Travaux Pratiques:

Ascenseur

On vous demande de construire un simulateur d'ascenseur (en Java ou dans un autre langage de votre choix).

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

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

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

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

Spécification : le système

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|.

Spécification : Simplification

Exemple de simplification :

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

Partie I : Le test.

1: Introduction au test de logiciels

Génie Logiciel

Toute fabrication de produit suit les étapes suivantes :

1-Conception 2-Réalisation 3-Test

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 : toute une panoplie de langages de programmation, méthodes de programmation, concepts, outils, méthodes, technologies, normes, etc. [+Constante évolution !]

Exemples : C, ADA, C++, Java, C#, POO, programmation événementielle Corba, .NET, architecture 3-tier/n-tier, XML, webservice, Ajax, etc.

Génie logiciel : domaine 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** => **défaut** => **anomalie**

(nature de l'erreur :spécification, conception, programmation…)

Le terme **bogue** est malheureusement utilisé pour désigner aussi bien défaut qu'une anomalie.

défaut ≠ **anomalie**

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…

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.

Le test : des définitions…(suite et fin)

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".

AFCIQ : Association Française pour le Contrôle Industriel et la Qualité

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 sur les définitions du test

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. (Le test du logiciel est également appelé vérification dynamique.)

Difficultés du test

1. Processus d'introduction des défauts très complexe
2. Mal perçu par les informaticiens et délaissé par les théoriciens

Difficultés du test : Testabilité

Testabilité : Facilité avec laquelle les tests peuvent être développés à partir des documents de conception

Facteurs de bonne testabilité :

- Précision, complétude, traçabilité des documents
- Architecture simple et modulaire
- Politique de traitements des erreurs clairement définie

Facteurs de mauvaise testabilité :

- Fortes contraintes d'efficacité (espace mémoire, temps)
- Architecture mal définie

Difficultés du test : Limites théoriques

1-Indécidabilité : une propriété indécidable est une propriété qu'on ne pourra jamais prouver dans le cas général (pas de procédé systématique)

Exemples de propriétés indécidables :

- L'exécution d'un programme termine
- Deux programmes calculent la même chose
- Un programme n'a pas d'erreurs

2-Explosion combinatoire : un programme a un nombre infini (ou extrêmement grand !) d'exécutions possibles

Le test n'examine qu'un nombre fini (ou très petit) d'exécutions

Heuristiques : approcher l'infini (ou l'extrêmement grand) avec le fini (très petit).

=> Choisir les exécutions à tester !

Difficultés du test : conclusion.

Conclusion : Impossibilité d'une automatisation complète satisfaisante !

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 trop souvent artisanale.

Demain, le test de logiciel devrait être :

- une activité rigoureuse,
- fondée sur des modèles et des théories
- De plus en plus « 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}

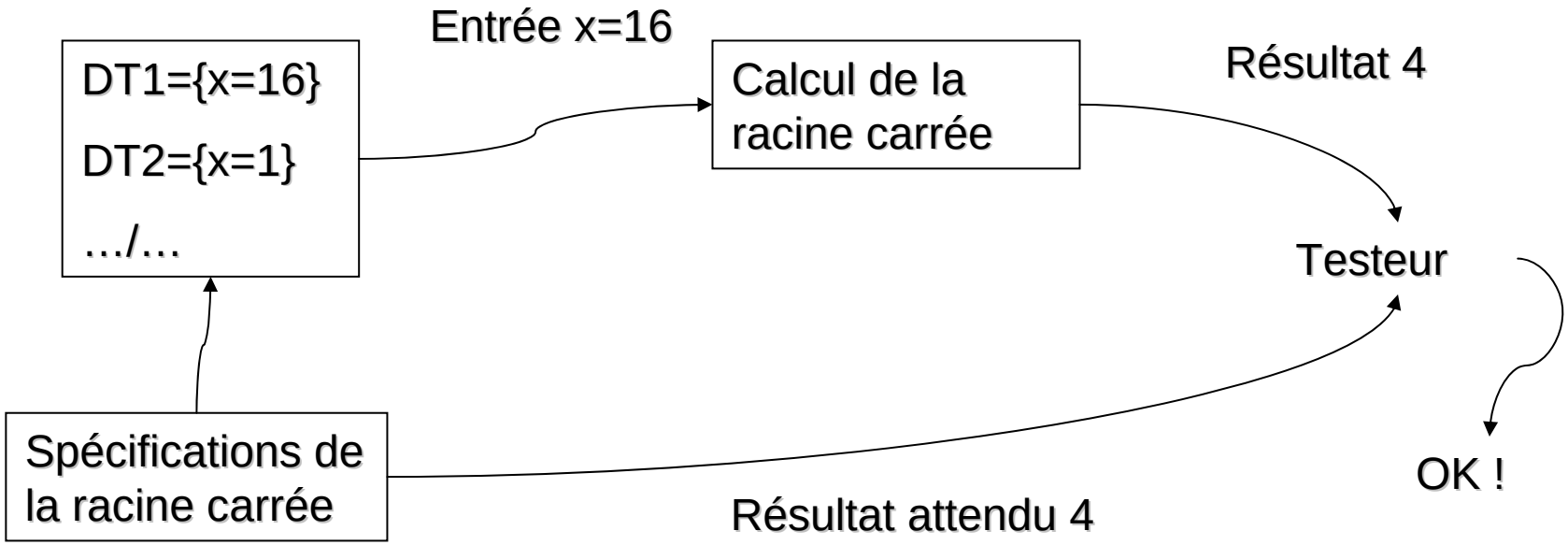
Jeu de test : est un ensemble de données de test.

Scénario de test : actions à effectuer avant de soumettre le jeu de test

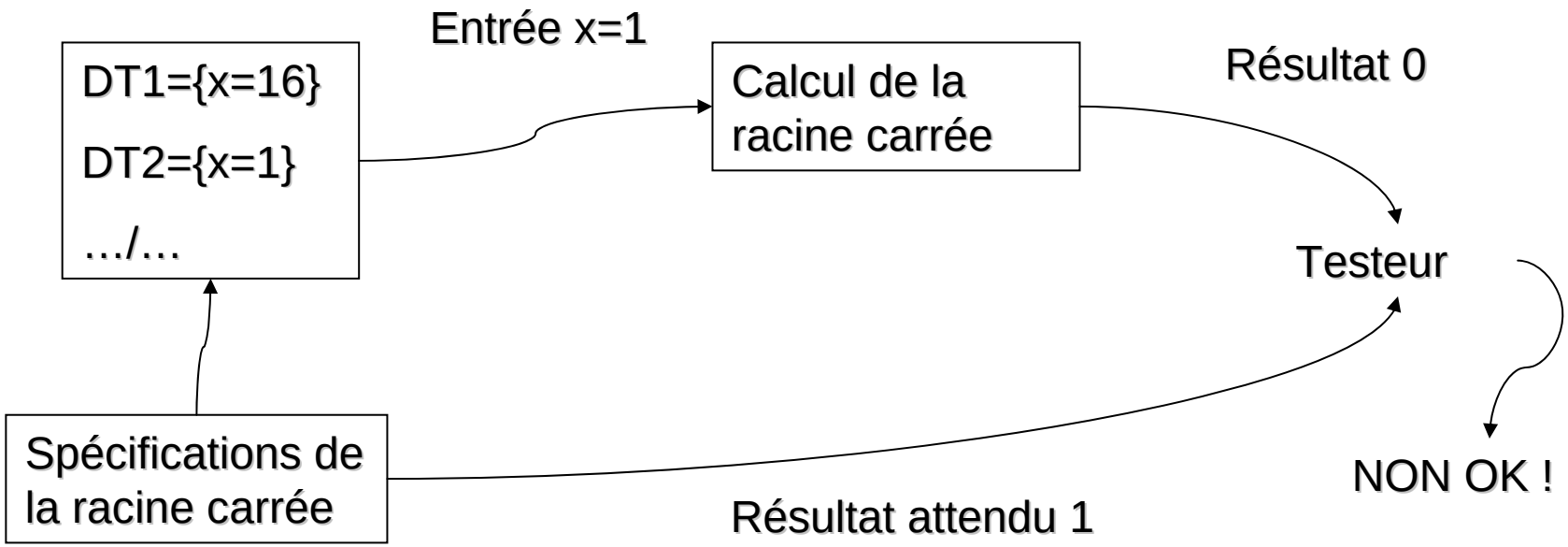
Le scénario de test produit un **résultat**

