Test et Validation du Logiciel

CNAM - Unité d'enseignement GLG101

Avril 2006

Patrick FELIX

patrick.felix@labri.fr

MVTsi - LaBRI

Plan

1. Pourquoi de la VVT ?
2. Introduction au test de logiciels
3. Le test dans un projet logiciel
4. Le test structurel
5. Le test fonctionnel
6. TP

Partie 1:
Pourquoi de la VVT ?

Des bogues, des conséquences désastreuses…

- Banque de New York [21 novembre 1985] : pertes financières énormes
- Le Therac-25 [juillet 1985 ->avril 1986] : 3 morts
- Le crash d'AT&T [15 janvier 1990] : pertes financières énormes + la réputation d'AT&T entachée.
- Le Pentium [juin 1994] : pertes financières énormes + psychose
- Ariane 5-01 [4 juin 1996]

Ariane 5-01 (4 juin 1996)

Le 23 juillet, la commission d'enquête remet son rapport : La fusée a eu un comportement nominal jusqu'à la 36ème seconde de vol. Puis les systèmes de référence inertielle (SRI) ont été simultanément déclarés défaillants. Le SRI n'a pas transmis de données correctes parce qu'il était victime d'une erreur d'opérande trop élevée du "biais horizontal" . . .

Les raisons :

1 Un bout de code d'Ariane IV (concernant le positionnement et la vitesse de la fusée) repris dans Ariane V

2 il contenait une conversion d'un flottant sur 64 bits en un entier signé sur 16 bits

3 pour Ariane V, la valeur du flottant dépassait la valeur maximale pouvant être convertie

4) défaillance dans le système de positionnement

5) la fusée a "corrigé" sa trajectoire

6) suite à une trop grande déviation, Ariane V s'est détruite !

Le coût d'un Bogue ?

- Coût du bogue de l'an 2000 ?
  - Quelques chiffres avancés : 300, 1600 ou même 5 000 milliards de dollars
- Quel impact ?
  - Sécurité des personnes,
  - Retour des produits,
  - Relations contractuelles,
  - Notoriété, image,
  - …

Sans méthodes formelles :

- Coût des tests : 50 à 60% du coût total, voire 70% !
- Interprétation(s) des termes usuels (-> utilisation d'UML)
- Ambiguïté des méthodes semi-formelles (# sémantiques UML).
- Maîtrise difficile de certains types de programmations
  [événementielle / parallèle / …]
- Maintenance évolutive difficile

Tendances actuelles ~ Méthodes formelles et certification

- Méthodes formelles :
  - preuve, vérification, test, validation
- Politique de certification
- Certains niveaux de certification exigent des méthodes formelles
- Obligation de certification
  - Grandes entreprises
  - Application à risques
  - Sous-traitance

Tendances actuelles ~ Test

Test populaire car :

- facile à mettre en oeuvre
- rapide

⇒ bon rapport qualité/temps

MAIS :

- Le test peut prouver la présence d'erreurs, pas leur absence, car méthode non exhaustive.
- Les Méthodes de vérification exhaustive souvent considérées comme trop coûteuses (temps/personnel/…).

Le test dans les méthodes formelles

- Objectif ~ Pouvoir raisonner sur les logiciels et les systèmes afin de :
  - Connaître leurs comportements
  - Contrôler leurs comportements
  - **Tester** leurs comportements.
- Moyen ~ Les systèmes sont des objets mathématiques.
- Processus :
  1. Obtenir un modèle formel du logiciel ou du système. [Si la taille le permet, le modèle peut être le logiciel ou le système]
  2. L'analyser par une technique formelle.
  3. **Générer des test** par une technique formelle
  4. **Transposer** les résultats obtenus sur les modèles aux logiciels et systèmes réels.
- Problèmes de l'approche :
  - Le modèle est-il fidèle ?
  - Peut-on tout vérifier ? Décidabilité.
  - Peut-on tout tester ? **Testabilité**.
  - La transposition des résultats est-elle toujours possible ? **Abstraction**.
  - Le test est-il **correct** ? Le test est-il **exhaustif** ?

Partie 2:
Introduction au test de logiciels

Génie Logiciel

Toute fabrication de produit suit les étapes suivantes :

1. Conception
2. Réalisation
3. Test
   - On s'assure que le produit final correspond à ce qui a été demandé selon divers critères
   - Exemples de critères : esthétique, performance, ergonomie, fonctionnalité, robustesse)

La fabrication de logiciel : Constante évolution !

1. nouveaux langages de programmation,
2. nouvelles méthodes de programmation,
3. toute une panoplie de concepts, outils, méthodes, etc.

Exemple : C, ADA, C++, Java, C#, Corba, .NET, etc.

Génie logiciel dont l'objectif essentiel est la maîtrise (conceptualiser, rentabiliser, etc.) de l'activité de fabrication des logiciels.

Assurance qualité

L'**assurance qualité** permet de mettre en œuvre un ensemble de dispositions qui vont être prises tout au long des différentes phases de fabrication d'un logiciel pour accroître les chances d'obtenir un logiciel qui corresponde à ses objectifs (son cahier des charges).

La définition et la mise en place des **activités de test** ne sont qu'un sous-ensemble des activités de l'assurance qualité, et le test aura pour but de minimiser les chances d'apparition d'une anomalie lors de l'utilisation du logiciel.

L'objet de ce qui suit consiste à étudier comment cet objectif peut être atteint.

Erreur, défaut et anomalie

Une **anomalie** (ou **défaillance**) est un comportement observé différent du comportement attendu ou spécifié.

Exemple. Le 4 juin 1996, on a constaté…

Chaîne de causalité :

Erreur (spécification, conception, programmation) => **défaut** => **anomalie**

Le terme **bogue** est malheureusement utilisé pour désigner aussi bien défaut (j'ai commis l'erreur de mettre un nom de variable *a* au lieu de *z*, ie erreur de frappe !) qu'une anomalie (ce programme est plein de bogues)

Exemple : Une anomalie (telle une maladie) trouve toujours son explication dans un défaut (agent pathogène) et un défaut (un microbe latent) ne provoquera pas nécessairement une anomalie.

Comme le test est en aval de l'activité de programmation, les erreurs (humaines) déjà commises, ainsi que la façon de les éviter ne nous préoccupent pas ! Nous porterons notre attention sur les défauts qui ont été malencontreusement introduits afin de minimiser les anomalies qui risquent de se produire.

Sans nuire à la suite de ce cours, nous pouvons confondre, par abus de langage, erreur et défaut (tendance humaine à confondre cause et conséquence !!!)

Classes de défaut

L'ensemble des défauts pouvant affecter un logiciel est infini.

Mais, des **classes de défaut** peuvent être identifiées : Calcul, logique, E/S, traitement des données, interface, définition des données…

Les moyens pour détecter des défauts peuvent être automatiques ou manuels et s'appliquent aussi bien sur le code source qu'à son comportement.

Donnons maintenant une définition de l'activité test

dans un projet logiciel.

Le test : des définitions…

Remarque : Le test du logiciel est également appelé vérification dynamique.

Définition (issue de 'Le test des logiciels [SX-PR-CK-2000]) : Le test d'un logiciel est une activité qui fait partie du processus de développement. Il est mené selon les règles de l'assurance de la qualité et débute une fois que l'activité de programmation est terminée. Il s'intéresse aussi bien au code source qu'au comportement du logiciel. Son objectif consiste à minimiser les chances d'apparitions d'une anomalie avec des moyens automatiques ou manuels qui visent à détecter aussi bien les diverses anomalies possibles que les éventuels défauts qui les provoqueraient.

Définition (issue de la norme IEEE-STD729, 1983) : Le test est un processus manuel ou automatique, qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification, ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification.

Définition (issue de l'A.F.C.I.Q) : "Le test est une technique de contrôle consistant à s'assurer, au moyen de son exécution, que le comportement d'un programme est conforme à des données préétablies".

Définition (issue de 'The art of software Testing' [GJM]) : « Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts".

Qq commentaires…

Par conséquent, le test d'un logiciel :

- a pour objectif de réduire les risques d'apparition d'anomalies avec des moyens manuels et informatiques.
- fait partie du processus de développement.
- n'a pas pour objectif de :
  - de corriger le défaut détecté (débogage ou déverminage)
  - de prouver la bonne exécution d'un programme.

Procédure de test : On applique sur tout ou une partie du système informatique un échantillon de données d'entrées et d'environnement, et on vérifie si le résultat obtenu est conforme à celui attendu. S'il ne l'est pas, cela veut dire que le système informatique testé présente une anomalie de fonctionnement.

AFCIQ : Association Française pour le Contrôle Industriel et la Qualité

Difficultés du test

1. Processus d'introduction des défauts très complexe
2. mal perçu par les informaticiens
3. délaissé par les théoriciens
4. non décidable
5. Etc.

Conclusions : Impossibilité d'une automatisation complète

Évolution du test

Aujourd'hui, le test de logiciel :

- est la technique de validation la plus utilisée pour s'assurer de la correction du logiciel.
- fait l'objet d'une pratique bien souvent artisanale.

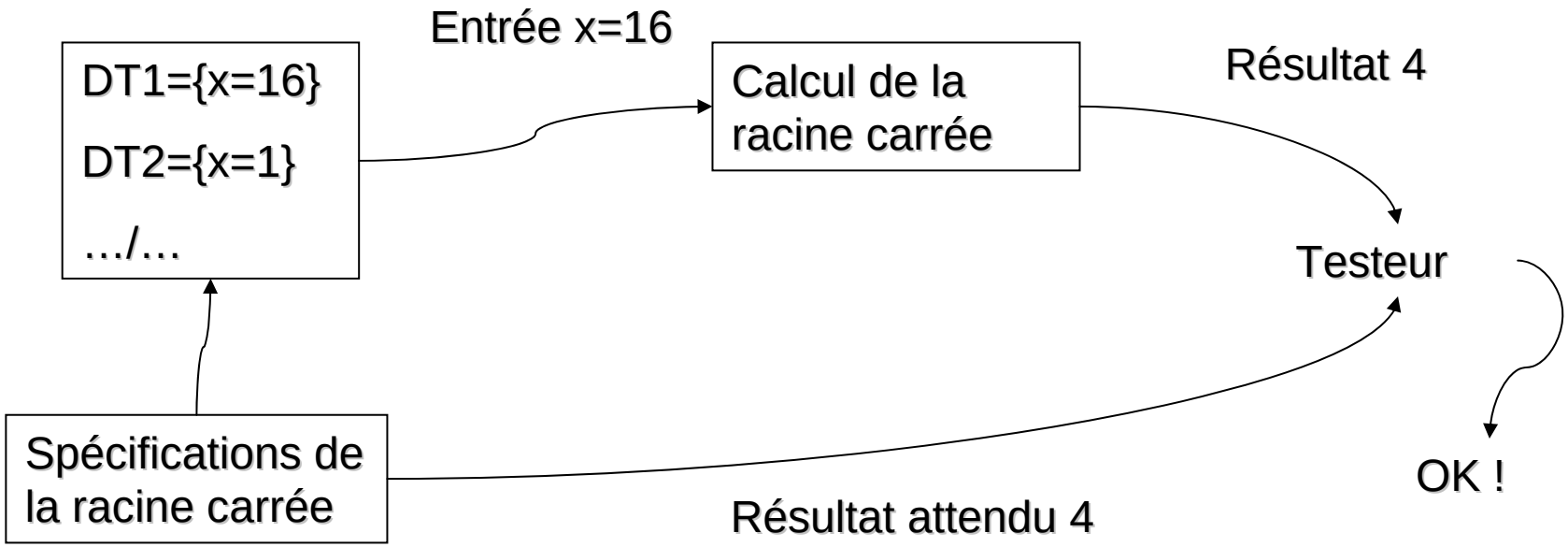
Demain, le test de logiciel devrait être :

- une activité rigoureuse,
- fondée sur des modèles et des théories
- automatique

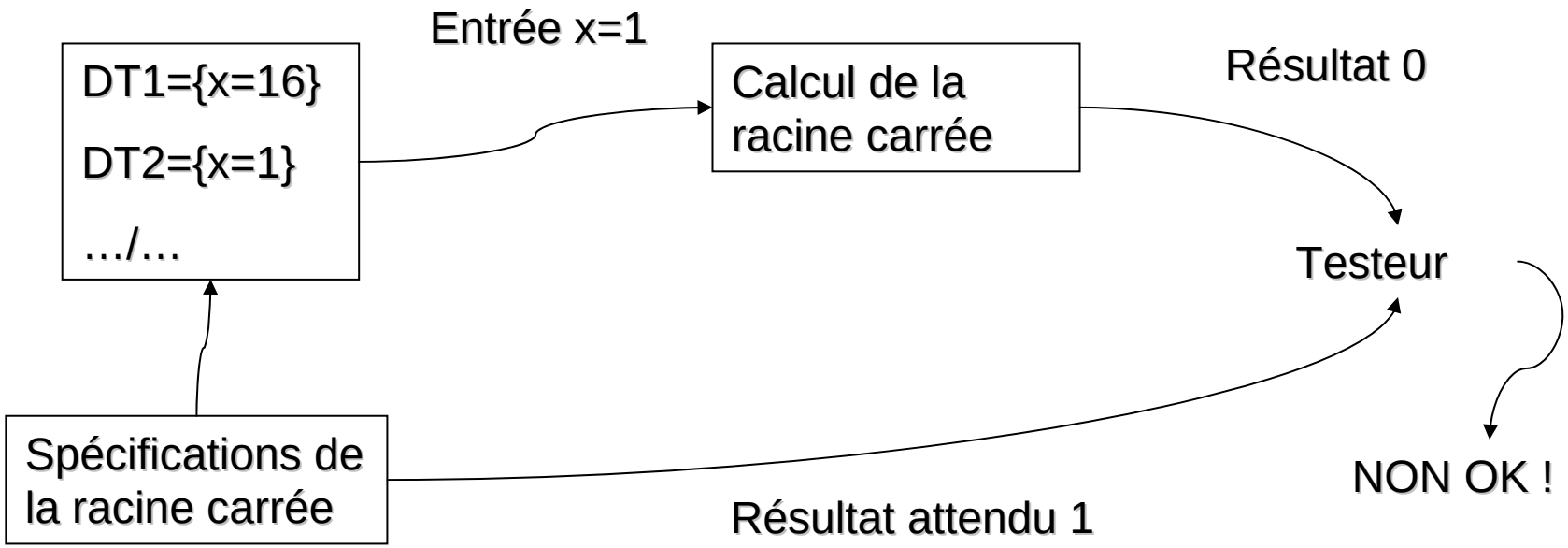
Approches du test

- L'activité de test se décline selon 2 approches :
  - rechercher statiquement des défaut simples et fréquents (contrôle)
  - définir les entrées (appelées 'données de test') qui seront fournies au logiciel pendant une exécution
- Exemple de données de test (DT)
  - DT1={a=2, z=4.3}
- Jeu de test : ensemble des DT produits pour tester
- Scénario de test : actions à effectuer avant de soumettre le jeu de test
- Le scénario de test produit un résultat
- Ce résultat doit être évalué de manière manuelle ou automatique pour produire un oracle

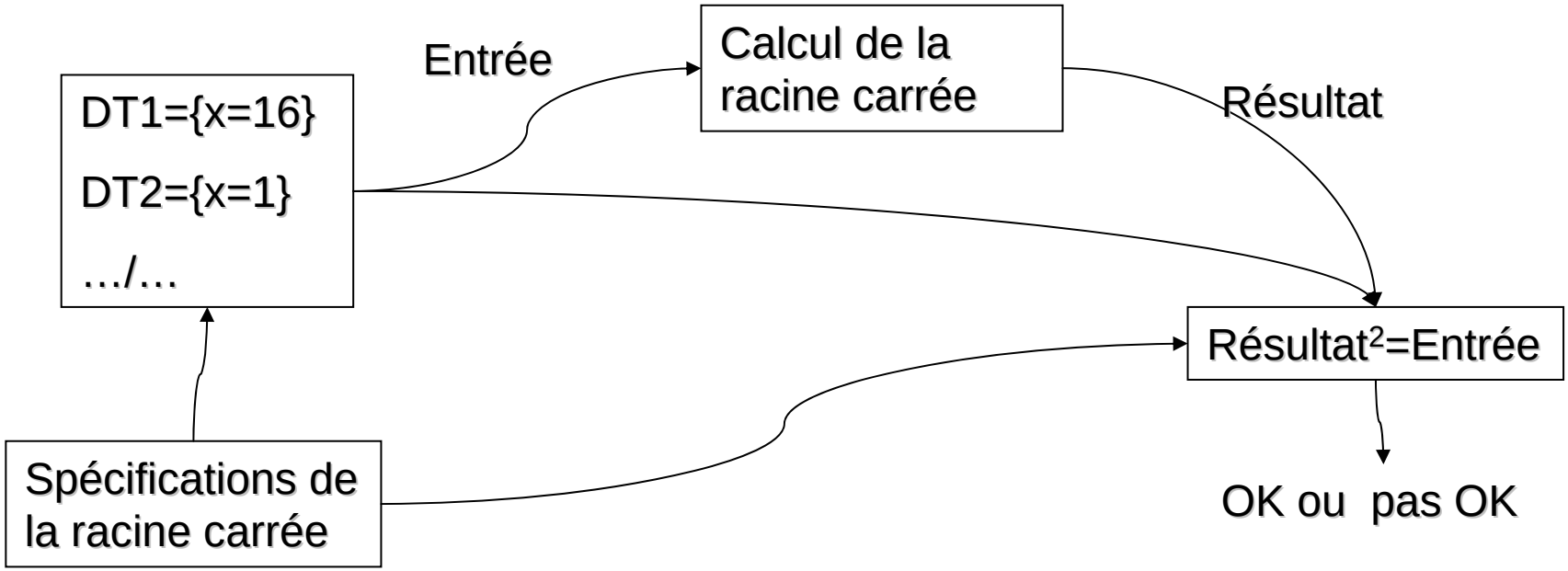
Exemple 1 de test avec oracle manuel



Exemple 2 de test avec oracle manuel



Exemple 3 de test avec oracle automatique



Choix des jeux de test

- Les données de test sont toutes les entrées possibles
  - test exhaustif
  - Idéal, mais non concevable !
- Les données de test constituent un échantillon représentatif de toutes les entrées possibles
  - exemple ‘Racine carrée’ :

    16, 1, 0, 2, 100, 65234, 826001234, -1, - 3
  - P:programme, F:spécification,

    $T \subset D$ est fiable $\Leftrightarrow$

    [pour tout $t \in T$ $F(t)=P(t) \Rightarrow$ pour tout $t \in D$ $F(t)=P(t)$]

=> Critère de sélection (procédure de définition) de jeux de test
  - Validité
  - Fiabilité

'Procédure de définition' valide? fiable?

