COT : COmposants et Test

2ème Partie

Génération de tests de conformité pour les systèmes réactifs

Thierry Jéron
IRISA / INRIA Rennes
Projet VerTeCS
jeron@irisa.fr
http ://www.irisa.fr/vertecs/

Génération de tests de conformité pour les systèmes réactifs

Plan du cours

I.1 Problématique du test de conformité

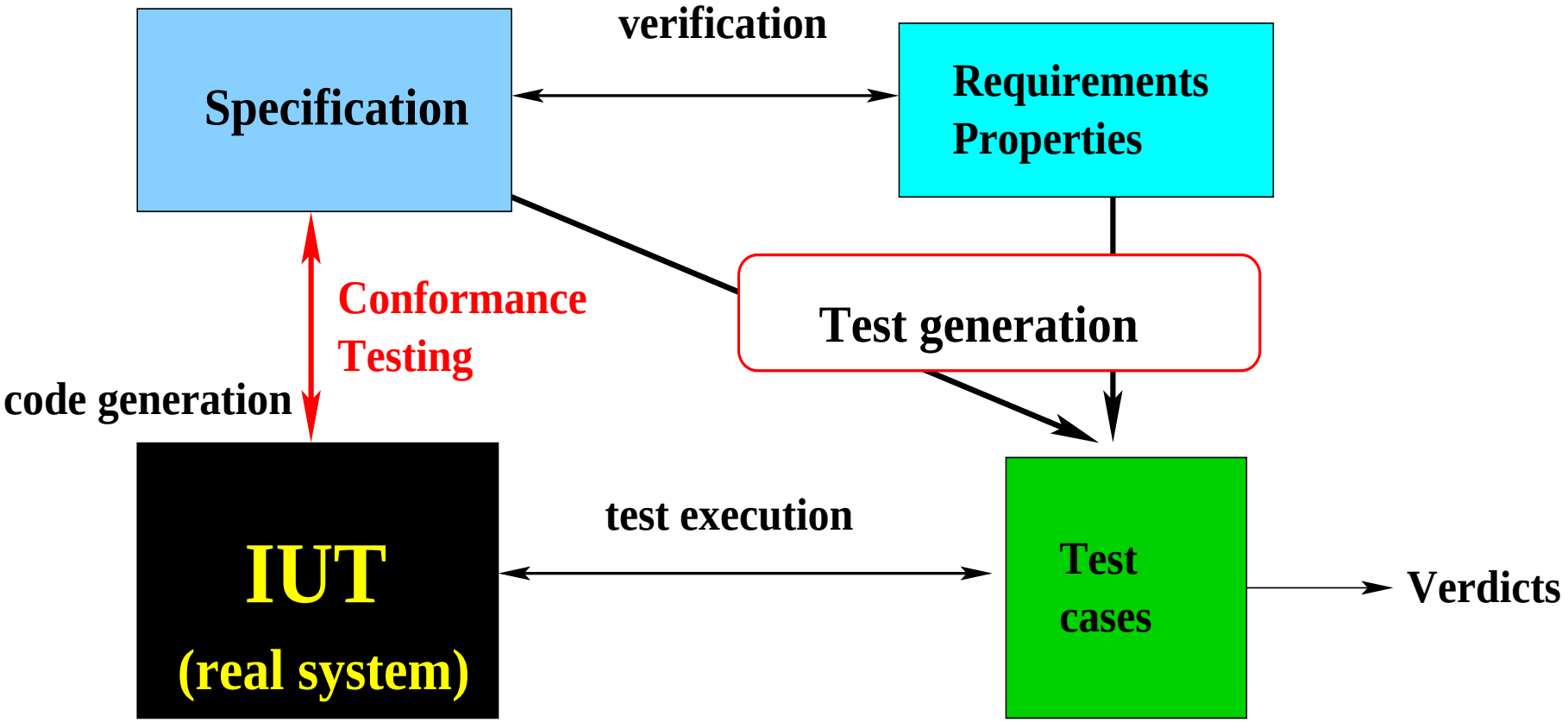
Tester qu'une implémentation boîte noire (IUT) d'un système est correcte par rapport à sa spécification fonctionnelle Spec.

IUT : *implementation under test*
Le système réel (matériel ou logiciel)

Différences principales avec le test structurel (boîte blanche) :

- IUT **boîte noire** : code inconnu, mais interface connue
- la Spec est la **référence**.
  $\Longrightarrow$ Oracle = comportements admissibles de la Spec.
  $\rightarrow$ utiliser des spécifications non ambigues
  $\Longrightarrow$ Problème de l'oracle résolu

Situation du test de conformité



Pratique Industrielle

Conception manuelle des suites de test
depuis des spécifications informelles

- plus de 30% du coût de développement ($\geq$ pour systèmes critiques)
- ad-hoc, long, répétitif, sujet à erreurs,
- maintenance difficile en cas de modification de Spec,
- pas de définition claire de la conformité et du processus de test

$\Rightarrow$ l'automatisation de la synthèse de tests
à partir de spécifications formelles
peut être profitable (effort $\Rightarrow$ coût)

$\rightarrow$ *model based testing/test generation*

Pourquoi formaliser ?

Spécification formelle :
description d'un système dans un langage (syntaxe)
dont la sémantique s'exprime dans un modèle mathématique.
$\Longrightarrow$ permet aussi de décrire et d'analyser ses propriétés : vérification.
Exples de langages : ACT ONE, Z, Lotos, SDL, UML, Lustre, etc
Modèles : spec algébriques, logique et ensembles, logique, automates ou systèmes de transition (étendus), etc.

Avantages d'une spécification formelle :
- non-ambiguité, abstraction, masquage des choix d'implémentation,
- propriétés attendues du logiciel / choix d'implémentation
- nécessaire pour les applications critiques
- utile à la vérification, génération de code ou implémentation
- référence pour la génération de tests (oracle)

Ingrédients principaux d'une théorie du test

Spécification, implémentation et conformité

Spécifications : modèle des comportements attendus

Implémentations : modèle des comportements réels observables

Relation de conformité : formalisation de "IUT conforme à Spec"

Les tests et leurs exécutions

Cas de test, suites de tests : modèle des tests

Exécution des tests : interaction test $\leftrightarrow$ IUT,
observations produites, **verdicts** associés (e.g. pass, fail)

Propriétés attendues : "IUT passe TS" $\leftrightarrow$ "IUT conforme à S"

Génération de tests

Algorithmes : tests = testgen(Spec) + preuve de propriétés attendues

Ingrédients de la formalisation du test (inspiré de Formal Methods in Conformance Testing)

Modèle de spécification :
$S \in SPEC$
Exprime la sémantique de la specification
Exples : spec. algébriques, logique et ensembles, logique, automates/systèmes de transition (étendus), etc.

Implémentation réelle boîte noire :
L'IUT est un objet réel, pas un modèle mathématique.
$\Rightarrow$ **Hypothèse fondamentale :**
$\exists I_{IUT} \in IMP$ (inconnue) qui modélise IUT.
Seule l'interface de IUT est connue.
$\Rightarrow$ modèle de comportements observables, proche de $SPEC$.

Formaliser la conformité

Conformité : choisir une relation $\mathbf{imp} \subseteq IMP \times SPEC$
IUT **conforme à** S définie par : I_{IUT} $\mathbf{imp}$ S

Pour S donné, **imp** *définit les IUT conformes,*
Complémentaire : IUT fautives qui devraient être rejetées par le test.

Le choix de **imp** dépend du potentiel de l'observateur à **distinguer** les IUT (e.g. observabilité, architecture de test),
Exples : égalité des résultats, satisfaction de propriété, équivalence de comportements, inclusion de traces, etc

Notion alternative : Modèle de faute : définit l'ensemble des IUT "mutantes" (non-conformes / S) et qu'on saura rejeter par le test.

Formalisation des Tests et leurs exécutions

Tests : choisir un modèle
cas de test : $t \in TEST$,
suite de tests $=$ ens. de cas de tests : $T \subset TEST$
Exples : donnée d'entrée, terme, séquence d'événements, automate/système de transition, etc

Observation de l'exécution d'un test :
$Obs(t, I_{IUT}) \in OBS$
Exples : résultat, trace d'éxécution, ensemble de traces de la composition parallèle, etc.

Verdict de l'exécution

Verdict : $Verdict : OBS \rightarrow \{fail, pass\}$

Exples : *fail* si résultat observé $\neq$ résultat attendu,
si trace observée $\neq$ trace attendue,
si $\neg\{$traces observées$\} \subset \{$traces attendues$\}$,
si formule/équation non satisfaite, etc.

Le verdict doit être cohérent avec la relation de conformité
$\Longrightarrow$ Propriétés des verdicts des tests/conformité .

Succès et échec d'un(e suite de) test

Echec d'un test

t **fails** $I_{IUT} \triangleq Verdict(Obs(t, I_{IUT})) = fail$

T **fails** $I_{IUT} \triangleq \exists t \in T, t$ **fails** I_{IUT}

Succès d'un test

t **passes** $I_{IUT} \triangleq \neg(t$ **fails** $I_{IUT})$

T **passes** $I_{IUT} \triangleq \neg(T$ **fails** $I_{IUT})$

Propriétés attendues des suites de test

Problème : les tests sont sensés détecter la (non-)conformité.
Il faut donc assurer la cohérence entre succès (ou échec) d'une suite de tests sur une IUT et conformité (ou non) par rapport à la Spec.

Complétude :
Idéalement, pour $S \in SPEC$ donnée on veut générer T_S **complète** i.e. tq :

$$\forall I_{IUT}, \quad [I_{IUT} \textbf{ imp } S \Leftrightarrow T_S \textbf{ passes } I_{IUT}]$$

i.e. $\forall I_{IUT}, \quad [I_{IUT} \textbf{ imp } S \Leftrightarrow \forall t \in T_S,\ Verdict(Obs(t, I_{IUT})) = pass]$

i.e. conformité $\Leftrightarrow$ passer tous les tests avec succès

Complet = Correct($\Rightarrow$) et Exhaustif ($\Leftarrow$)

avec :

Correction et exhaustivité

Correction, non-biais (*soundness, unbias*)
T **correcte**/ imp,S $\triangleq \forall I_{IUT}, \quad [I_{IUT}$ **imp** $S \Rightarrow T$ **passes** $I_{IUT}]$
Autrement dit : $\Leftrightarrow \forall I_{IUT}, \quad [T$ **fails** $I_{IUT} \Rightarrow \neg(I_{IUT}$ **imp** $S)]$

i.e. seules les IUT non conformes peuvent être rejetées.
Cas limite : T tq $\forall I_{IUT}, T$ **passes** I_{IUT} est correcte !

Exhaustivité :
T est **exhaustive**/imp, S $\triangleq \forall I_{IUT}, \quad [T$ **passes** $I_{IUT} \Rightarrow I$ **imp** $S]$
Autrement dit $\Leftrightarrow \forall I_{IUT}, \quad [\neg(I_{IUT}$ **imp** $S) \Rightarrow T$ **fails** $I_{IUT}]$

i.e. toutes les IUT non-conformes sont rejetées, peut-être d'autres.
Cas limite : T tq $\forall I_{IUT}, T$ **fails** I_{IUT} est exhaustif

Problématique de la génération de suites de test

Trouver un algorithme $gen : SPEC \rightarrow TEST$ tq
$\forall S, T_S = testgen(S)$ soit **complète** (correcte et exhaustive).

En pratique, la correction peut être atteinte, pas la complétude.
La complétude nécessite en général une suite de test infinie.

Program testing can be used to prove the presence of bugs,
not their absence
$\Longrightarrow$ sélectionner un ensemble fini (raisonnable) de cas de tests corrects et supposés détecter toutes les non-conformités
$\Rightarrow$ utiliser des hypothèses et critères de sélection

Hypothèses de sélection (Bernot, Gaudel (SA), Phalippou (LTS))

Formaliser les hypothèses faites par les praticiens du test pour limiter les suites de test. Inspiré de Goodenough et Gerhart (voir notions de critères de couverture, Partie 1).

Principe :

- partir d'un jeu de test complet $complet(S)$ (correct et exhaustif) mais généralement infini
- faire une hypothèse H sur I_{IUT} (fonction du contexte d'utilisation, de la spécification),
- réduire $complet(S)$ en T tq $\forall I_{IUT}$,
  I_{IUT} **sat** $H \Rightarrow [complet(S)$ **passes** $I_{IUT} \Leftrightarrow T$ **passes** $I_{IUT}]$

Exemples d'hypothèses de sélection

Uniformité ($\rightarrow$ fiabilité en partie 1) :
H_U : partition des comportements, uniformité dans chaque classe
i.e t **passes** $I \Leftrightarrow \forall t' \in Classe(t), t'$ **passes** I
$T =$ choix d'un test dans chaque classe.
I **sat** $H_U \Rightarrow [complet(S)$ **passes** $I \Leftrightarrow T$ **passes** $I]$
Exples : partition des domaine d'entrée : $\mathbb{Z} = \mathbb{Z}^- \cup \{0\} \cup \mathbb{Z}^+$,
$\forall z \in \mathbb{Z}, I(z)$ passe ssi $I(0)$, $I(1)$, $I(-1)$ passent ;
Autres exples : critères de couverture (branche, conditions, etc)

Régularité : fonction $length : TEST \rightarrow \mathbb{N}$ et borne M telle que
$H_R = [\forall t, length(t) \leq M, t$ **passes** $I] \Leftrightarrow [\forall t, t$ **passes** $I]$
$T = \{t \in TEST | \ length(t) \leq M\}$
Exple : borne sur la taille des listes, termes, borne sur la longueur de séquences, nbre d'appels récursifs, etc.

Autres hypothèses de sélection

Robustesse : se limiter aux comportements décrits dans S suffit.
Exple : la relation de conformité **ioco** sur les IOLTS porte sur Traces(S).

Equité bornée : borne sur le nombre de fois où un état ayant des choix non-déterministes doit être visité pour montrer tous ses comportements.

Autres modes de sélection :

Objectif de test : comportements de la spec ciblés par le test, focalise le test sur une fonctionalité
$\rightarrow$ un ou plusieurs cas de test.

Critère de sélection/couverture : permettant de limiter les cas de test, e.g. couverture de branches/d'états/etc.

Applications possibles

Spécifications algébriques (Bernot, Gaudel)
conformité = satisfaction des axiomes

Automates (voir Lee-Yannakakis)
conformité = équivalence d'automates

Systèmes de transition, algèbres de processus (Tretmans)
conformité = pré-ordre de traces

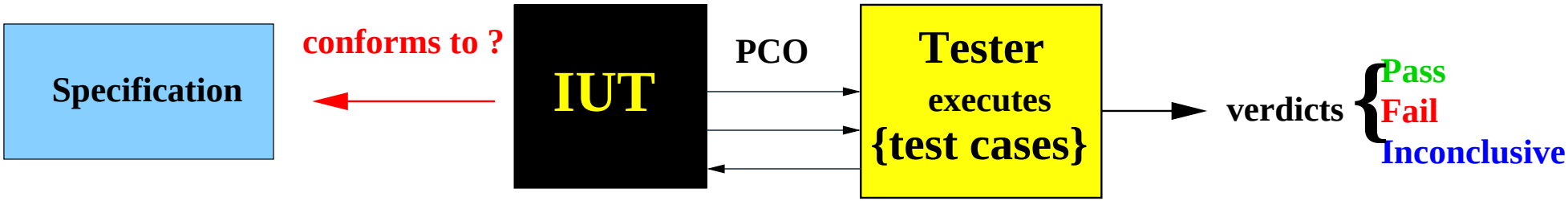
Le test de conformité des système réactifs

Système réactif : système qui réagit à son environnement à travers ses interfaces.

- Environnement : humain, logiciel, matériel
- Interfaces : commandes, capteurs, canaux de communications, opérations, méthodes.
- Différence contrôle/observation : entrée/sortie, appel/réponse, etc

Pratique du test

- Interaction contrôlée entre l'**IUT** et le **Testeur** à travers des **Points de Contrôle et d'Observation (PCOs)**,
- Le testeur **exécute** une **suite de test** = { cas de test }
- Produit des **verdicts**
- But : trouver des erreurs ou augmenter sa confiance dans l'IUT



Automatisation de la synthèse de tests de conformité

Spécifications formelles exécutables

peuvent être utilisées pour la simulation, vérification, preuve.

Principales difficultés de l'automatisation

- taille et complexité des spécifications
- observation partielle, non-déterminisme,
- communication synchrone/asynchrone
- répartition du système / des testeurs
- prise en compte du contrôle et des données
- intégration au cycle développement

II Techniques de génération de tests de conformité

II.1 Techniques fondées sur les automates
- Modèles
- Algorithmes

II.2 Techniques fondées sur les systèmes de transition
- Modèles
- Algorithmes

II.3 Techniques avancées

Les deux types principaux de méthodes

II.1 Méthodes fondées sur les automates

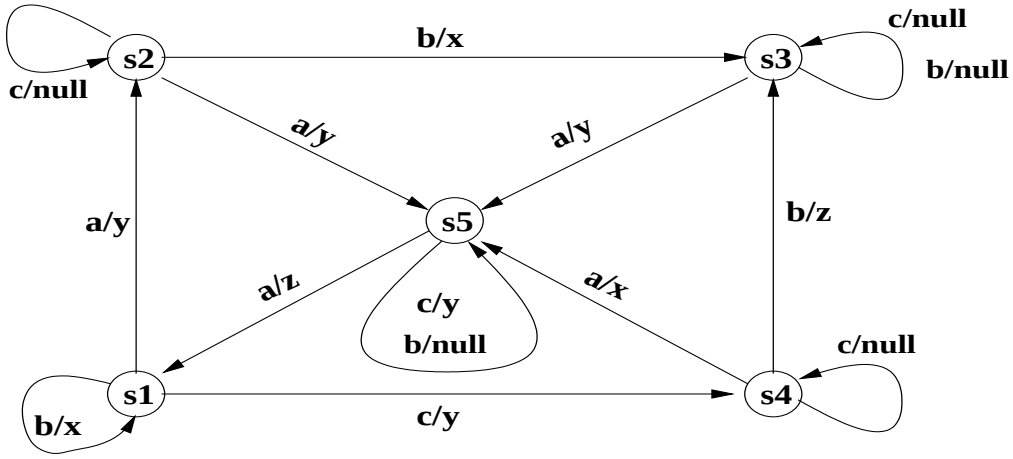
Origine : pb d'identification de machine, test de circuit
Modèle : automates + hypothèses (déterminisme, forte connexité, etc),
Conformité : équivalence de traces entre IUT et Spec,
Algorithmique : voyageur de commerce $\rightarrow$ + courte séquence
Propriétés : $\rightarrow$ suite de tests complète

II.2 Méthodes fondées sur les Systèmes de transition

Origine : équivalences et pré-ordres de tests de processus
Modèle : systèmes de transitions
Conformité : inclusion de traces,
Algorithmique : automates, model-checking.
Propriétés : correction, "exhaustivité"

II.1 Techniques fondées sur les automates

Machine de Mealy $M = (S, I, O, \delta, \omega, s_0)$ où
S : ensemble fini d'états, s_0 : état initial
I : alphabet d'entrée, O : alphabet de sortie,
$\delta \subseteq S \times I \to S$: fonction de transfert (étendue en $\delta^* \subseteq S \times I^* \to S$),
$\omega \subseteq S \times I \to O$: fonction de sortie (étendue en $\omega^* \subseteq S \times I^* \to O^*$).



