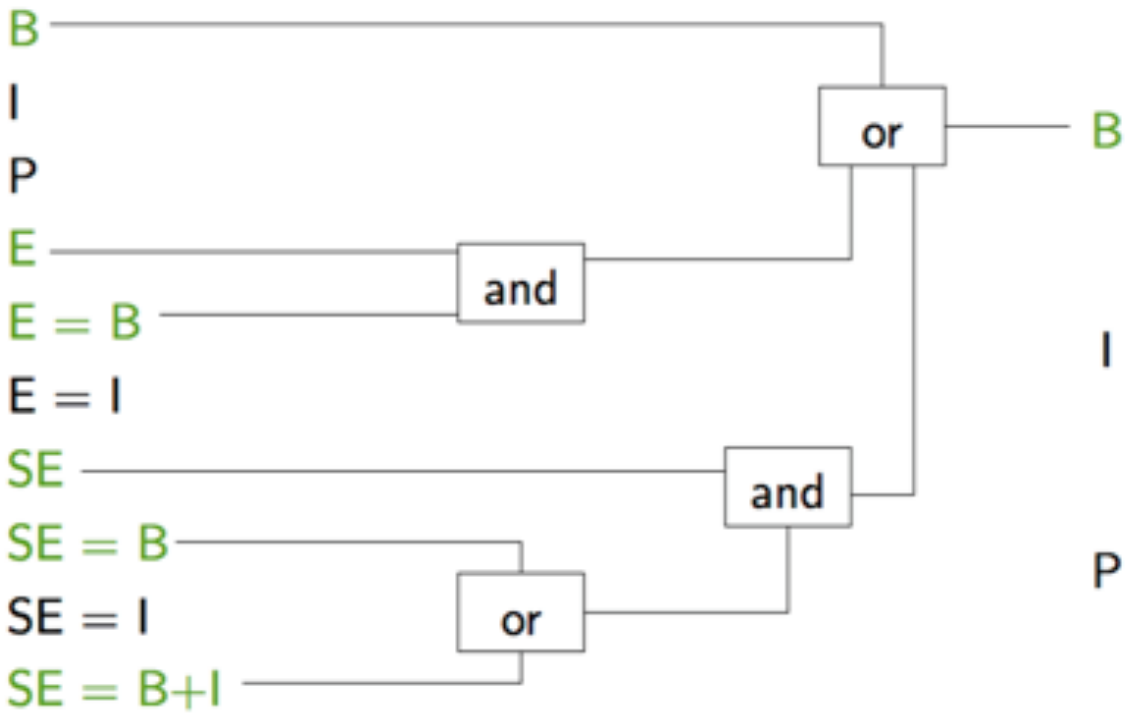


Table de décision simplifiée

Cause 1	x	0	0	1	1
Cause 2	x	0	1	0	1
Cause 3	0	1	1	1	x
Effet	x				x

Test 1 : Cause 3 fausse
Test 2 : Cause 1 et 2 vraies

Graphe cause-effet



4 tests pour l'effet B :