Exemples de **critères de sélection**

- C1: J={n1,n2} avec n1<10 et n2 impair
  Ex: J1={4,3}, J2={9,291}, J1={2,3}, J1={9,31}, etc.
- C2: J sous ensemble de {1,2,3}
  Ex: J1={1}, J2={2}, J1={2,3}, J1={1,2,3}, etc.

- **Critère valide**: s'il induit au moins un jeu de test non réussi dans le cas d'un programme non correct
- **Critère fiable**: s'il produit uniquement des jeux de tests réussis ou uniquement des jeux de test non réussis

Ex: programme qui produit le n-ème nombre premier (3 étant le premier nombre premier)

```
lire(n)
p:= 2*n+1;
écrire (p);
```

Exemples de critères valides et fiables

- C3: J sous ensemble de {4,5,6,7,8}
- C4: J sous ensemble de {p/4<p<q}

Rappel : le but est de fournir des « DT représentatifs »

Autres approches : **critère adéquat**

Un jeu de test adéquat comporte des DT capables d'exclure la possibilité d'existence d'un certain type de défaut.

Classes de techniques de test

Les techniques de test peuvent être regroupées en différentes classes selon

- les critères adoptés pour choisir des DT représentatives,
- Les entités utilisées (spécification, code source, exécutable, etc.) pour s'assurer de l'absence de certains défauts

Exemples de classes :

1. Statique / dynamique
2. Structurelle / fonctionnelle
3. Manuel / automatique
4. Unitaire / intégration / système/ non-régression
5. Caractéristiques

Les modalités de test

1. Test statique : Test «par l'humain», sans machine, par lecture du code
   - inspection ou revue de code;
   - réunions (le programmeur, le concepteur, un programmeur expérimenté, un testeur expérimenté, un modérateur)
   - le but : trouver des erreurs dans une ambiance de coopération

2. Test dynamique : Test par l'exécution du système
   - qui teste-t-on ? une implantation du système (IUT = Implementation Under Test)
   - que teste-t-on ? une propriété/caractéristique à vérifier
   - un test réussit (Passes) si les résultats obtenus sont les résultats attendus, sinon il échoue (Fails);

Les niveaux de tests

Tests unitaires : s'assurer que les composants logiciels pris individuellement sont conformes à leurs spécifications et prêts à être regroupés.

Tests d'intégration :s'assurer que les interfaces des composants sont cohérentes entre elles et que le résultat de leur intégration permet de réaliser les fonctionnalités prévues.

Tests système : s'assurer que le système complet, matériel et logiciel, correspond bien à la définition des besoins tels qu'ils avaient été exprimés. [validation]

Tests de non-régression : vérifier que la correction des erreurs n'a pas affecté les parties déjà testées. [Cela consiste à systématiquement repasser les tests déjà exécutés]

Les méthodes de test

Dépendent de la connaissance de l'implémentation du programme testé

1. Les méthodes **structurelles** : production de jeux de test en analysant le code source
   - Connaissance totale de l'implantation;
   - Aussi appelées test en boîte blanche, ou test basé sur l'implantation.
   - possibilité de fixer finement la valeur des entrées pour sensibiliser des chemins particuliers du code;
   - conception des tests uniquement pour le code déjà écrit.

2. Les méthodes **fonctionnelles** : production de jeux de test en considérant les résultats fournis par l'exécution du programme sans se soucier de sa structure interne.
   - Aucune connaissance de l'implantation;
   - Aussi appelées test en boîte noire, ou test basé sur la spécification.
   - Repose exclusivement sur la spécification
   - permet d'écrire les tests avant le codage;

3. En pratique : Combinaison des deux méthodes fonctionnelles et structurelles.

3. Les tests **orientés-erreurs** :

   Ces tests permettent de découvrir la présence d'erreurs dans les programmes. On pourra citer parmi les techniques utilisées dans ce cas, les méthodes statistiques, le "semage" d'erreurs et les tests de mutation.

Test manuel / test automatisé

1. Test **manuel**
   - le testeur entre les données de test par ex via une interface;
   - lance les tests;
   - observe les résultats et les compare avec les résultats attendus;
   - fastidieux, possibilité d'erreur humaine;
   - ingérable pour les grosses applications;
2. Test **automatisé**
   - Avec le support d'outils qui déchargent le testeur :
     - du lancement des tests;
     - de l'enregistrement des résultats;
     - parfois de la génération de l'oracle;
     - test unitaire pour Java: JUnit
     - génération automatique de cas de test : de plus en plus courant (cf Objecteering).
3. **Built-in** tests
   - Code ajouté à une application pour effectuer des vérifications à l'exécution:
     - par ex des assertions !
     - ne dispense pas de tester ! test embarqué différent de code auto-testé !
     - permet un test unitaire "permanent", même en phase de test système;
     - test au plus tôt;
     - assertions: permettent de générer automatiquement l'oracle;

Test de caractéristiques

Quelques exemples :

- test de **robustesse** : permet d'analyser le système dans le cas où ses ressources sont saturées ou bien d'analyser les réponses du système aux sollicitations proche ou hors des limites des domaines de définition des entrées. La première tâche à accomplir est de déterminer quelles ressources ou quelles données doivent être testées. Cela permet de définir les différents cas de tests à exercer. Souvent ces tests ne sont effectués que pour des logiciels critiques, c'est-à-dire ceux qui nécessitent une grande fiabilité.

- test de **performance** : permet d'évaluer la capacité du programme à fonctionner correctement vis-à-vis des critères de flux de données et de temps d'exécution. Ces tests doivent être précédés tout au long du cycle de développement du logiciel d'une analyse de performance, ce qui signifie que les problèmes de performances doivent être pris en compte dès les spécifications.

Classification des tests

Classement des techniques de tests de logiciels selon :

- Critères adoptés pour choisir des DT représentatives
- Entités utilisées (spécification, code source, ou code exécutable)

- Techniques fonctionnelles / structurelles
- Techniques statiques / dynamiques
- Techniques combinant fonctionnelles, structurelles, dynamiques et statiques (c'est le cas du test boîte grise)

Un exemple de classement selon trois axes :

- le **niveau de détail** (étape dans le cycle de vie)
- le **niveau d'accessibilité**
- la **caractéristique**

Classification des tests (suite)

Niveau de détail

- tests unitaires : vérification des fonctions une par une,
- tests d'intégration : vérification du bon enchaînement des fonctions et des programmes,
- tests de non-régression : vérification qu'il n'y a pas eu de dégradation des fonctions par rapport à la version précédente,

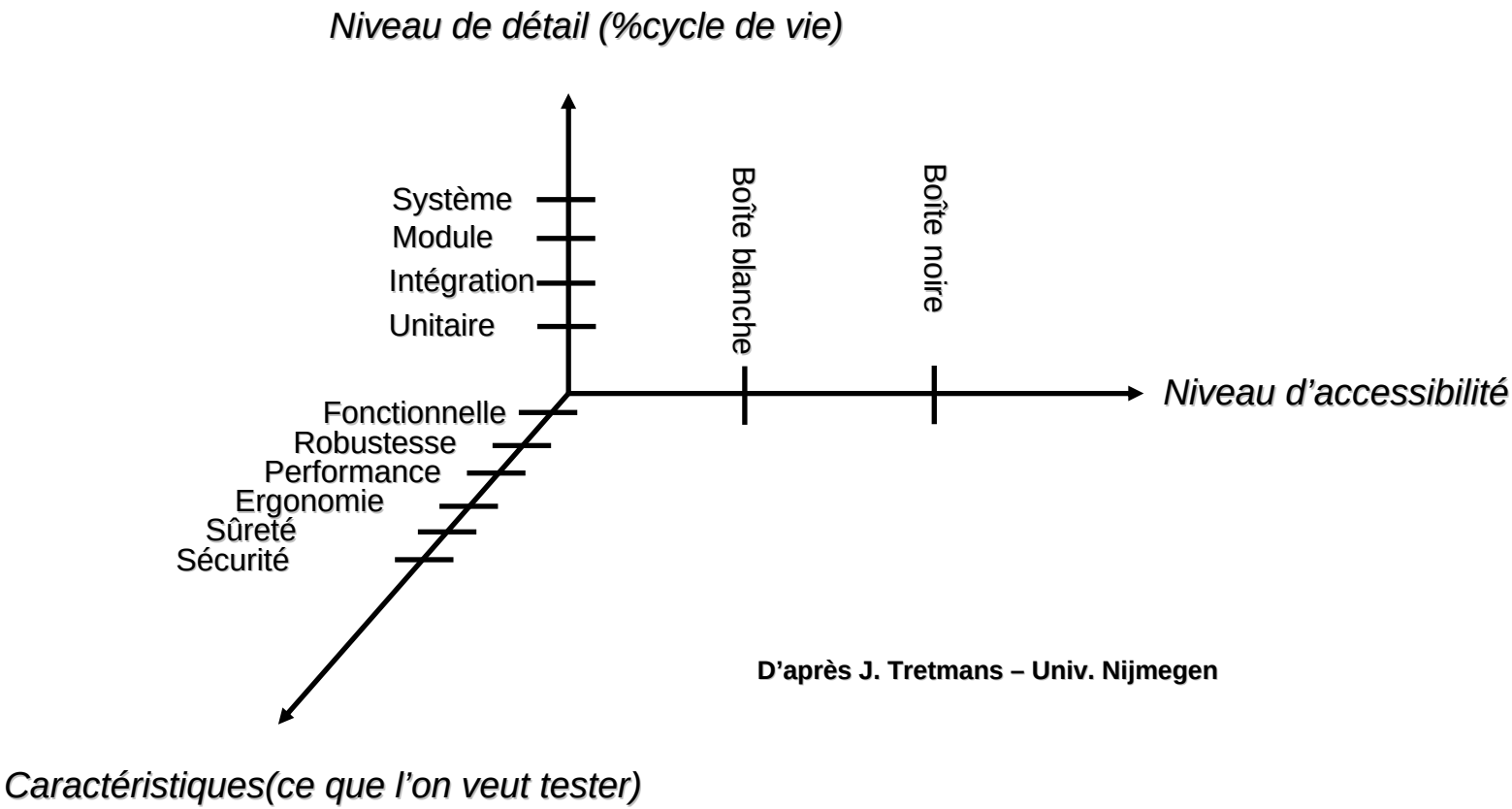
Niveau d'accessibilité

- Boîte noire : à partir d'entrée définie on vérifie que le résultat final convient.
- Boîte blanche : on a accès à l'état complet du système que l'on teste.
- Boîte grise : on a accès à certaines information de l'état du système que l'on teste.

Caractéristique :

- test fonctionnel
- test de robustesse
- test de performance

Classification des tests (suite)



Quelques exemples d'application

Test de programmes impératifs

- modèles disponibles : ceux issus de l'analyse de leur code source
- Donc : méthodes de test structurelles pour couvrir le modèle
- Couverture suivant des critères liés au contrôle ou aux données.

Test de conformité des systèmes réactifs

- Modèle disponible : la spécification
- Donc : méthodes de test fonctionnelles
- génération automatique de tests de conformité,

Test de systèmes

- Techniques de test d'intégration lors de la phase d'assemblage
- Aspects méthodologiques
- Test système.

Stratégie de test

Une technique de test doit faire partie d'une stratégie de test

- adéquation avec le plan qualité
- Intégration dans le processus de développement des logiciels
- Une technique de test puissante restera sans effet si elle ne fait pas partie d'une stratégie de test…

La stratégie dépend :

- de la criticité du logiciel
- du coût de développement

Une stratégie définit :

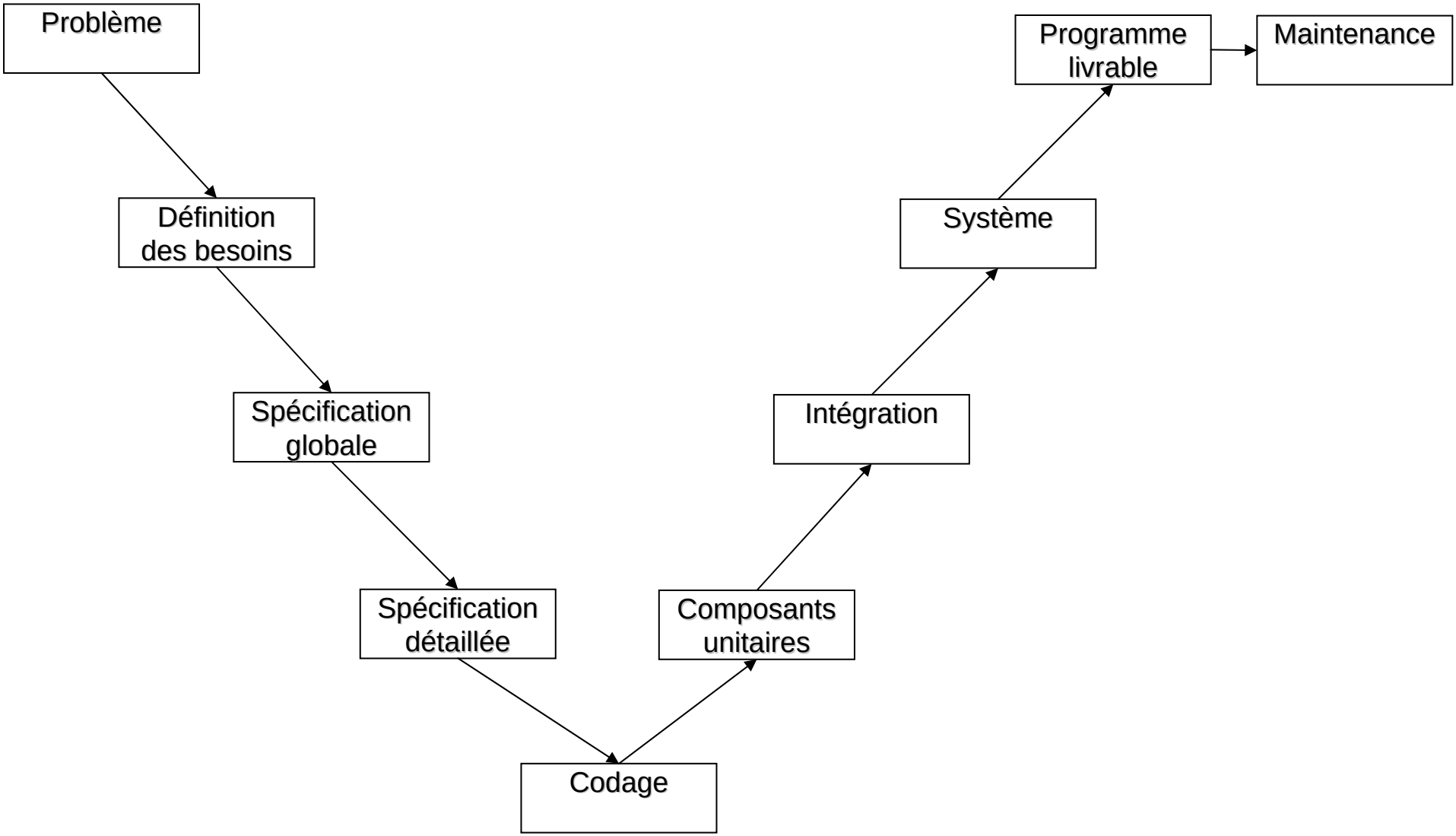
- Des ressources mises en œuvre (équipes, testeurs, outils, etc.)
- Les mécanismes du processus de test (gestion de configuration, évaluation du processus de test, etc.)

Une stratégie tient compte :

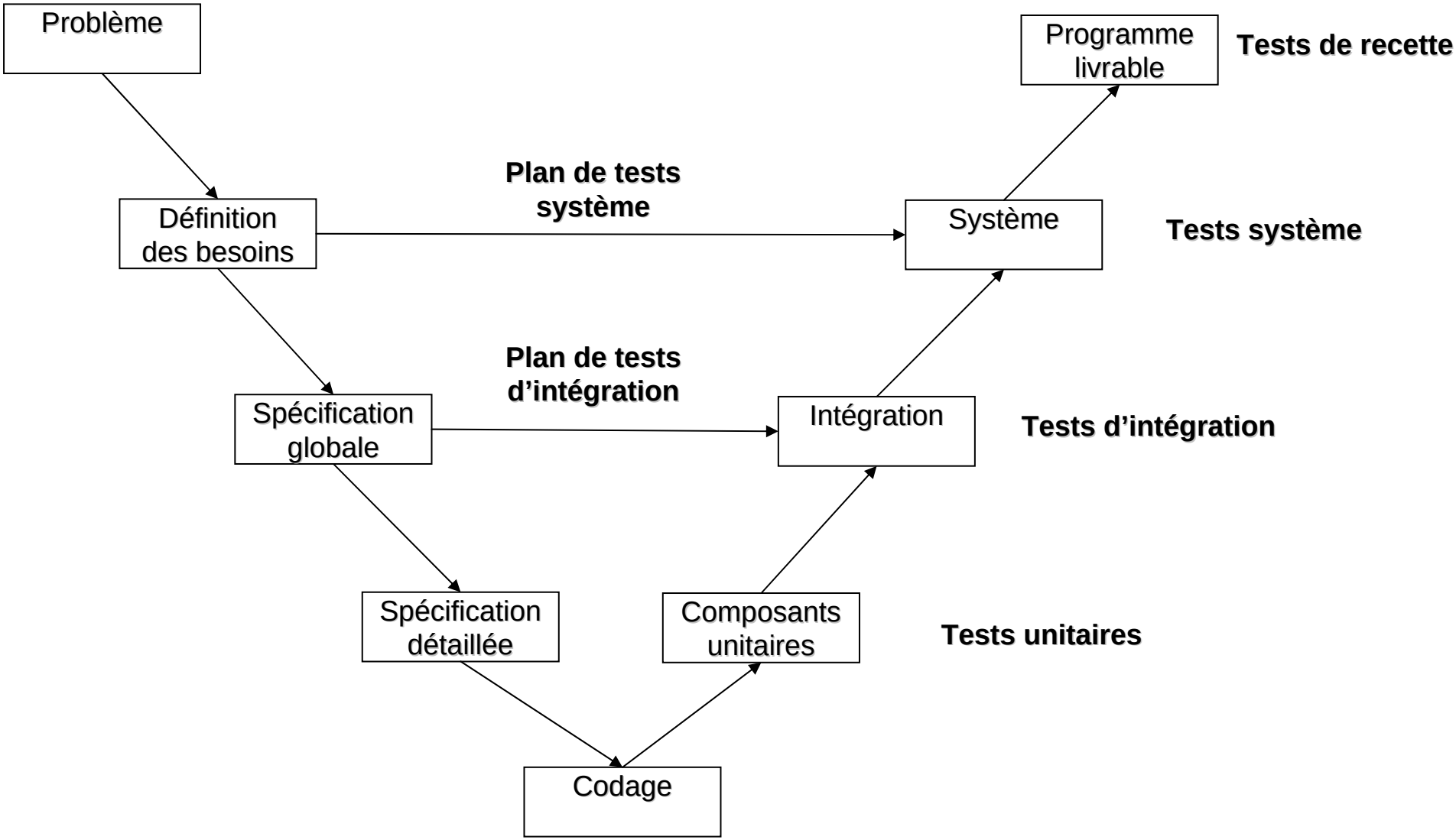
- Des méthodes de spécif, conception
- Langages de programmation utilisés
- Du types d'application (temps réel, protocole, base de données…)
- L'expérience des programmeurs
- Etc.