S={s1,s2,s3,s4,s5}

I={a,b,c}

O={x,y,z}

Spec et IUT modélisées par des machines de Mealy M_{Spec} et M_{IUT}

Hypothèses sur M_{Spec} et M_{IUT}

Les machines de Mealy sont supposées déterministe :
$\forall s \in S, \forall a \in I$, $\delta(s,a)$ et $\omega(s,a)$ sont uniques
(vrai par définition : δ et ω fonctions).

Soit une machine $M = (S, I, O, \delta, \omega, s_0)$

M est complète en entrée si $\forall s \in S, \forall i \in I$, $\delta(s,i)$ et $\omega(s,i)$ sont définis.
i.e. accepte toute entrée dans tout état
(sinon, on complète par $s \xrightarrow{i/null} s$)

M est fortement connexe si pour toute paire d'états (s, s')
$\exists \sigma, \sigma' \in I^*$ tq $\delta^*(s,\sigma) = s'$ et $\delta^*(s',\sigma') = s$.
i.e. de tout état, on peut rejoindre tout autre état (en particulier l'état initial)

Autres hypothèses sur M_{Spec} et M_{IUT}

Équivalence :
s et s' sont équivalents (noté $s \simeq s'$) ssi $\forall \sigma \in I^*, \omega^*(s,\sigma) = \omega^*(s',\sigma)$.
M_1 et M_2 sont équivalentes (noté $M_1 \simeq M_2$) ssi $s_{0_1} \simeq s_{0_2}$.

Minimalité :
M est minimale si elle n'a aucun couple d'états équivalents
NB : Toute machine non minimale peut être minimisée (raffinement de partition).

Taille bornée : M_{IUT} et M_{Spec} ont même nombre d'états $|M_{\text{IUT}}| = |M_{\text{Spec}}|$
Dans la suite, les états de M_{IUT} et M_{Spec} ont même nom.
(extension possible : $|M_{\text{IUT}}| - |M_{\text{Spec}}| \leq \Delta$)

Conformité

Conformité $\triangleq$ équivalence de machines
Soient M_{IUT} et M_{Spec} deux machines de Mealy de mêmes alphabets, minimales, fortement connexes, et ayant même nombre d'états,
$M_{IUT}\ imp\ M_{Spec} \triangleq M_{IUT} \simeq M_{Spec}$

Le but du test est donc de chercher à prouver (ou invalider) l'équivalence entre une machine de Mealy M_{IUT} inconnue et une autre machine de Mealy connue M_{Spec}.

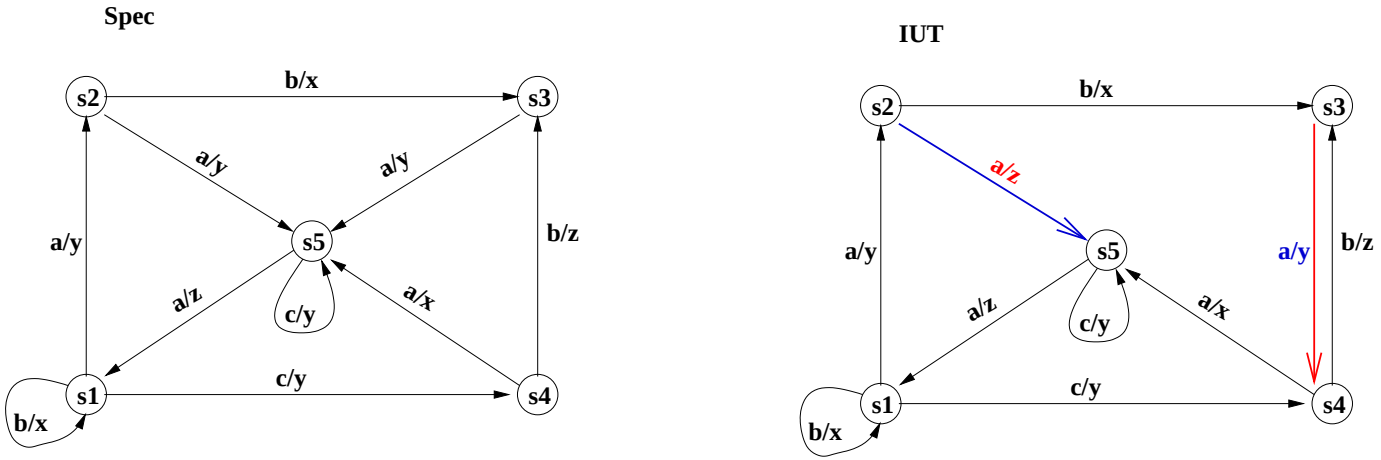
NB : noter qu'avec les hypothèses de minimalité, même nombre d'états, mêmes alphabets
équivalence de machine = isomorphisme de machine

Autre vue : modèle de faute

Modèle de faute : ensemble des M_{IUT} "mutantes" de M_{Spec} respectant les hypothèses, et non-conformes (i.e. non isomorphes).

Pour une transition $s \xrightarrow{i/o} s'$ de M_{Spec}, fautes possibles :

- faute de sortie : $s \xrightarrow{i/o'} s'$ et $o \neq o'$
- faute de transfert : $s \xrightarrow{i/o} s''$ et $s' \neq s''$



Principe du test

Non-isomorphisme $\Leftrightarrow$ existence d'une faute

Détection de toutes les fautes du modèle de faute.

$\Rightarrow$ tester chaque transition de IUT pour ses fautes de sortie et de transfert.

Test élémentaire : test d'une transition $s \xrightarrow{i/o} s'$:

- aller en s
- appliquer i, vérifier o (sortie)
- identifier s' (transfert)

NB : L'identification d'état consiste en une séquence d'entrées/sorties

$\rightarrow$ Problème d'optimisation : rechercher une séquence la plus courte possible contenant tous les tests élémentaires

Différentes techniques

Dépendant des séquences d'identification possibles.

Conditions d'applicabilité sur M_{Spec} :

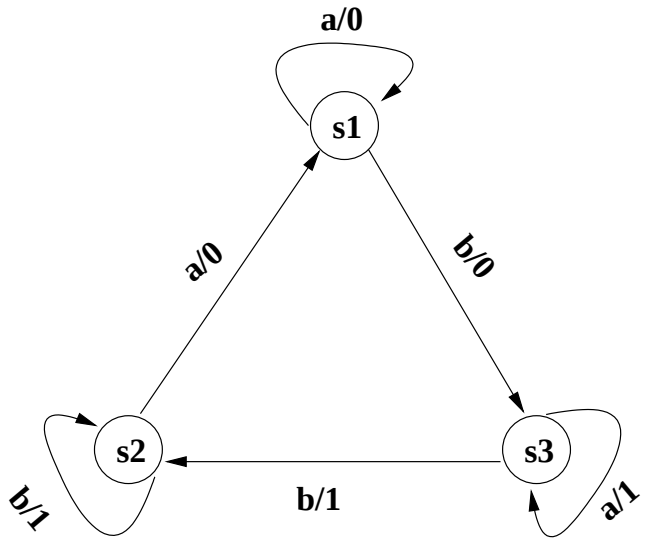
Transition Tour : pas d'identification ($\Longrightarrow$ incomplet pour transfert)
ou $\exists$ interrogation d'état : $\forall s, \exists s \xrightarrow{status/s} s$

DS : $\exists DS \in I^*, \forall s_i \in S, \forall s_j \not\simeq s_i \in S, \omega^*(s_i, DS) \neq \omega^*(s_j, DS)$

UIO : $\forall s_i \in S, \exists UIO_i \in I^*, \forall s_j \not\simeq s_i \in S, \omega^*(s_i, UIO_i) \neq \omega^*(s_j, UIO_i)$

W : $\forall s_i \in S, \forall s_j \not\simeq s_i \in S, \exists c_{ij} \in I^*, \omega^*(s_i, c_{ij}) \neq \omega^*(s_j, c_{ij})$
NB : W est exactement la condition de minimalité

Différence entre DS, UIO et W



W : machine minimale. $cs_{12} = cs_{13} = b$, $cs_{23} = a$.

UIO : $UIO(s_1) = b$, $UIO(s_3) = a$, pas d'UIO pour s_2.

DS : n'existe pas.

Algorithmes pour le Tour de Transitions

Hyp : M_{Spec} et M_{IUT} sont fortement connexes, complètes, minimales, même nombre d'états, status valide sur M_{IUT}.

Génération de suite de test :
trouver un circuit (orienté) qui couvre chaque transition.

Pb d'optimisation :
trouver un circuit de longueur minimale couvrant toutes les transitions.
$\Rightarrow$ Pb du Postier Chinois

Circuit eulérien et Postier Chinois

Circuit eulérien : circuit (orienté) qui passe une et une seule fois par toutes les transitions.

Proposition : un graphe possède un circuit eulérien ssi il est fortement connexe et symétrique.

où $G = (V, E)$ est symétrique ssi pour tout sommet $s \in V$, $d^o_{in}(s) = d^o_{out}(s)$.
$d^o_{in}(s) =$ degré entrant de $s = Card(\{s' \to s \in E\})$
$d^o_{out}(s) =$ degré sortant de $s = Card(\{s \to s' \in E\})$

On utilisera la proposition suivante :

circuit de longueur minimale couvrant toutes les transitions de G

$=$

circuit eulérien de l'augmentation symétrique de coût minimal $\hat{G}$ de G.

Augmentation symétrique de coût minimal

$\hat{G}$: graphe symétrique obtenu en dupliquant de manière minimale certaines transitions de G .

$X_{i,j} =$ nbre de fois où une transition $s_i \to s_j$ de G est dupliquée ds $\hat{G}$.

$\hat{G}$ est une solution du problème d'optimisation :

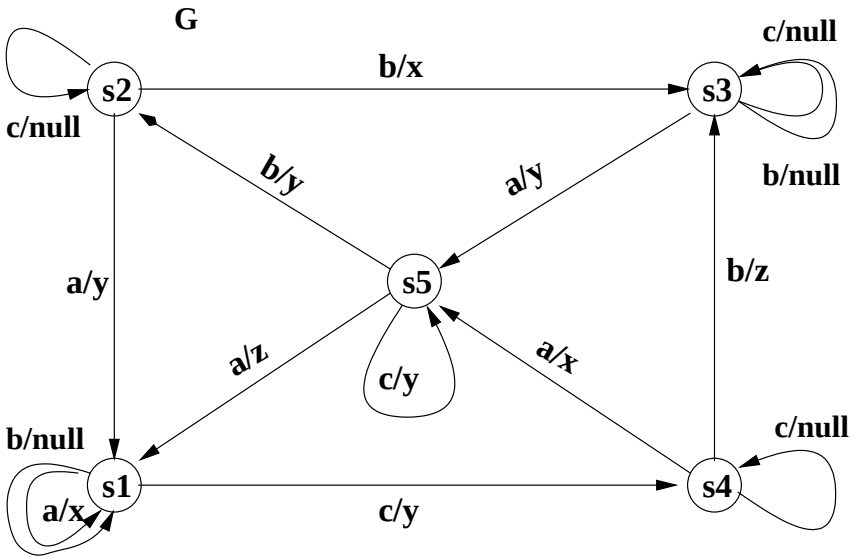
$$\text{Minimiser } \Sigma_{i,j} X_{i,j} \text{ tq } \forall j, d^o_{in}(s_j, \hat{G}) = d^o_{out}(s_j, \hat{G}) \text{ i.e.}$$

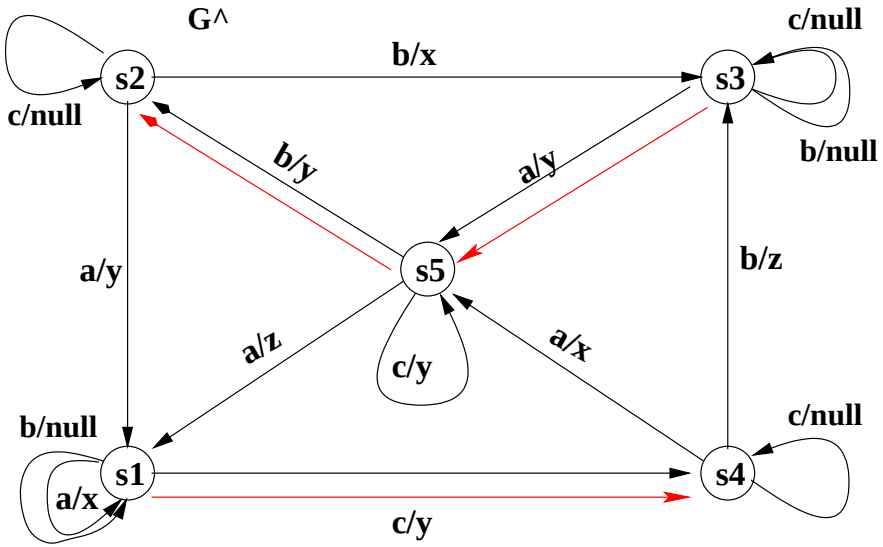
$$\forall j, d^o_{in}(s_j, G) + \Sigma_{\{i, s_i \to s_j\}} X_{i,j} = d^o_{out}(s_j, G) + \Sigma_{\{k, s_j \to s_k\}} X_{j,k}$$

NB : on peut ignorer les boucles (influence nulle).

Résolution : programmation linéaire ou algorithme de flot.

Exemple


G
s2
s3
s5
s1
s4
b/x
c/null
b/y
a/y
b/null
b/z
a/z
c/y
a/x
c/null
b/null
a/x
c/y


G^
s2
s3
s5
s1
s4
b/x
c/null
b/y
a/y
b/null
b/z
a/z
c/y
a/x
c/null
b/null
a/x
c/y

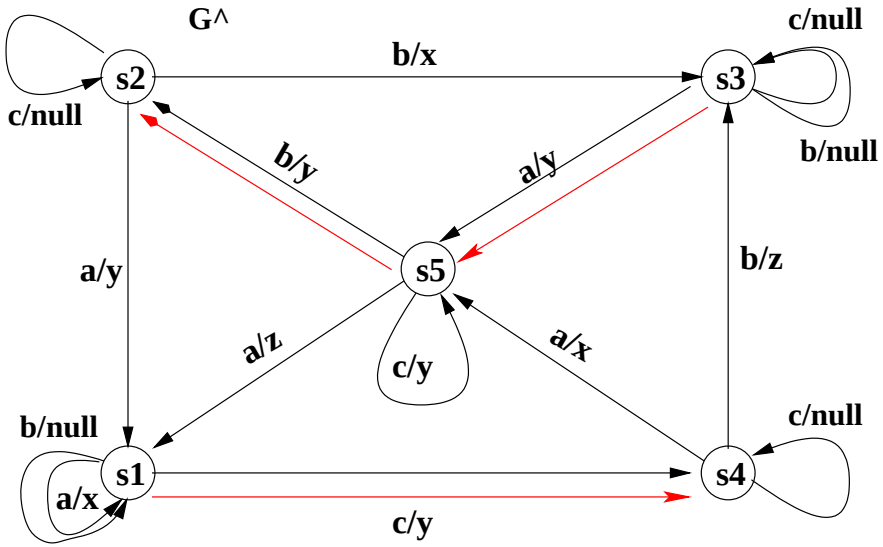
Exemple : résolution du problème d'optimisation

Minimiser $\Sigma_{i,j} X_{i,j}$ tq

$$
\begin{aligned}
S_1 &: X_{21} + X_{51} + 4 = X_{14} + 3 \\
S_2 &: X_{52} + 2 = X_{21} + X_{23} + 3 \\
S_3 &: X_{23} + X_{43} + 4 = X_{35} + 3 \\
S_4 &: X_{14} + 2 = X_{45} + X_{43} + 3 \\
S_5 &: X_{45} + X_{35} + 3 = X_{52} + X_{51} + 3
\end{aligned}
$$

Solution : $X_{14} = X_{52} = X_{35} = 1$, $X_{i,j} = 0$ sinon.

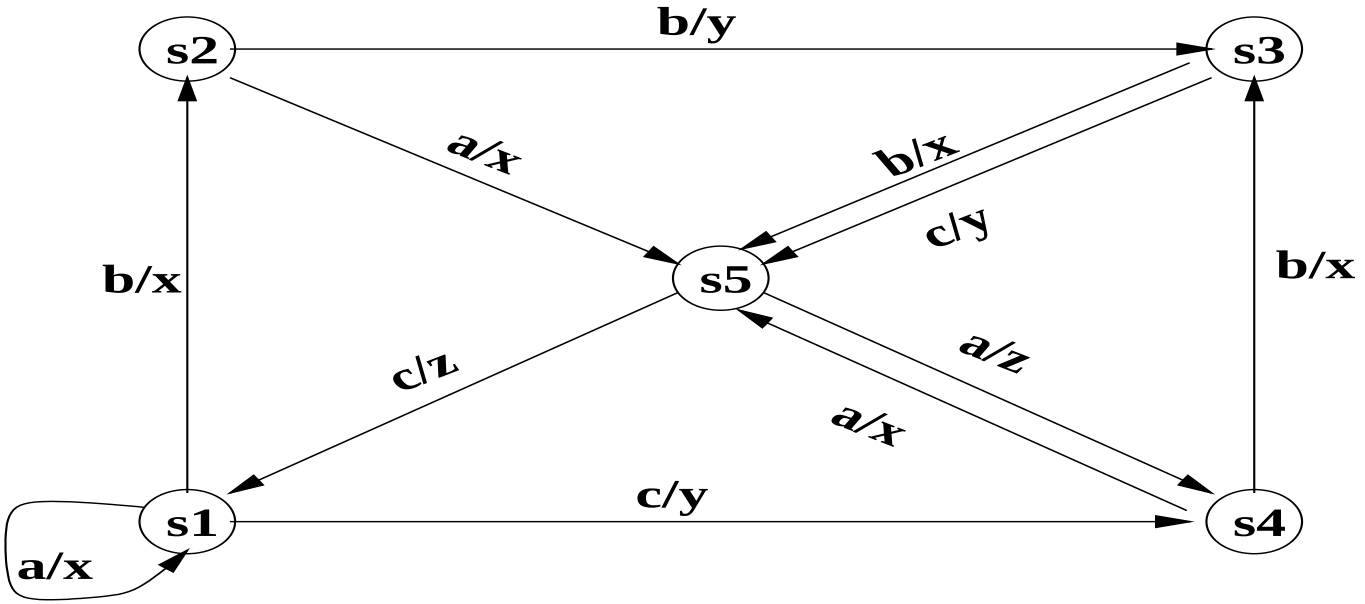
Exemple : calcul du circuit eulérien



Circuit eulérien sur $\hat{G}$ $\rightarrow$ séquence de test (status omis) :

$$s_1 \xrightarrow{a/x} s_1 \xrightarrow{b/null} s_1 \xrightarrow{c/y} s_4 \xrightarrow{c/null} s_4 \xrightarrow{b/z} s_3 \xrightarrow{b/null} s_3 \xrightarrow{c/null} s_3 \xrightarrow{a/y} s_5$$
$$\xrightarrow{c/y} s_5 \xrightarrow{a/z} s_1 \xrightarrow{c/y} s_4 \xrightarrow{a/x} s_5 \xrightarrow{b/y} s_2 \xrightarrow{c/null} s_2 \xrightarrow{b/x} s_3 \xrightarrow{a/y} s_5 \xrightarrow{b/y} s_2 \xrightarrow{a/y} s_1.$$