Ce résultat doit être évalué de manière manuelle ou automatique pour produire un **oracle**

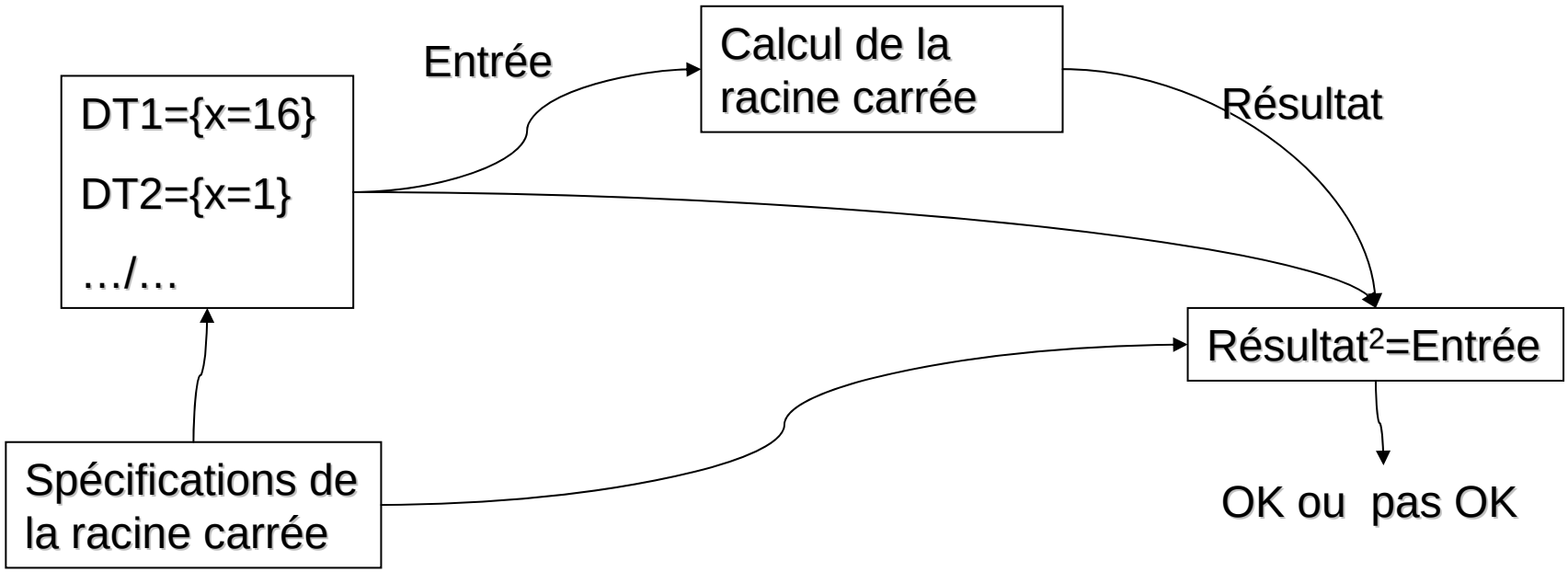
Exemple 1 de test avec oracle manuel



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

⇒ **Critère de test** (ou de sélection) : Un critère permet de spécifier formellement un objectif (informel) de test. Un critère de test peut, par exemple, indiquer le parcours de toutes les branches d'un programme, ou l'examen de certains sous-domaines d'une opération.

- Validité
- Fiabilité

Validité, fiabilité, complétude d'un critère de test

Validité : Un critère de test est dit valide si pour tout programme incorrect, il existe un jeu de test non réussi satisfaisant le critère.

- P:programme, F:spécification,

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

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

Fiabilité : Un critère est dit fiable s'il produit uniquement des jeux de test réussis ou des jeux de test non réussis.

Complétude : Un critère est dit complet pour un programme s'il produit uniquement des jeux de test qui suffisent à déterminer la correction du programme (pour lequel tout programme passant le jeu de test avec succès est correct)

Remarque : Tout critère valide et fiable est complet.

La complétude : un rêve…

Hypothèse de test : La complétude étant hors d'atteinte en général, on peut qualifier un jeu de test par des hypothèses de test qui caractérisent les propriétés qu'un programme doit satisfaire pour que la réussite du test entraîne sa correction

Classification des tests

Différentes classes de tests selon :

- les critères de test utilisées
- Les entités utilisées (spécification, code source, exécutable…)

Exemples de classes :

1. Les modalités de test : **Statique** / **Dynamique**
2. Les méthodes de test : **Structurelle** / **Fonctionnelle**
3. **Manuel** / **Automatique**
4. Les niveaux de tests : **Unitaire** / **Intégration** / **Système** / **Non-régression**
5. Les caractéristiques de test : **Robustesse** / **Conformité** / **Performance** /…

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
   - **Implantation** du système (IUT = Implementation Under Test)
   - Une propriété / caractéristique à tester
   - un test réussit (Passes) si les résultats obtenus sont les résultats attendus, sinon il échoue (Fails);

Les niveaux de tests

Tests **unitaires** (ou test de composant): 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

1. Les méthodes **structurelles** : repose sur des analyses du code source
   - Examen de la structure du programme (flot de contrôle ou de données)
   - 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** : repose sur une spécification (formelle ou informelle) du programme, le code source du programme n'est pas utilisé.
   - Aucune connaissance de l'implantation;
   - Aussi appelées test en boîte noire, ou test basé sur la spécification.
   - permet d'écrire les tests avant le codage;

Parfois : Combinaison des deux méthodes fonctionnelles et structurelles.

3. Les tests **orientés-erreurs** :
   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:
     - À l'aide d'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,