Partie 3:
Le test dans un projet logiciel

Cycle de développement en V



Hiérarchisation des tests



Le test dans le cycle de développement

tester dès que possible

- tester dès l'analyse

processus de développement itératif

- tester à chaque incrément
- test de régression (ou non-régresssion): nouveau test du système après une modification pour vérifier qu'elle n'a pas apporté d'autres fautes;

eXtreme programming: on écrit les tests puis on code

- formellement déconseillé d'utiliser les tests comme spécification !

test de montée en charge

- test des performances;

test de recette

- pour obtenir l'approbation du client avant la livraison.

Test unitaire

Test d'une **unité logicielle** : test de base réalisé par le programmeur au fil du développement

- dépend fortement du paradigme de programmation utilisé :
  - procédural : la procédure
  - OO : la classe et ses méthodes

    On écrit pour chaque classe testée une ou plusieurs classes contenant une suite de test, avec pour chaque méthode un ou plusieurs cas de test.
- pour détecter des fautes dans son comportement individuel;
- que faire si le comportement dépend d'autres unités ? Différents points de vue :
  1. on ne teste pas (unitairement);
  2. on teste en utilisant les autres unités si elles sont disponibles, mais alors c'est plutôt du test d'intégration;
  3. on simule le comportement des autres unités par des bouchons [ou stubs].
- que faire si le comportement dépend d'éléments non contrôlables ? (ex réseau, base de données, etc) : on utilise des bouchons.
- attention aux inter-dépendances entre méthodes.
- en boîte blanche ou boîte noire.

Test unitaire

Les réticences : '*Tester est comme manger des légumes quand on n'aime pas ça, on sait que c'est bon pour la santé mais on préfère manger des …*'.

- ça prend trop de temps d'écrire des tests !
- ça prend trop de temps d'exécuter des tests !
- ce n'est pas mon boulot de tester mon code !
- je suis payé pour écrire du code, pas pour le tester !

Écrire des programmes testables :

- La classe de test est extérieure à la classe testée;
- Il faut aussi tester les méthodes privées;
- Il faut pouvoir :
  - accéder à l'état d'un objet;
  - amener un objet dans un état propice au test.

Ne pas hésiter devant le refactoring

- Tester amène souvent à revoir son code en l'améliorant…

Test unitaire : Que tester ?

Vient avec l'expérience…mais quelques repères [Hunt Thomas] :

1. Au préalable, découper le comportement de la méthode en sous comportements (classes d'équivalences) qu'il faudra tester individuellement :
   - dans tel cas, la méthode doit lancer une exception;
   - dans tel autre cas, elle doit retourner ça;
   - dans tel autre cas encore, elle doit retourner autre chose;
   - etc.

2. Le résultat est-il juste ?
   - S'assurer qu'on a bien testé le retour d'une fonction/la levée des exceptions,
     [en général c'est le plus facile et c'est ce qu'on fait en premier]

3. Conditions aux limites : C'est souvent les conditions aux limites qui posent pb dans une application !
   - **Conformance** : Est-ce que la donnée est conforme à un format pré-défini ? (Ex. sur le traitement d'une adresse email : ? @ ? . ?
   - **Ordering** : Dans le cas où on travaille sur une collection ordonnée :
     - si on cherche une valeur : vérifier qu'on la trouve bien en tête/milieu/fin de collection
     - si une méthode prend une collection en entrée : le code présuppose-t-il un ordre particulier pour cette collection ?
     - si une donnée interne doit être maintenue triée, le vérifier ;

Test unitaire : Que tester ? (suite)

- **Range**
  - Cas où une variable peut prendre ses valeurs dans un intervalle donné, souvent plus grand que celui qui nous intéresse (un age codé sur un entier par exemple);
    - éviter le codage sur un type simple "trop grand", créer son propre type à la place, gardé par des assertions
    - utiliser intensivement des pré-cond et des invariants de classe ;
    - penser à tester les valeurs litigieuses : une valeur nominale, mais aussi la plus petite valeur, et la plus grande.
- **Reference**
  - Cas où votre méthode référence d'autres méthodes ou classes : dans quelles conditions peuvent-elles être utilisées ?
    - regarder scrupuleusement les documentations à la recherche de pré-post condition explicites ou non.
- **Cardinality**
  - Quand il faut compter... par exemple si on doit maintenir et publier un top-ten des meilleures ventes :
    - peut-on publier un top-ten vide ? à un élt ? à moins de 10 elts ? à 10 elts ?
    - et si la société ne vend que 5 articles ? 0 ?
    - et si brusquement on passe à un top-5 ?
- **Time**
  - problèmes de gestion du temps réel (quel calendrier, changements d'heures, etc)

Test unitaire : Que tester ? (suite)

3. Check Inverse RelationShips
   - Symétrie et fonction inverse : Si on calcule une racine carrée, vérifier que le résultat élevé au carré donne la valeur initiale.

4. Cross-checking with other means
   - Re-calculer un résultat en utilisant une autre version (version plus ancienne abandonnée car moins efficace mais déjà testée par ex).
   - Vérifier au moyen d'invariants les choses du style "si j'emprunte un livre j'en ai un de plus emprunté, un de moins libre, au total toujours le même nombre".

5. Force Error Conditions
   - Vérifier le comportement de la méthode dans les mauvais cas qui finissent toujours par se produire :
     - plus de mémoire, ou d'espace disque;
     - erreur réseau, plus de réseau;
     - base de données plantée;
     - etc.

6. Performance characteristics

Test d'intégration

Test d'un **ensemble d'unités** qui coopèrent;

- But : détecter des erreurs dans leur interopérabilité, la mauvaise utilisation d'une interface;
- interconnexion de composants (niveau macro);
- commence très tôt en objet (niveau micro): une classe est typiquement composée d'objets d'autres classes;
- bien repérer l'inter-dépendance des classes pour choisir un ordre d'intégration :
  - si les dépendances forment un arbre (un ordre partiel), alors on peut intégrer simplement de bas en haut;
  - s'il y a un cycle de causalité (A dépend de B qui dépend de A), fréquent :
    - on émule une des classes (par ex A);
    - on teste B avec l'émulation de A;
    - on teste A avec B;
    - on reteste B avec le vrai A.
- typiquement en boîte noire.

Test système

- L’application à tester est complètement intégrée dans son environnement :
  - inclut les autres applications utilisées,
  - l'environnement opérationnel (par exemple la JVM).
- on teste les scénarios intéressants déterminés lors de l'analyse (use cases, sequence diagrams);
- en boîte noire uniquement :
  - Le spécification est alors le seul critère de référence.

Des règles de bon sens

Concernant la forme des cas de test :

- inclure dans un cas de test des entrées pour le programme mais aussi le résultat attendu (sortie calculée, émission d'une exception, impression d'un message, etc);
- toujours déterminer le résultat attendu par rapport à la spécification du programme (pas au code);
- stocker les cas de tests pour pouvoir les exécuter à nouveau;
- soigner la traçabilité des tests.

Concernant le processus de test :

- Si possible faire tester par un autre développeur que celui du code sous test;
- examiner très attentivement les rapports de test, les stocker aussi;
- À chaque modification : relancer tous les cas de tests (non régression).

Concernant le choix des objectifs de test :

- vérifier que le programme se comporte bien dans les cas attendus comme dans les cas invalides;
- si exception levée : vérifier qu'elle l'est;

Nécessaire recours au test

Apport et limite du test

Stratégie de test

Panorama des techniques de test

Les phases de test dans le projet logiciel

Partie 4:
Le test structurel

Statique / Dynamique

- Analyse **dynamique** : nécessite l'exécution du code binaire

Principe : à partir du code source et spécification, produire des DT qui exécuteront un ensemble de comportements, comparer les résultats avec ceux attendus…

1. Techniques de couverture du graphe de contrôle
   a) Couverture du flot de contrôle
   b) Couverture du flot de données
2. Test mutationnel (test par injection de défaut)
3. Exécution abstraite
4. Test évolutionniste (algorithme génétique)
5. …

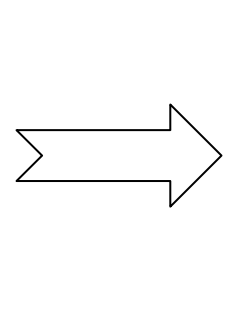
- Analyse **statique** : ne nécessite pas l'exécution du code binaire
  1. Revue de code
  2. Estimation de la complexité
  3. Preuve formelle (prouveur, vérifieur ou model-checking)
  4. Exécution symbolique
  5. Interprétation abstraite

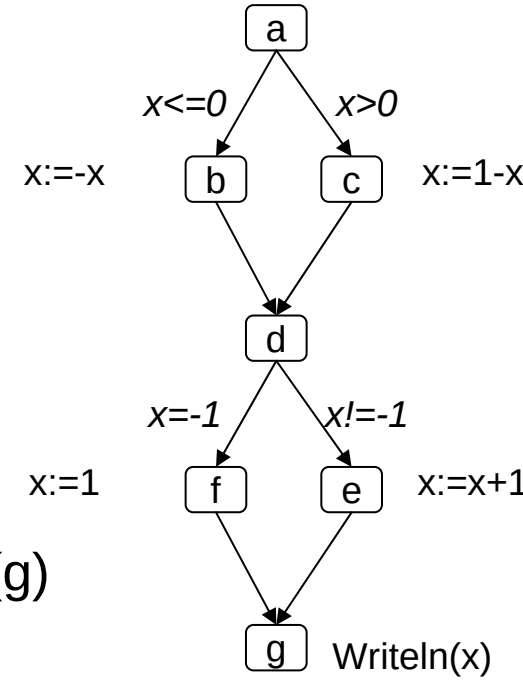
Test structurel dynamique avec technique de couverture du graphe de contrôle

But: produire des DT qui exécuteront un ensemble de comportements du programme

- Utilise : spécification, code source et code exécutable
- Un programme => un graphe de contrôle

```
begin
if (x<=0)then x:=-x
else x:=1-x;
if (x=-1)then x:=1
else x:=x+1;
end
```





- Un sommet entrée (a) et un sommet sortie (g)
- Un sommet = un bloc d'instructions
- Un arc = la possibilité de transfert de l'exécution d'un nœud à un autre

- Une **exécution possible** = un **chemin de contrôle** dans le graphe de contrôle

  [a,c,d,e,g] est un chemin de contrôle

  [b,d,f,g] n'est pas un chemin de contrôle

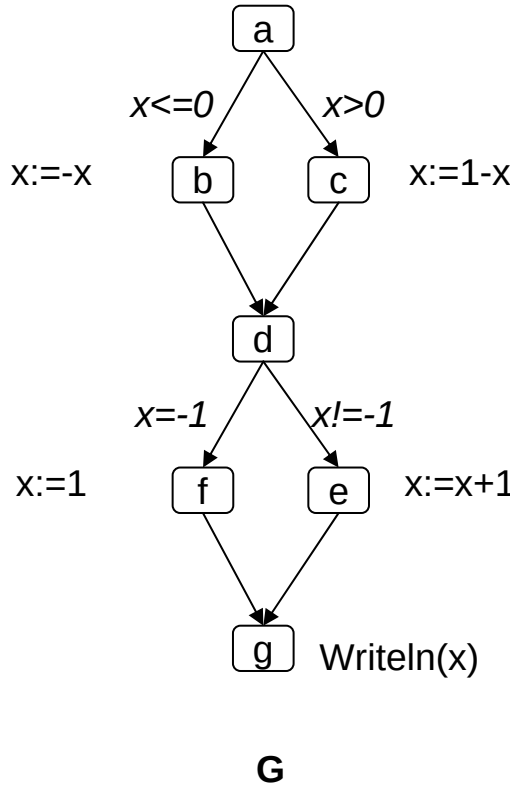
Expression des chemins d'un graphe de contrôle

Soit M l'ensemble des chemins de contrôle du graphe G :

M = abdfg+abdeg+acdfg+acdeg

= a.(bdf+bde+cdf+cde).g

= a.(b+c)d.(e+f).g (expression des chemins de G)



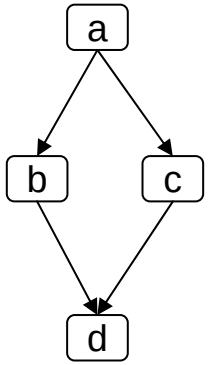
G

Construction de l'expression des chemins :

séquentielle

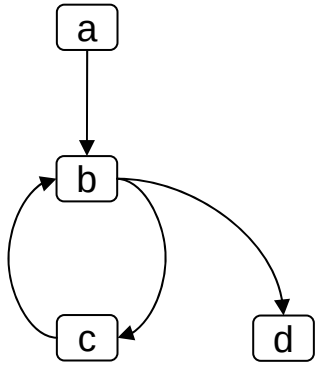
ab

alternative



a.(b+c).d

répétitive

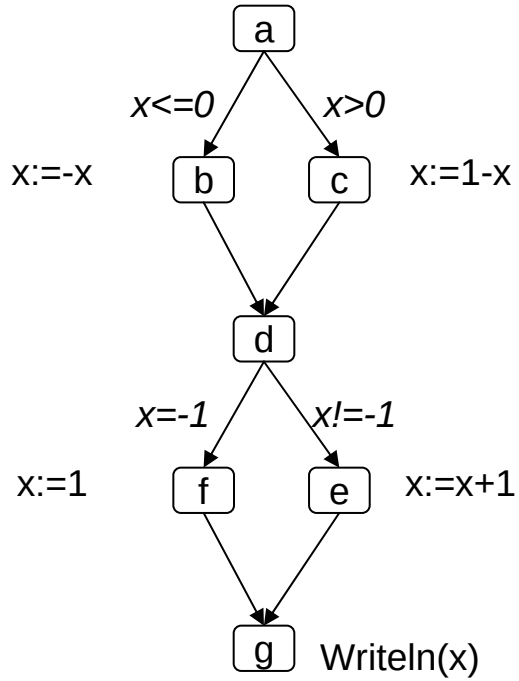


$ab.(cb)^{*}.d$

$ab.(cb)^{\underline{4}}.d$

Chemin exécutable

```
begin
if (x<=0)then x:=-x
else x:=1-x;
if (x=-1)then x:=1
else x:=x+1;
end
```



DT1={x=2}

DT1 sensibilise le chemin [acdfg] : [acdfg] est un chemin exécutable

[abdgf] est un chemin non exécutable : aucune DT capable de sensibiliser ce chemin

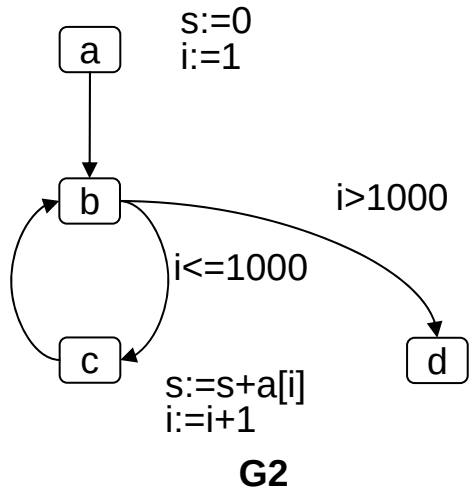
Sensibiliser un chemin peut parfois être difficile : intérêt des outils automatiques (mais attention problème de trouver des DT qui sensibilise un chemin est non décidable)

Existence de chemins non exécutables : signe de mauvais codage ?