Algorithmes pour la technique des UIO

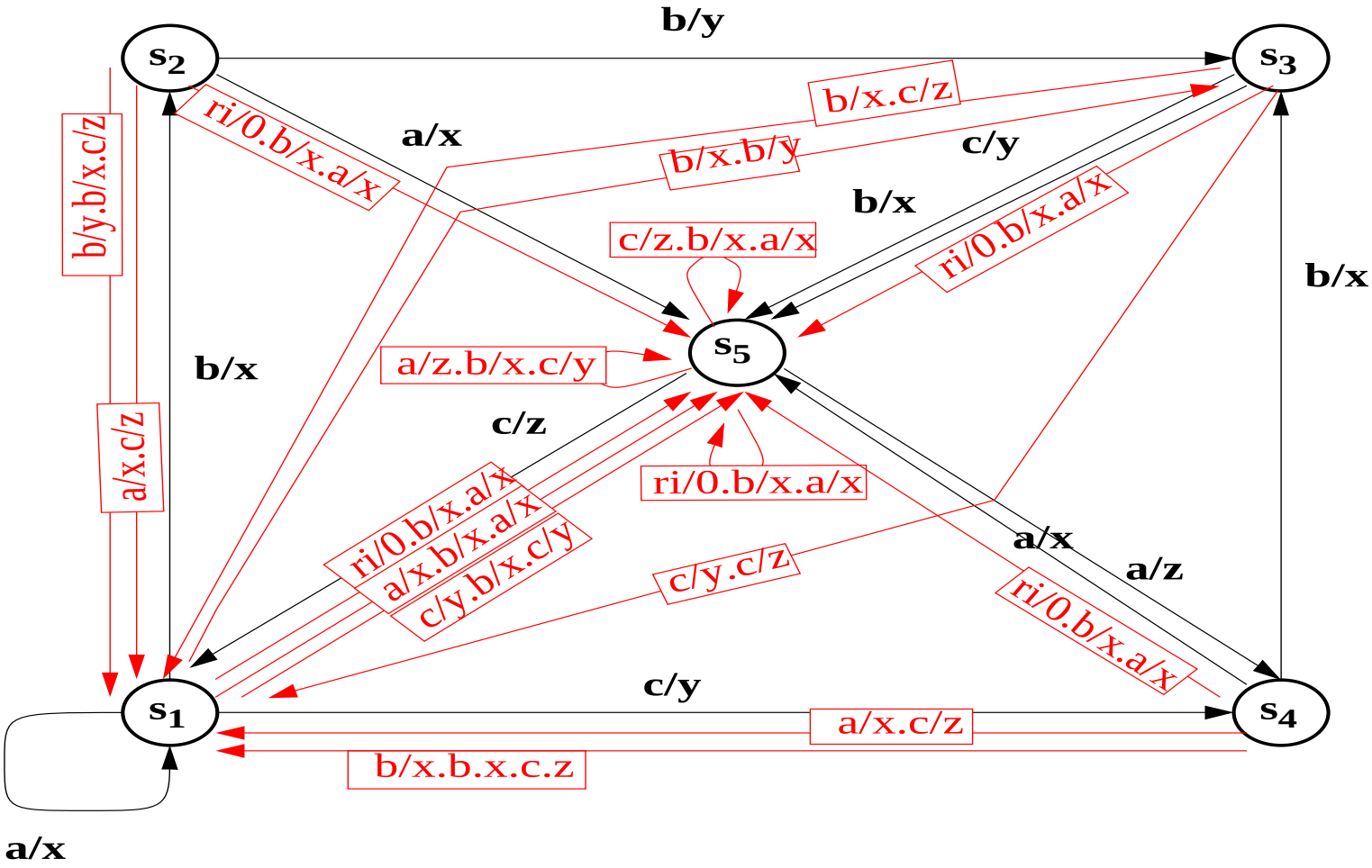


NB : on suppose de + ds chaque état s_i, une transition $s_i \xrightarrow{ri/0} s_1$.

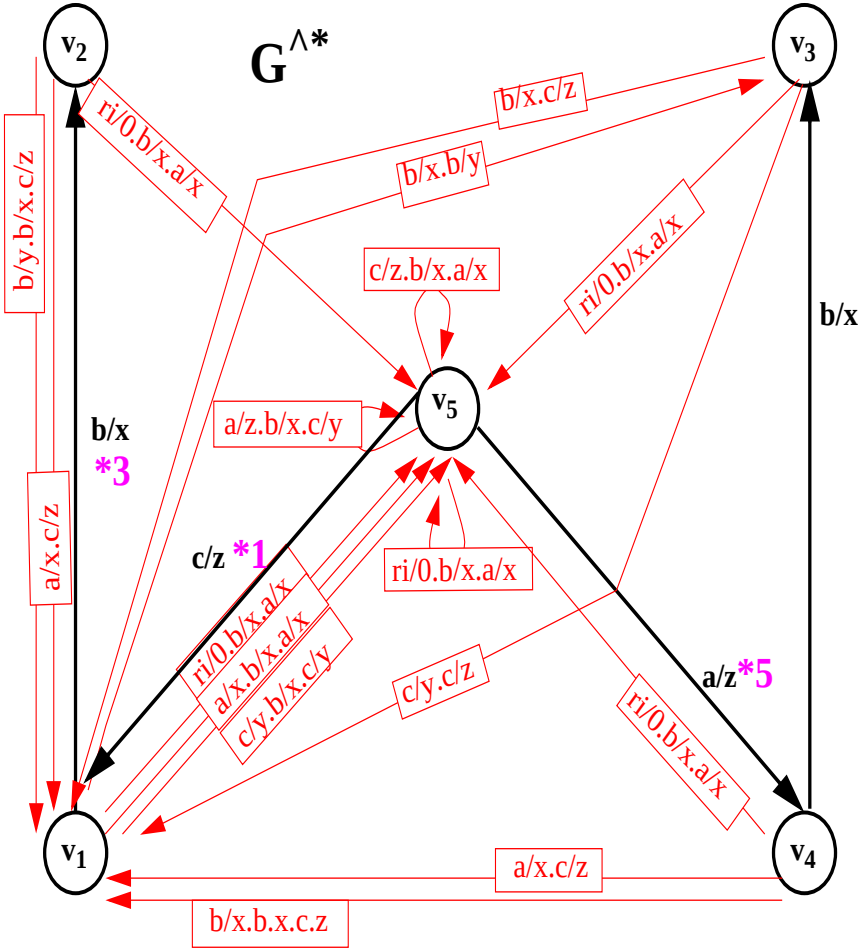
Calculer les UIO de chaque état : $UIO(s_1) = b/x.a/x$, $UIO(s_2) = b/y$, $UIO(s_3) = b/x.c/z$, $UIO(s_4) = b/x.c/y$, $UIO(s_5) = c/z$.

Test élémentaire : Test($s_i \xrightarrow{i/o} s_j$) $= i/o.UIO(s_j)$

Problème du postier chinois rural



Augmentation symétrique



Euler tour of G^*

Optimal test sequence:

ri/0 b/x.b/y c/y.c/z a/x.b/x.a/x

a/z.b/x.c/y c/z.b/x.a/x ri/0.b/x.a/x a/z

a/x.c/z c/y.b/x.c/y a/z b/x

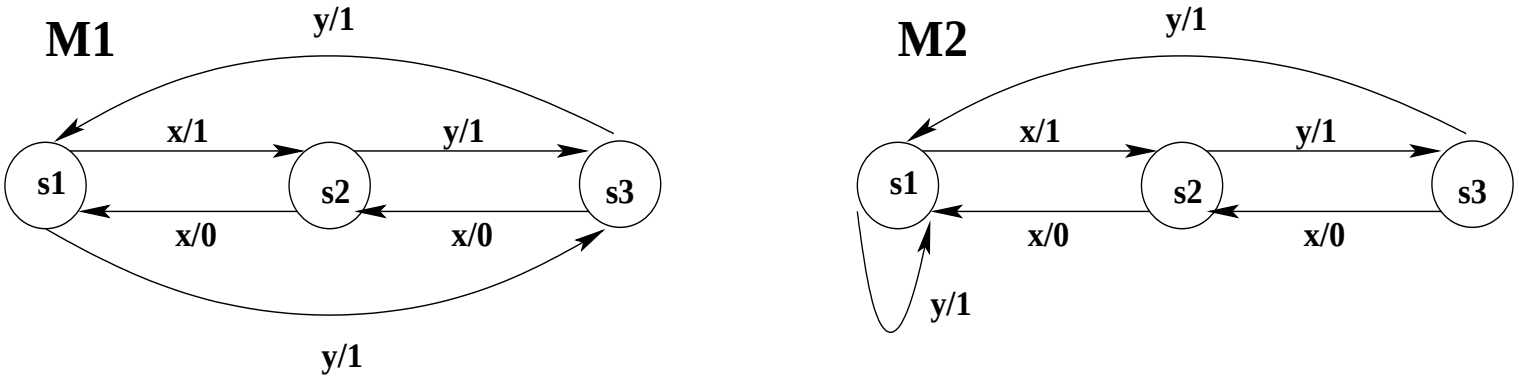
b/x.c/z b/x b/y.b/x.c/z b/x

a/x.c/z b/x ri/0.b/x.a/x a/z

b/x ri/0.b/x.a/x a/z b/x.b/x.c/z

ri/0.b/x.a/x a/z ri/0.b/x.a/x c/z.

Pb d'incomplétude d'UIO



$UIO(s_3) = y/1.x/1$ dans M_1 donc $Test(s_1 \xrightarrow{y/1} s_3) = y/1.y/1.x/1.$
$\delta^*(s_1, y.y.x) = 1.1.1$ aussi dans M_2

alors que M_2 est non conforme à M_1 sur la transition $s_1 \xrightarrow{y/1} s_3$:

faute de transfert $s_1 \xrightarrow{y/1} s_1$

D'où vient le problème ?

Quelle est la solution ?

Analyse

- Suite de test complète pour le modèle de faute (sous hypothèses)
- Identification d'état

- Hypothèses fortes sur Spec et IUT
- Complexité

Extensions :
- Ajout d'états dans le modèle de faute,
- Automates non-déterministes,
- Automates étendus avec variables
- Calcul de séquences distinguantes d'ensembles d'états sur des EFSM (Petrenko).

II.2 Techniques fondées sur les systèmes de transitions

Origines :

- Modèles : systèmes de transitions étiquetés (LTS)
- Pré-ordres de test (Henessy, De Nicola), refusal testing (Philips), testeur canonique (Brinksma)

Pas utilisable directement pour la génération de tests.
Problème : pas de distinction entre observation et contrôle.

Nouveaux modèles : LTS avec distinction entre entrées et sorties (IOSM, IOTS, IOLTS) : Phalippou (CNET), Tretmans (U. Twente)

Outils : TVeda (CNET), TGV (Irisa/Verimag), TorX (U. Twente), TestComposer (Telelogic).

Plan de la partie

Modèle des IOLTS
- Définition,
- Comportements
- Blocages

Théorie du test
- Spécifications, implémentations
- Relation de conformité
- Cas de tests
- Propriétés des cas de tests

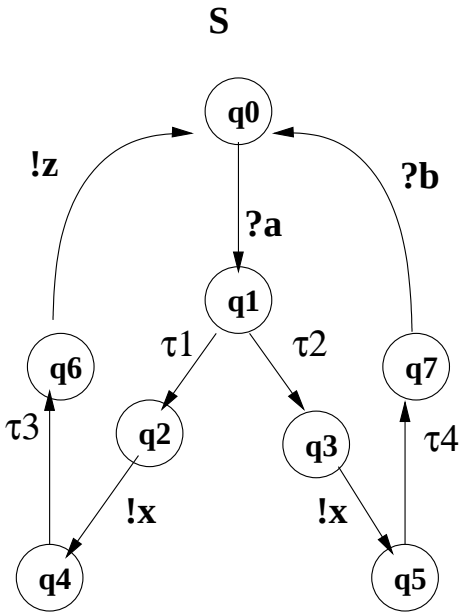
Génération de tests
- Génération aléatoire : algorithme et propriétés
- Génération guidée par un objectif : algorithme et propriétés

Modèle des IOLTS : système de transition (LTS) à entrées/sorties

$M = (Q, A, \rightarrow, Q_0)$ avec

- Q : ensemble dénombrable d'états,
- $\Lambda = \Lambda_? \cup \Lambda_! \cup \mathcal{T}$: alphabet d'actions, où
  $\Lambda_?$: entrées(?a), $\Lambda_!$: sorties (!x),
  $\mathcal{T}$: actions internes (τ_i),
  $\Lambda_{\text{VIS}} = \Lambda \setminus \mathcal{T} = \Lambda_? \cup \Lambda_!$: actions visibles
- $\rightarrow \subseteq Q \times \Lambda \times Q$: **relation** de transition,
- $Q_0 \subseteq Q$: ens. d'états initiaux.

Hyp : pas de divergence de τ.
i.e. pas de séquence infinie de τ
traversant une infinité d'états



Comportements des IOLTS

Soit $M = (Q, \Lambda, \rightarrow, Q_0)$ un IOLTS,

Runs : $\rho = q_0 \stackrel{\lambda_0}{\rightarrow} q_1 \stackrel{\lambda_1}{\rightarrow} \ldots \stackrel{\lambda_{n-1}}{\rightarrow} q_n$ tq $q_0 \in Q_0$ est un *run*.

$Runs(M) \subseteq Q_0.(\Lambda.Q)^*$: ens. des runs de M.

Langage : une séquence est la projection $\mu = proj_{\Lambda}(\rho) = \lambda_0.\lambda_1 \ldots \lambda_{n-1}$ d'un run ρ sur Λ.

$L(M) = proj_{\Lambda}(Runs(M)) \subseteq \Lambda^*$: langage généré par M.

Traces : projection $\sigma = proj_{\Lambda_{VIS}}(\rho) = a_1.a_2 \ldots a_k$ d'un run ρ sur Λ_{VIS}

$Traces(M) = proj_{\Lambda_{VIS}}(Runs(M)) \subseteq \Lambda^*_{VIS}$: ens. des traces de M.

IOLTS vus comme automates

Soit $M = (Q, \Lambda, \rightarrow, Q_0)$ un IOLTS,
et $X \subseteq Q$ un ensemble d'états marqués,
on interprètera parfois l'IOLTS M muni de X comme un automate :

run accepté en X : $\rho = q_0 \stackrel{\lambda_0}{\rightarrow} q_1 \stackrel{\lambda_1}{\rightarrow} \ldots \stackrel{\lambda_{n-1}}{\rightarrow} q_n \in Runs(M)$ tq $q_n \in X$.

séquence acceptée = projection sur Λ^* d'un run accepté.
trace acceptée = projection sur Λ^*_{VIS} d'un run accepté.

$\rightarrow$ $Runs_X(M) \subseteq Runs(M) = Runs_Q(M)$,
$\rightarrow$ $L_X(M) = proj_\Lambda(Runs_X(M)) \subseteq L(M) = L_Q(M)$,
$\rightarrow$ $Traces_X(M) = proj_{\Lambda_{\text{VIS}}}(Runs_X(M)) \subseteq Traces(M) = Traces_Q(M)$.

Relation $\Rightarrow$ et after

Soit $M = (Q, \Lambda, \rightarrow, Q_0)$ un IOLTS.
La sémantique de traces induit la relation $\Rightarrow \subseteq Q \times \Lambda^*_{\text{VIS}} \times Q$ définie par :

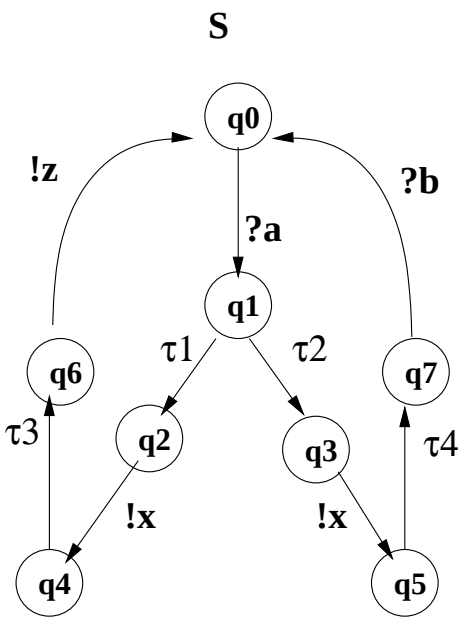
- $q \stackrel{\varepsilon}{\Rightarrow} q' \triangleq q = q'$ ou $\exists \tau_1, \tau_2 \ldots \tau_n \in \mathcal{T} \ : \ q \stackrel{\tau_1 . \tau_2 \cdots \tau_n}{\rightarrow} q'$
- $a \in \Lambda_{\text{VIS}}$, $q \stackrel{a}{\Rightarrow} q' \triangleq \exists q_1, q_2 \ : q \stackrel{\varepsilon}{\Rightarrow} q_1 \stackrel{a}{\rightarrow} q_2 \stackrel{\varepsilon}{\Rightarrow} q'$,
- $\sigma = a_1 \cdots a_n \in \Lambda^*_{\text{VIS}}$, $q \stackrel{\sigma}{\Rightarrow} q' \triangleq \exists q_0, \ldots q_n \ : q = q_0 \stackrel{a_1}{\Rightarrow} \cdots \stackrel{a_n}{\Rightarrow} q_n = q'$

Notation After

- q after $\sigma \triangleq \{q' \in Q \mid q \stackrel{\sigma}{\Rightarrow} q'\}$
- pour $P \subseteq Q$, P after $\sigma \triangleq \bigcup_{q \in P} q$ after σ

On note M after $\sigma \triangleq Q_0$ after σ l'ensemble des états où M peut résider après l'observation de σ depuis un état initial.

Illustration



$q_1 \stackrel{\varepsilon}{\Rightarrow} q_2$ et $q_1 \stackrel{\varepsilon}{\Rightarrow} q_3$
$q_1 \stackrel{!x}{\Rightarrow} q_4$, $q_1 \stackrel{!x}{\Rightarrow} q_6$, $q_1 \stackrel{!x}{\Rightarrow} q_5$, $q_1 \stackrel{!x}{\Rightarrow} q_7$,
$q_1 \stackrel{!x.!z}{\Rightarrow} q_0$

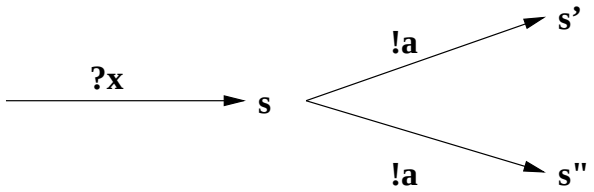
q_0 after $?a.!x = \{q_4, q_5, q_6, q_7\}$,
q_0 after $?a.!z = \emptyset$

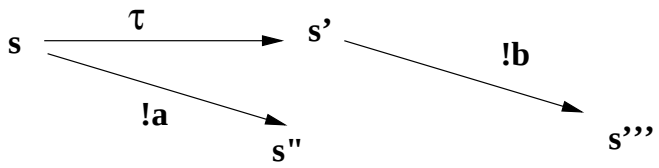
$\{q_2, q_3\}$ after $!x = \{q_4, q_5, q_6, q_7\}$

$Traces(S) = \{\varepsilon, ?a, ?a.!x, ?a.!x.!z, ?a.!x.?b, \ldots\}$

Non-déterminisme et choix

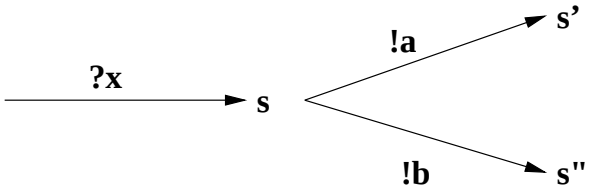
Non-déterminisme au sens des automates : existence d'actions internes τ et $\rightarrow$ est une relation





Déf : M est déterministe s'il n'a aucune action interne, $Card(Q_0) = 1$ et $\forall q \in Q, \forall a \in \Lambda_{\text{VIS}}$, $Card(\{q' \mid q \xrightarrow{a} q'\}) \leq 1$.