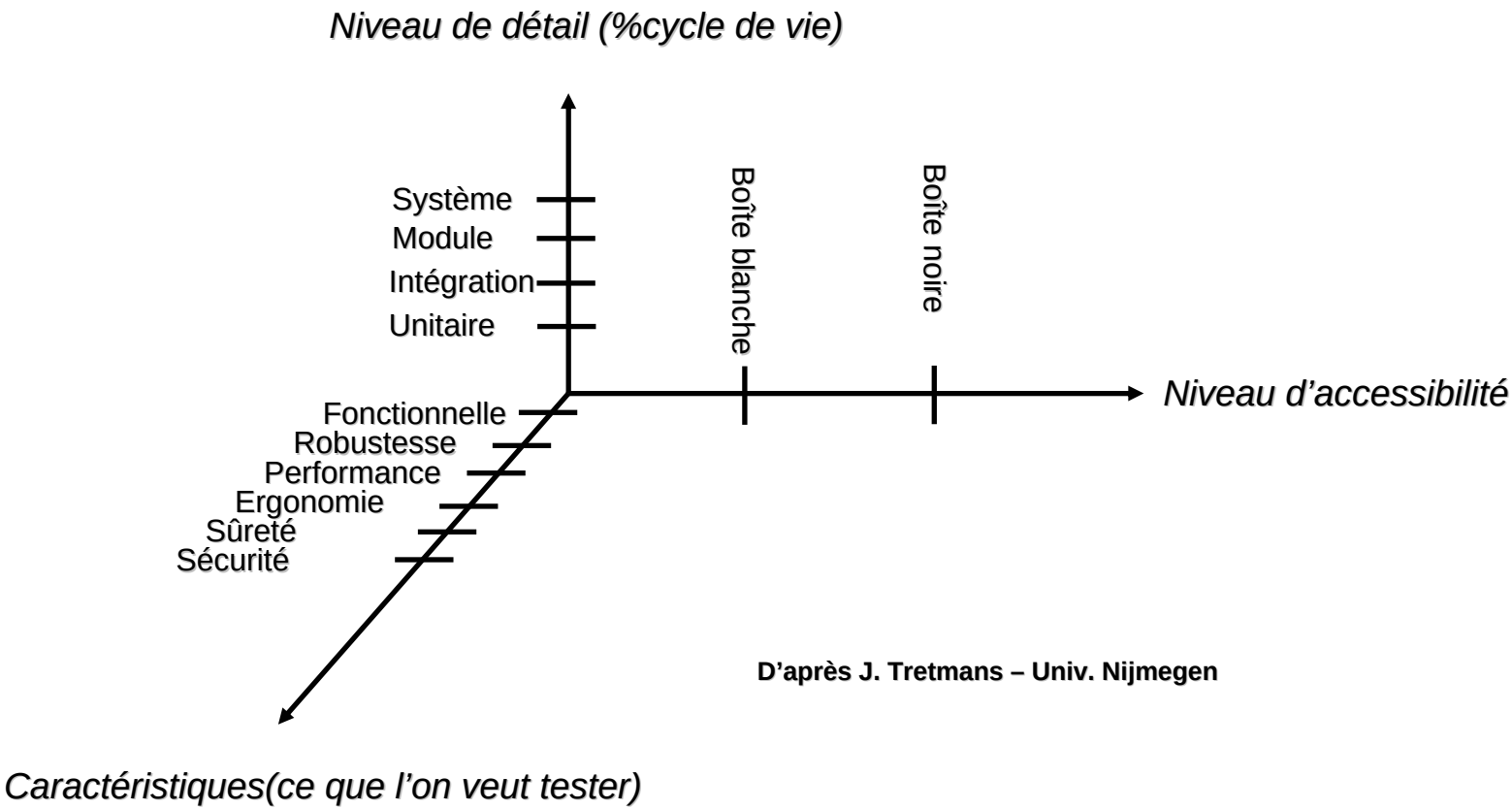
Niveau d'accessibilité

- 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 I : Le test.

2: Le test fonctionnel

1.2: Le test fonctionnel
1.2.1: Le test de conformité de systèmes réactifs

Types d'application

Typologie du test

Théorie du test de conformité

Test d'interopérabilité

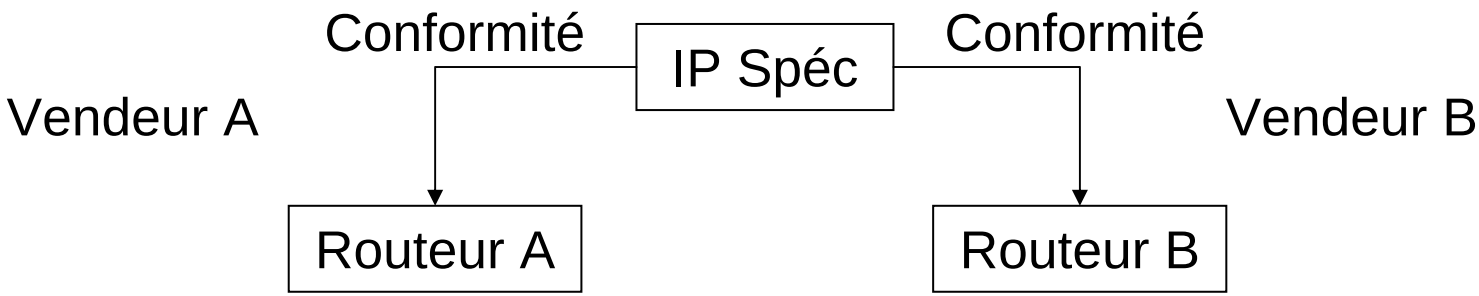
Types d'application

Application composée du plusieurs entités communicantes

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

Typologie du test

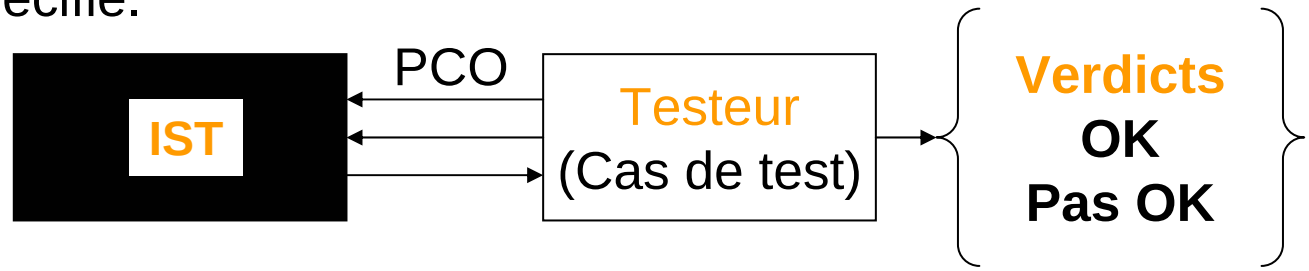
- **Test de Conformité**
  - Une seule entité est testée.
    Objectif : déterminer si une implémentation est conforme à sa spécification.
- **Test d'Interopérabilité**
  - Deux ou plusieurs entités sont testées.
    Objectif : déterminer si ces implémentations peuvent interagir ensemble comme prévu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