Chemin exécutable / chemin non exécutable

- Nombre de chemins de contrôle de G :
  - se déduit directement de l'expression des chemins de G
  - a(b+c)d(e+f)g => 1.(1+1).1.(1+1).1 = 4 chemins de contrôle
  - Nb chemins exécutables + Nb chemins non exécutables
  - Parfois le Nb chemins non exécutables peut être important :

```
begin
s:=0;
for i:=1 to 1000 do s:=s+a[i];
end
```



Expression des chemins de G2 : $a.b.(cb)^{\underline{1000}}.d$

Nombre de chemins :

$1.1.(1.1)^{\underline{1000}}.1 = 1^{\underline{1000}} = 1+1^1+1^2+ \ldots +1^{1000}=1001$

Parmi ces 1001 chemins, un seul est exécutable: $a.b.(cb)^{1000}.d$

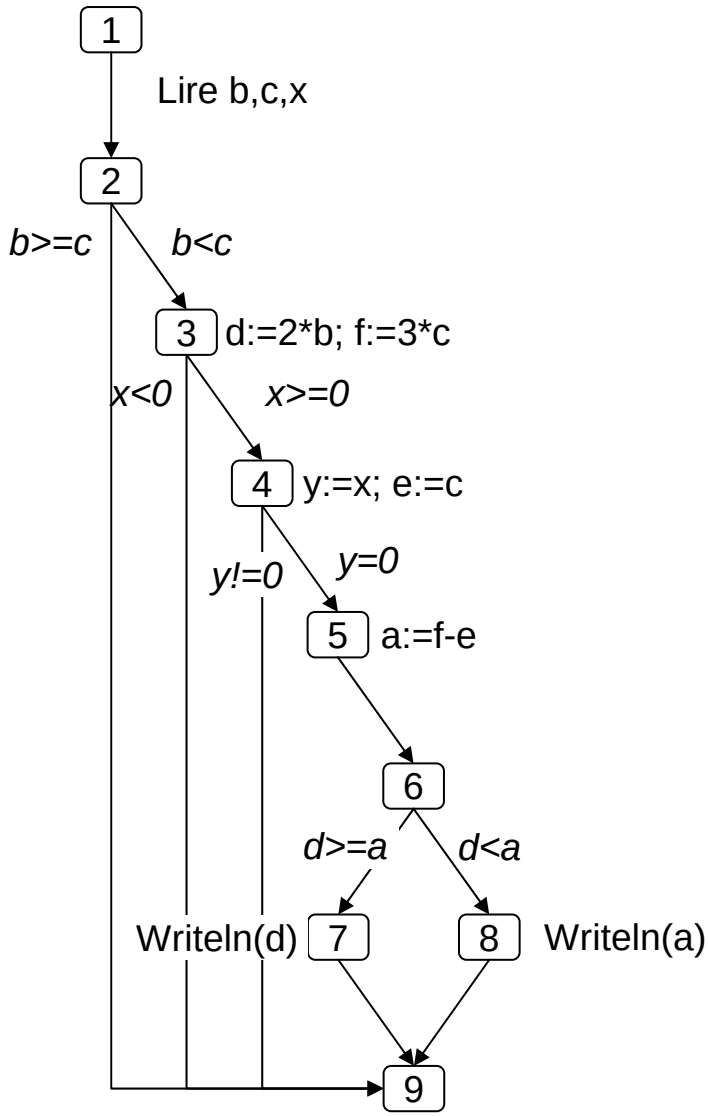
Exercice 1

```
lire(b,c,x);
if b<c
then begin
     d :=2*b ;
     f :=3*c
     if x>=0
     then begin
          y := x ;
          e := c ;
          if (y=0)
          then begin
               a :=f-e ;
               if d<a
               then begin
                    writeln(a)
                    end
               else begin
                    writeln (d)
                    end
               end
          end
     end
```

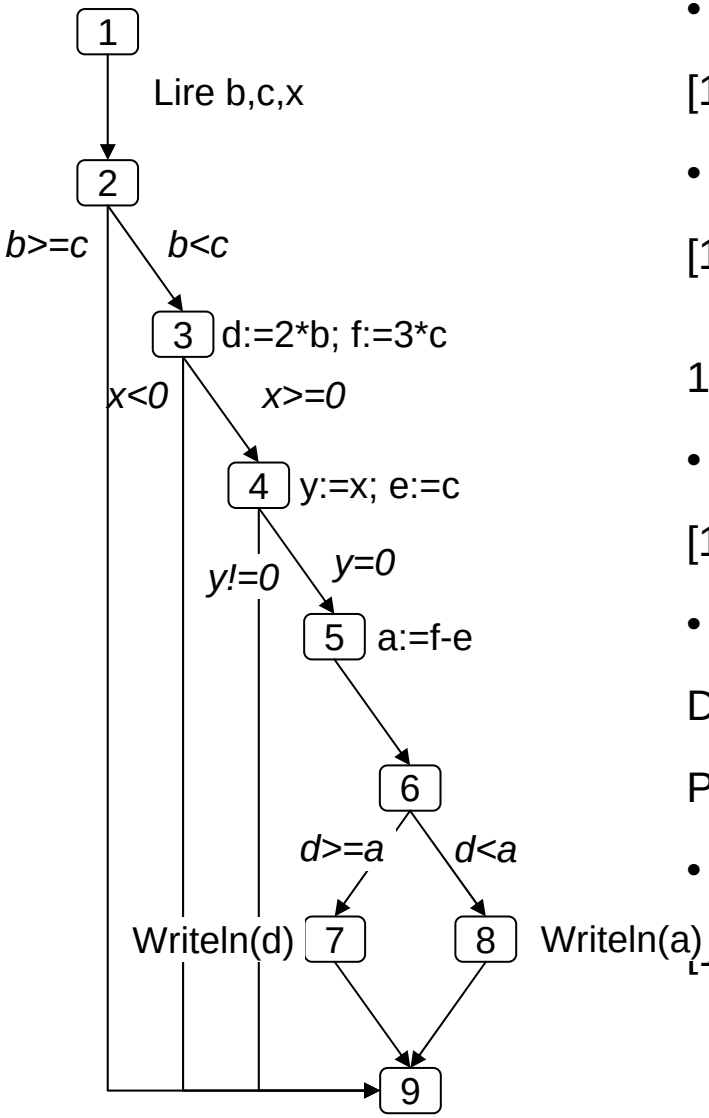
- Donner le graphe de contrôle G(P3) associé au programme P3.
- Donner 3 chemins de contrôle du graphe G(P3).
- Donner l'expression des chemins de contrôle de G(P3).
- Soit DT1={b=1,c=2,x=2}. Donner le chemin sensibilisé par DT1.
- On s'intéresse aux instructions en italique… Donner des DT qui vont couvrir ces instructions.
- Donner un chemin de contrôle non exécutable de G(P3).

Exercice 1 (correction)

```
lire(b,c,x);
if b<c
then begin
     d :=2*b ;
     f :=3*c
     if x>=0
     then begin
          y := x ;
          e := c ;
          if (y=0)
          then begin
               a :=f-e ;
               if d<a
               then begin
                    writeln(a)
                    end
               else begin
                    writeln (d)
                    end
               end
          end
     end
```



Exercice 1 (correction)



- 3 chemins de contrôle :

[12345689] [12345679] [129]

- Expression des chemins de contrôle :

[12345689]+[12345679]+[123459]+[12349]+
[1239]+[129]=

12.[34568+34567+345+34+ 3+ε].9

- Chemin sensibilisé par DT1={b=1,c=2,x=2}:

[12349]

- DT qui couvrent les instructions en italique:

DT2={b=1,c=2,x=0} couvre «writeln(a)»

Pas possible de couvrir «writeln(d)».

- Un chemin de contrôle non exécutable :

[12345679]

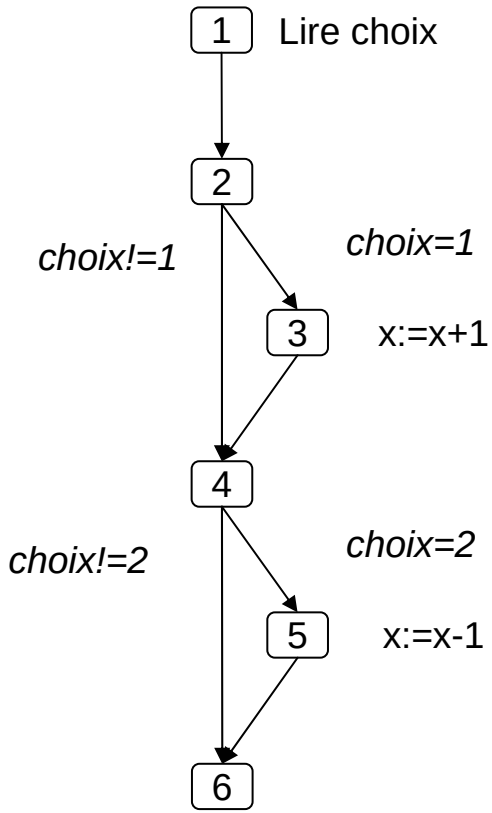
Exercice 2

```
Lire(choix)

if choix=1

then x=x+1 ;

if choix=2

then x=x-1 ;

writeln(choix ;
```

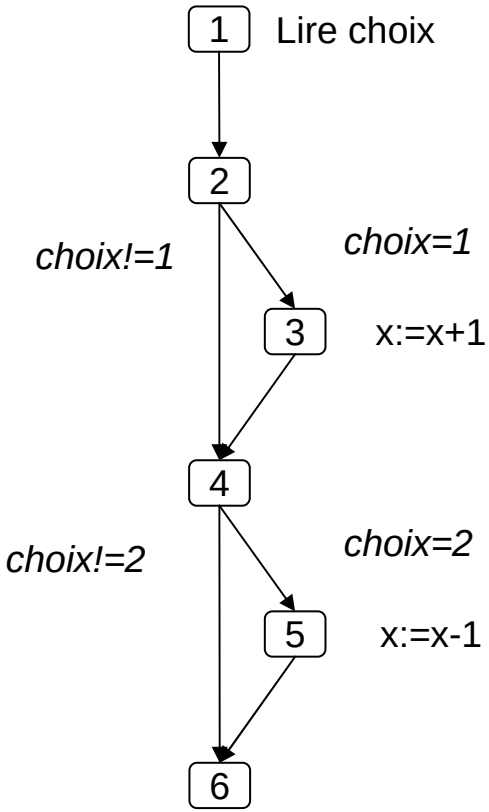
1. Donner le graphe de contrôle correspondant au programme P4.
2. Donner l'expression des chemins de contrôle de G(P4). En déduire le nombre de chemins de contrôle.
3. Donner les chemins de contrôle non exécutables. Conclure.
4. Proposer une nouvelle solution pour ce programme. Construisez son graphe de contrôle et donner l'expression des chemins de contrôle ainsi que le nombre de chemins de contrôle.

Exercice 2

```
Lire(choix)

if choix=1

then x=x+1 ;

if choix=2

then x=x-1 ;

writeln(choix) ;
```



Exercice 2 (correction)



1. Graphe de contrôle.

2. Expression des chemins de contrôle :

[123456]+[12346]+[12456]+[1246]=

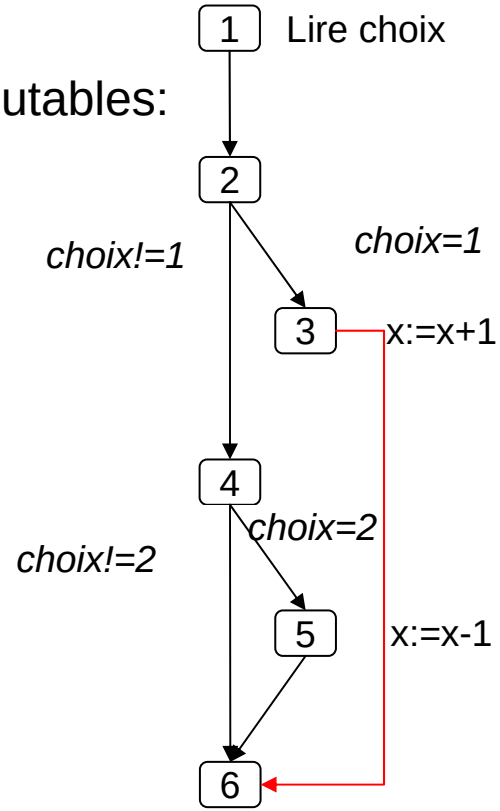
12.[3+ε].4.[5+ ε].6

3. Chemins de contrôle non exécutables:

[123456]

4. Une nouvelle solution :

```
Lire(choix)
if choix=1
then x=x+1 ;
else if choix=2
        then x=x-1;
writeln(choix);
```



Exercice 3

1. Écrivez un algorithme de recherche de l'emplacement d'un élément e dans un tableau T (on suppose que T contient l'élément e).

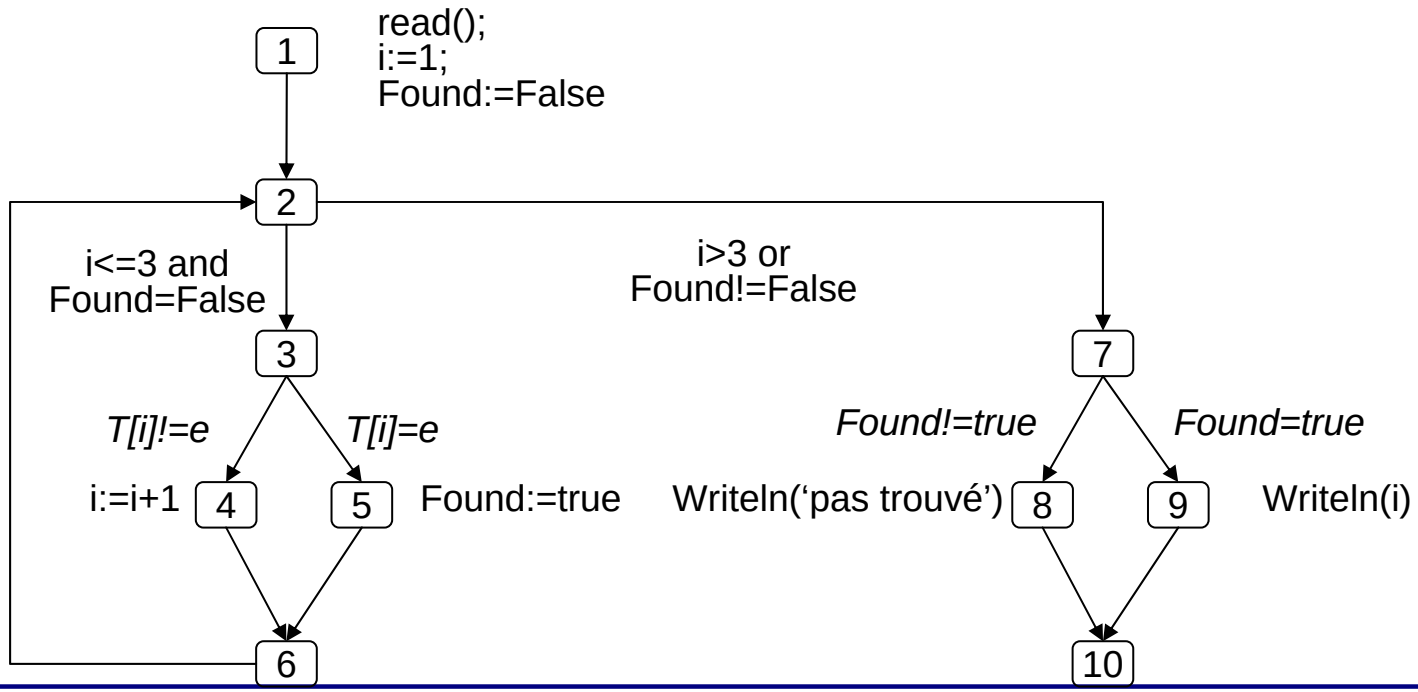
2. Donner le graphe de contrôle associé.

3. Donner l'expression des chemins.

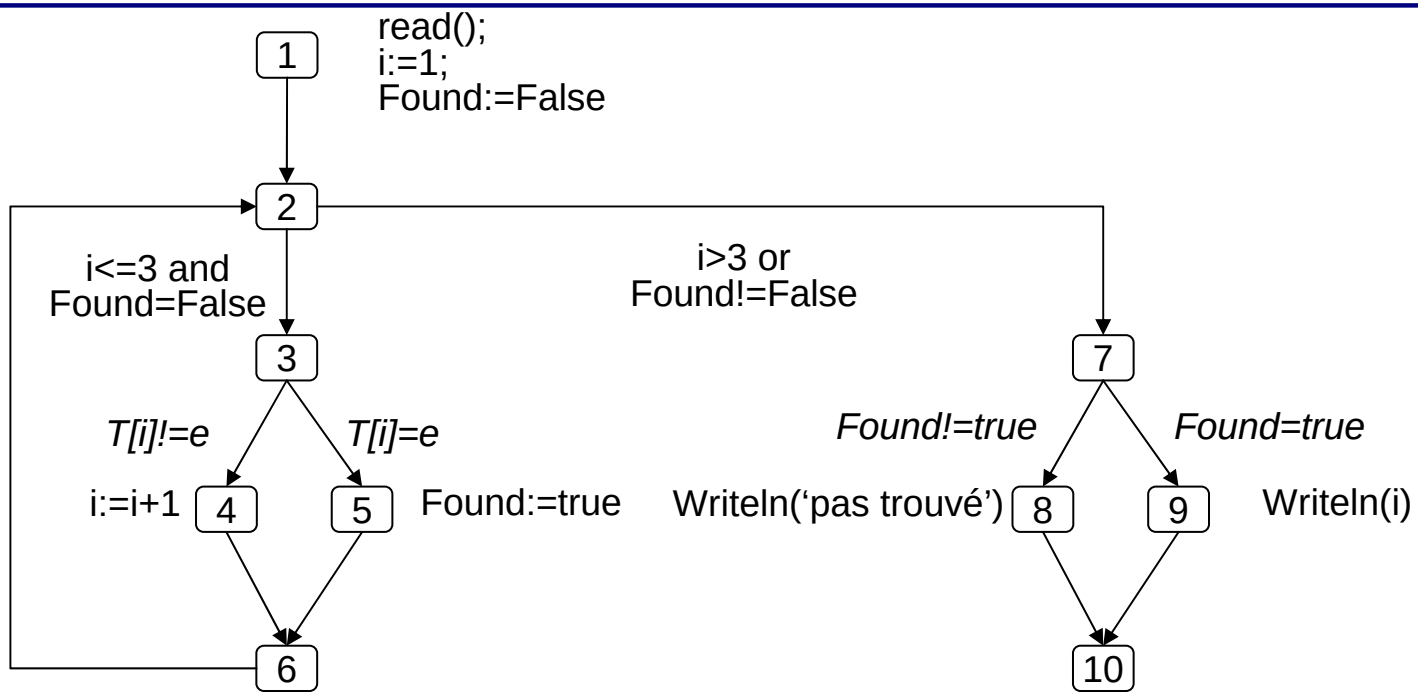
4. Dans le cas où le tableau a une taille de 3, donner le nombre de chemins de contrôle.