Choix non contrôlé : parfois appelé *non-déterminisme observable*



Déterminisation d'IOLTS

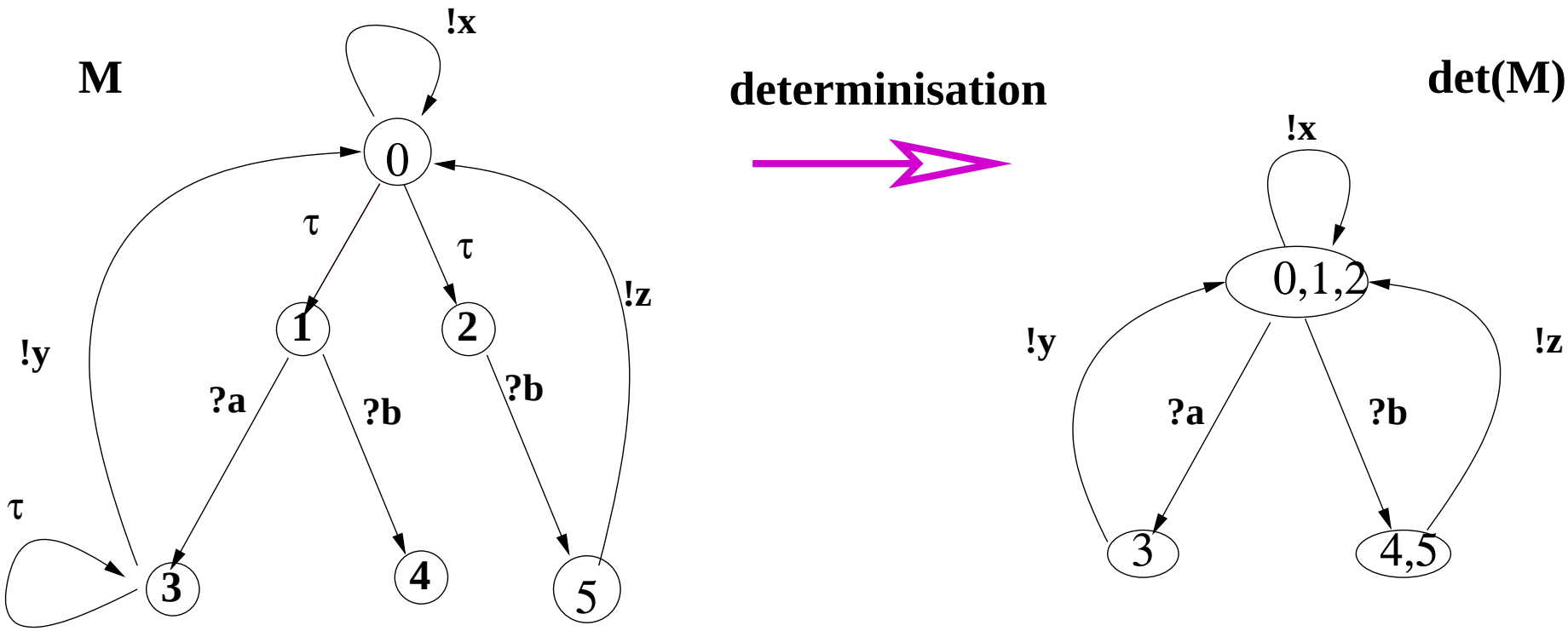
Pour un IOLTS $M = (Q, \Lambda, \rightarrow, Q_0)$, on peut construire un IOLTS déterministe $det(M)$ et de mêmes traces que M.

$det(M) = (2^Q, \Lambda_{\text{VIS}}, , \rightarrow_{det}, Q_0 \text{ after } \varepsilon)$ où

- $2^Q = \mathcal{P}(Q)$ est l'ensemble des parties de Q
- pour $P, P' \in 2^Q$ et $a \in \Lambda^{\text{VIS}}$, $P \xrightarrow{a}_{\text{det}} P'$ ssi $P' = P \text{ after } a$

$$Traces(M) = Traces(det(M))$$

Déterminisation : exemple



IOLTS complets

Soit $M = (Q, \Lambda_{\text{VIS}} \cup \mathcal{T}, \rightarrow, Q_0)$ un IOLTS, $q \in Q$ un état de M et $A \subseteq \Lambda_{\text{VIS}}$ un sous-alphabet d'actions visibles.

q est fortement A-complet si toute action de A est tirable en q :
$\forall \lambda \in A, q \xrightarrow{\lambda}$

q est faiblement A-complet si toute action de A est tirable après un nombre quelconque d'action interne :
$\forall \lambda \in A, q \stackrel{\lambda}{\Rightarrow}$

M (faiblement/fortement) A-complet si tous les états de Q sont A-complets

Prop : M déterministe $\Rightarrow$ fortement $=$ faiblement

Prop : M déterminisme et Λ_{VIS}-complet $\Rightarrow$ $Traces(M) = \Lambda^*_{\text{VIS}}$

Spécification et implémentations

Spécification : $S = (Q^{\mathsf{S}}, \Lambda, \rightarrow_{\mathsf{S}}, q_0^{\mathsf{S}})$, avec $\Lambda = \Lambda_? \cup \Lambda_! \cup \mathcal{T}^{\mathsf{S}}$

Implémentation : $I = (Q^{\mathsf{I}}, \Lambda^{\mathsf{I}}, \rightarrow_{\mathsf{I}}, q_0^{\mathsf{I}})$ avec $\Lambda^{\mathsf{I}} = \Lambda_? \cup \Lambda_! \cup \mathcal{T}^{\mathsf{I}}$

L'implémentation est inconnue, sauf son interface, identique à S

$\Rightarrow$ mêmes actions visibles : $\Lambda_{\mathsf{VIS}} = \Lambda_! \cup \Lambda_?$,

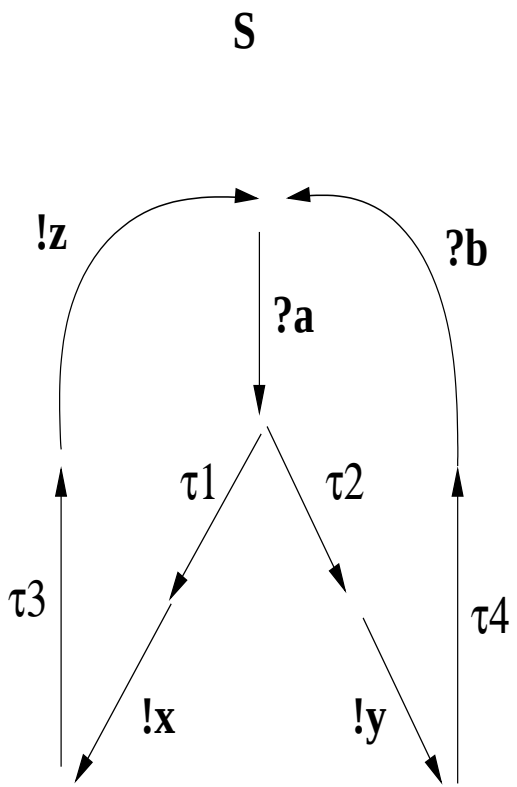
actions internes *a priori* différentes

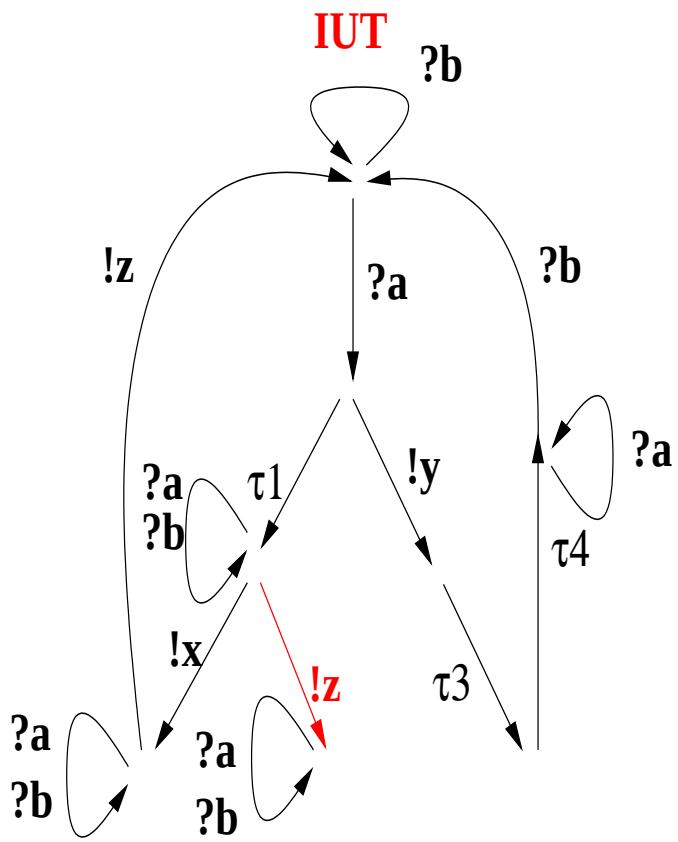
Hyp. : I est faiblement $\Lambda_?$-complète : $\forall q \in Q^{\mathsf{I}}, \forall a \in \Lambda_?, q \stackrel{a}{\Rightarrow}$.

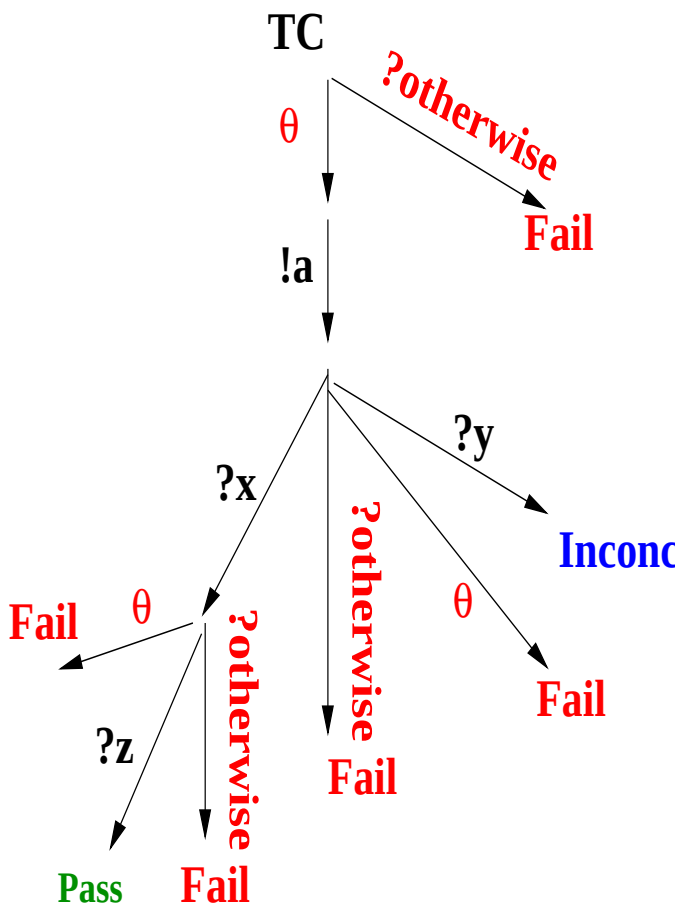
Justification : dans tous ses états, I accepte toute entrée, après d'éventuelles actions internes, quitte à renvoyer une erreur.

NB : l'hypothèse ne sera utile que pour éviter les blocages dans la modélisation de l'interaction testeur/IUT.

Spécification et implémentation : exemple

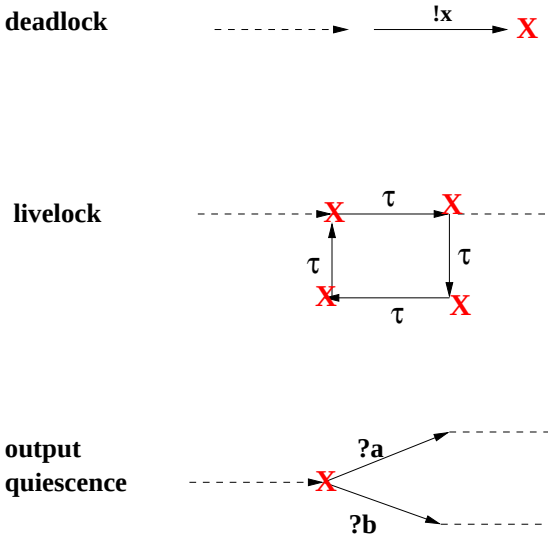






Observation des blocages

En pratique, le test peut observer les traces de l'IUT, mais aussi ses blocages grâce à des temporisateurs.
Seuls les blocages de l'IUT non spécifiés par S doivent être rejetés.



Notation : $\Gamma(q) \triangleq \{a \in \Lambda \mid q \xrightarrow{a}\}$

$q \in deadlock(M) \triangleq \Gamma(q) = \emptyset$

$q \in livelock(M) \triangleq \exists \tau_1, \ldots, \tau_n, q \xrightarrow{\tau_1 \cdots \tau_n} q$

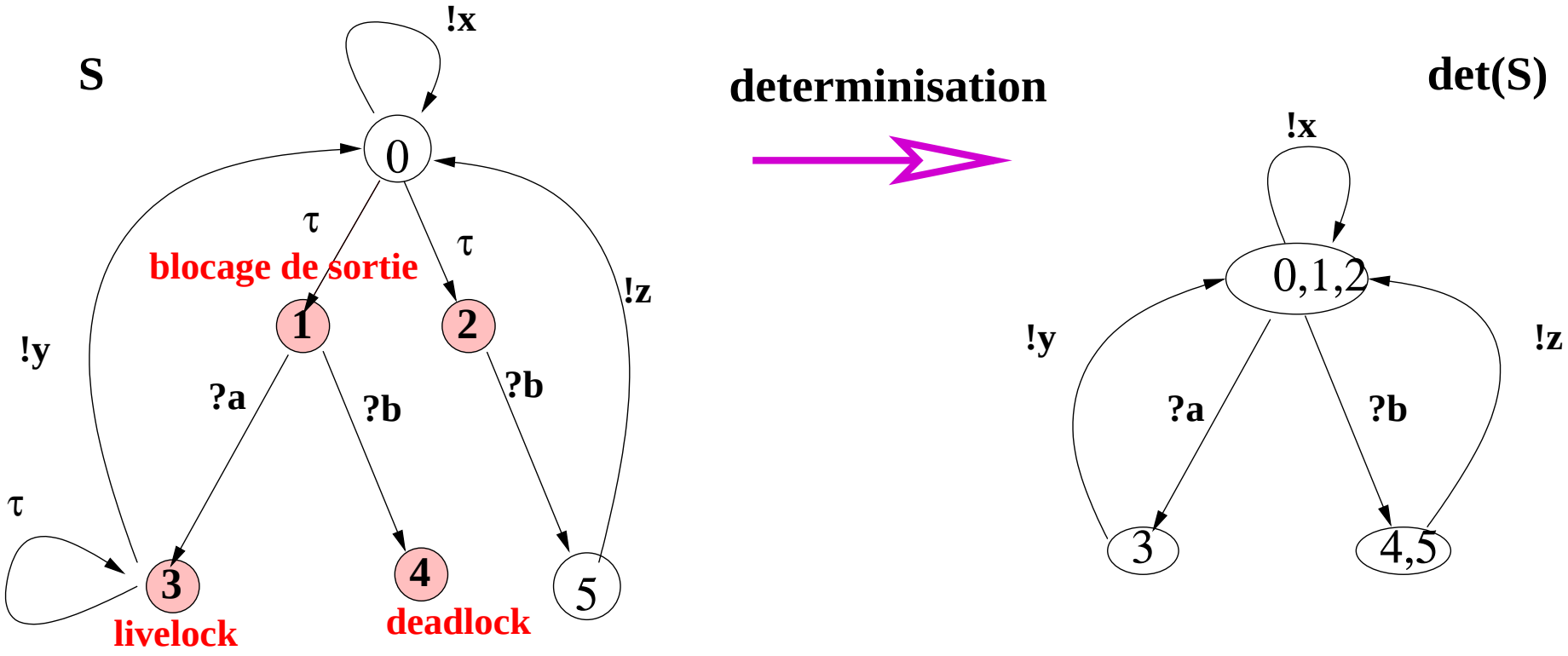
$q \in outputlock(M) \triangleq \Gamma(q) \subseteq \Lambda_?$

$$quiescent(M) = deadlock(M) \cup livelock(M) \cup outputlock(M)$$

Perte des blocages par déterminisation

L'équivalence de traces (la déterminisation) ne préserve pas les blocages.

$\Rightarrow$ il faut expliciter les blocages avant déterminisation.