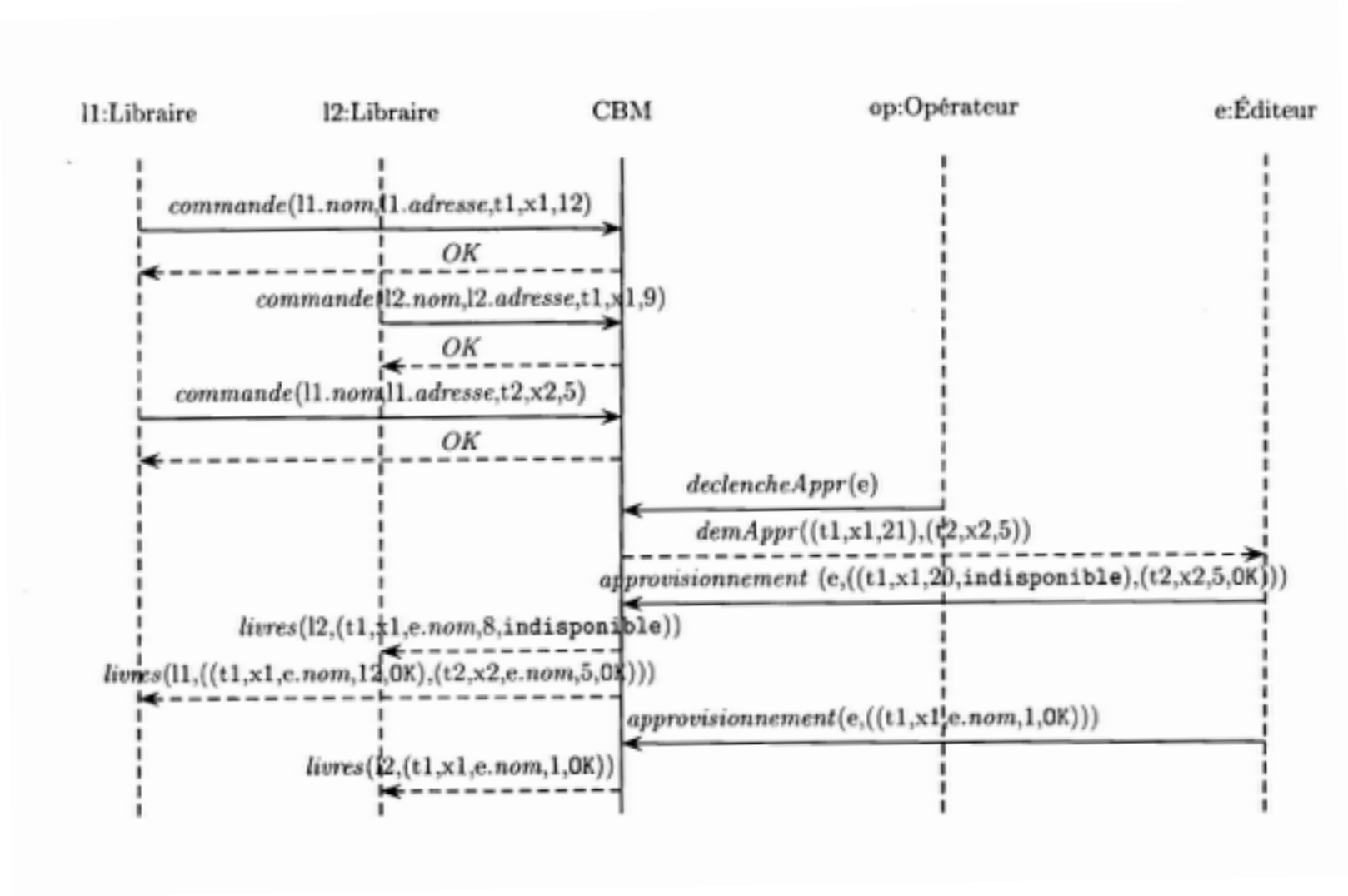
- B
- E et E = B
- SE et SE = B
- SE et SE = B+I

B : bold, I : italic, P : plain, E : emphasized, SE : super emphasized

Graphes Causes-Effets

- Cette méthode de test peut s'avérer une méthode de test fonctionnel très complète et précise.
- De plus, le fait qu'elle utilise un langage de représentation formalisé permet l'automatisation de certaines de ces phases.
- Cependant, les graphes peuvent devenir très complexe quand une fonction fait intervenir un très grand nombre de causes. Dans ce cas, le testeur se voit contraint d'ajouter des nœuds intermédiaires pour maîtriser la complexité, mais le choix approprié de ces nœuds est loin d'être évident.
- Un autre problème dans l'utilisation de cette méthode provient de la difficulté de vérifier l'exactitude du graphe.
- Enfin, ces graphes sont difficiles à mettre à jour lors de la modification des spécifications ou lorsque le testeur réalise que certaines informations lui font défaut.
- En remplaçant des spécifications informelles en graphes cause-effet, le testeur remplace une spécification complexe par une autre. Cet effort supplémentaire est compensé par le fait qu'on effectue en réalité une relecture critique et attentive de la spécification d'origine qui peut s'avérer très efficace.

Test à partir de diagrammes de séquences



Test à partir de diagrammes de séquences

En pratique:

- Instancier les diagrammes de séquence avec des valeurs précises
  (ex: scénarii, diagrammes associés aux cas d'utilisation)

- La séquence doit se terminer par un **effet « observable »** (messages vers les agents extérieurs)

- Il faudrait aussi pouvoir **vérifier l'état du système**
  (=> observateurs à prévoir en plus)

- Construire un ensemble d'instances dans le bon état pour pouvoir jouer ce scénario !