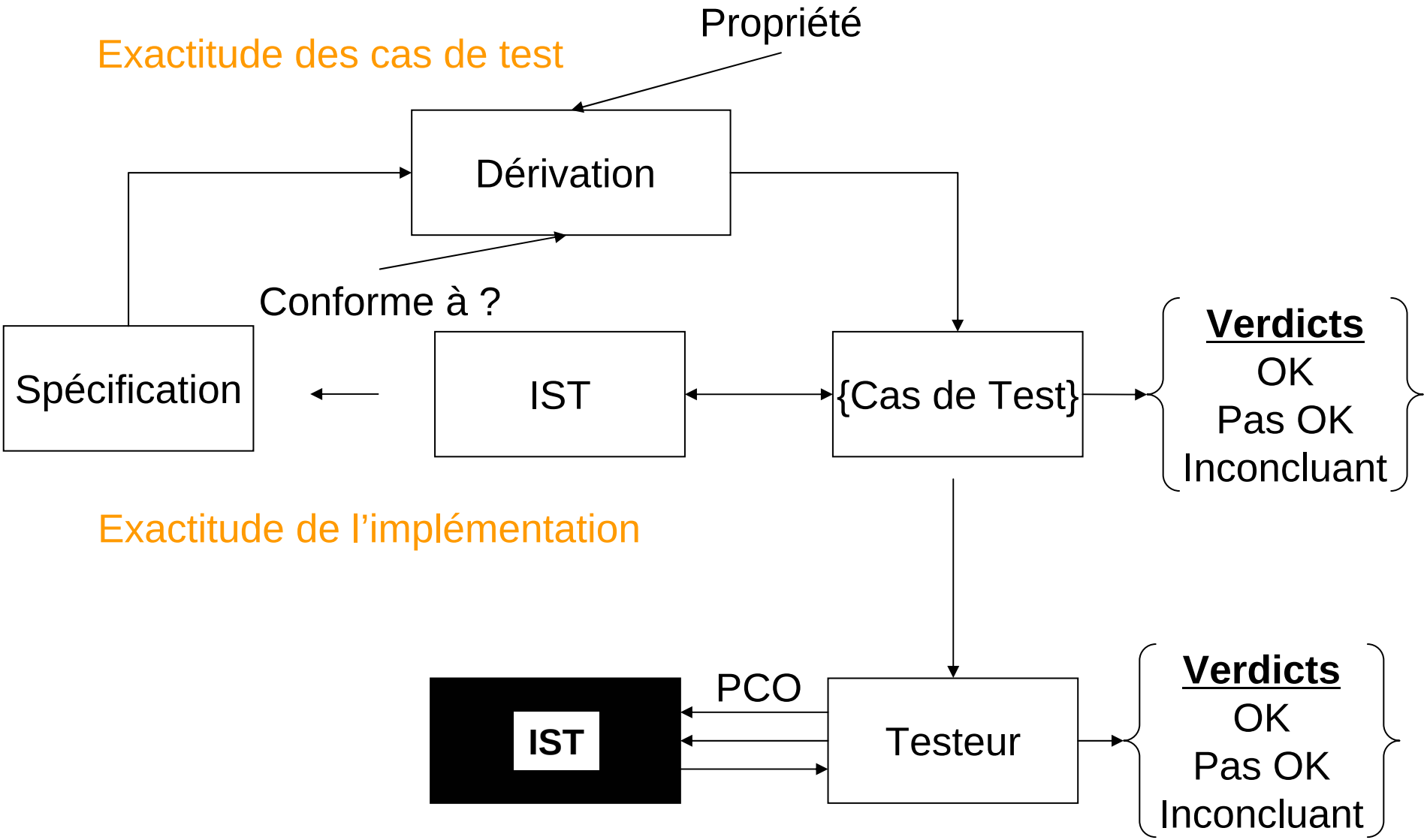
Pratique industrielle

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 !**

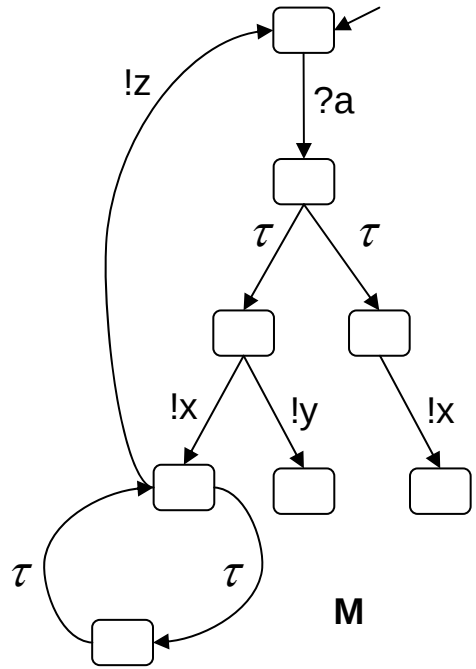
Théorie du test de conformité



Systèmes de transitions étiquetées à entrée/sortie

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

- Q^M un ensemble fini d'états avec $q_0{}^M$ comme état initial,
- $\sum^M = \sum^M{}_I \cup \sum^M{}_o$ un ensemble d'événements observables
  - $\sum^M{}_I$ un ensemble fini d'entrée (?a),
  - $\sum^M{}_o$ un ensemble fini de sortie (!a),
- $\rightarrow_M \subseteq Q^M \times (\sum^M \cup \{\tau\}) \times Q^M$ *la relation de transition*
  - τ une action interne (pas dans $\sum^M$)



Notations

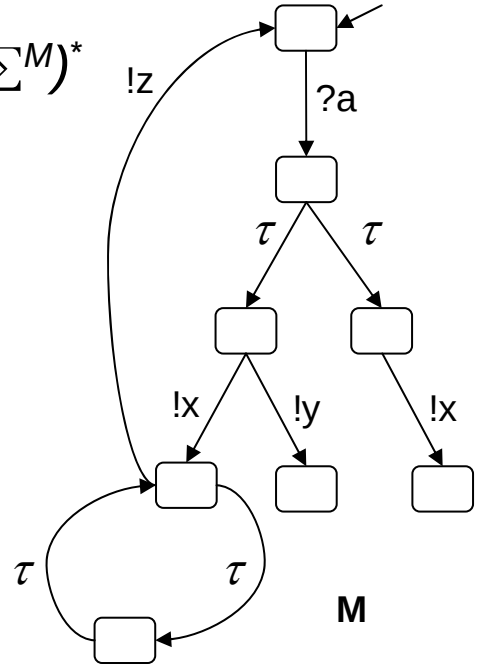
M =(Q^M,$\sum^M$, $\rightarrow^M$,$q_0{}^M$):IOLTS ; q,q'$\in Q^M$; a$\in\sum^M$; $\varepsilon,\sigma,\sigma' \in(\sum^M)^*$

$\Rightarrow$ décrit des comportements visibles de M :

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

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

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



q after σ définit l'ensemble des états accessibles à partir de q par la séquence visible σ :

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

M after σ $\equiv$ {q'$\in Q^M \mid q_0{}^M \Rightarrow^{\sigma} q'$}

$Out_M(q)$ est l'ensemble des actions de sortie à partir de q :

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

Traces(q) décrit les séquences visibles à partir de q :

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

Traces(M) décrit les comportements visibles de M : Traces(M) $\equiv$ Trace($q_0{}^M$)

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,

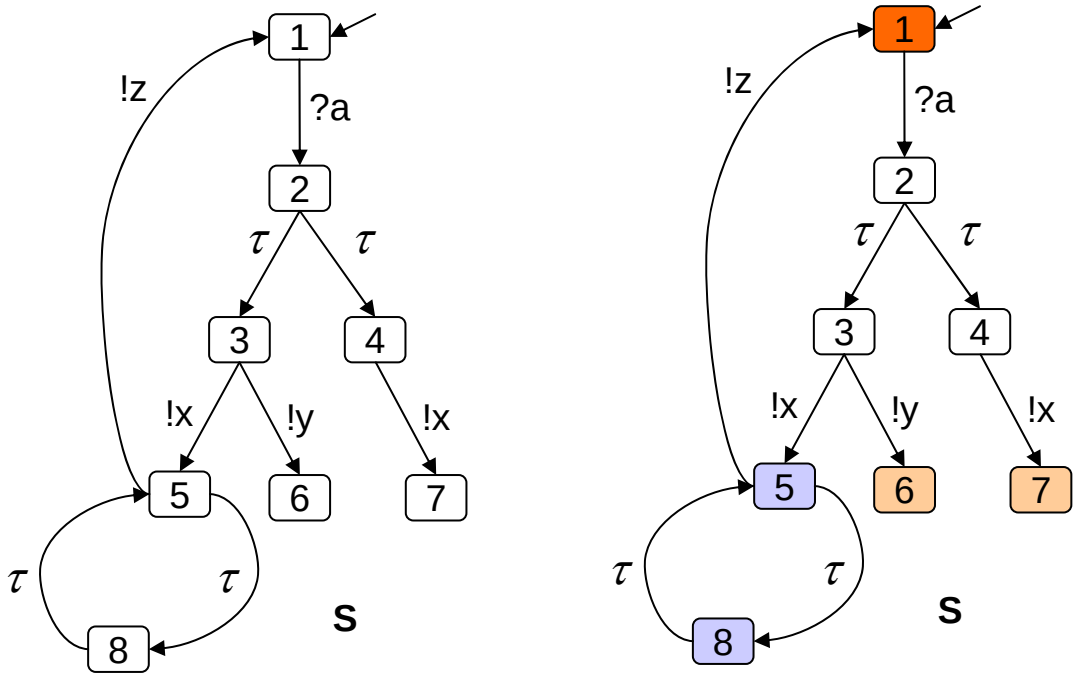
$$\forall q \in Q^M, \forall a \in \sum^M{}_i, \exists q' \in Q^M, q \Rightarrow^a q'$$

Comportement visible : traces + silences

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

– Silence: absence de comportement 'visible'

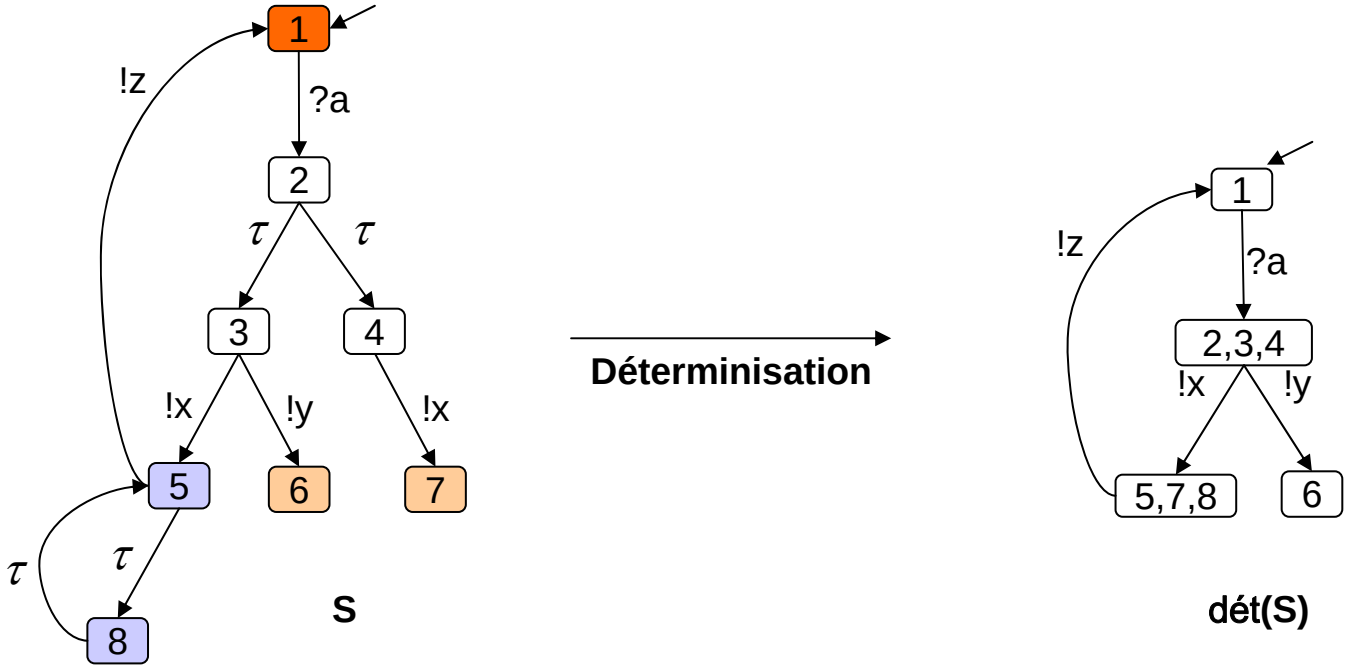
– 3 types de silence : deadlock, livelock et **output-lock**.