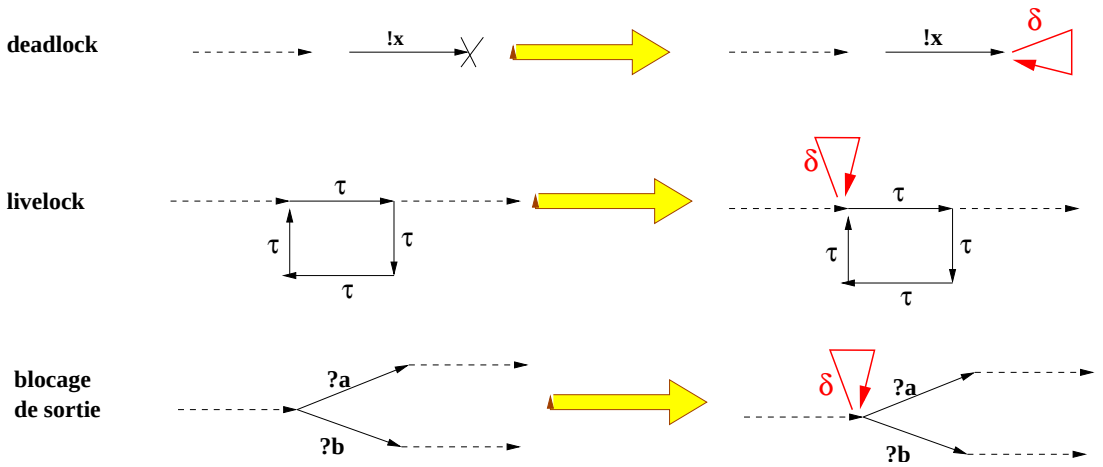


Explicitation des blocages

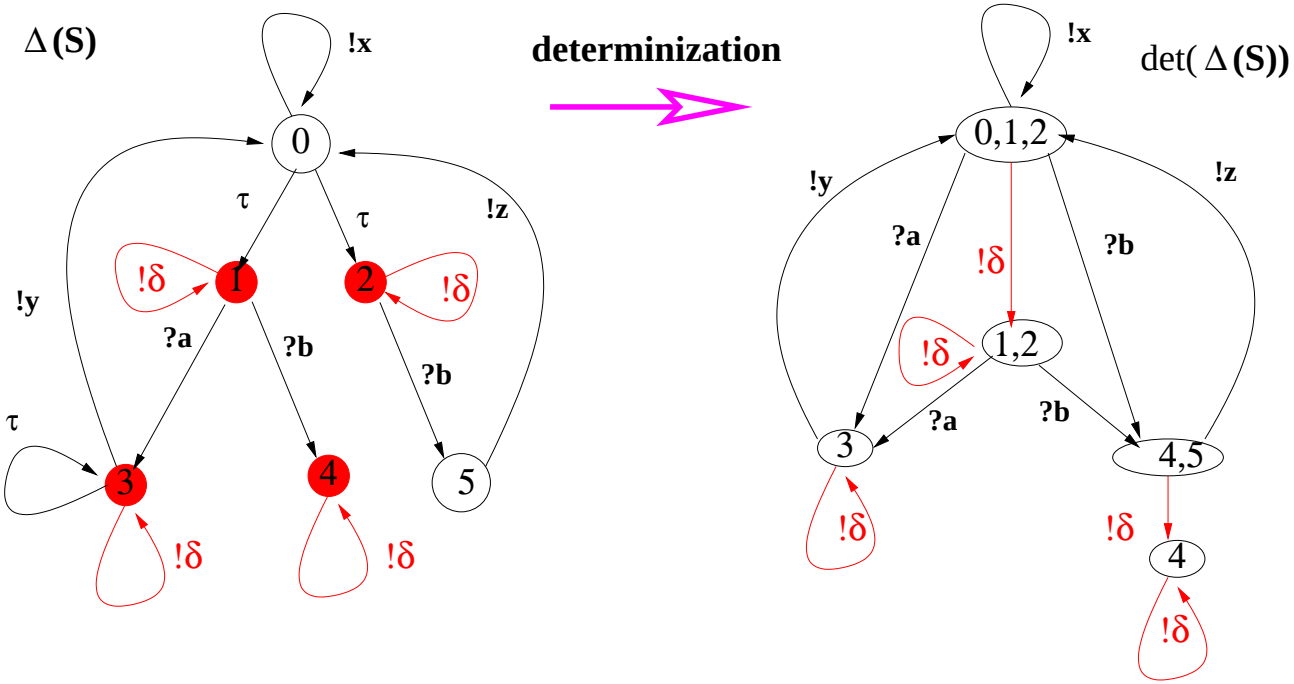
Soit $M = (Q, \Lambda, \rightarrow, q_0)$ avec $\Lambda = \Lambda_! \cup \Lambda_? \cup \mathcal{T}$

L'IOLTS suspendu de M est l'IOLTS $\Delta(M) = (Q, \Lambda^\delta, \rightarrow_\delta, q_0)$ avec

- $\Lambda^\delta = \Lambda \cup \{\delta\} = \Lambda_! \cup \{\delta\} \cup \Lambda_? \cup \mathcal{T}$ et $\Lambda_!^\delta = \Lambda_! \cup \{\delta\}$
  δ est considéré comme une sortie (car observable)
- $\rightarrow_\delta = \rightarrow \cup \{(q, \delta, q) \mid q \in quiescent(M)\}$



Traces suspendues



Traces suspendues : $STraces(M) \triangleq Traces(\Delta(M)) = Traces(det(\Delta(M)))$

$STraces(S)$ et $STraces(I)$ représentent exactement les comportements visibles de S et $I \Rightarrow$ servent de base à la définition de la conformité.

Relation de conformité

La relation de conformité définit l'ensemble des IUT I conformes à S.

Définition : Soient $S = (Q^{\mathsf{S}}, \Lambda_! \cup \Lambda_? \cup \mathcal{T}^{\mathsf{S}}, \rightarrow_{\mathsf{S}}, q_0^{\mathsf{S}})$ une spécification et $I = (Q^{\mathsf{I}}, \Lambda_! \cup \Lambda_? \cup \mathcal{T}^{\mathsf{I}}, \rightarrow_{\mathsf{I}}, q_0^{\mathsf{I}})$ une implémentation de même interface

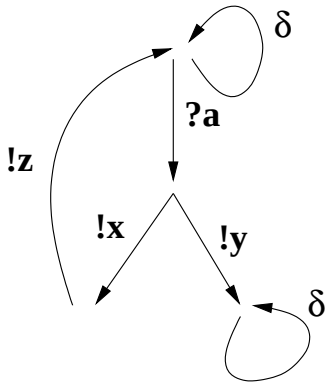
$$I \textbf{ ioco } S$$
$$\triangleq$$
$$\forall \sigma \in STraces(S), Out(\Delta(IUT) \text{ after } \sigma) \subseteq Out(\Delta(S) \text{ after } \sigma)$$

où $Out(P) \triangleq \Gamma(P) \cap \Lambda_!^{\delta}$ est l'ens. des sorties et blocages en P

Intuition : I est conforme à sa spécification S si et seulement si, après toute trace suspendue de S, toutes les sorties et blocages de I sont autorisés dans S.

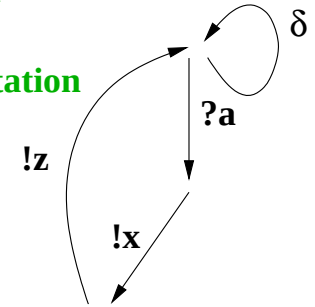
ioco par l'exemple

Specification



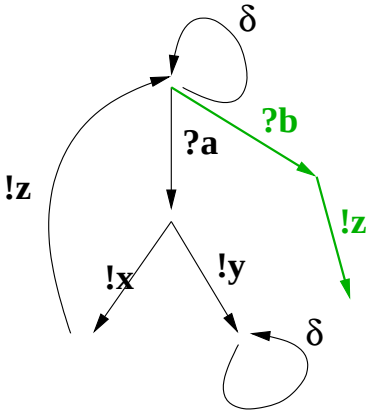
IUT1

choix d'implementation



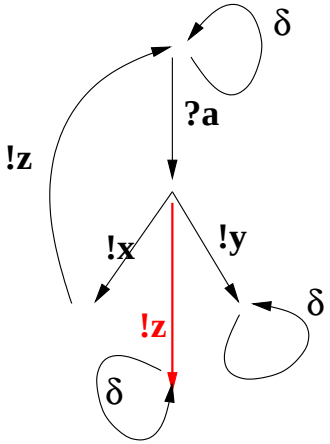
IUT2

specification partielle



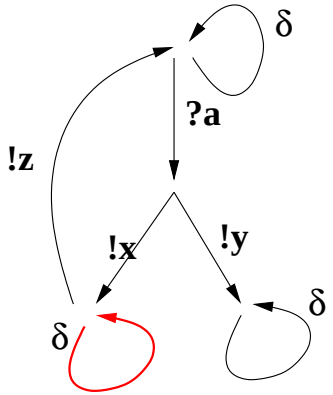
IUT3

sortie interdite



IUT4

blocage interdit



Caractérisation de ioco en termes de $STraces$

$$I \textbf{ ioco } S \iff STraces(I) \cap [STraces(S).\Lambda_!^\delta] \subseteq STraces(S) \quad (1)$$
$$\iff STraces(I) \cap [STraces(S).\Lambda_!^\delta \setminus STraces(S)] = \emptyset \quad (2)$$

$STraces(I)$= comportements visibles de I
$STraces(S)$ = comportements visibles de S
$STraces(S).\Lambda_!^\delta$ = comportements visibles de S prolongés par sortie ou δ.

(1) : les $STraces$ de I prolongeant les $STraces$ de S par des sorties ou blocages doivent rester des $STraces$ de S.

(2) : I n'a aucune STrace qui soit une STrace de S prolongée par une sortie ou un blocage sans être une STrace de S.

Observateur de non-conformité / S

Soit l'observateur $Can(S) = (Q^c, \Lambda^{\delta}_{\mathsf{VIS}}, \rightarrow_c, q_0^c,)$ muni de $Fail \in Q_c$
construit depuis $det(\Delta(S)) = (Q^d, \Lambda^{\delta}_{\mathsf{VIS}}, \rightarrow^d, q_0^d)$ par :

- Ajout d'un nouvel état $Fail$:
  $Q_c \triangleq Q_d \cup \{Fail\}$, $Fail \notin Q_d$
  $q_0^d \triangleq q_0^c$
- Complétion en sortie vers $Fail$:
  $\rightarrow_c \triangleq \rightarrow_d \cup \{q \xrightarrow{!a}_c Fail \mid q \in Q_d, !a \in \Lambda^{\delta}_{!} \wedge \neg(q \xrightarrow{!a}_d)\}$
  On notera $q \xrightarrow{!othw}_c Fail$.

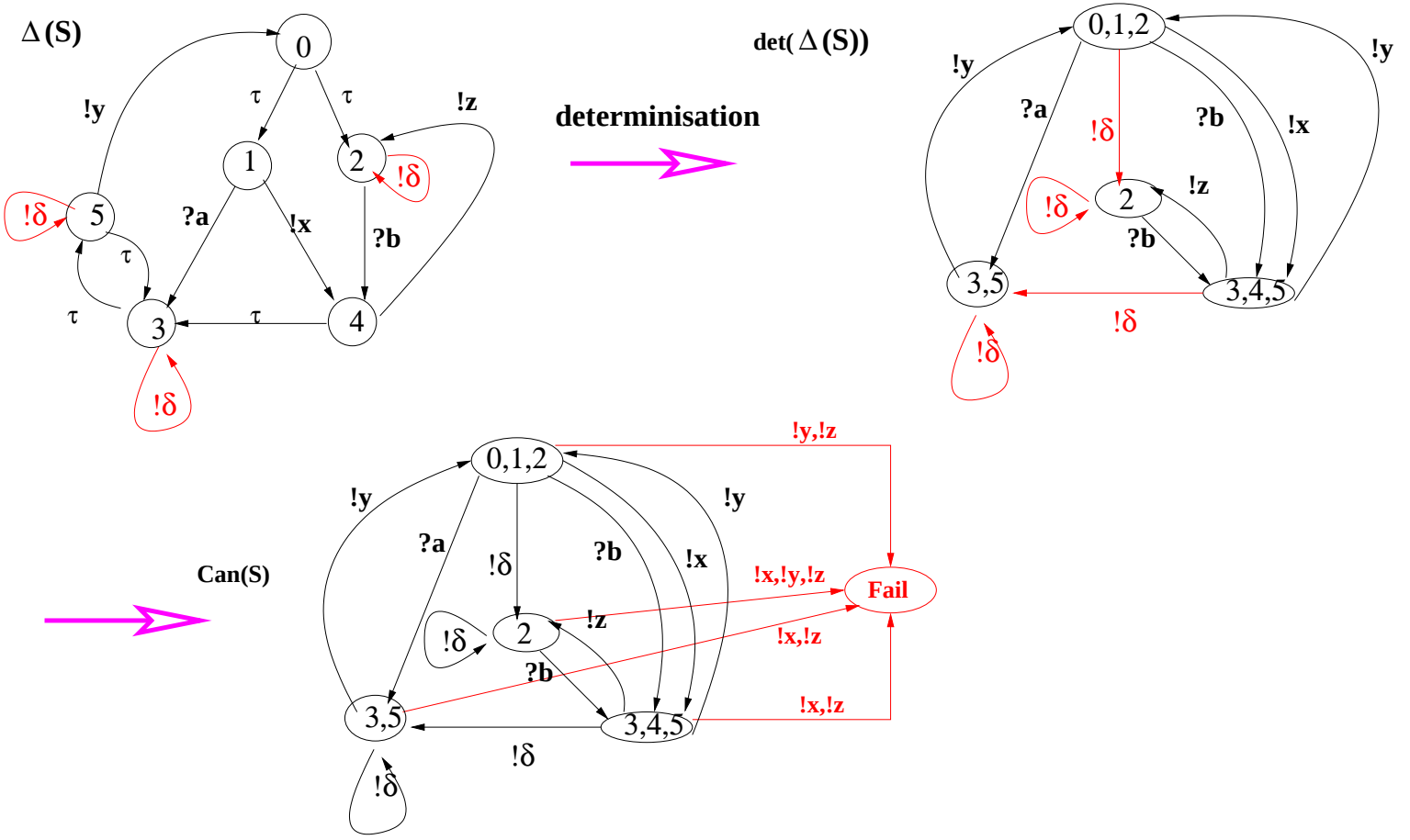
$Can(S)$ reconnait le language $Traces_{Fail}(Can(S)) = STraces(S).\Lambda^{\delta}_{!} \setminus STraces(S)$

Prop : $Can(S)$ est un observateur de non-conformité i.e.

$$I \textbf{ ioco } S \iff STraces(I) \cap Traces_{\mathsf{Fail}}(Can(S)) = \emptyset$$

NB : Pour S donné, I **ioco** S est une propriété de sûreté sur $STraces(I)$.
$Can(S)$ est l'observateur de la négation de cette propriété.

Observateur de non-conformité : exemple



Cas de test

Un cas de test pour S est un IOLTS $TC = (Q^{TC}, \Lambda^{TC}, \rightarrow_{TC}, q_0^{TC})$ tq :

- TC est déterministe
- $\Lambda_!^{TC} = \Lambda_?$ et $\Lambda_?^{TC} = \Lambda_!^{\delta} = \Lambda_! \cup \{\delta\}$ (inversion entrées/sorties)
- TC est équipé de 3 ensembles d'états puits (les seuls états puits) représentant les verdicts :
  Verdicts = **Pass** $\cup$ **Fail** $\cup$ **Inconc** $\subseteq Q^{TC}$
  Etat Puits : q tel que $\forall a \in \Lambda^{TC}, \neg(q \xrightarrow{a})$
- les états de TC, excepté **Verdicts**, sont $\Lambda_?^{TC}$-complets
  i.e. TC est prêt à recevoir toute entrée de $\Lambda_?^{TC}$ = sortie de $\Lambda_!^{\delta}$
  $?otherwise$ dénotera le complémentaire d'un ensemble d'entrées.

$\rightarrow Traces(TC)$,
$Traces_{\textbf{Verdict}}(TC) = Traces_{\text{Fail}}(TC) \cup Traces_{\text{Pass}}(TC) \cup Traces_{\text{Inconc}}(TC)$

Exécution des tests

Modélisé par la composition parallèle $TC \| \Delta(I)$ avec synchronisation sur les actions de l'interface commune $\Lambda^{\delta}_{\mathsf{VIS}}$:

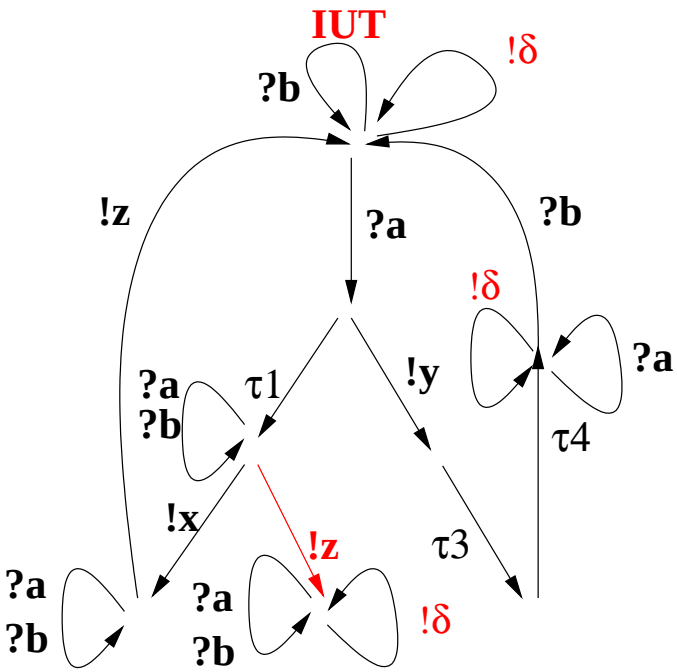
$TC \| \Delta(I) = (Q^{\mathsf{TC}} \times Q^{\mathsf{I}}, \Lambda^{\mathsf{I}}, \rightarrow_{TC \| \Delta(I)}, (q_0^{\mathsf{TC}}, q_0^{\mathsf{I}}))$
où $\rightarrow_{TC \| \Delta(I)}$ est défini par :

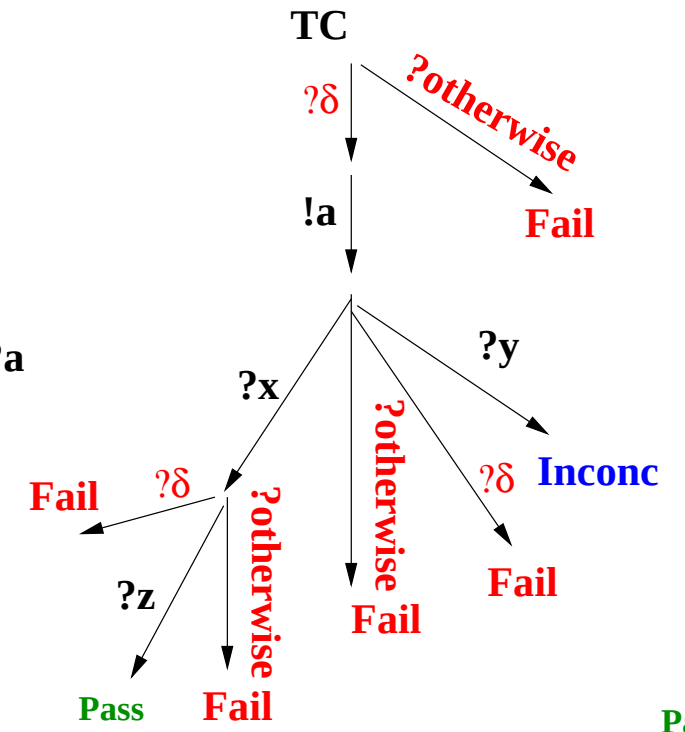
$$\frac{tc \xrightarrow{a}_{\mathsf{TC}} tc' \quad q \xrightarrow{a}_{\Delta(I)} q' \quad a \in \Lambda^{\delta}_{\mathsf{VIS}}}{(tc, q) \xrightarrow{a}_{TC \| \Delta(I)} (tc', q')} \qquad \frac{q \xrightarrow{\tau}_{\Delta(I)} q' \quad \tau \in \mathcal{T}^{\mathsf{I}}}{(tc, q) \xrightarrow{\tau}_{TC \| \Delta(I)} (tc, q')}$$