Exercice 3 (correction)

```
read(e,T[1],T[2],T[3])
i:=1;
Found:= false;
While i<=3 and Found=False
Do begin
   if T[i]=e then found:=true;
   else i:=i+1;
   end
If found then writeln(i);
Else writeln('pas trouvé');
```



Exercice 3 (correction)



Expression des chemins de contrôle : $1.2.[3.[4+5].6]^{*}.7.[8+9].10$

Nombre de chemins de contrôle : $1x1x[1x[1+1]x1]^{\underline{3}}x1x[1+1]x1=$
$[2]^{\underline{3}}x[2]=(2^0+2^1+2^2+2^3)x2=(1+2+4+8)x2=30$

Satisfaction d'un test structurel avec couverture

Soit T un test structurel qui nécessite la couverture d'un ensemble de chemins $\{\delta_1, \ldots, \delta_k\}$ du graphe de contrôle.

On notera : $T = \{\delta_1, \ldots, \delta_k\}$

Soit DT une donnée de test qui sensibilise le chemin de contrôle C.

- Définition: DT **satisfait** T ssi C couvre tous les chemins de T.
- Exemple : considérons le graphe de contrôle G5

  Soient δ_1=cdebcde et δ_2=ce et $T1 = \{\delta_1, \delta_2\}$.

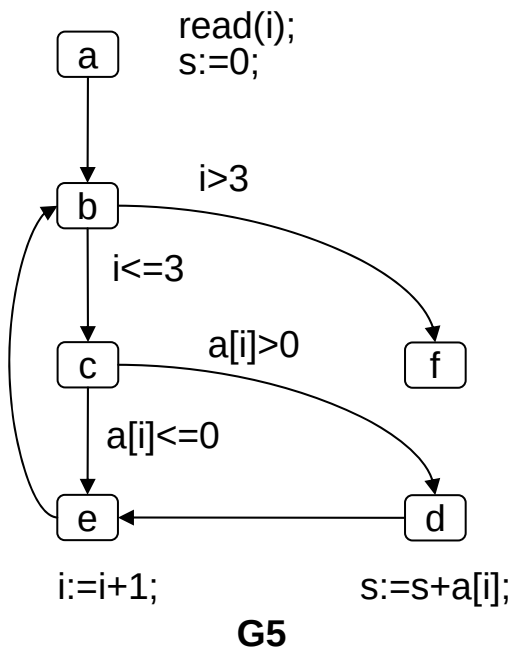
  DT1={a[1]=-2, a[2]=3, a[3]=17,i=1} sensibilise :

  M1=abcebcdebcdebf

  M1= abcebcdebcdebf couvre δ_1=cdebcde

  M1=abcebcdebcdebf couvre δ_2=ce

  Donc DT1 satisfait T1



G5

Hiérarchie des techniques de test structurel

- Exemple 2 : considérons le graphe de contrôle G5

  Soient δ_1=de et δ_2=b et δ_3=cd et T2= {δ_1, δ_2 , δ_3}.

  DT1={a[1]=-2, a[2]=3, a[3]=17,i=1} sensibilise :

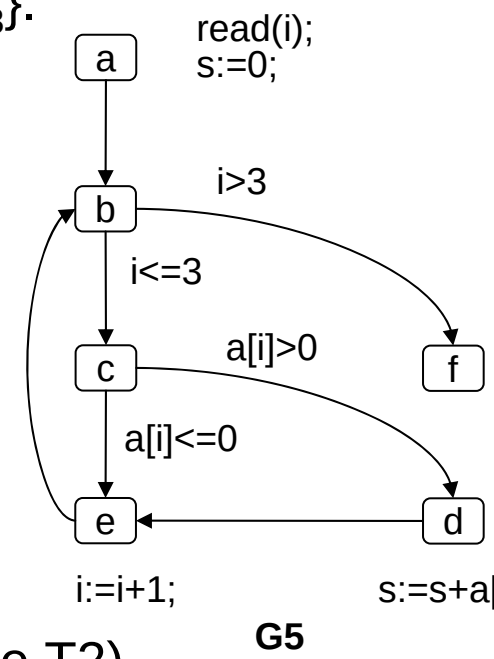
  M1=abcebcdebcdebf

  M1= abcebcdebcdebf couvre δ_1=de

  M1=abcebcdebcdebf couvre δ_2=b

  M1=abcebcdebcdebf couvre δ_3=cd

  Donc DT1 satisfait T2



G5

- Lorsque T1 est satisfait, T2 l'est aussi :

  T1 ⇒ T2 (T1 est un test plus **fiable** (ie. 'fort') que T2)

- T1 ⇒ T2 et T2 ⇒ T3 alors T1 ⇒ T3 (Transitivité)

- Hiérarchie entre les différentes techniques structurelles de test (relation d'ordre partielle)

Deux Catégories de critère de couverture

- Approche '**Flot de contrôle**' avec couverture de tous les arcs :

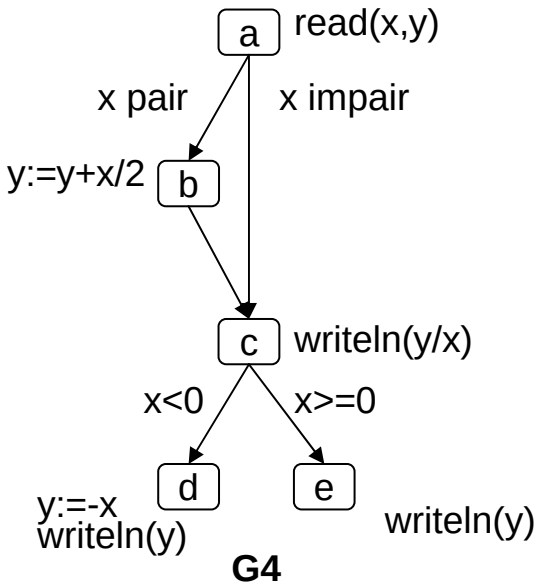
  DT1={x=-2, y=0} sensibilise le chemin M1=abcd

  DT2={x=1, y=0} sensibilise le chemin M1=ace

  - Si l'affectation du nœud b est erronée, cette erreur ne sera pas détectée par DT1 et DT2.

- Approche '**Flot de données**'

  L'affectation de y au nœud b n'est pas utilisée par DT1 et DT2 : il faudrait tester le chemin abce sensibilisé par la DT3={x=2, y=0}



Couverture sur le flot de contrôle

- Couverture de **tous-les-nœuds**

  But : sensibiliser tous les chemins de contrôle qui nous permettent de visiter tous les nœuds du graphe.

  Nom du critère de couverture : « tous les nœuds »

  Taux de couverture : TER1 (Test Effectiveness Ratio 1 ou C1)

  TER1 = |{nœuds couverts}| / |{nœuds}|

- Couverture de **tous-les-arcs**

  TER2 = |{arcs couverts}| / |{arcs}|

Exercices

1. Donner un graphe de contrôle G et une donnée de test DT montrant que le critère « tous les noeuds » est insuffisant pour détecter une erreur.
   - Calculer le taux de couverture TER1 associé au critère « tous les noeuds » pour votre DT.
   - Calculer le taux de couverture TER2 associé au critère « tous les arcs » pour votre DT.

2. Complétez le programme suivant qui calcule l'inverse de la somme des éléments, d'indice entre `inf` et `sup`, d'un tableau a contenant des entiers strictement positifs :

```
lire (inf,sup);
i:=inf;
sum:=0;
while (i<= sup)
do begin
   sum:=sum+a[i];
   …/…
```

   - Tester le programme avec DT1={a[1]=1; a[2]=2; a[3]=3;inf=1;sup=3}. Que se passe-t-il ?
   - Calculer TER1 et TER2.

Couverture sur le flot de contrôle (suite)

- Couverture de tous les **chemins indépendants**

  V(G) (le nombre de Mc Cabe ou nombre cyclomatique) donne le nombre de chemins indépendants.

  V(G)=#arcs - #noeuds + 2 [Si que des décisions binaires : V(G)=Nombre de nœuds de décision + 1]

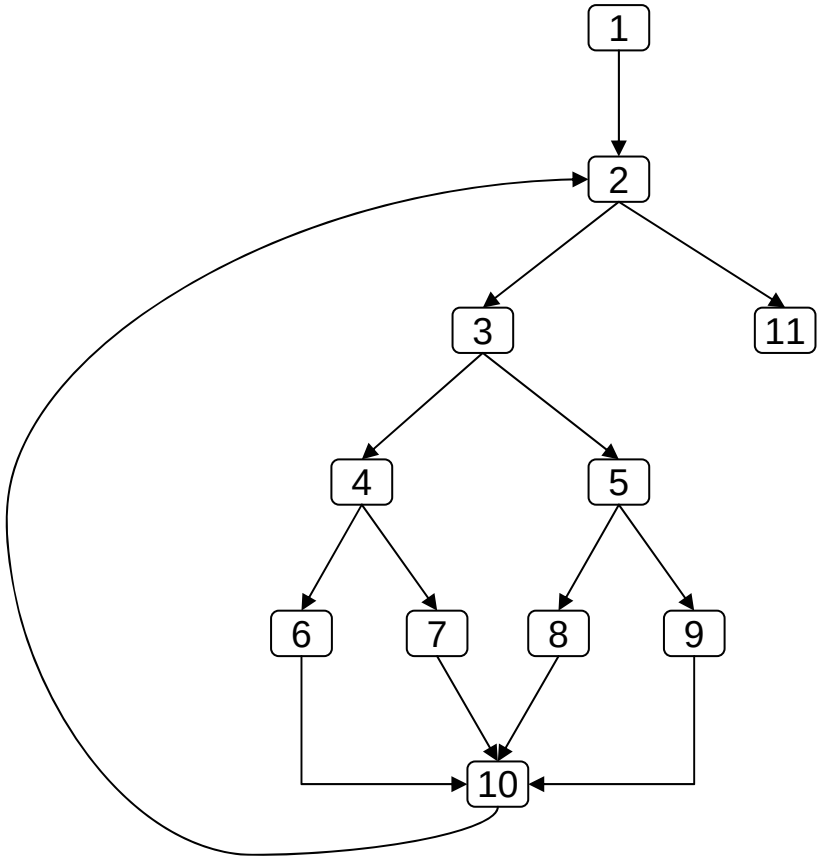
  Taux de couverture : Nbre de chemins indépendants couverts / V(G)

- Exercice : Donner le nombre Mc Cabe du programme suivant :

```
if C1
then        while (C2)
    Do      begin
            X1;
            end
else X2;
X3;
```

- Dans la pratique, la limite supérieure du nombre cyclomatique serait de 30… Au delà le test est difficile à réaliser…

- Rmq: Le ‘selon’ (switch) peut donner un nombre cyclomatique catastrophique avec une compréhension fort simple du code !!!


1
2
3
11
4
5
6
7
8
9
10

Couverture sur le flot de contrôle (suite)

- Couverture des **PLCS** (Portion Linéaire de Code Suivie d'un Saut ou linear code sequence and jump) :

PLCS : séquence d'instructions entre deux branchements.

Définition. Un **saut** est une arête (s,s') du graphe de contrôle tq :
- s est le sommet initial du graphe ou bien
- s' est le sommet terminal du graphe ou bien
- s est une condition et s' est le sommet atteint dans le cas ou s est évalué à faux ou bien
- s' est la condition d'une boucle et s le sommet terminal du corps de cette boucle

- Définition. Une **PLCS** est un couple (c,s) où c est un chemin $c=s_1s_2\ldots s_k$ tel que :
  - s_1 est le sommet d'arrivée d'un saut
  - $s_1s_2\ldots s_k$ est sans saut
  - (s_k,s) est un saut

- Le critère de complétude associé s'appelle TER3

  TER3=PLCS couvertes / #PLCS

- Hiérarchie : TER3=1⇒ TER2=1⇒ TER1=1

Exercices

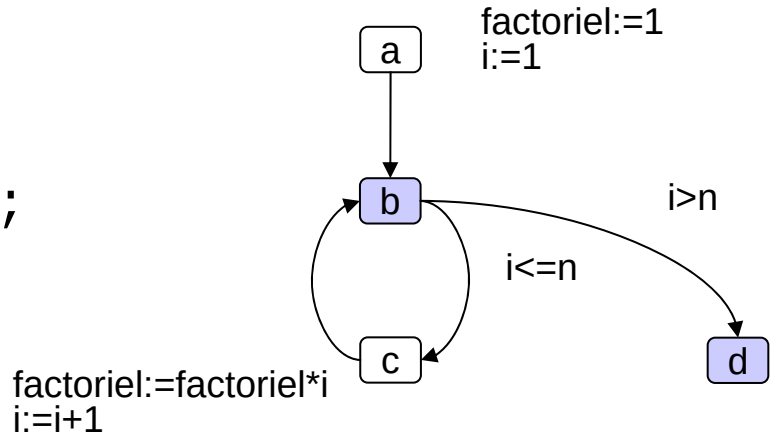
Donnez les PLCS du programme suivant:

```
main()
{
int i,factoriel;
factoriel=1;
for(i=1;i<=n;i++)
   {factoriel=factoriel*i;
   }
printf("%d\n",factoriel);
}
```

Exercice (correction)

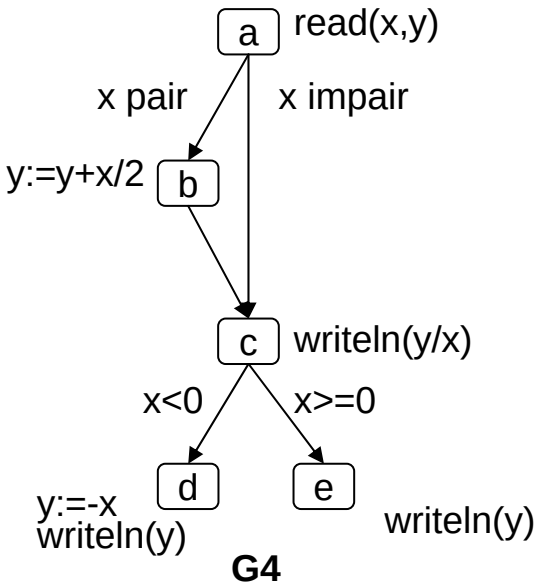
Donnez les PLCS du programme suivant:

```
main()
{
        int i,factoriel;
[a]     factoriel=1;
[b]     for(i=1;i<=n;i++)
[c]     {factoriel=factoriel*i;
        }
[d]     printf("%d\n",factoriel);
}
```



- Les sauts : (a,b),(b,d),(c,b)
- Sommets d'arrivée de saut : b,d
- Les PLCS : ([b],d), ([bc],b)

Couverture sur le flot de contrôle : pas assez fine !



G4

Couverture sur le flot de données

Analyse fine des relations entre instructions en tenant compte des variables qu'elles utilisent/définissent

- Couverture de toutes les définitions (DEF)

Chaque définition est exécutée au moins une fois par un chemin qui atteint une utilisation.

- Couverture de toutes les utilisations (C_REF et P_REF)

Chaque définition est exécutée au moins une fois pour toutes les utilisations qu'elle atteint et tous les arcs issus de ces utilisations sont couverts.

- Couverture de tous les P_REF

On se limite aux utilisations dans d'un prédicat

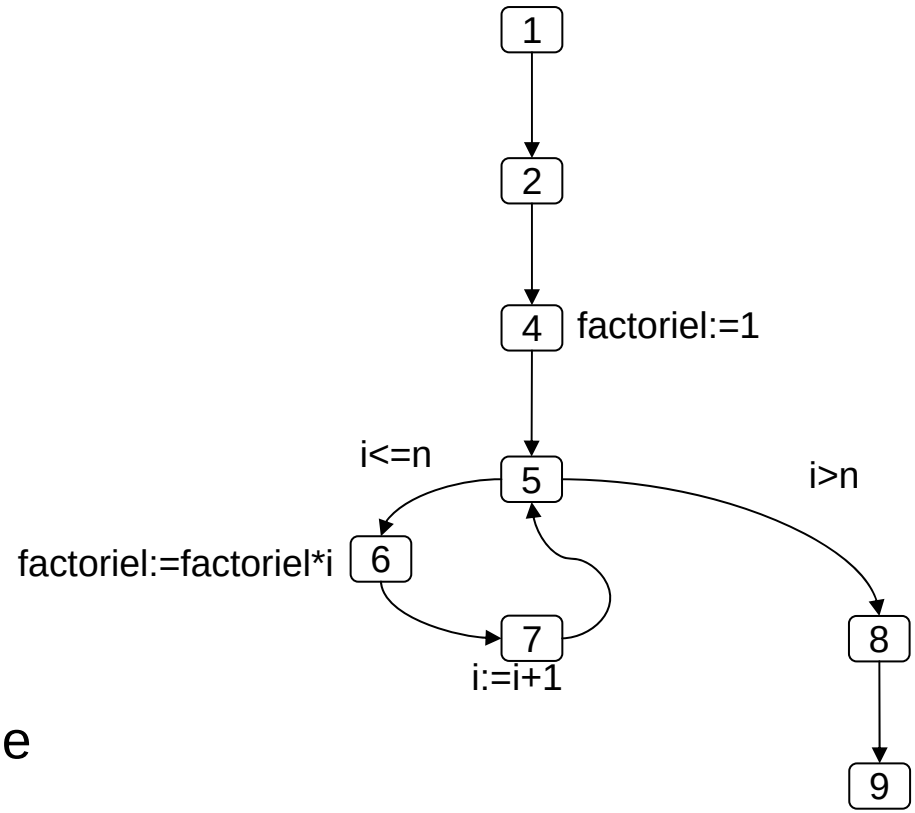
- Couverture de tous les DEF_REF chemins

Chaque définition est exécutée au moins une fois pour toutes les utilisations qu'elle atteint et tous les arcs issus de ces utilisations sont couverts. De plus chaque sous chemin entre une définition et une référence qu'elle atteint doit être exécuté.