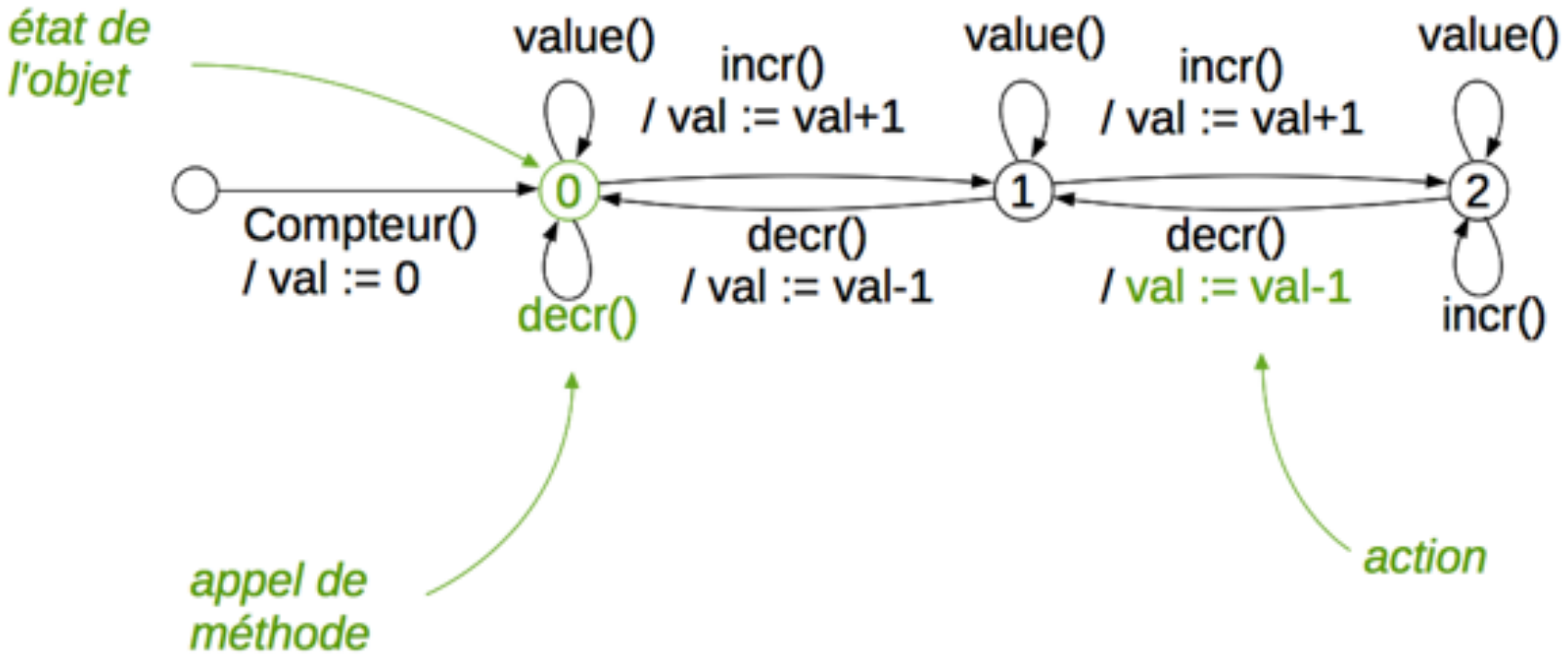
Test à partir de machines à états

- Diagramme états-transitions modélisant le cycle de vie d'un objet :

  - état : moment de la vie de l'objet caractérisé par la valeur des attributs, les actions possibles...
  - transition : changement d'état déclenché par un événement (appel d'une méthode, condition devenue vraie, passage du temps,...)