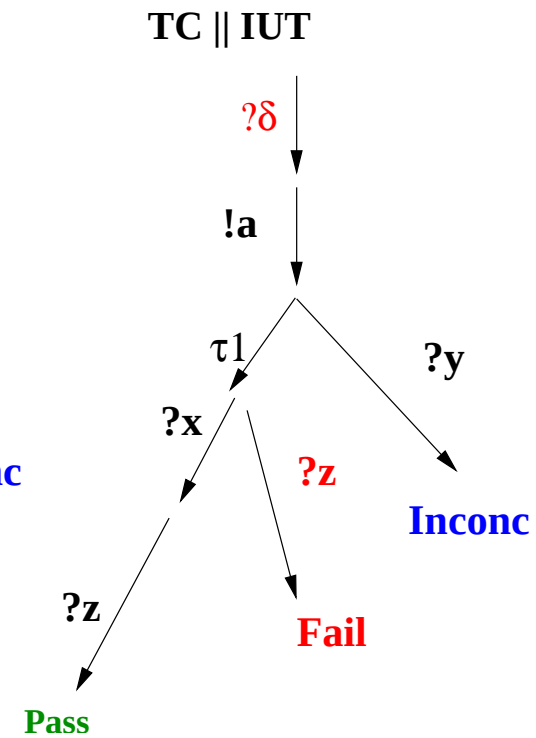
Prop : $Traces(TC \| \Delta(I)) = Traces(TC) \cap STraces(I)$

Prop : I est faiblement $\Lambda_?$-complet et TC est $\Lambda^{\delta}_!$-complet (hors **Verdicts**)
$\Rightarrow TC \| \Delta(I)$ n'est bloquant que dans les états de **Verdicts** de TC

Exemple


IUT
?b
!δ
!z
?a
?b
!δ
?a
?a
?b
τ1
!y
τ4
!x
!z
τ3
?a
?b
?a
?b
!δ


TC
?δ
?otherwise
!a
Fail
?y
?x
?otherwise
Fail
?δ
Inconc
Fail
?δ
?otherwise
?z
Fail
Pass
Fail


TC || IUT
?δ
!a
τ1
?y
?x
?z
Inconc
?z
Fail
Pass

Observations et verdicts

Observations de TC sur I $\triangleq$ Traces maximales finies de $TC \| \Delta(I)$
i.e. traces finissant dans **Verdicts**.

$$
\begin{aligned}
Mtraces(TC\|\Delta(I)) &\triangleq \{\sigma \in {\Lambda^{\delta}_{VIS}}^{*} \mid TC\|\Delta(I) \text{ after } \sigma \subseteq \textbf{Verdicts} \times Q^{!}\} \\
&= STraces(I) \cap Traces_{\textbf{Verdict}}(TC) \\
&= STraces(I) \cap \\
&\quad (Traces_{\text{Fail}}(TC) \cup Traces_{\text{Pass}}(TC) \cup Traces_{\text{Inconc}}(TC))
\end{aligned}
$$

Soit $\sigma \in Mtraces(TC\|\Delta(I))$ une observation de TC sur I

$$
\begin{aligned}
verdict(\sigma) = Fail &\triangleq \sigma \in Traces_{\text{Fail}}(TC) \\
verdict(\sigma) = Pass &\triangleq \sigma \in Traces_{\text{Pass}}(TC) \\
verdict(\sigma) = Inconc &\triangleq \sigma \in Traces_{\text{Inconc}}(TC)
\end{aligned}
$$

Verdicts

On dira qu'un cas de test TC rejette (*fails*) I ssi une exécution maximale de $TC \| \Delta(I)$ produit **Fail**

Ceci exprime seulement une *possibilité* de rejet.

$$\begin{array}{rcl} TC \text{ fails } I & \triangleq & \exists\, \sigma \in Mtraces(TC \| \Delta(I)),\ verdict(\sigma) = fail \\ & \Longleftrightarrow & STraces(I) \cap Traces_{\text{Fail}}(TC) \neq \emptyset \end{array}$$

! À cause des choix non contrôlables de I, un même cas de test peut produire différents verdicts sur la même IUT !

Voir exple précédent où $STraces(I) \cap Traces_{\text{Fail}}(TC) \neq \emptyset$

mais aussi $STraces(I) \cap Traces_{\text{Pass}}(TC) \neq \emptyset$

et $STraces(I) \cap Traces_{\text{Inconc}}(TC) \neq \emptyset$

Propriétés des suites de tests (I)

Correction d'un cas de test/ d'une suite de test :]
Un cas de test TC est correct pour S et **ioco** si il ne peut rejeter que des IUT non-conformes :

$$TC \text{ correct/}\mathbf{ioco}, S \quad \triangleq \quad \forall\, I, [TC \text{ fails } I \Rightarrow \neg(I \ \mathbf{ioco}\ S)]$$

$$\iff \quad \forall\, I, [I \ \mathbf{ioco}\ S \Rightarrow \neg(TC \text{ fails } I)]$$

Une suite de test TS est correcte si tous ses cas de tests sont corrects.

$$TS \text{ correcte/}\mathbf{ioco}, S \triangleq \forall TC \in TS, TC \text{ correct/}\mathbf{ioco}, S$$

Condition nécessaire et suffisante de correction

On rappelle que

- I **ioco** $S \iff STraces(I) \cap Traces_{\text{Fail}}(Can(S)) = \emptyset$
- TC fails $I \iff STraces(I) \cap Traces_{\text{Fail}}(TC) \neq \emptyset$
- TC correct/**ioco**, $S \iff \forall\, I, [I$ **ioco** $S \Rightarrow \neg(TC$ fails $I)$

D'où la condition nécessaire et suffisante de correction :

Prop : TC correct/**ioco**, $S \iff Traces_{\text{Fail}}(TC) \subseteq Traces_{\text{Fail}}(Can(S))$

TS correcte/**ioco**, $S \iff \bigcup_{TC \in TS} Traces_{\text{Fail}}(TC) \subseteq Traces_{\text{Fail}}(Can(S))$

Propriétés des suites de tests (II)

Exhaustivité : une suite de test TS est exhaustive pour S et **ioco** si pour toute IUT non-conforme il existe un cas de test de TS qui **peut la rejeter** :

$$TS \text{ exhaustive} \quad \triangleq \quad \forall\, I, [\neg(I \textbf{ ioco } S) \Rightarrow [\exists\, TC \in TS, TC \text{ fails } I]]$$

$$\iff \quad \forall\, I, [[\forall\, TC \in TS, \neg(TC \text{ fails } I)] \Rightarrow I \textbf{ ioco } S]$$

i.e. pour toute I, si aucun TC ne peut rejetter I,
alors I est conforme à S

Prop : TS exhaustive/**ioco**, $S \iff$
$Traces_{\text{Fail}}(Can(S)) \subseteq \bigcup_{TC \in TS} Traces_{\text{Fail}}(TC)$

Testeur canonique et CNS correction et exhaustivité

TS correcte et exhaustive/**ioco**, $S \iff$
$\bigcup_{TC \in TS} Traces_{\text{Fail}}(TC) = Traces_{\text{Fail}}(Can(S))$

En particulier $TS = \{Can(S)\}$ est correcte et exhaustive.

$$\begin{array}{lcl} I \textbf{ ioco } S & \iff & STraces(I) \cap Traces_{Fail}(Can(S)) = \emptyset \text{ (def de } \textbf{ioco}) \\ & \iff & \neg(Can(S) \text{ fails } I) \text{ (def. de fails)} \end{array}$$

$Can(S)$ est un testeur canonique pour S et **ioco**
i.e. le cas de test le plus général.

Pb : $Can(S)$ a trop de comportements $\Rightarrow$ sélection.

Sélection des cas de test

But : Trouver un algorithme qui, pour **ioco** et pour S donné, produise une suite de test TS qui soit

- correcte (facile à partir de $Can(S)$)
- exhaustive à la limite i.e. en considérant la suite infinie de tous les tests pouvant être produits

Deux techniques :

1. Sélection non-déterministe (à la TorX)

2. Sélection guidée par un objectif de test (à la TGV)

Sélection non-déterministe : vue simplifiée

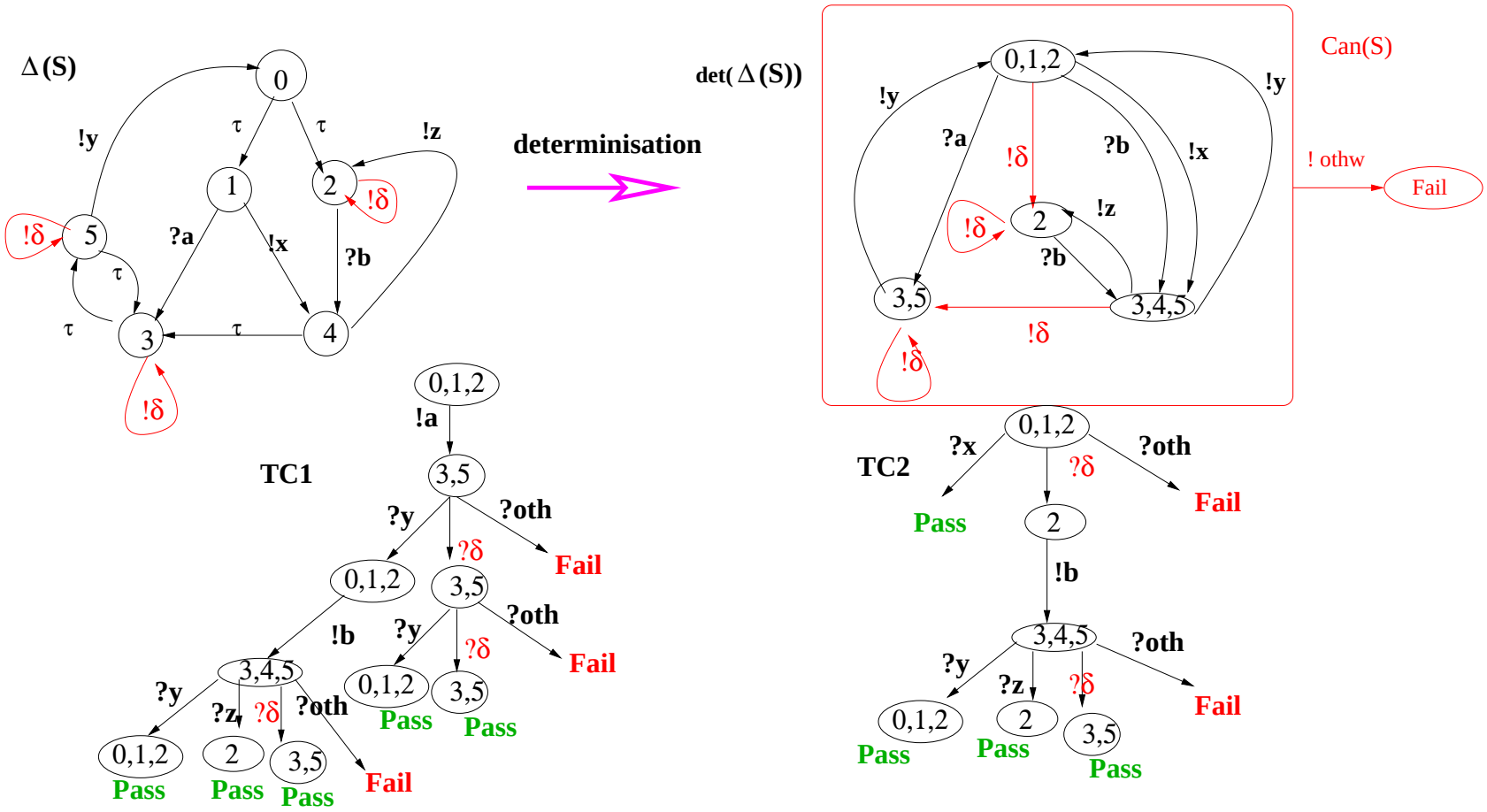
Remarques

1. La suite $TS = \{Can(S)\}$ est correcte et exhaustive
2. Après toute trace σ dans $Can(S)$, on peut faire un choix entre
   - s'arrêter et produire Pass,
   - observer une émission de $\Delta(I)$ parmi toutes les émissions (correctes ou non) de $Can(S)$ after σ
   - choisir une réception de $\Delta(I)$ parmi celles de $Can(S)$ after σ.

Conséquence :

Soit TS l'ensemble *infini* de tous les TC obyenus par dépliage *fini* (arrêt sur $Pass$) et *contrôlable* (choix 1 entrée, toutes les sorties) de $Can(S)$.
TS est correcte et exhaustive.

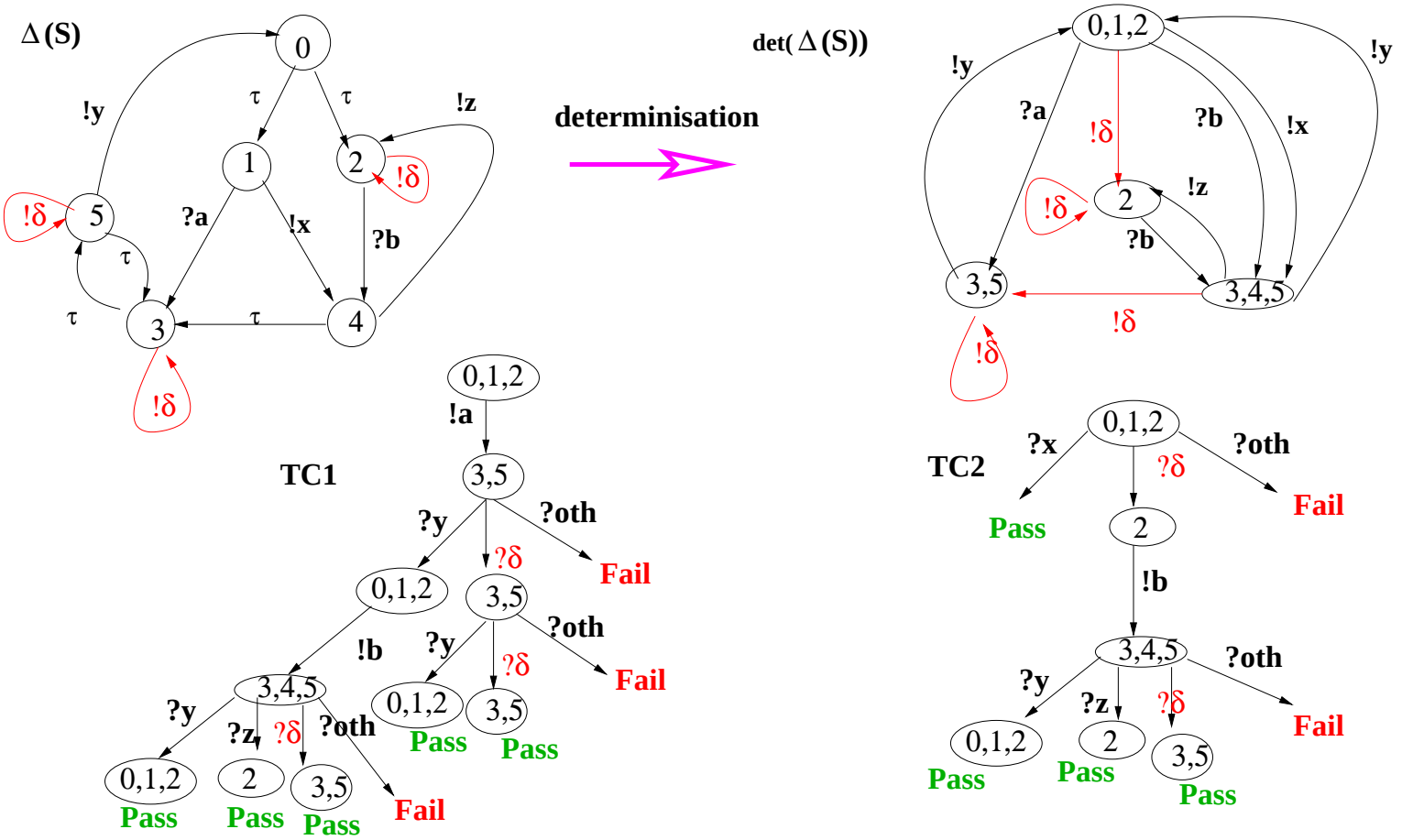
Exemple


Δ(S)
0
!y
τ
τ
!z
1
2
!δ
!δ
5
?a
!x
?b
τ
τ
τ
3
4
!δ
determinisation
det(Δ(S))
0,1,2
!y
?a
!δ
?b
!x
!y
Can(S)
! othw
Fail
2
!δ
!z
?b
3,5
3,4,5
!δ
!δ
TC1
0,1,2
!a
3,5
?y
?δ
?oth
Fail
0,1,2
3,5
!b
?y
?δ
?oth
Fail
3,4,5
?y
?z
?δ
?oth
0,1,2
3,5
Pass
Pass
0,1,2
2
3,5
Pass
Pass
Pass
Fail
TC2
0,1,2
?x
?δ
?oth
Fail
Pass
2
!b
3,4,5
?y
?z
?δ
?oth
Fail
0,1,2
2
3,5
Pass
Pass
Pass

Sélection non-déterministe : algo. original

- $P_0 = q_0^{\Delta(S)}$ after $_{\Delta(S)}\ \varepsilon = \{$ états accessibles par τ depuis $q_0^{\Delta(S)}\}$
  $gen\_test(P_0)$ avec
- $gen\_test(P)$ pour $P = q_0^{\Delta(S)}$ after $_{\Delta(S)}\ \sigma$ et $\sigma \in Traces(\Delta(S))$ est calculé par :
  Choisir de manière non-déterministe une des règles :
  - **Arrêt :** $P \in Pass$; return ;
  - **Choix d'une entrée :** choisir x dans $In(P)$ (si $\neq \emptyset$) ;
    ajouter la transition $P \xrightarrow{x} P' = P$ after $_{\Delta(S)}x$;
    et appliquer $gen\_test(P')$; return ;
  - **Toutes les sorties :** pour toute sortie a dans $Out(P)$
    ajouter la transition $P \xrightarrow{a} P' = P$ after $_{\Delta(S)}a$;
    et appliquer $gen\_test(P')$;
    puis compléter P en entrée avec $P \xrightarrow{?otherwise} fail$;return ;

Exemple


Δ(S)
det(Δ(S))
determinisation
!y
τ
τ
!z
!δ
!δ
?a
!x
?b
τ
τ
τ
!δ
0
1
2
3
4
5
0,1,2
!y
?a
!δ
?b
!x
!y
2
!δ
!z
?b
3,5
3,4,5
!δ
!δ
TC1
0,1,2
!a
3,5
?y
?δ
?oth
Fail
0,1,2
3,5
!b
?y
?δ
?oth
Fail
3,4,5
0,1,2
3,5
Pass
Pass
?y
?z
?δ
?oth
0,1,2
2
3,5
Fail
Pass
Pass
Pass
TC2
0,1,2
?x
?δ
?oth
Pass
2
Fail
!b
3,4,5
?y
?z
?δ
?oth
Fail
0,1,2
2
3,5
Pass
Pass
Pass

Propriétés des suites de test

NB : Pas de verdict **Inconc** car pas d'objectif de test.

Théorème : La suite de test composée de l'ensemble infini de cas de tests produits par gen_test est correcte et exhaustive.