Exercice

Soit le programme suivant:

```
1 main()
2 {
3 int i,factoriel;
4 factoriel=1;
5 for(i=1;i<=n;i++)
6   {factoriel=factoriel*i;
7   }
8 printf("%d\n",factoriel);
9 }
```



Question : donner des exemples de couverture de flot de données.

Partie 5:
Le test fonctionnel

Un exemple de test fonctionnel : le test de conformité de systèmes réactifs

Types d'application

Typologie du test

Théorie du test de conformité

Test d'interopérabilité

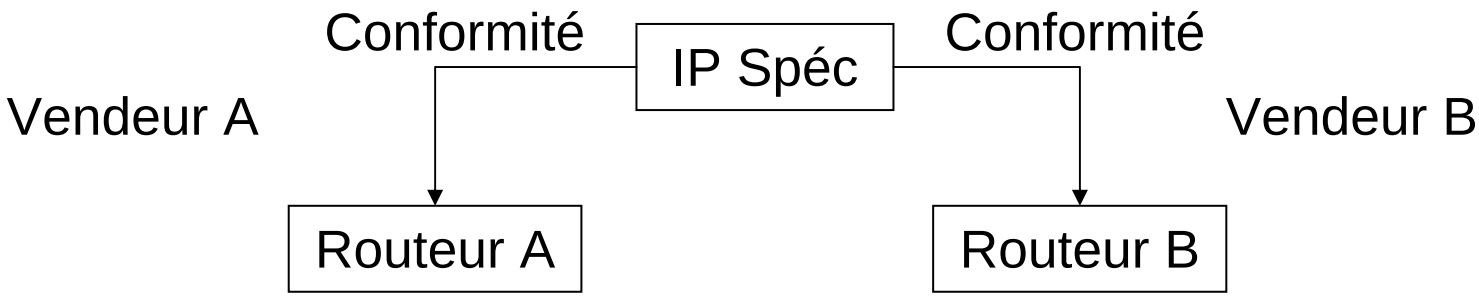
Types d'application

Application composée du plusieurs entités communicantes

- Système embarqué
- Système à base de composants
- Protocole
- Peut être temporisé
- …/…

Typologie du test

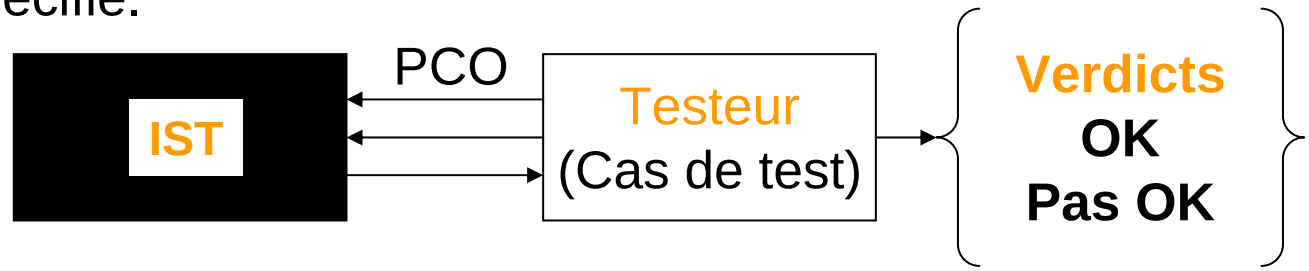
- **Test de Conformité**
  - Une seule entité est testée.
    Objectif : déterminer si une implémentation est conforme à sa spécification.
- **Test d'Interopérabilité**
  - Deux ou plusieurs entités sont testées.
    Objectif : déterminer si ces implémentations peuvent interagir ensemble comme prévu par la spécification.



Ces routeurs interagissent-ils correctement ?

Test de conformité

- But : être sûr qu'une implémentation fait ce que le standard spécifie.



IST (implémentation Sous Test) :

- Architecture boîte noire
- PCO (Points de contrôle et d'observation) : contrôle restreint et observation avec certaines interfaces

Testeur :

- Sortie : événements pour contrôler l'IST (cas de test),
- Entrée : observation de l'IST

Verdict :

- Résultat d'un cas de test (OK – Pas OK – Inconcluant)
- Un même cas de test peut conduire à différents verdicts

Test d'Interopérabilité

Test d'Interopérabilité:

- But final dans le développement d'un produit de systèmes communicants (exemple : routeur)
- Grand champ d'application : systèmes distribués (systèmes réactifs, systèmes embarqués, systèmes à base de composants, protocoles…)

Des implémentations peuvent être considérées conformes à leur spécification, mais peuvent ne pas interopérer.

Contrairement à la conformité, peu de travaux théoriques sur le test d'interopérabilité (définitions, méthodologies, algorithmes…)

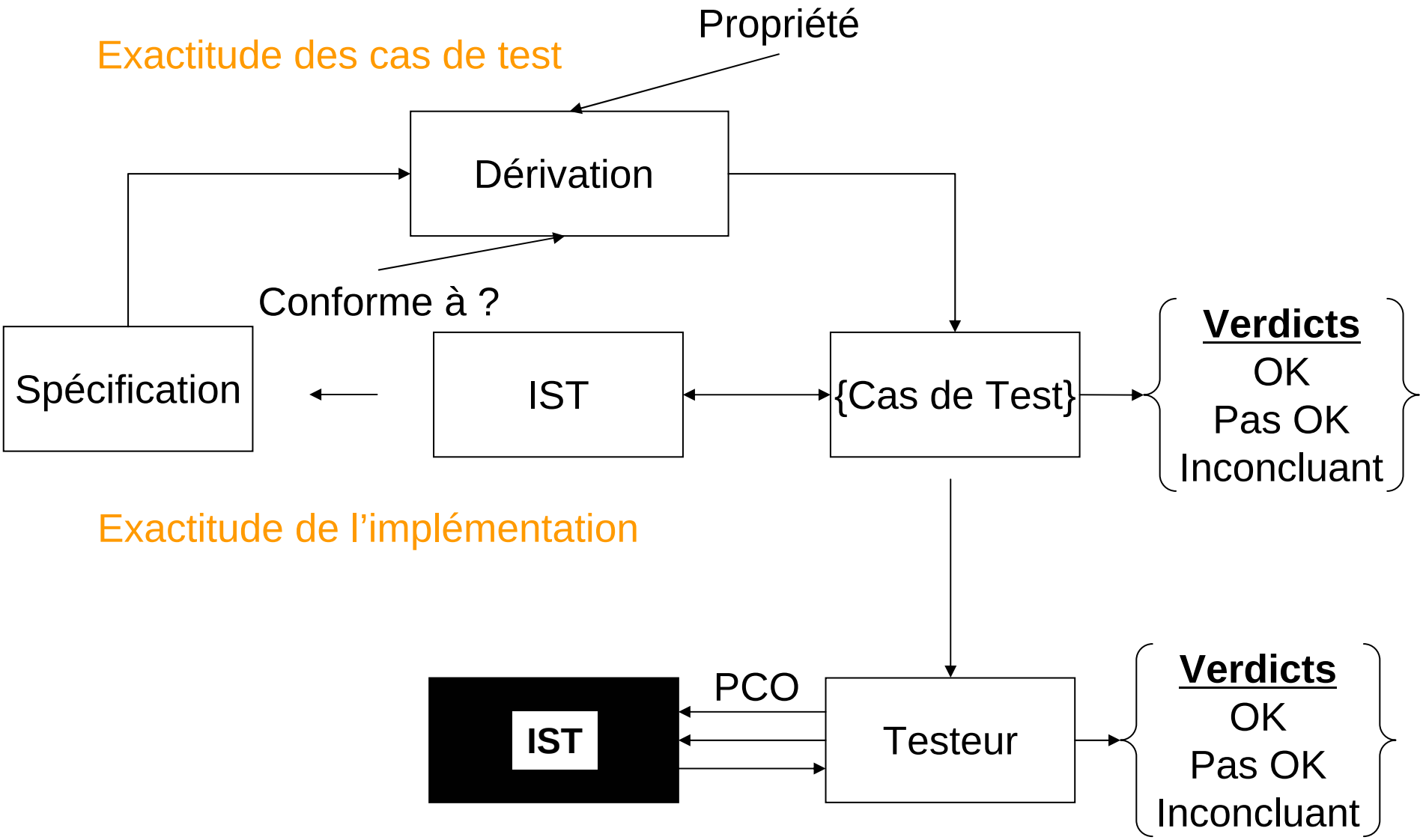
Pratique industrielle

En général : conception manuelle des cas de test à partir d'une spécification informelle

- processus long et répétitif
- coût important
- Aucune assurance sur le correction des cas de test
- Délicate maintenance des cas de test cases

⇨**La génération automatique des cas de test à partir de la spécifications formelle est très intéressante !**

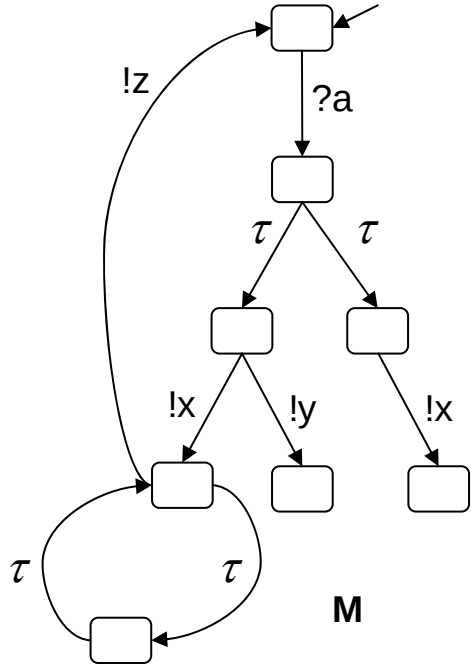
Théorie du test de conformité



Input-Output Labeled Transition Systems

IOLTS $M=(Q^M,\sum^M,\rightarrow^M,q_0{}^M)$, where

- Q^M a finite set of states with $q_0{}^M$ as the initial state,
- $\sum^M = \sum^M{}_I \cup \sum^M{}_o$ a set of observable events
  - $\sum^M{}_I$ a finite set of inputs (?a),
  - $\sum^M{}_o$ a finite set of outputs (!a),
- $\rightarrow_M \subseteq Q^M \times (\sum^M \cup \{\tau\}) \times Q^M$ *the transition relation*
  - τ an internal action (not in $\sum^M$)



Notation

M:IOLTS ; $q,q' \in Q^M$; $a \in \sum^M$; $\varepsilon,\sigma \in (\sum^M)^*$

$\Rightarrow$ describes visible behaviors of M

$q \Rightarrow^{\varepsilon} q'$ ≡ $q=q'$ or $q \rightarrow^{\tau^*} q'$

$q \Rightarrow^{a} q'$ ≡ $\exists q1,q2 \in Q^M \mid q \Rightarrow^{\varepsilon} q1 \rightarrow^{a} q2 \Rightarrow^{\varepsilon} q'$

$q \Rightarrow^{a\sigma} q'$ ≡ $\exists q1 \in Q^M \mid q \Rightarrow^{a} q1 \Rightarrow^{\sigma'} q'$

q after σ defines the set of states reachable from q by the visible sequence σ:

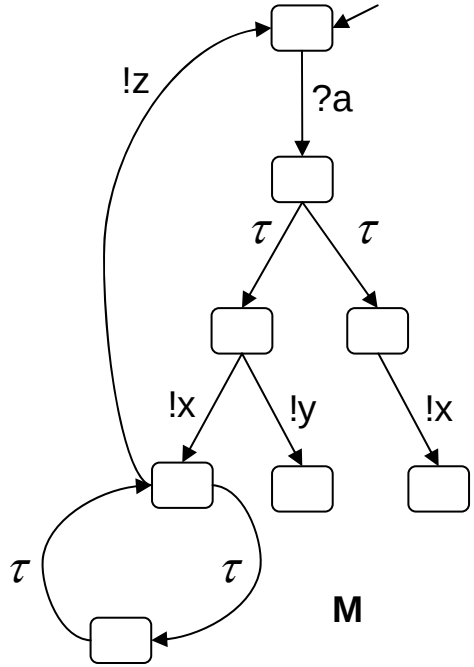
q after σ ≡ $\{q' \in Q^M \mid q \Rightarrow^{\sigma} q'\}$

$Out_M(q)$ is the set of the output actions in q:

$Out_M(q)$ ≡ $\{a \in \sum^M{}_o \mid \exists q' \in Q^M, q \rightarrow_M^a q'\}$

Traces(q) describes the sequence of visible actions firable from q

Traces(q) ≡ $\{\sigma \in (\sum^M)^* \mid \exists q' \in Q^M, q \Rightarrow^{\sigma} q'\}$



Hypothèses de Test

Modèle pour la Spécification : IOLTS $S = (Q^s, \sum^s, \rightarrow^s, q_0^s)$

Modèle pour l'Implémentation : IOLTS $IUT = (Q^{IUT}, \sum^{IUT}, \rightarrow^{IUT}, q_0^{IUT})$
avec $\sum^s{}_I \subseteq \sum^{IUT}{}_I$ et $\sum^s{}_o \subseteq \sum^{IUT}{}_o$ et input-complet.

Un IOLTS M est dit input-complet (input-enable) si M accepte toutes les entrées dans tous ses états. Formellement,

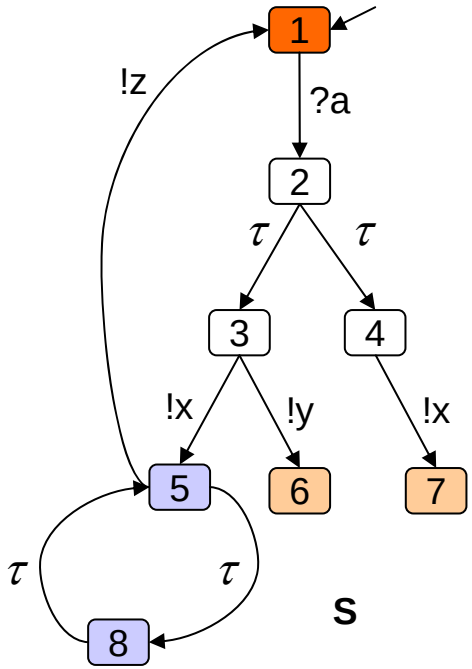
$$\forall q \in Q^M, \forall a \in \sum^M{}_i, \exists q' \in Q^M, q \Rightarrow^a q'$$

Comportement visible : traces + silence

Un testeur observe des traces de l'IST, mais aussi les silences.

– Silence: absence de comportement 'visible'

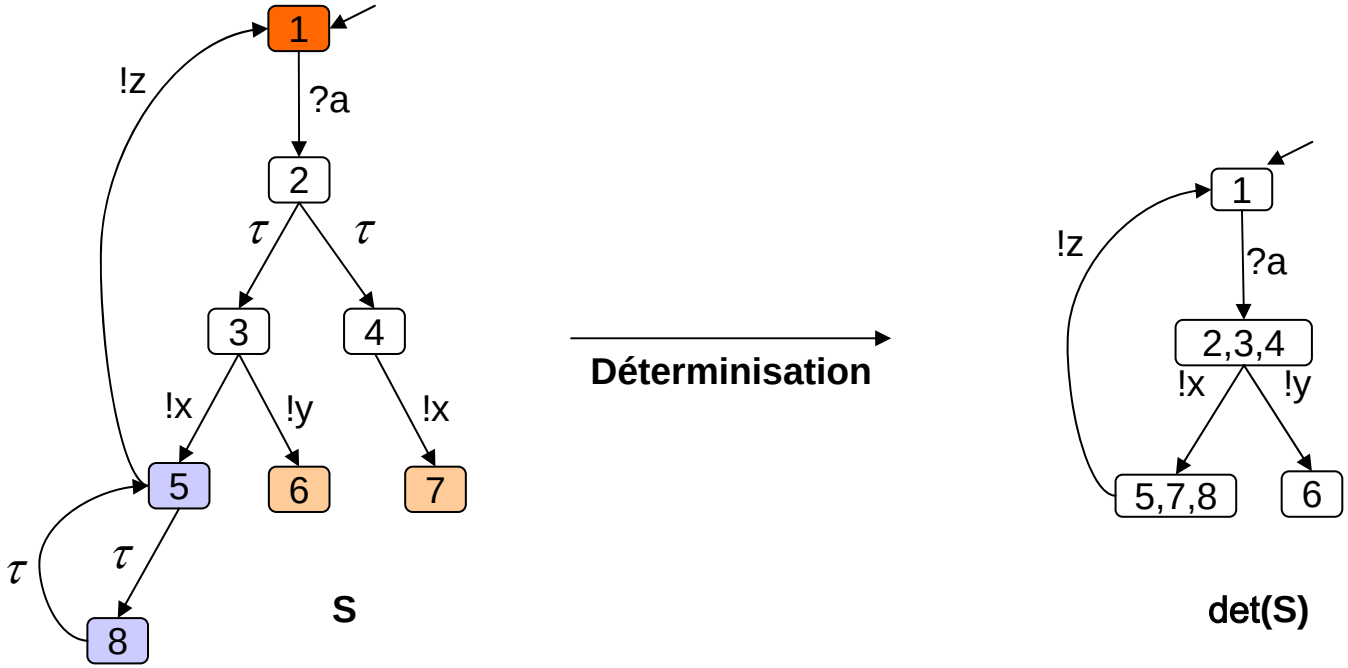
– 3 types de silence : deadlock (6,7), livelock (5,8) et output-lock (1).



Caractérisation des traces d'un IOLTS

Deux séquences avec la même trace ne peuvent être distinguées.

=> Nous considérons le IOLTS déterministe dét(S) qui a les mêmes traces que S.



Mais la déterminisation ne préserve pas les silences !