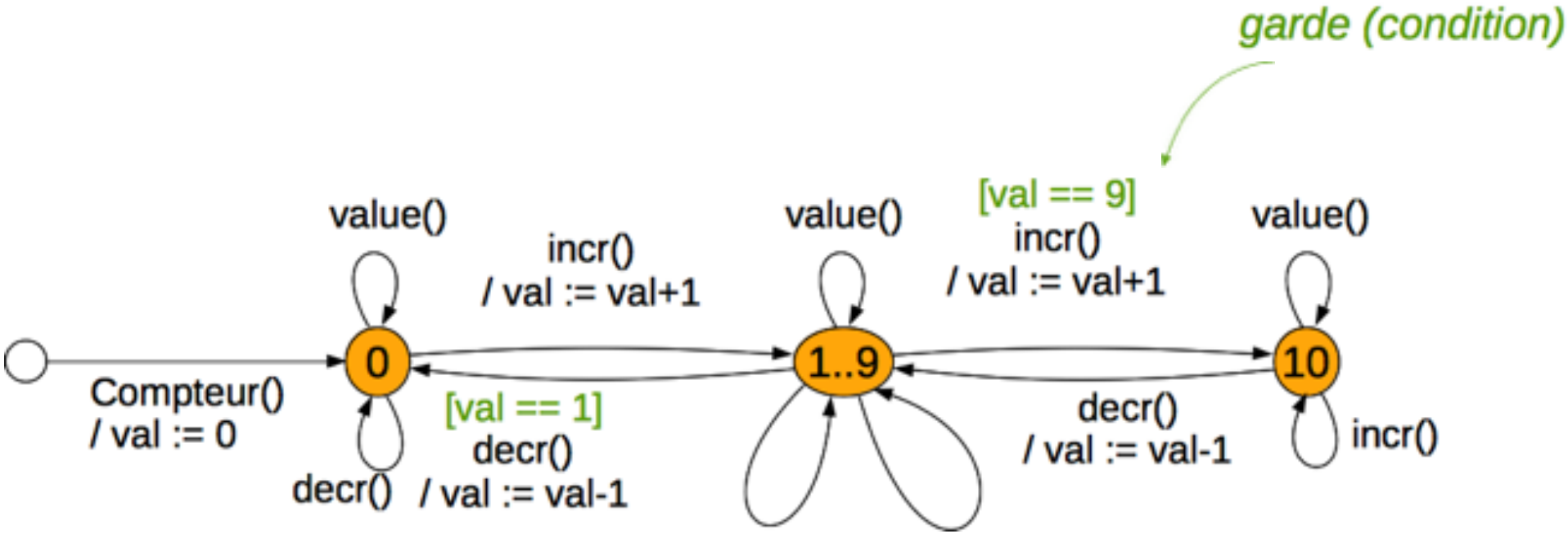
Machines à états

- Compteur à trois valeurs



Machines à états

- Compteur à dix valeurs



états regroupés par comportements similaires

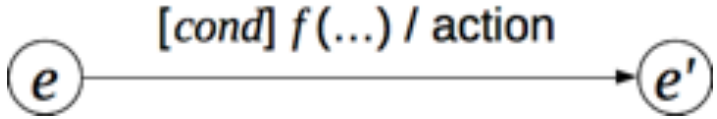
Test à partir d'une machine à états

- couverture des transitions : un test par transition
  - un test pour chaque action possible dans chaque état
  - actions testées de façon indépendante

- couverture des chemins : un test par scénario d'utilisation de l'objet
  - un test pour chaque séquence d'actions depuis l'état initial
  - tests des dépendances entre actions