Rappel : I **ioco** $S \triangleq STraces(I) \cap STraces(Can(S)) = \emptyset$

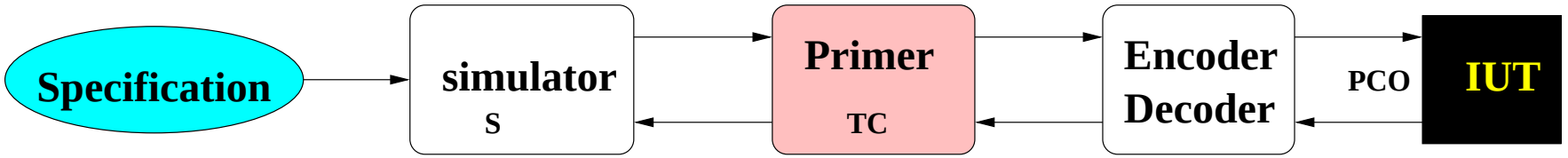
Correction : par construction, il est facile de voir que pour tout TC, $Traces_{\text{Fail}}(TC) \subseteq Traces_{\text{Fail}}(Can(S))$ (CNS de correction)

Exhaustivité : si $\neg(I$ **ioco** $S)$ alors $\exists \sigma' \in STraces(I) \cap Traces_{\text{Fail}}(Can(S))$. Soit σ tq $\sigma' = \sigma.x$. On a $\sigma \in STraces(S)$, $x \in \Lambda_!^\delta$ et $\sigma' = \sigma.x \notin STraces(S)$. $det(\Delta(S))$ est sans blocage, donc il existe $y \in \Lambda_!^\delta$ tq $\sigma.y \in STraces(S)$. Dans gen_test, faire les choix input/output en fonction de $\sigma'.y$ puis "Toutes les sorties", puis Pass. Le TC résultant peut rejeter I.

Implémentation dans TorX

Le choix non-déterministe est implémenté par un choix aléatoire ou interactif.

Sélection et exécution à la volée des cas de test.



Langages : Promela (SPIN), Lotos (CADP)

Sélection de test guidée par un objectif

Principales différences avec TorX :

- sélection des tests par des objectifs de test.
- algorithmes plus efficaces
- sélection off-line, exécution *a posteriori*

Objectif de test : décrit un ensemble de comportements à tester, ciblé par le cas de test.
⇒ modélisation par des observateurs d'accessibilité (langage accepté)
⇒ algorithmes de sélection type "model-checking".

Objectifs de test

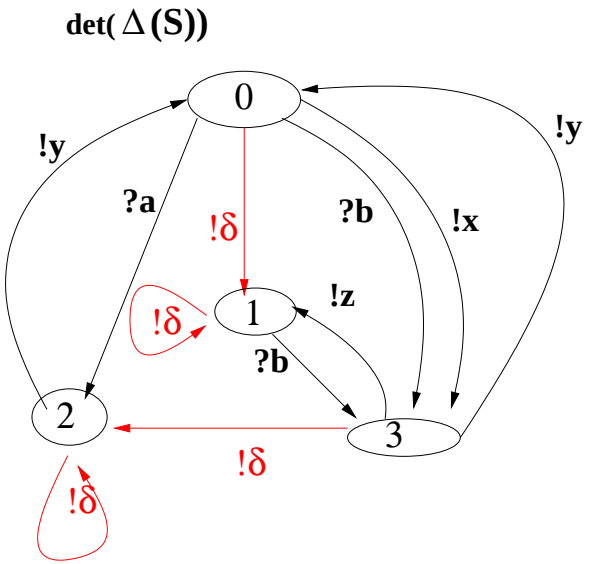
Un objectif de test est un IOLTS $TP = (Q^{\mathsf{TP}}, \Lambda^{\delta}_{VIS}, \rightarrow_{\mathsf{TP}}, q_0^{\mathsf{TP}})$,
muni d'un ensemble d'états puits $Accept^{\mathsf{TP}}$
(puits : $\forall a \in \Lambda^{\delta}_{VIS}, Accept \xrightarrow{a}_{\mathsf{TP}} Accept$).

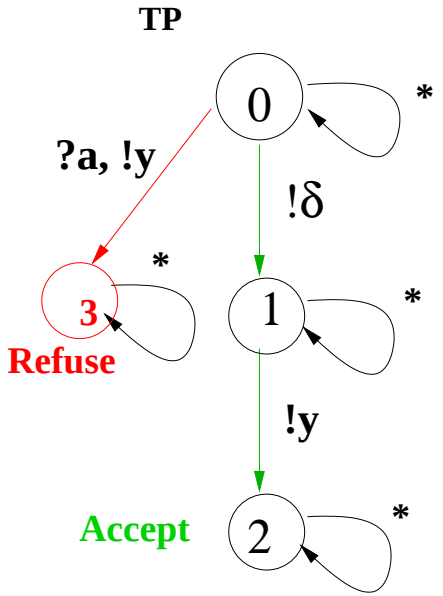
On le supposera déterministe et complet sur Λ^{δ}_{VIS}
(complet : $\forall q, \forall a \in \Lambda^{\delta}_{VIS}, q \xrightarrow{a}_{\mathsf{TP}}$).

$Traces_{\mathsf{Accept}}(TP) = L_{\mathsf{Accept}}(TP)$ est suffixe-clos
et $Traces(TP) = L(TP) = (\Lambda^{\delta}_{\mathsf{VIS}})^*$

TP peut être vu comme un observateur de propriété d'accessibilité.

Exemple d'objectif





Principe de la sélection

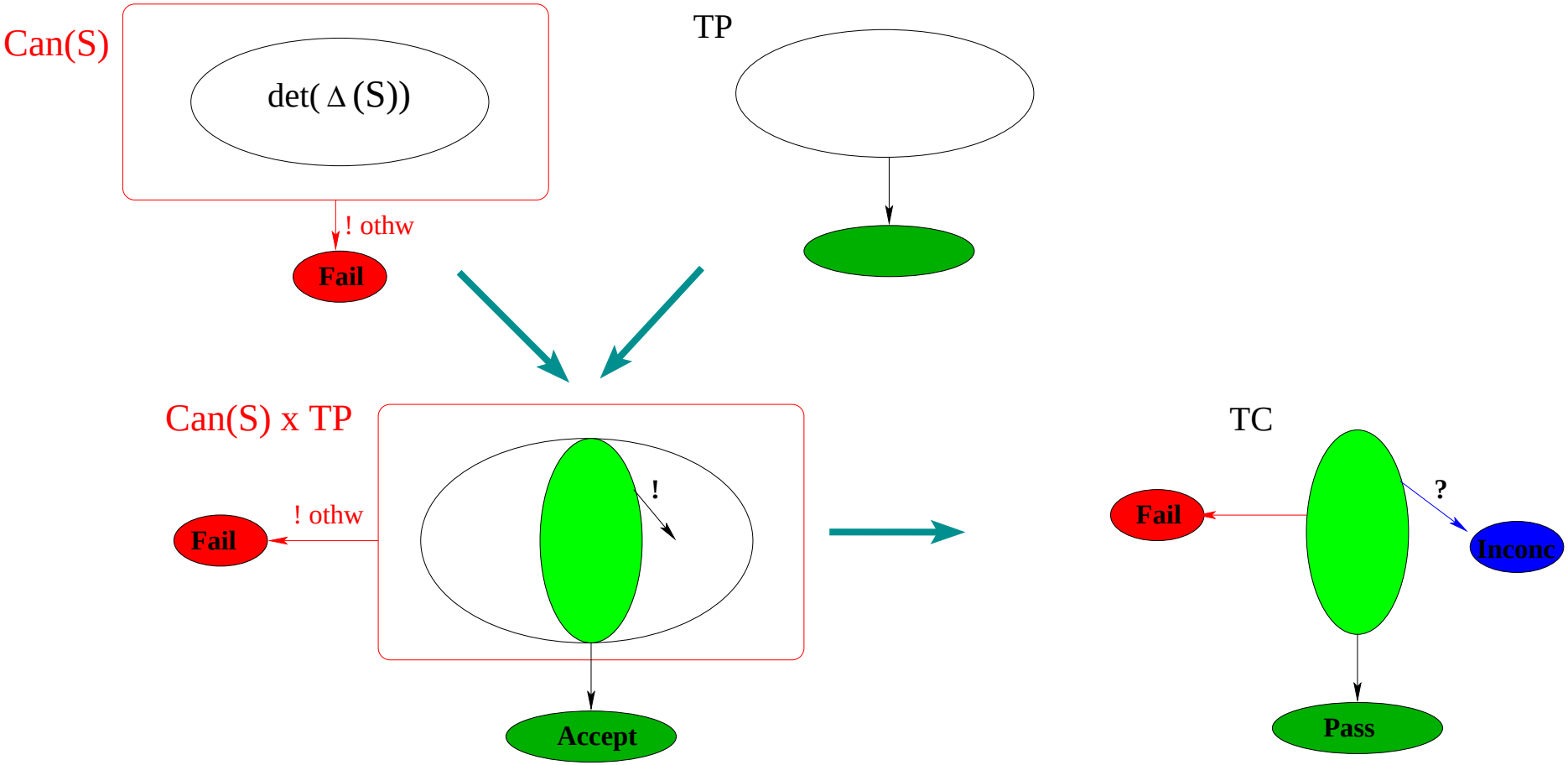
Générer des cas de tests qui soient à la fois

- observateurs de non-conformité
  $Traces_{\text{Fail}}(TC) \subseteq Traces_{\text{Fail}}(Can(S))$
  (ce qui implique la correction)

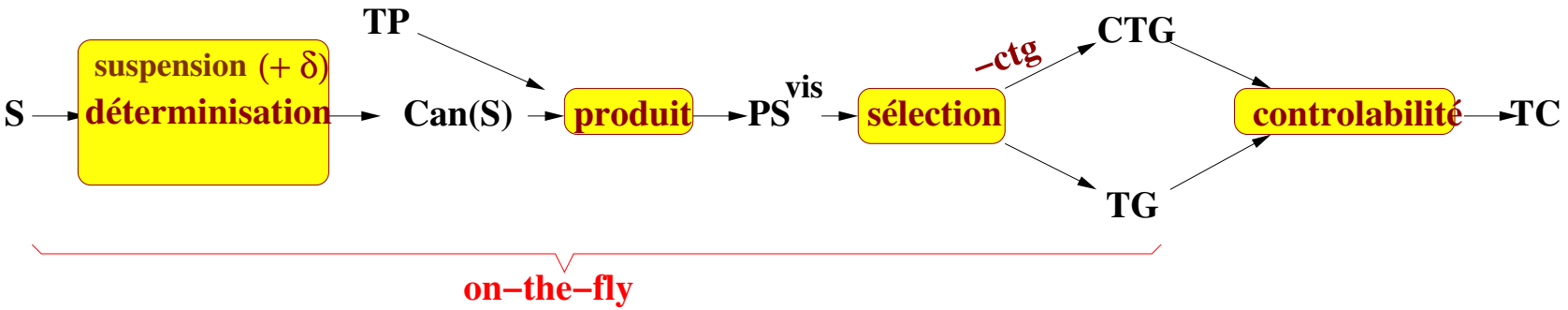
- observateurs d'accessibilité
  $Traces_{\text{Pass}}(TC) \subseteq Traces_{\text{Accept}}(TP)$

Mais en privilégiant les traces acceptées $STraces(S) \cap Traces_{\text{Accept}}(TP)$

Schéma de sélection



Opérations nécessaires à la sélection



Produit synchrone

Soient $M_1 = (Q^1, A, \rightarrow_1, q_0^1)$ muni de F_1
et $M2 = (Q^2, A, \rightarrow_2, q_0^2)$ muni de F_2 deux IOLTS de même alphabet A
Le produit synchrone de M_1 et M_2 est l'IOLTS
$M_1 \times M_2 = (Q^1 \times Q^2, A, \rightarrow, (q_0^1, q_0^2))$ muni de $F_1 \times F_2$
où $\rightarrow$ est définie par la règle :

$$\frac{q_1 \stackrel{a}{\rightarrow}_1 q_1' \quad q_2 \stackrel{a}{\rightarrow}_2 q_2'}{(q_1, q_2) \stackrel{a}{\rightarrow} (q_1', q_2')}$$

On a alors :
$L(M_1 \times M_2) = L(M_1) \cap L(M_2)$ et
$L_{F_1 \times F_2}(M_1 \times M_2) = L_{F_1}(M_1) \cap L_{F_2}(M_2)$.

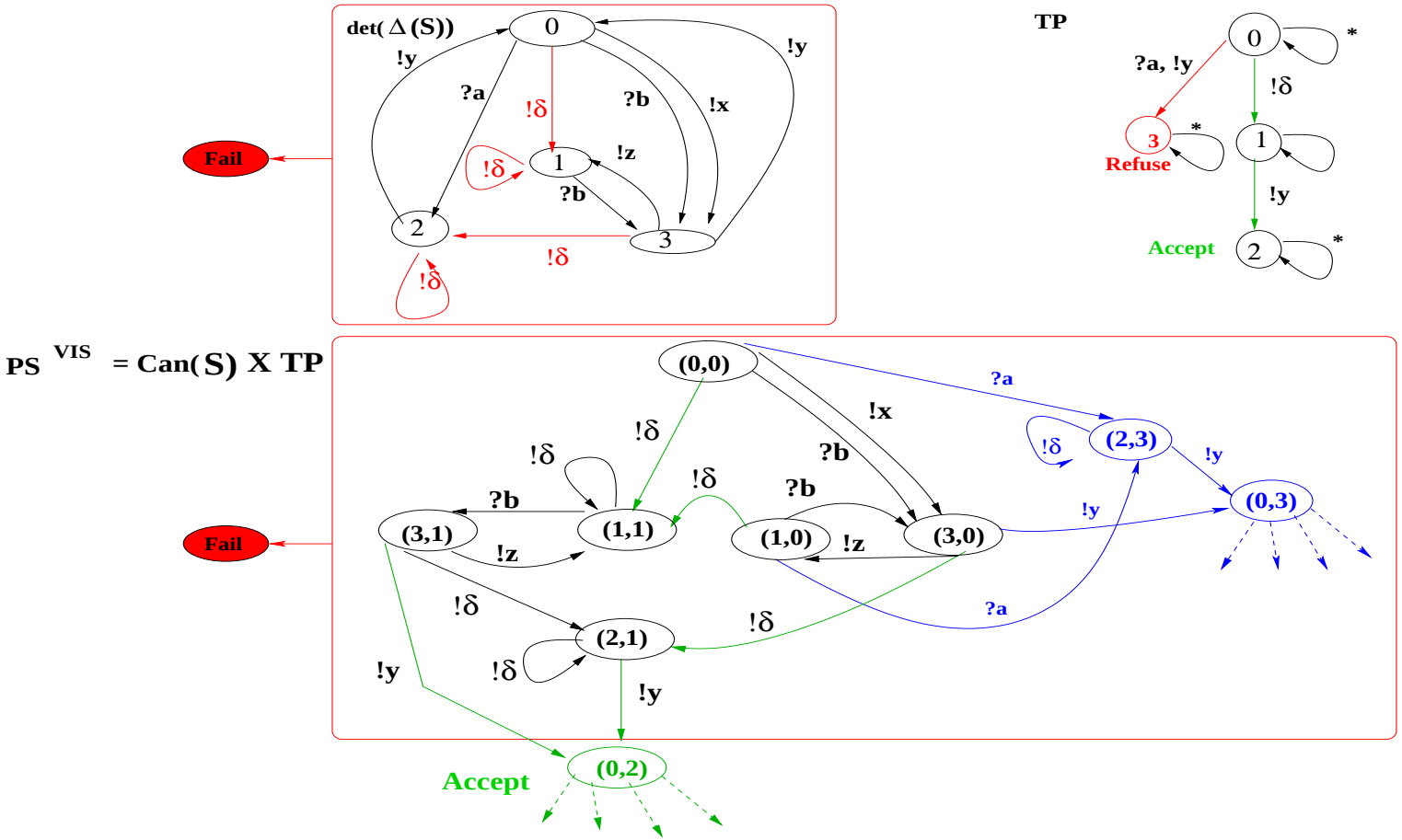
Produit synchrone $Can(S) \times TP$

$Can(S) = (Q^c, \Lambda^{\delta}_{\text{VIS}}, \rightarrow_c, q_0^c,)$ muni de $Fail \subseteq Q^c$ et
$TP = (Q^{\text{TP}}, \Lambda^{\delta}_{\text{VIS}}, \rightarrow_{\text{TP}}, q_0^{\text{TP}})$ muni de $Accept_{\text{TP}} \subseteq Q^{\text{TP}}$
sont deux IOLTS de même alphabet $\Lambda^{\delta}_{\text{VIS}}$.

Soit $PS^{\text{VIS}} = Can(S) \times TP$ et les ensembbles d'états
$Accept_{\text{VIS}} = Q_c \setminus \{Fail\} \times Accept_{\text{TP}}$ et $Fail_{\text{VIS}} = \{Fail\} \times Q^{\text{TP}}$

TP étant complet on a $Traces(TP) = (\Lambda^{\delta}_{\text{VIS}})^*$, d'où
$Traces(PS^{\text{VIS}}) = Traces(Can(S)) \cap Traces(TP) = Traces(Can(S))$
$Traces_{\text{Accept}}(PS^{\text{VIS}}) = STraces(S) \cap Traces_{\text{Accept}}(TP)$
$Traces_{\text{Fail}}(PS^{\text{VIS}}) = Traces_{\text{Fail}}(Can(S)) \cap Traces(TP) = Traces_{\text{Fail}}(Can(S))$

Exemple


det(Δ (S))
0
1
2
3
!y
?a
!δ
?b
!x
!z
!δ
?b
!δ
!δ
Fail
TP
0
1
2
3
?a, !y
!δ
!y
*
*
*
Refuse
Accept
PS VIS = Can(S) X TP
(0,0)
?a
!x
!δ
!δ
?b
!δ
?b
(2,3)
!δ
!y
(0,3)
!y
(3,1)
(1,1)
(1,0)
(3,0)
?b
!z
!z
Fail
!δ
!δ
?a
(2,1)
!y
!δ
!y
!y
Accept
(0,2)

Sélection

But : Extraire de PS^{VIS} un cas de test avec les verdicts adéquats.