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 !

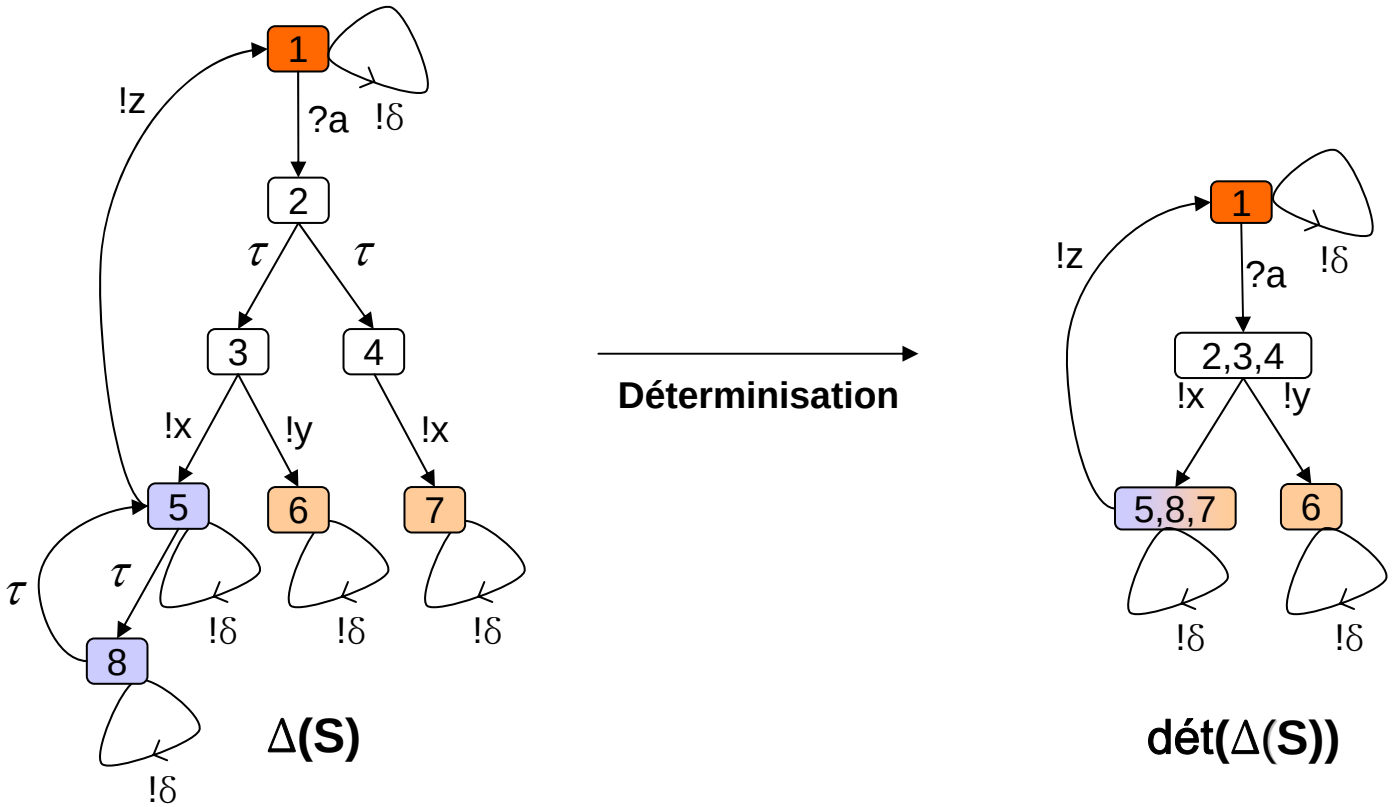
⇒ Une solution consiste à expliciter les silences [TRE96].

Automate de suspension

Tretmans96

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

- 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 $\equiv \forall \sigma \in STraces(S) : out(\Delta(IUT) \text{ after } \sigma) \subseteq out(\Delta(S) \text{ after } \sigma)$

Littéralement :

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

Intuition :

IUT ioco-conforme à S, ssi :

si IUT produit la sortie x après la trace σ,
alors S peut produire x après σ.

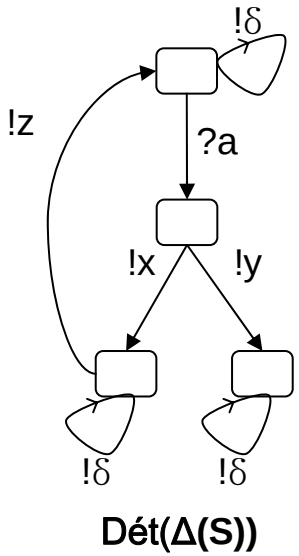
si IUT ne peut pas produire de sortie après la trace σ,
alors S ne peut pas produire de sortie après σ (silence δ).

Relation de Conformité

Tretmans96

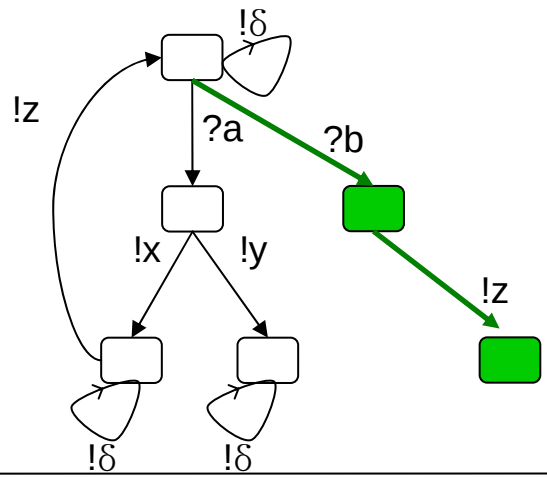
IUT ioco S ≡ ∀σ ∈ STraces(S) : out(Δ(IUT) after σ)⊆ out(Δ(S) after σ)

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



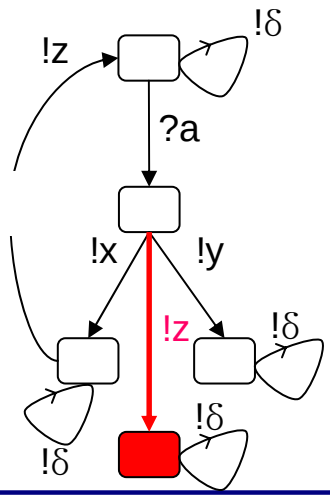
Dét(Δ(S))

IUT1 ioco S



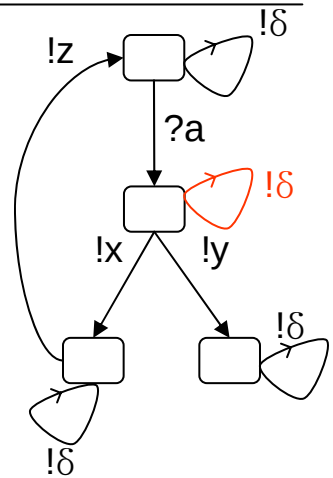
Δ(IUT1)

¬(IUT2 ioco S)



Δ(IUT2)

¬(IUT3 ioco S)

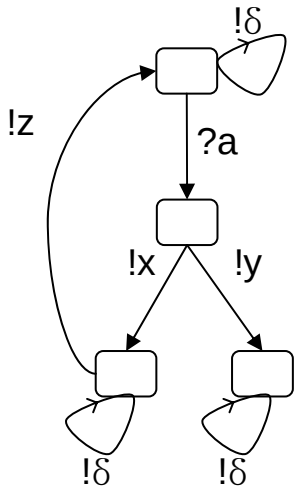


Δ(IUT3)