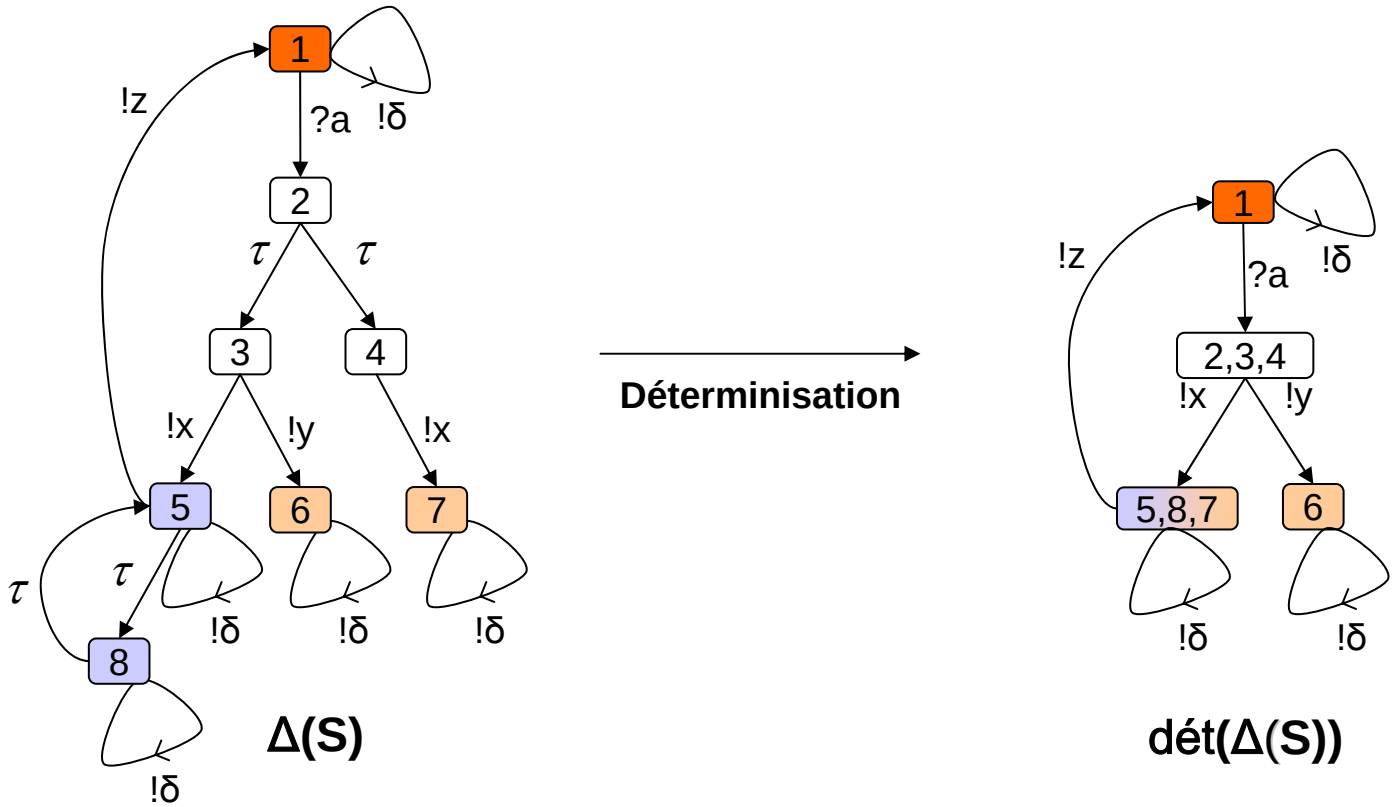
⇒ Une solution consiste à expliciter les silences [TRE96].

Automate de suspension

Tretmans96

Pour expliciter les silences, l'absence de comportement visible est modélisée par un événement de sortie !δ :

- Automate de suspension Δ(S) = S + boucle de !δ sur chaque 'état de silence'
- Traces Suspendues de S : STraces(S) = Traces(Δ(S)).
- dét(Δ(S)) caractérise les comportements visibles de S.

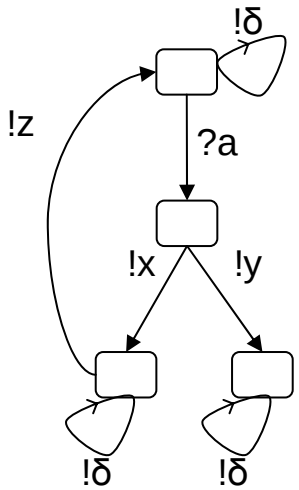


Relation de Conformité

Tretmans96

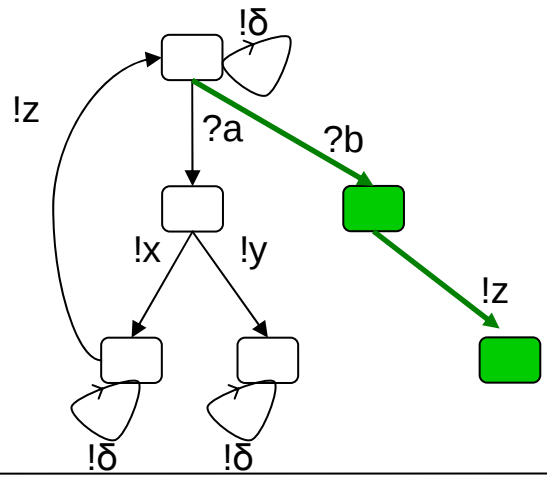
IUT ioco S ≡ ∀σ ∈ Traces(S) : out(Δ(IUT) after σ)⊆ out(Δ(S) after σ)

Pour tout comportement observable σ de S, une possible (observable) sortie de l'IST après σ est aussi une possible (observable) sortie de S après σ.



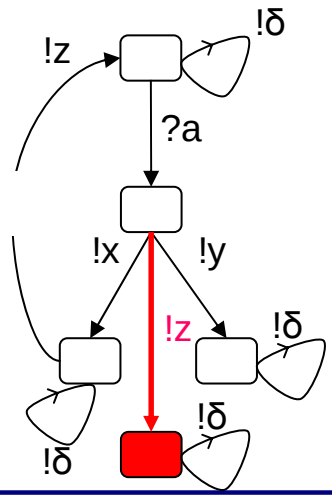
Dét(Δ(S))

IUT1 ioco S



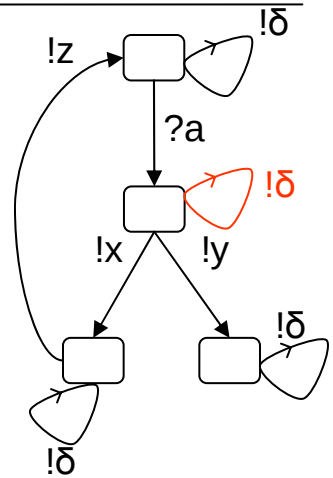
Δ(IUT1)

¬(IUT2 ioco S)



Δ(IUT2)

¬(IUT3 ioco S)

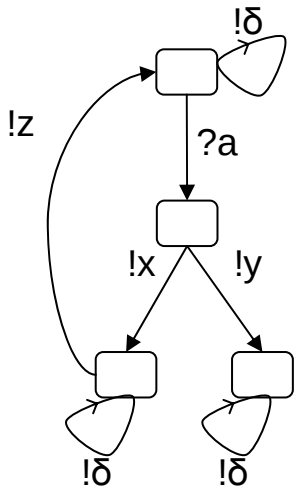


Δ(IUT3)

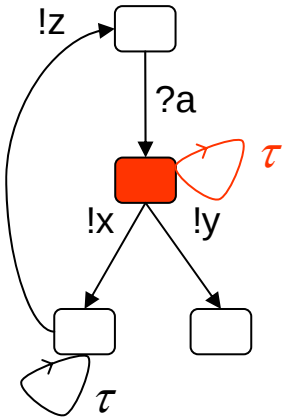
Relation de Conformité

IUT ioconf S ≡ ∀σ ∈ Traces(S) out(IUT after σ)⊆ out(S after σ)

IUT ioco S ≡ ∀σ ∈ STraces(S) out(Δ(IUT) after σ)⊆ out(Δ(S) after σ)



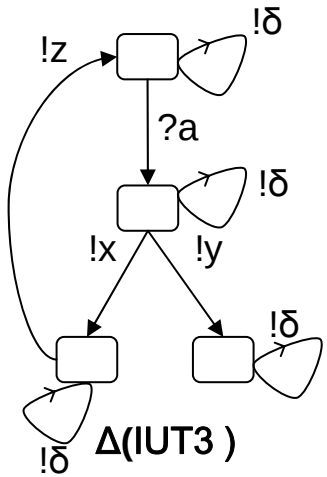
Det(Δ(S))



IUT3

IUT3 ioconf S

¬(IUT3 ioco S)



Δ(IUT3)

Cas de Test

- Un cas de test décrit les interactions entre Testeur et IST (Implémentation).
- Un testeur :
  - exécute un cas de test,
  - observe les réactions/comportements de l'IST,
  - les compare avec les comportements attendus du cas de test et
  - déduit le verdict correspondant.

Formellement, un cas de test TC est un *IOLTS acyclique avec une notion de verdicts :*

$TC = (Q^{TC}, \Sigma^{TC}, \rightarrow^{TC}, q_0^{TC})$ avec ${\Sigma^{TC}}_o \subseteq {\Sigma^{s}}_I$ et ${\Sigma^{TC}}_I \subseteq {\Sigma^{IUT}}_o$ U {?δ}

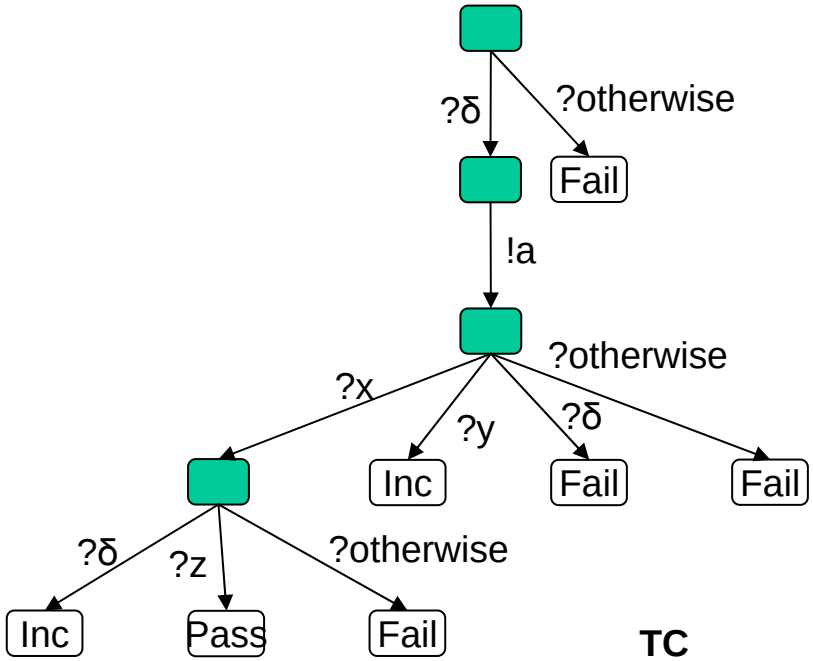
- $Q_{Pass} \subseteq Q^{TC}$ *and* $Q_{Fail} \subseteq Q^{TC}$ *and* $Q_{Inconc} \subseteq Q^{TC}$ (verdict states).

[!δ: silence visible / ?δ: expiration du temporisateur]

Cas de Test : un exemple

Hypothèses sur les cas de test :

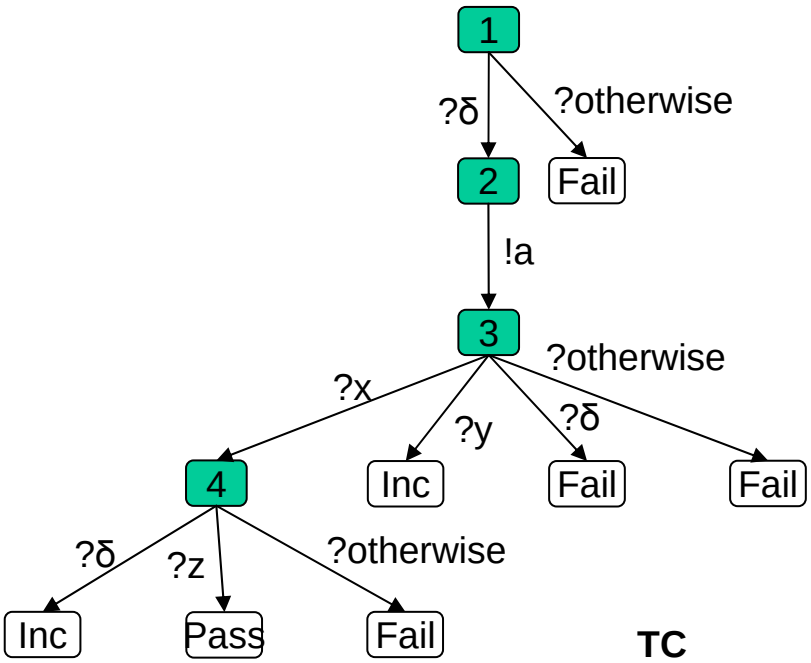
- contrôlable : pas de choix entre une sortie et une autre action,
- Input-complet dans tous les états où une entrée est possible (?otherwise)
- Les états de verdict sont uniquement atteints après des entrées.



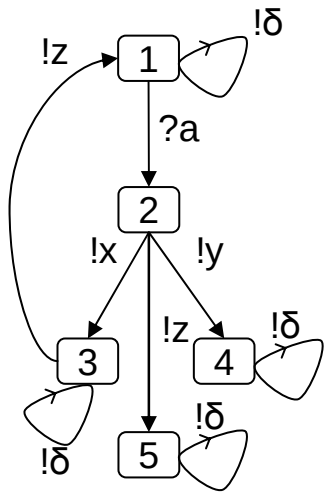
Exécution d'un cas de test

L'exécution d'un cas de test TC avec une implémentation IST est modélisée par la composition parallèle TC//Δ(IUT) avec une synchronisation sur les actions visibles communes (l'événement ?a synchronisé avec !a).

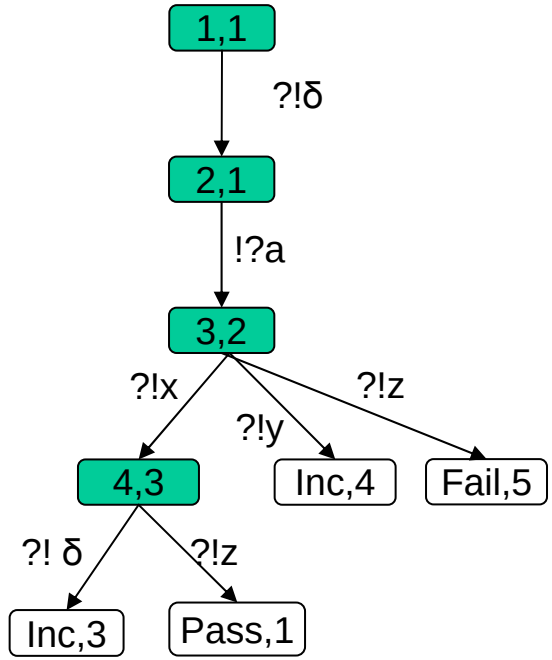
Exemple : Exécution de TC avec IUT2



TC

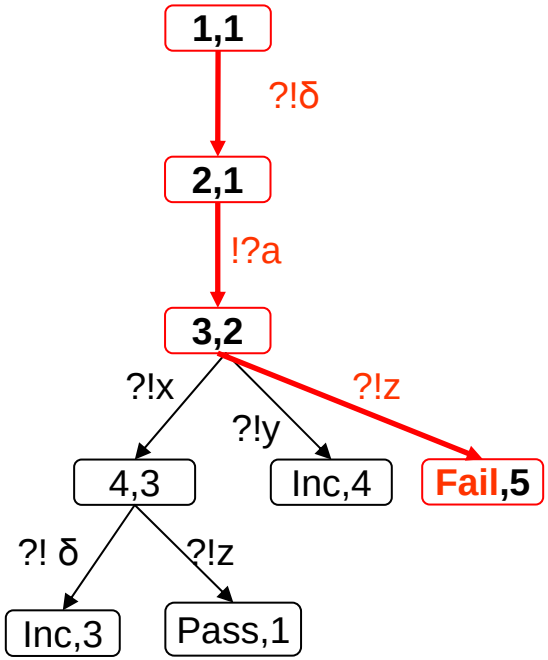


Δ(IUT2)



TC // Δ(IUT2)

Exécution d'un cas de test et verdict



TC//Δ(IUT2)

- TC//Δ(IUT) donne toutes les exécutions possibles.
- TC//Δ(IUT) s'arrête uniquement dans un état de : $(Q_{Pass} \cup Q_{Fail} \cup Q_{Inc}) \times Q^{IUT}$

Exécution : une trace maximale de *TC // Δ(IUT).*
Verdict : un état de TC atteint à la fin de l'exécution.

Formellement, nous définissons :

1. Une exécution σ d'un cas de test TC avec l'implémentation IUT est un élément de *MaxTraces(TC//Δ(IUT))*
2. verdict(σ) = fail (resp. *pass*, *inconc*) ≡ (TC after σ) $\subseteq Q_{Fail}$ (resp Pass, Inconc)

Avec :
MaxTraces(M) ≡ {σ∈(∑*M*)* | Γ(q0 after σ)=∅}
[Γ(q) est l'ensemble des actions tirables dans q]

Cas de Test et verdicts

Un cas de test peut produire case différents verdicts :

Exemple: *TC1* peut produire *Inc* ou *Pass*

Formellement, un rejet possible de l'implémentation IUT par le cas de test *TC* est défini par :

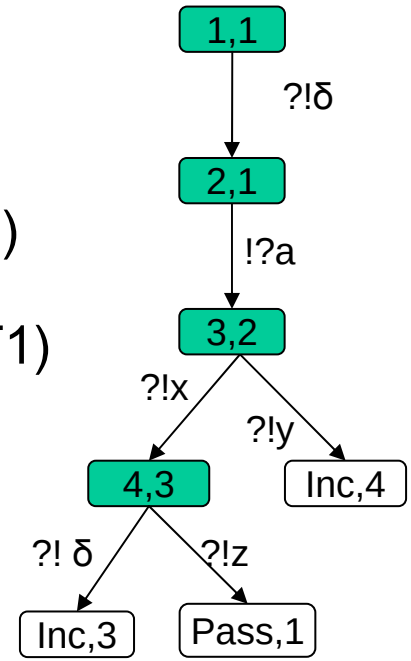
TC may reject IUT

≡

$\exists\ \sigma \in \text{MaxTraces}(TC // \Delta(IUT)), \text{verdict}(\sigma)=\text{fail}$

('may pass' et 'may inconc' sont définis de la même manière)

Exemple : (TC1 may pass IUT1) mais ¬(TC1 may fail IUT1)



TC1 // Δ(IUT1)

Propriétés attendues des cas de test

Non biaisé (Soundness) : un cas de test ne doit pas rejeter une implémentation conforme.