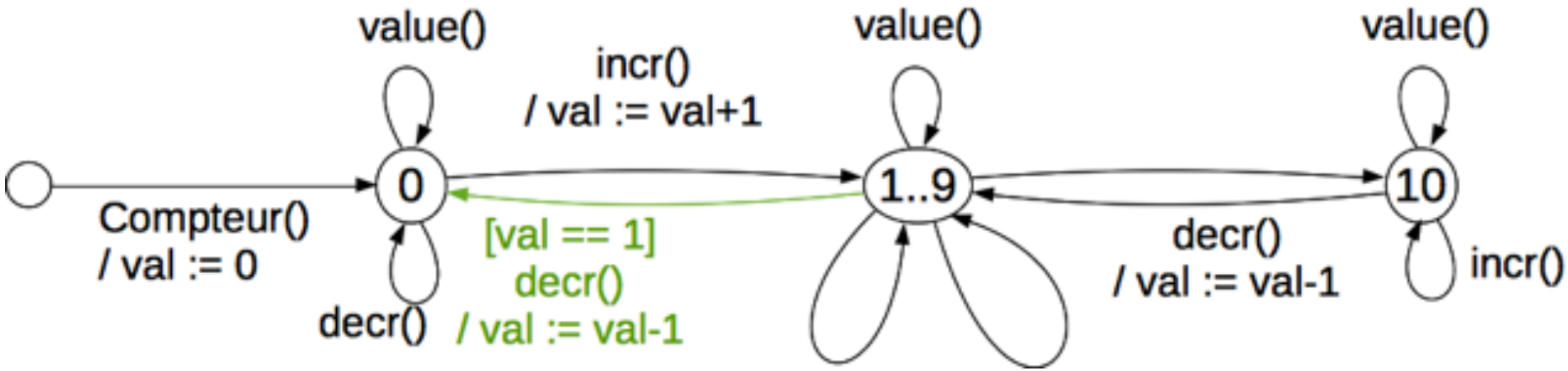
Couverture des transitions

- Test pour une transition



  - construire un objet et l'amener dans l'état e
  - Vérifier que l'objet est dans l'état e
  - Vérifier que la garde cond est satisfaite
  - Activer la transition en appelant la méthode f
  - Verifier que l'action a été exécutée et que l'objet est dans l'état e'
  - Réinitialiser l'objet

Test d’une transition



	actions	observation
préambule	c = new Compteur (); c.incr();	
corps de test	c.value();	1
identification	c.decr();	
postambule	c.value(); c = null;	0

Couverture des transitions

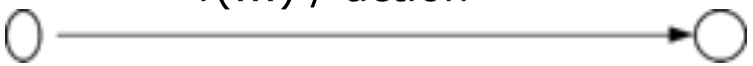
- Méthode : tester chaque transition indépendamment des autres (réinitialiser l'objet entre chaque test)

- Avantages et inconvénients
  - ✓ nombre de transition fini
  - ✓ Génération de tests et oracle automatisable
  - ✓ Fautes ciblées précisément

  - – Redondance des préambules
  - – Couverture incomplète des comportements

Couverture des transitions

- Raffinements
  - conditions multiples : un test pour chaque condition



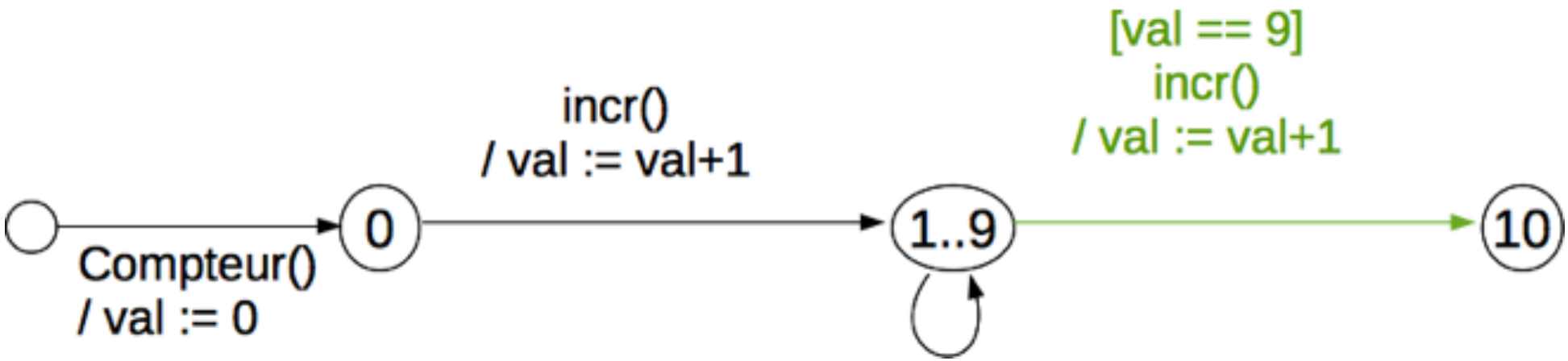
un test pour chaque x < 0 et
un test pour x ≥ 10

  - tests aux limites des conditions

un test avec x = 45 et
un test avec x = 99 (si x entier)

Couverture des chemins

- Problème : couverture des transitions n'implique pas couverture des chemins
  - incrémenter le compteur jusqu'à son maximum : couvert