Relation de Conformité

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

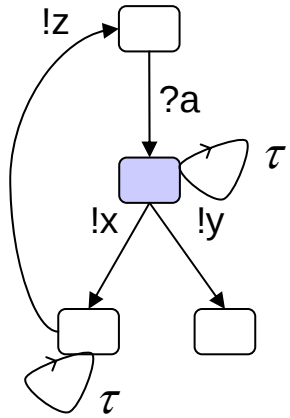
IUT ioco S ≡ ∀σ ∈ STraces(S) out(Δ(IUT) after σ)⊆ out(Δ(S) after σ)



Det(Δ(S))

out(Δ(IUT3) after ?a) = {!x,!y,!δ}

out(Δ(S) after ?a) = {!x,!y}



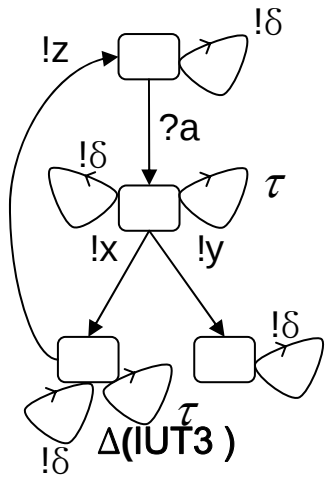
IUT3

IUT3 ioconf S ?

OUI

IUT3 ioco S ?

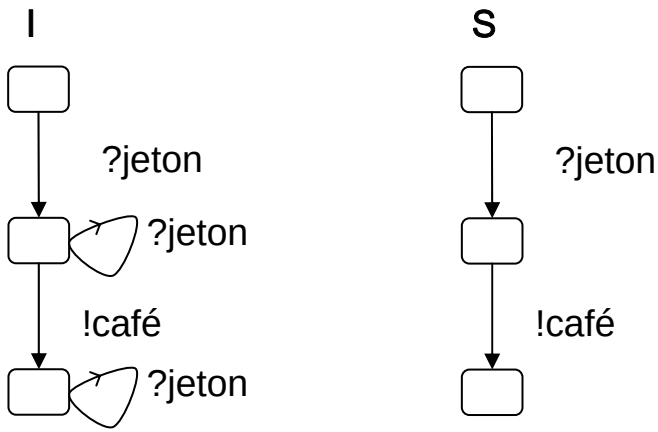
NON



Δ(IUT3)

Exercice

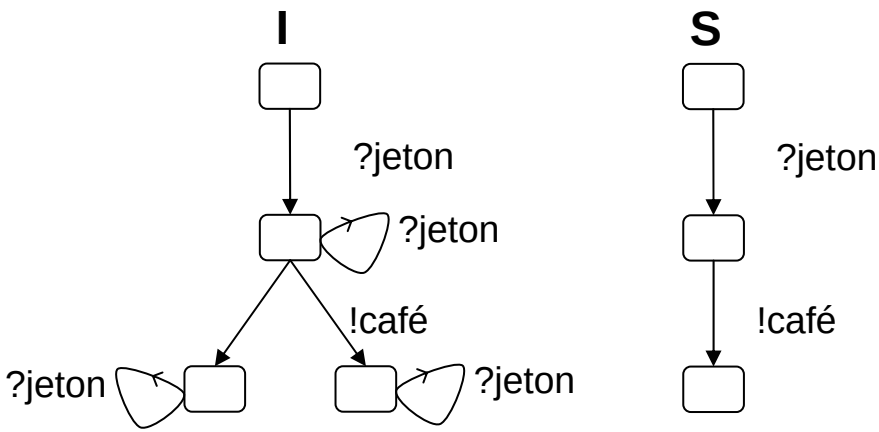
I ioco S $\equiv \forall \sigma \in$ STraces(S) : out(Δ(I) after σ) $\subseteq$ out(Δ(S) after σ)



I ioco S ?

Exercice

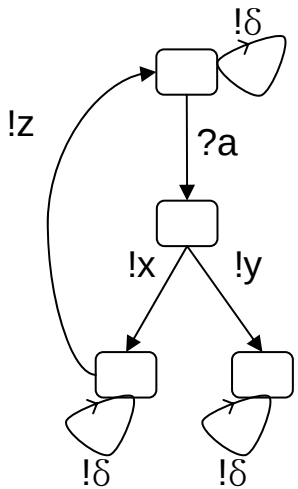
I ioco S $\equiv \forall \sigma \in$ STraces(S) : out(Δ(I) after σ) $\subseteq$ out(Δ(S) after σ)



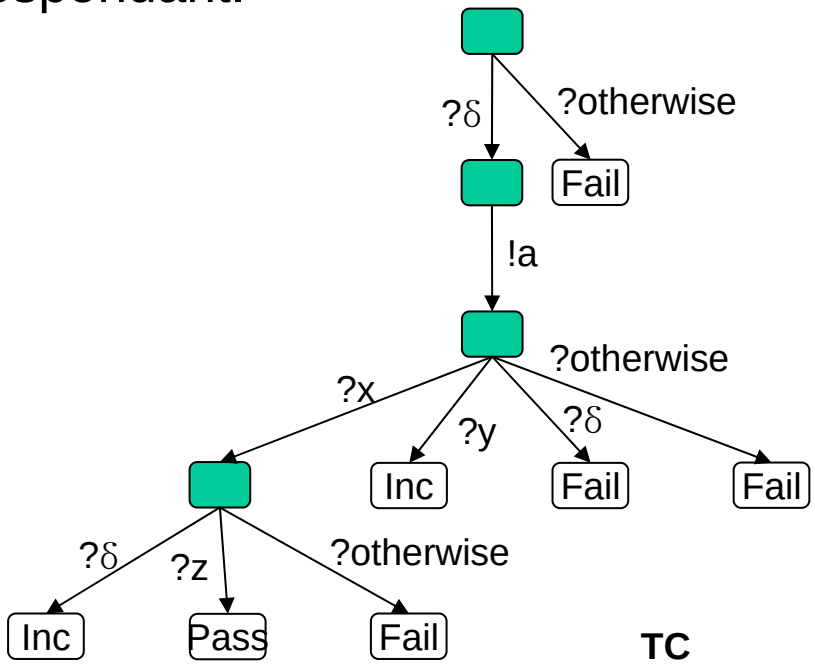
I ioco S ?

Cas de Test : Exemple

- 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.



Det(Δ(S))



TC

Cas de Test : Définition

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 $\{?\delta\}$

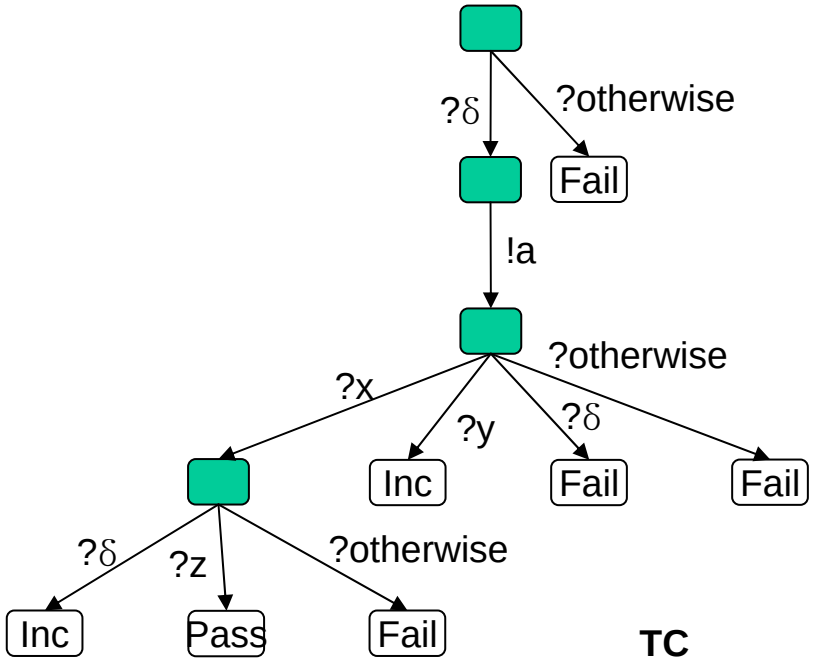
- $Q_{Pass} \subseteq Q^{TC}$ *and* $Q_{Fail} \subseteq Q^{TC}$ *and* $Q_{Inconc} \subseteq Q^{TC}$ (verdict states).