Formellement, une suite de test *TS* est non biaisée pour S et ioco ssi :

$\forall IUT, \forall TC \in TS, TC \text{ may reject } IUT \Rightarrow \neg(IUT \text{ ioco } S)$

Exhaustivité : une non-conforme implémentation devrait être rejetée par un acs de test.

Formellement, une suite de test *TS* est exhaustive pour *S* et *ioco* ssi :

$\forall IUT, \neg(IUT \text{ ioco } S) \Rightarrow \exists TC \in TS, TC \text{ may reject } IUT$

Problème:

Etant donnée une spécification *S*, une relation de conformité *R* (comme *ioco*), et une propriété *P*, comment **générer** des cas de test pour une implémentation de *S* vérifiant *P*.

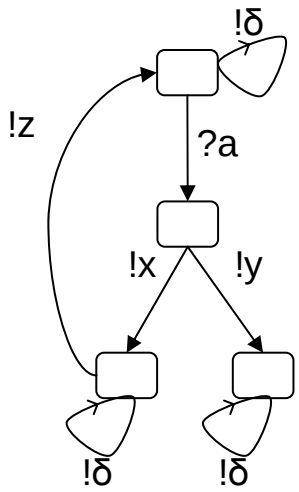
⇒ *TS=GenTest(S,R,P).*

Propriétés peuvent être : *non biaisé*, *exhaustif*, *critère de couverture (permet de définir une méthode de sélection), …/…*

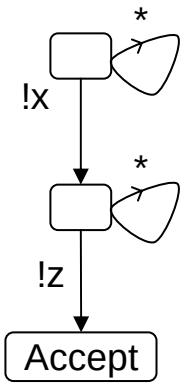
Un exemple de Génération

Spécification: *S*
Relation de Conformité: *ioco*
Propriété: Objectif de Test *OT*

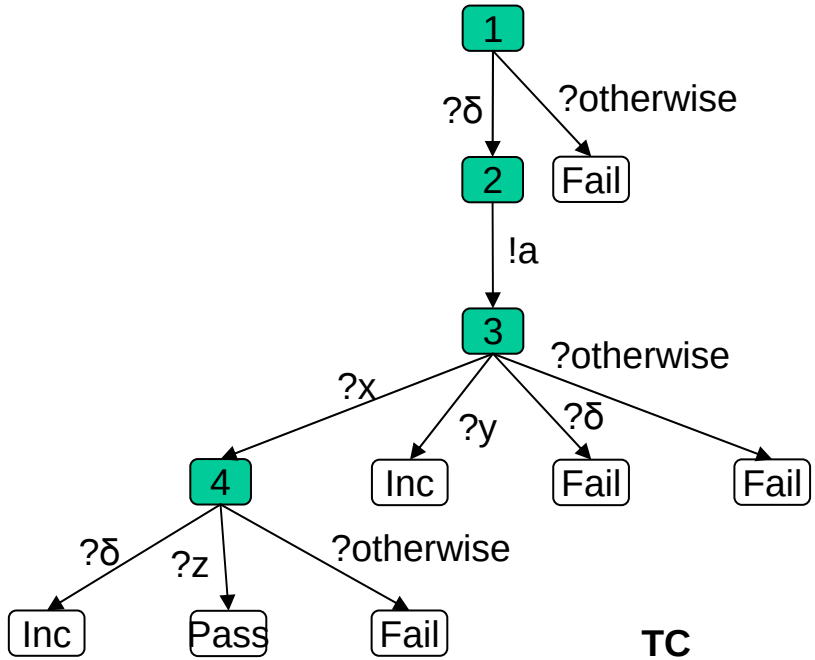
- Accessibilité de *Det(Δ(S))* : trace suspendue de *S* 'acceptée' par *OT*
- Autres algorithmes de génération



Det(Δ(S))



OT



TC

Théorie du test, en résumé

Modèles pour spécifier :

– Spécification *S*, implémentation *I*, cas de test *TC*, et suite de test *TS*

Formalisation de :

– l'exactitude du test avec une relation (*I ioco S* pour le test de conformité),

– l'exécution du test (par le testeur) et de sesverdicts (*verdicts(exec(I,TC))*, *I pass TS…)*

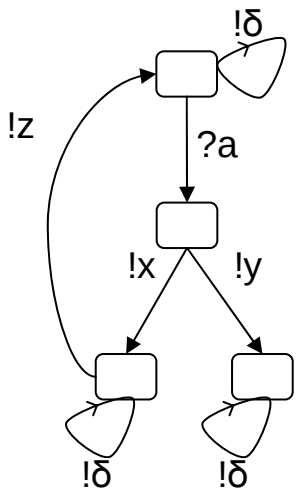
– Définition de propriétés attendues : *soundness*, *exhaustively*. *coverage*

⇨ *Définir des méthodes automatiques (algorithme) de dérivation de suite de test avec la bonne propriété : TC=gen_test(S).*

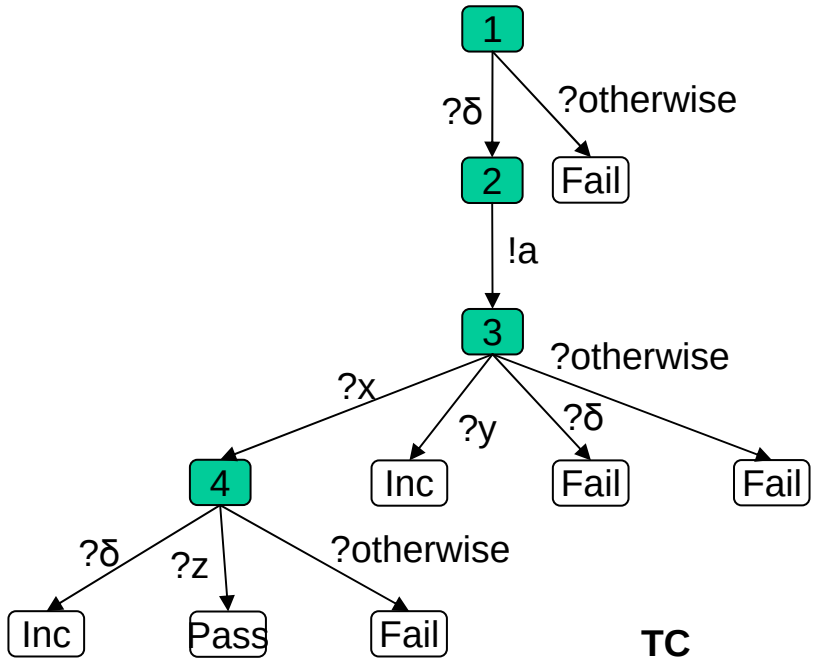
Quelques exercices…

Exercices

Ex1. En considérant la relation de conformité *ioco*, donner un cas de test *TC* pour la spécification *S* vérifiant la propriété *« Après la réception de a, j'envoie un x suivi d'un z »*.



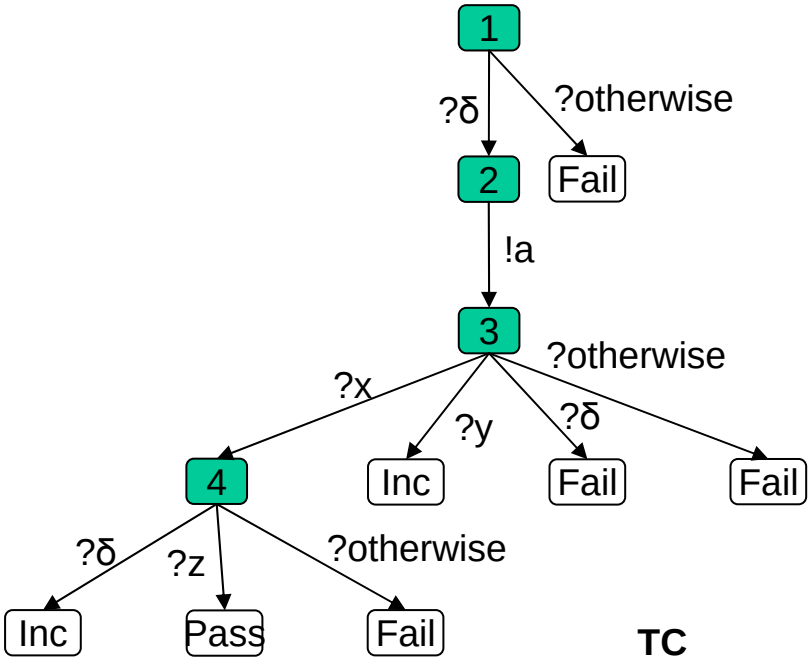
Dét(Δ(S))



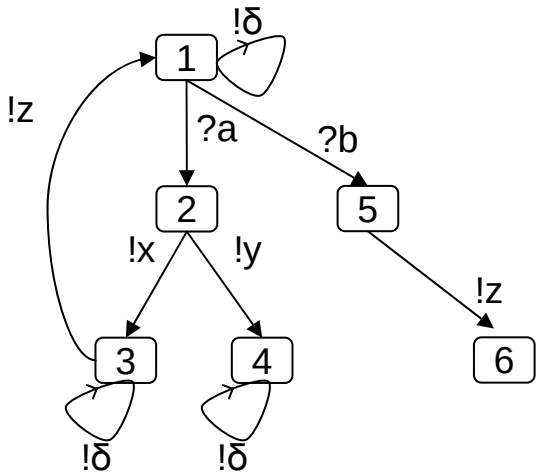
TC

Exercices (suite)

Ex2. Donner les exécutions possibles du cas de test *TC* avec *IUT1*



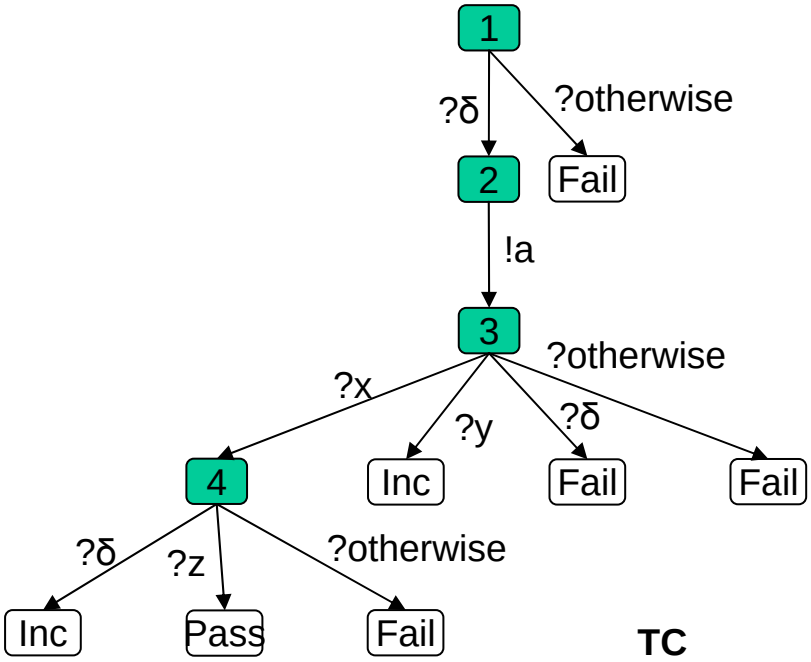
TC



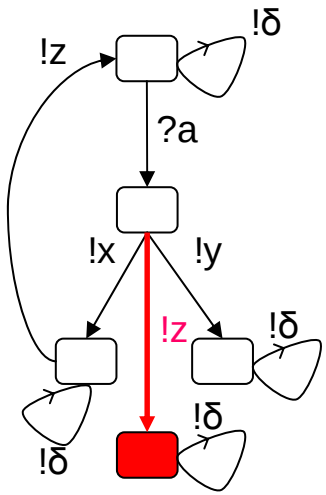
Δ(IUT1)

Exercices (suite)

Ex3. Donner les exécutions possibles du cas de test *TC* avec *IUT2.*



TC



Δ(IUT2)

Travaux Pratiques:
Test d'un contrôleur d'ascenseur

1. Introduction

On fait appel à vous pour tester un contrôleur d'ascenseur. Comme il s'agit du début du projet avec une équipe sans expérience avec ce genre de système, il est décidé de construire une simulation en Java afin de valider la conception du contrôleur. On vous livre une implantation du simulateur d'ascenseur en Java et on vous demande d'en effectuer le test.

Une spécification en langue naturelle du contrôleur d'ascenseur est fournie ci-dessous.

1. Comportement local
2. Comportement global

2. Spécification : les portes

Variables

`étage` : l'étage de la porte

Comportement

- Attendre que l'ascenseur soit à l'arrêt à l'étage de la porte
- Ouvrir la porte
- Attendre un certain laps de temps
- Fermer la porte
- Signaler à l'ascenseur qu'il peut redémarrer

2. Spécification : les usagers

Variables

`étage` : l'étage courant de l'usager
`direction` : la direction que l'usager veut emprunter
`destination` : la destination de l'usager

Comportement

- Si un appel a été signalé au même étage en direction opposée :
  - Attendre
  - Sinon, appeler l'ascenseur
- Attendre que la porte s'ouvre
- Décider d'entrer ou non (l'usager peut être distrait)
- Si la porte est encore ouverte, entrer dans l'ascenseur
- Entrer la destination
- Attendre que la porte se ferme
- Attendre que l'ascenseur soit à destination
- Attendre que la porte s'ouvre et sortir

2. Spécification : l'ascenseur

Variables

- `étage` : l'étage courant de l'ascenseur
- `direction` : la direction courante de l'ascenseur
- `destinations` : un vecteur des destinations entrées par les usagers
- `appels` : un vecteur des appels effectués par les usagers

Comportement

- Choisir la direction
- Monter ou descendre d'un étage selon la direction
- Renverser la direction si l'ascenseur atteint l'étage le plus haut (resp. le plus bas)
- Si le nouvel étage correspond à un appel ou une destination
- Effacer tout appel ou destination pour l'étage courant
- Signaler d'ouvrir la porte
- Attendre la fermeture de la porte

2. Spécification : l'ascenseur (suite)

Comportement

Choisir la direction :

- Selon la direction courante, chercher un appel ou une destination vers le haut ou le bas
  - S'il n'y a aucune direction courante, commencer à chercher vers le haut
- S'il existe un appel à l'étage courant, indiquer qu'il n'y a pas de direction courante
- S'il existe un appel ou une destination vers la direction courante et que l'ascenseur n'est pas à l'étage le plus haut (resp. le plus bas)
  - Maintenir la direction courante
  - Sinon, chercher pour un appel ou une destination dans la direction opposée
- S'il existe un appel ou une destination dans la direction opposée et que l'ascenseur n'est pas à l'étage le plus bas (resp. le plus haut)
  - Changer la direction à la direction opposée
  - Sinon, indiquer qu'il n'y a pas de direction courante

Comportement global

- Lorsque l'ascenseur est en mouvement, aucune porte n'est ouverte.
- Il est toujours vrai qu'un usager qui demande l'ascenseur y entrera fatalement
- Il n'y a jamais plus d'une porte d'ouverte à la fois.
- La distance parcourue par un usager est toujours égale à |source − destination|.

2. Spécification : Simplification

Exemple de simplification :

- Un seul ascenseur,
- Deux usagers au même étage demandent l'ascenseur : si leurs directions sont différentes, un usager attend.

Exemple de trace d'exécution

0. # Usager[0]: # effectue l'appel 1-UP
1. # Usager[1]: # effectue l'appel 2-DOWN
2. + Ascenseur: + direction: UP
3. + Ascenseur: + Etage: 1
4. + Ascenseur: + arrêt a l'étage 1
5. * Porte[1]: * ouverture
6. # Usager[0]: # entre ds l'ascenseur
7. # Usager[0]: # entre la destination 2
8. * Porte[1]: * fermeture
9. + Ascenseur: + fin de l'arrêt
10. + Ascenseur: + direction: UP
11. + Ascenseur: + Etage: 2
12. + Ascenseur: + arrêt à l'étage 2
13. * Porte[2]: * ouverture
14. # Usager[0]: # destination atteinte
15. # Usager[1]: # entre ds l'ascenseur
16. # Usager[1]: # entre la destination 0
17. * Porte[2]: * fermeture
18. + Ascenseur: + fin de l'arrêt
19. + Ascenseur: + direction: DOWN
20. + Ascenseur: + Etage: 1
21. + Ascenseur: + direction: DOWN
22. + Ascenseur: + Etage: 0
23. + Ascenseur: + arrêt à l'étage 0
24. * Porte[0]: * ouverture
25. # Usager[1]: # destination atteinte

Dossier à remettre

La date de remise de votre dossier est fixée au 12 juin 2006 : votre dossier peut être soit envoyé par mail (felix@labri.fr), soit rendu lors du cours le 12 juin à 18h30.

Ce dossier comprendra :

- Le code source de l'application augmenté du code que vous avez écrit pour effectuer vos tests. Le code écrit fera l'objet de commentaires clairs et précis qui seront très appréciés du correcteur…
- Un rapport avec :
  - une modélisation UML de l'application (facultatif),
  - une description des différents tests effectués,
  - une évaluation de la couverture de vos tests,
  - un mode d'emploi succinct et précis permettant au correcteur de passer l'ensemble de vos tests,
  - une synthèse de votre campagne de test (sous forme de tableau, graphique, etc.),
- Dans le cas de découverte d'erreur, on vous demande de présenter dans votre rapport une analyse de l'erreur (le préambule pour lequel l'erreur est apparue, la séquence d'événements dans la quelle l'erreur a été trouvée, etc.).