  - incrémenter le compteur jusqu'à son maximum puis le remettre à zéro : pas testé en couvrant les transitions

Couverture des chemins

- Méthode : tester chaque séquence d'actions depuis l'état initial

- Avantages et inconvénients :
  - ✓ Factorisation des préambules
  - ✓ Détection des fautes liées aux enchaînements d'actions
  - – Nombre de chemins généralement infini

- En pratique :
  - ➢ Seulement les chemins non couverts par les tests des transitions
  - ➢ Seulement les chemins «intéressants» (intuition/expérience)

Test à partir de machines à états

Méthode générale

1. Spécifier :
   - identifier les différents états
   - pour chaque état, identifier les actions possibles dans cet état, leurs conditions et leurs conséquences (changement d'état)
2. construite des tests :
   - au moins un test par transition (plus si nécessaire)
   - des test pour les chemins d'intérêt non couverts

Test fonctionnel à partir de UML-OCL

- Fonctionnel => la spécification sert pour la sélection des tests et comme oracle pour décider du succès des tests
  - N'a-t-on pas oublié des cas = couvre-t-on tous les cas mentionnés dans les documents d'analyse et de conception (cas d'utilisation, diagrammes de séquence, scénario)
- Cas où la spécification est une formule logique :
  - On la met sous **forme normale disjonctive**, en assurant que les cas **disjoints**
    - C1 v C2 v ... v Cn, avec Ci ≠ Cj si i ≠j
  - on prévoit un test par cas ...

Exemple

$max \geq x \wedge max \geq y \wedge (max = x \vee max = y)$

devient :

$(max \geq x \wedge max \geq y \wedge (max = x \wedge max = y))$

$\vee (max \geq x \wedge max \geq y \wedge (max = x \wedge max \neq y))$

$\vee (max \geq x \wedge max \geq y \wedge (max \neq x \wedge max = y))$

soit encore :

$(max = x \wedge max = y) \vee (max = x \wedge max > y) \vee (max > x \wedge max = y)$

dont on peut déduire un jeu de test :

{(10,10,10), (121, -72, 121), (2, 54, 54)}

Modèles d'opérations avec pré/post conditions

Test de conformité : couvrent tous les cas satisfaisant la pré-condition

Test de robustesse : couvrent les cas qui ne satisfont pas la pré-condition (pas toujours utiles, comportement pas toujours défini)

Comme avant : on met la **conjonction** de la pré-condition et de la post-condition sous forme normale disjonctive !

context max(x,y : integer) : integer

pre: x≥ 0 and y ≥ 0

post: max ≥x and max ≥ y and (max = x ∨ max = y)

on se ramène à : $(x \geq 0 \wedge y \geq 0 \wedge max = x \wedge max = y)$

$\vee (x \geq 0 \wedge y \geq 0 \wedge max = x \wedge max > y)$

$\vee (x \geq 0 \wedge y \geq 0 \wedge max > x \wedge max = y)$

Si la clause ***post*** est un ensemble d'implicatins

- si les prémisses des implications sont disjointes : on construit , pour chaque implication, un cas qui satisfait la prémisse et la pré-condition
- sinon on réorganise la clause de manière à avoir des prémisses disjointes...

context max(x,y : integer): integer

pre: x ≥ 0 and y ≥ 0

post: x ≥ y implies max = x

y ≥ x implies max = y

donnera

(x ≥ 0 and y ≥ 0 and x > y) implies max = x

(x ≥ 0 and y ≥ 0 and x = y) implies max = x and max = y

(x ≥ 0 and y ≥ 0 and y > x) implies max = y

Conclusion sur le test fonctionnel

- Le test fonctionnel ou test en boite noire constitue en général la partie la moins formalisée et donc la moins automatisée du processus de test des logiciels.

- La technique la plus utilisée dans l'industrie reste le test partitionnel avec une répartition adaptée entre les tests nominaux, aux limites (en général les plus efficaces pour la détection des défauts) et les tests de robustesse.
- Les autres types de test, comme l'utilisation de graphes cause-effet, s'adressent plutôt à des profils spécifiques.
- Enfin le test aléatoire s'avère efficace, surtout si on peut l'automatiser, pour les premières phases de test.