[$!\delta$: silence visible / $?\delta$: expiration du temporisateur]

Cas de Test : Hypothèses

Hypothèses sur les cas de test :

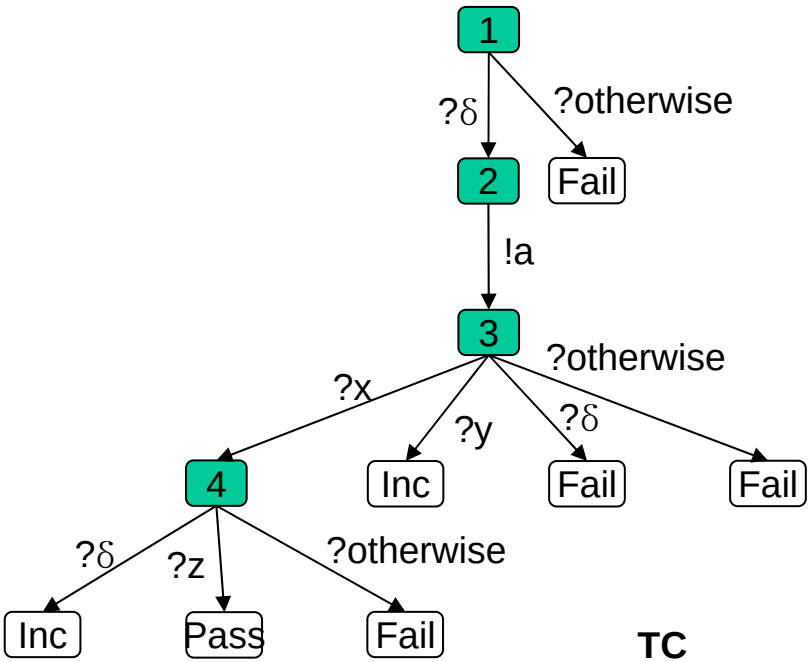
- contrôlable : pas de choix entre une sortie et une autre action,
- Input-complet dans tous les états où une entrée est possible (?otherwise)
- Les états de verdict sont uniquement atteints après des entrées.



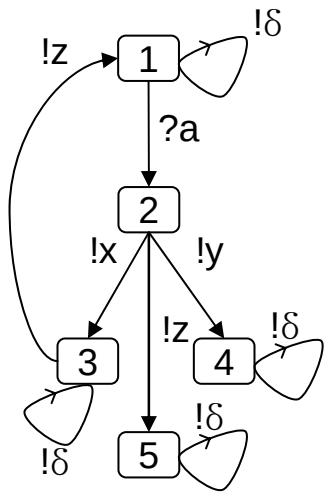
Exécution d'un cas de test

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).

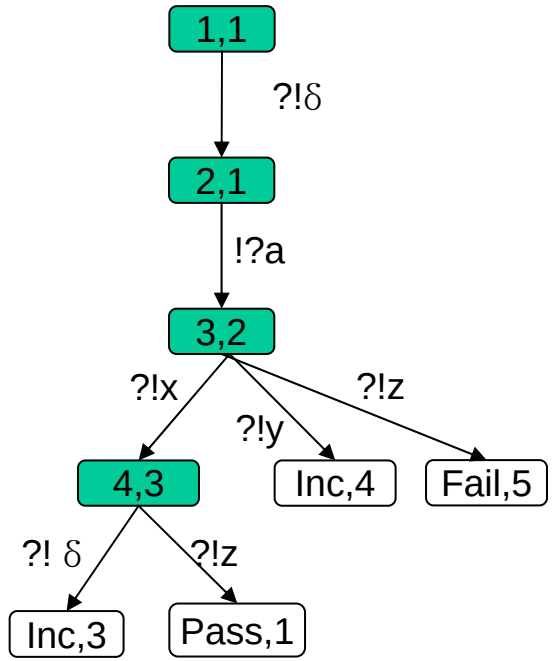
Exemple : Exécution de TC avec IUT2



TC

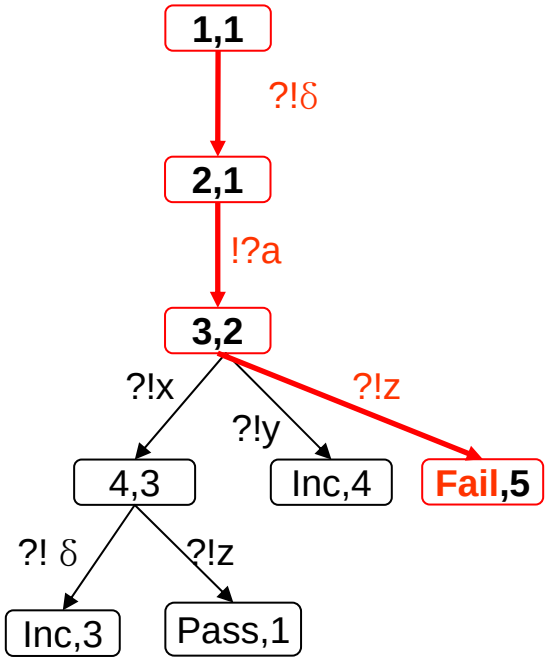


Δ(IUT2)



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*) $\equiv$ (TC after σ) $\subseteq Q_{Fail}$ (resp Pass, Inconc)

Avec :
MaxTraces(M) $\equiv \{\sigma \in (\sum M)^* \mid \Gamma(q0 \text{ after } \sigma) = \varnothing\}$
[$\Gamma(q)$ 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 :

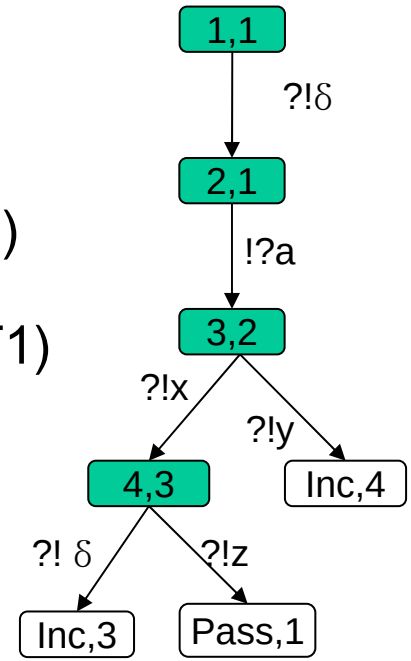
TC may reject IUT

≡

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

('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*.

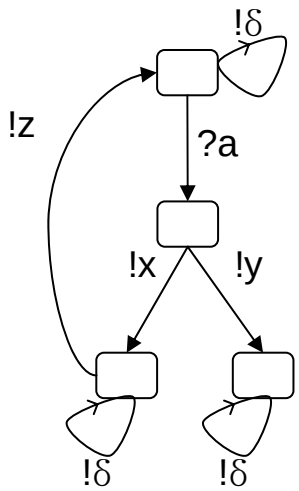
⇒ *TS=GenTest(S,R,P).*

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

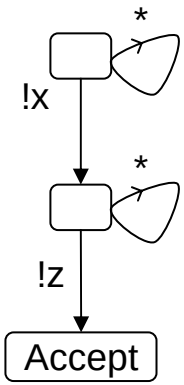
Un exemple de Génération

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

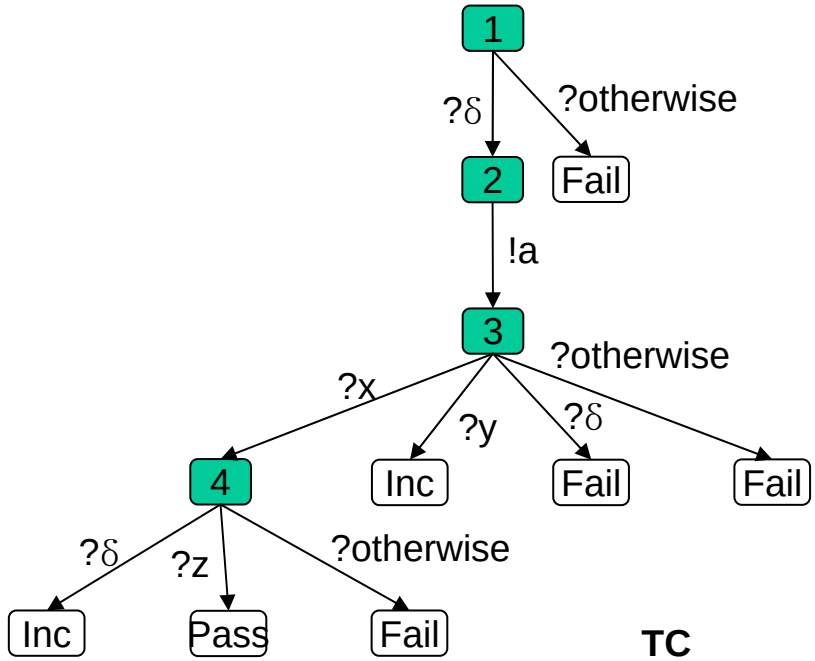
- Accessibilité de *Det(Δ(S))* : trace suspendue de *S* 'acceptée' par *OT*
- Autres algorithmes de génération



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…)*

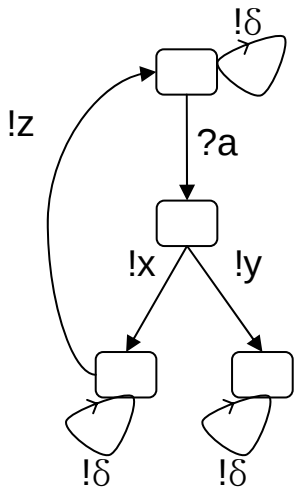
– Définition de propriétés attendues : *soundness*, *exhaustively*. *coverage*

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

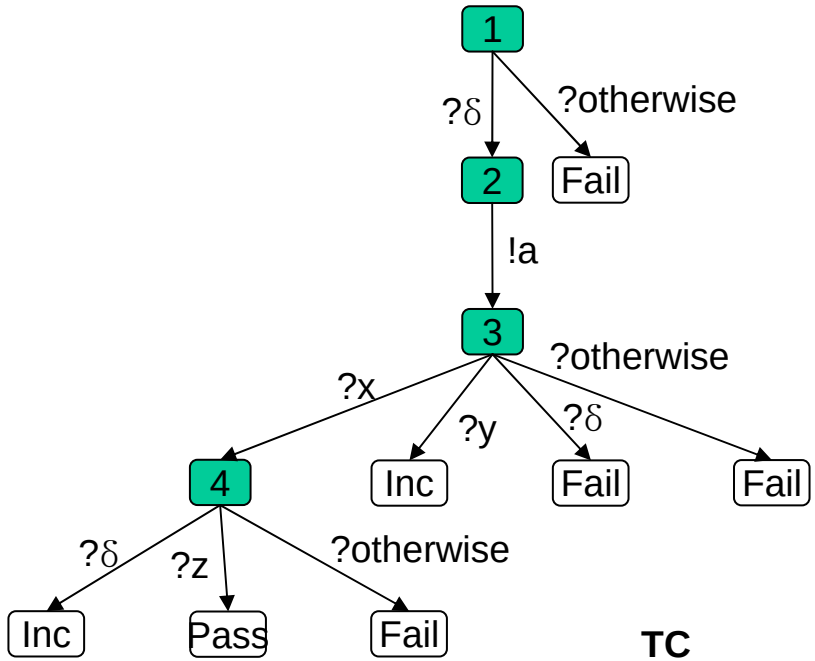
Quelques exercices…

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 »*.



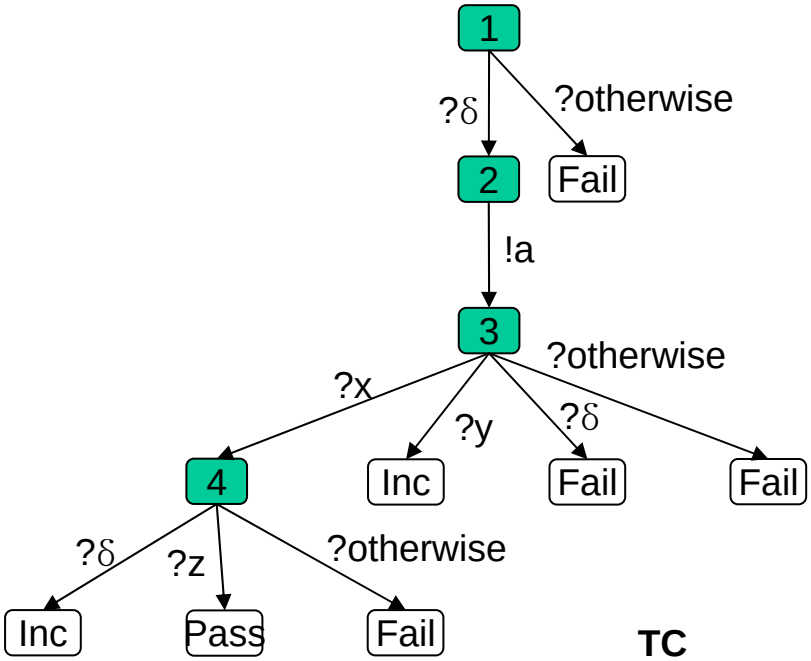
Dét(Δ(S))



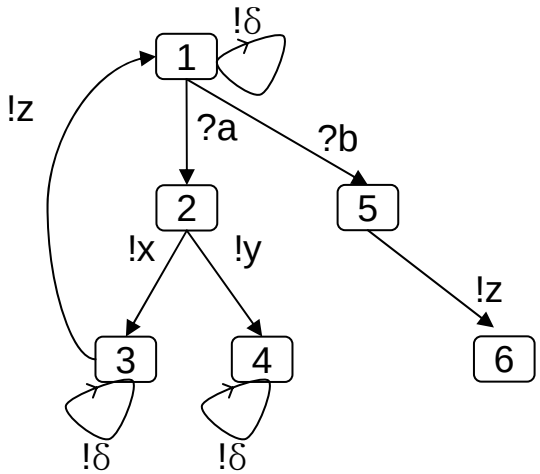
TC

Exercices (suite)

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



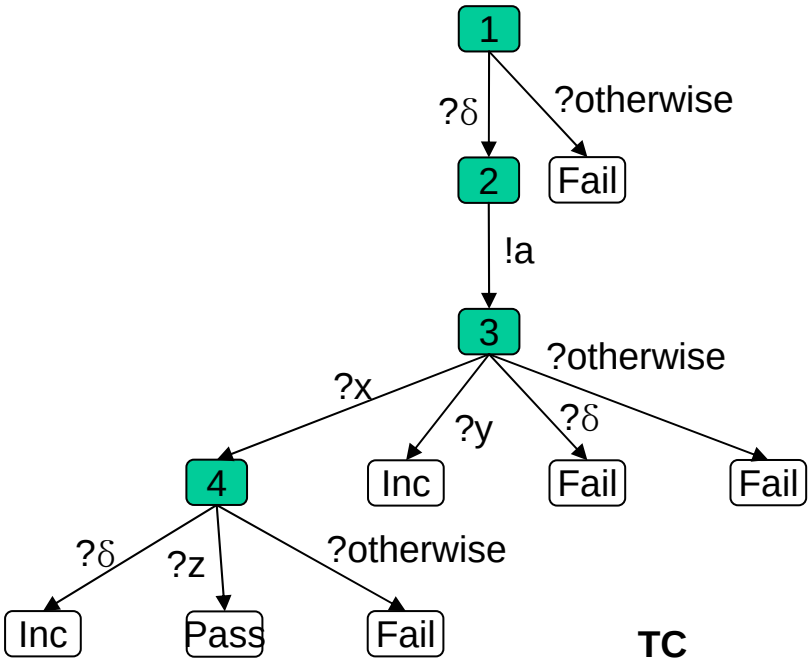
TC



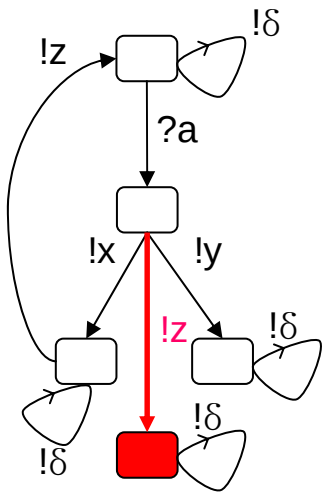
Δ(IUT1)

Exercices (suite)

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



TC



Δ(IUT2)