Fail : détecter $STraces(S).\Lambda_!^\delta \setminus STraces(S)$
On a $Traces_{\text{Fail}}(PS^{\text{VIS}}) = Traces_{\text{Fail}}(Can(S))$

Pass : détecter $STraces(S) \cap Traces_{\text{Accept}}(TP)$
On a $Traces_{\text{Accept}}(PS^{\text{VIS}}) = STraces(S) \cap Traces_{\text{Accept}}(TP)$
Donc **Pass** $= Accept_{\text{VIS}}$

Inconc : détecter $Rtraces(PS^{\text{VIS}}) \triangleq STraces(S) \setminus pref_{\leq}(Traces_{\text{Accept}}(PS^{\text{VIS}}))$
i.e. traces suspendues de S qu'on ne peut pas prolonger en traces acceptées.
Les entrées (contrôlables) peuvent être coupées
Observation : $pref_{\leq}(Traces_{\text{Accept}}(PS^{\text{VIS}})) = Traces_{coreach(Accept_{\text{VIS}})}(PS^{\text{VIS}})$
$\Rightarrow$ analyse de co-accessibilité à $Accept_{\text{VIS}}$

Illustration

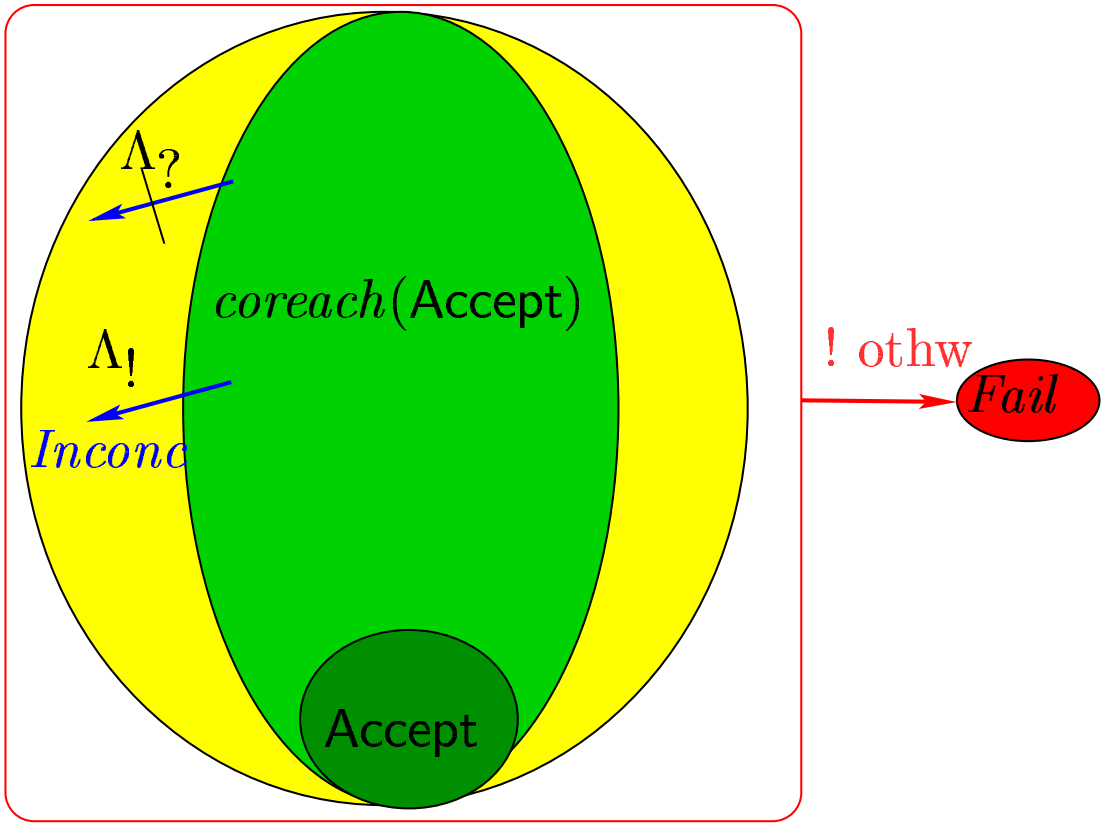
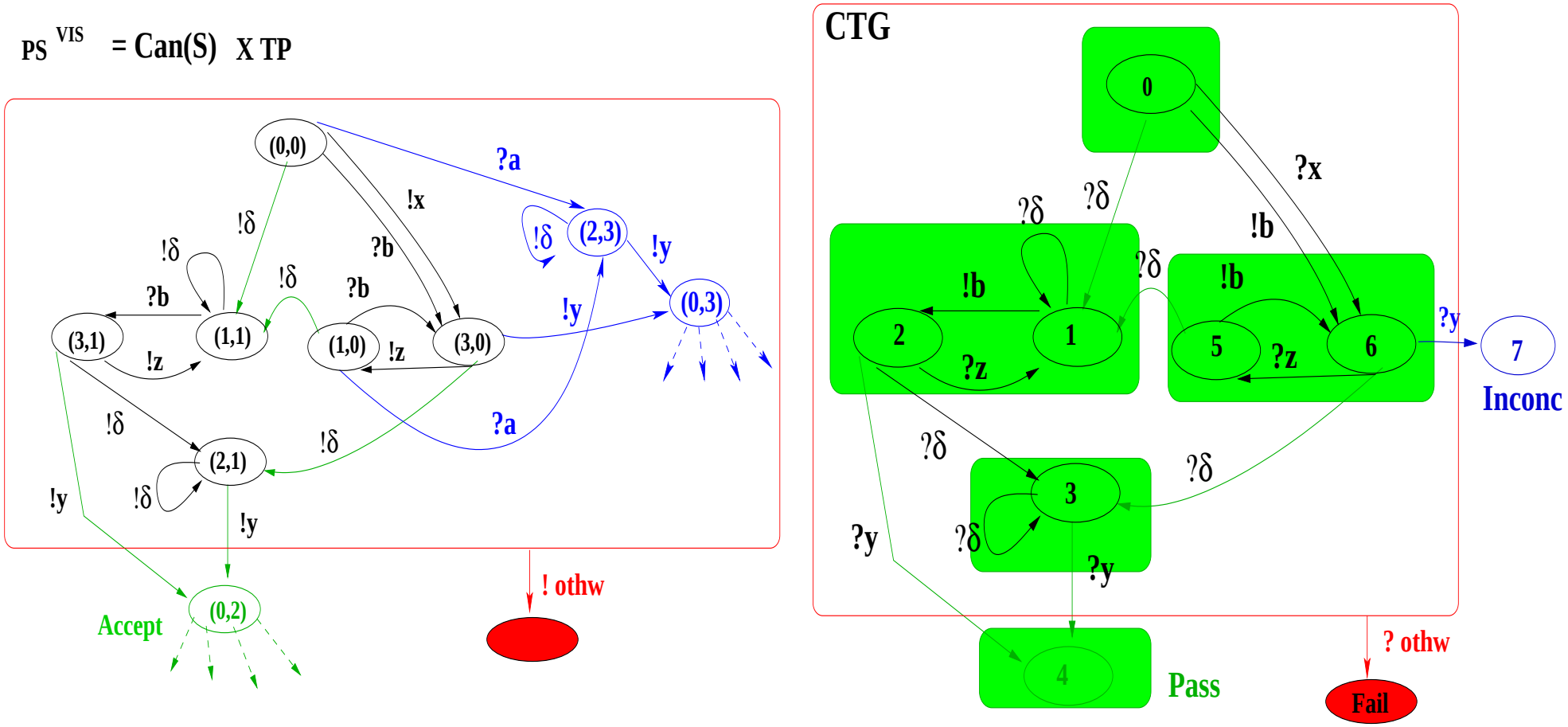


Illustration de l'extraction du graphe de test complet



Accessibilité et co-accessibilité

$post_B(P) = \{q' \in Q^{\mathsf{M}} \mid \exists b \in B, \exists q \in P,\ q \xrightarrow{b}_{\mathsf{M}} q'\}$
i.e. successeurs immédiats de P par des actions de B

$reach_B(P) = \mu X.P \cup post_B(X) = \bigcup_{i \geq 0} post^i_B(P)$
i.e. accessibles de P par des actions de B

$pre_B(P) = \{q' \in Q^{\mathsf{M}} \mid \exists b \in B, \exists q \in P,\ q' \xrightarrow{b}_{\mathsf{M}} q\}$
i.e. prédécesseurs immédiats de P par des actions de B,

$coreach_B(P) = \mu X.P \cup pre_B(X) = \bigcup_{i \geq 0} pre^i_B(P)$
i.e. co-accessibles de P par des actions de B.

Définition du Graphe de Test Complet (CTG)

Soit $Can(S) \times TP = PS^{\text{VIS}} = (Q^{\text{VIS}}, \Lambda^{\delta}_{\text{VIS}}, \rightarrow_{\text{VIS}}, q_0^{\text{VIS}})$, muni de $Accept_{\text{VIS}}$ et $Fail_{\text{VIS}}$.
Le graphe de test complet est l'IOLTS $CTG = (Q^{\text{VIS}}, \Lambda^{\delta}_{\text{VIS}}, \rightarrow_{\text{CTG}}, q_0^{\text{VIS}})$
muni de **Pass** $\triangleq Accept_{\text{VIS}}$, **Inconc** $\subseteq Q^{\text{VIS}}$ et **Fail** $=$ **Fail** $_{\text{VIS}}$,
où **Inconc** et $\rightarrow_{\text{CTG}}$ sont définis par les règles :

$$\textbf{Keep :}\ \frac{\begin{array}{l} q \in coreach(Accept_{\text{VIS}}) \\ q' \in coreach(Accept_{\text{VIS}}) \cup \{\textbf{Fail}_{\text{VIS}}\} \end{array} \qquad q \xrightarrow{\alpha}_{\text{VIS}} q' \quad \alpha \in \Lambda^{\delta}_{\text{VIS}}}{q \xrightarrow{\alpha}_{\text{CTG}} q'}$$

$$\textbf{Inconc :}\ \frac{\begin{array}{l} q \in coreach(Accept_{\text{VIS}}) \\ q' \notin (coreach(Accept_{\text{VIS}}) \cup \{\textbf{Fail}_{\text{VIS}}\}) \end{array} \qquad q \xrightarrow{\alpha}_{\text{VIS}} q' \qquad \alpha \in \Lambda^{\delta}_{!}}{q \xrightarrow{\alpha}_{\text{CTG}} q' \qquad q' \in \textbf{Inconc}}$$

Propriétés de CTG en termes de traces

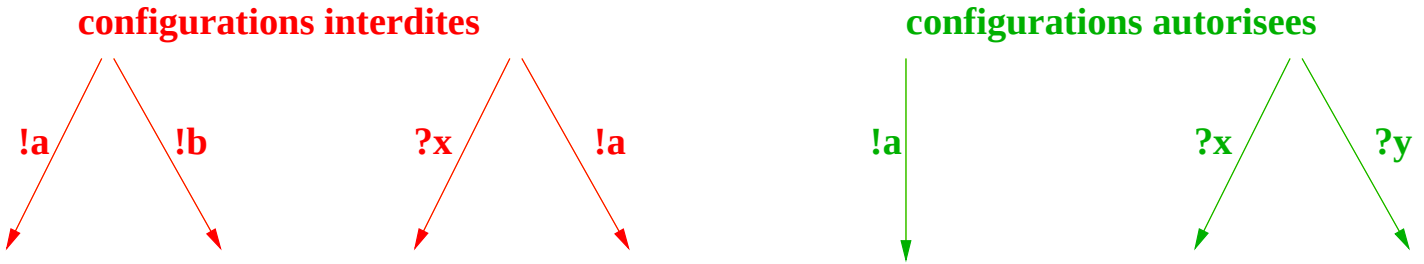
$Traces_{\text{Pass}}(CTG) = Traces_{\text{Accept}}(PS^{\text{VIS}}) = STraces(S) \cap Traces_{\text{Accept}}(TP)$
le verdict *pass* est produit sur toute trace suspendue de S acceptée par TP.

$Traces_{\text{Inconc}}(CTG) =$
$[STraces(S) \cap pref_{\leq}(Traces_{\text{Accept}}(TP))].\Lambda_!^{\delta} \cap STraces(S) \setminus pref_{\leq}(Traces_{\text{Accept}}(TP))$
le verdict *Inconc* est produit sur toute trace suspendue de S, dont la dernière action est une sortie ou δ qui ne peut pas mener à la satisfaction de TP.

$Traces_{Fail}(CTG)$
$= [STraces(S) \cap pref_{\leq}(Traces_{\text{Accept}}(TP))].\Lambda_!^{\delta} \setminus STraces(S)$
$\Rightarrow Traces_{Fail}(CTG) \subseteq Traces(Can(S))$ (CNS de correction)
Le verdict *Fail* est produit sur toute trace, qui est obtenue à partir d'un préfixe de trace suspendue de S pouvant mener à Accept, poursuivie par une sortie non admise dans S.

Conflits de contrôlabilité : calcul des Cas de test

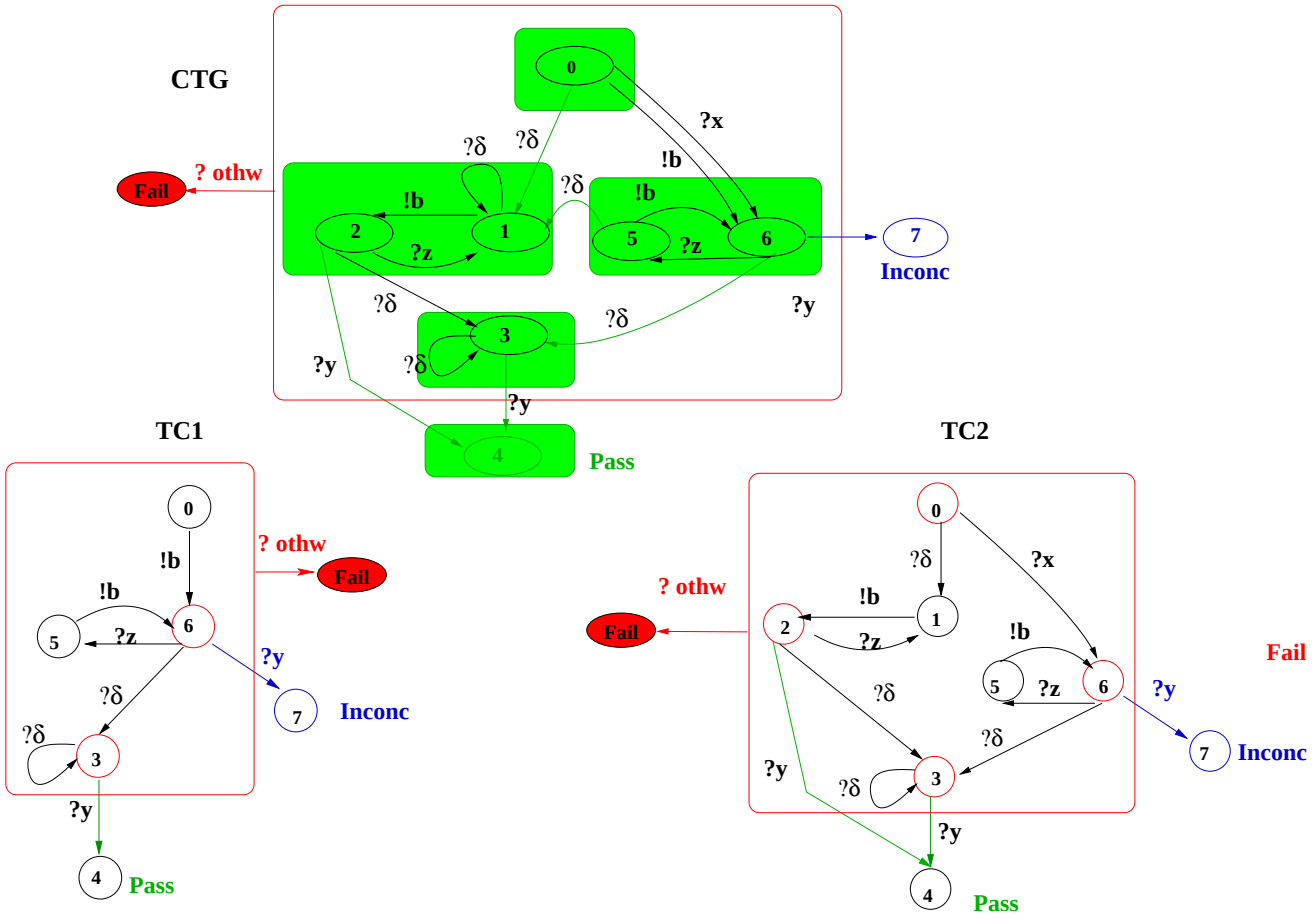
En pratique, un cas de test doit être contrôlable :



Algorithme : parcours en arrière de CTG depuis les états Pass vers l'état initial et élagage des conflits. Adapté de $coreach(Pass)$.

TC est un sous-IOLTS de CTG donc
$Traces_{\text{Fail}}(TC) \subseteq Traces_{\text{Fail}}(CTG) \subseteq Traces(Can(S))$
$\Rightarrow$ la correction est conservée

Élagage : exemple



Propriétés des cas de test (I)

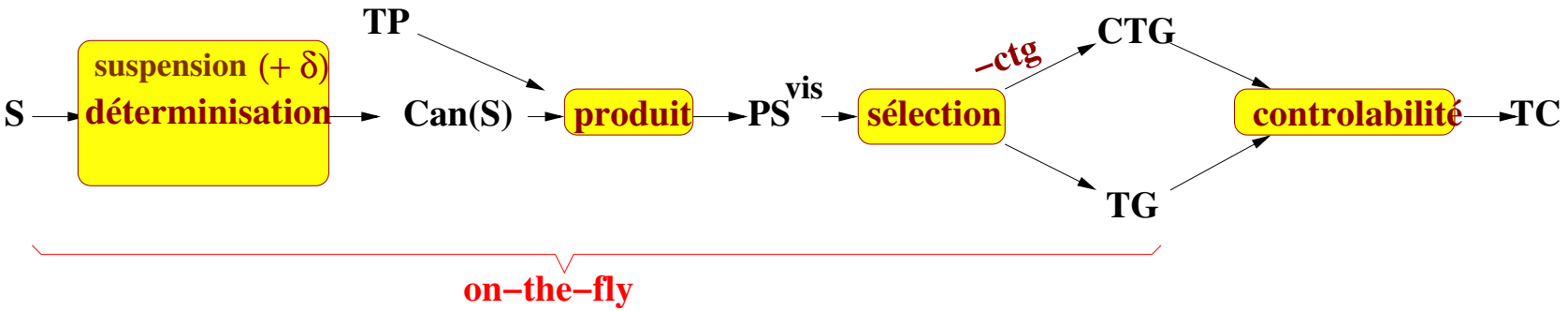
Théorème : La suite de tests composée de l'ensemble (infini) des cas de test que peut produire l'algorithme est correcte, exhaustive.

Correction : $Traces_{\text{Fail}}(TC) \subseteq Traces(Can(S))$
$\Rightarrow TC$ non-biaisé.

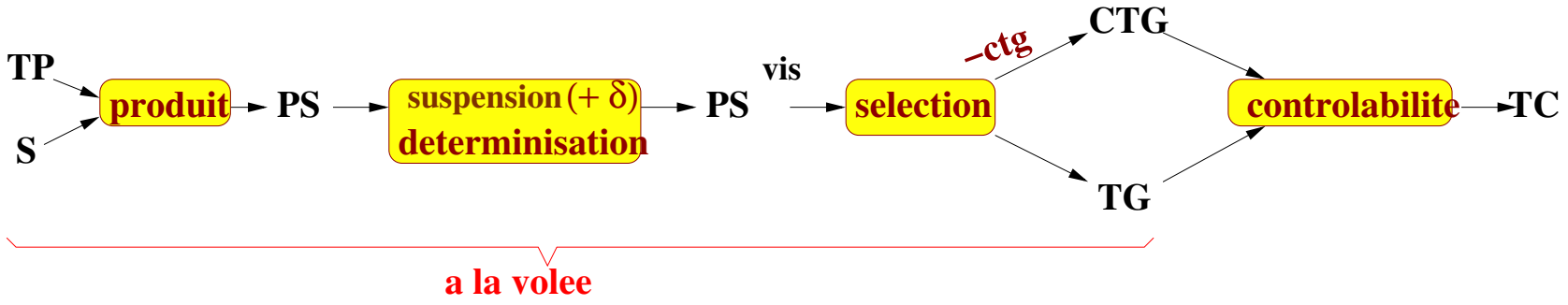
Exhaustivité : si $\neg(I$ **ioco** $S)$ alors $\exists \sigma \in STraces(S), \exists x \in \Lambda_!^\delta$,
$x \in Out(\Delta(I)$ after $\sigma) \wedge x \notin Out(\Delta(S)$ after $\sigma)$.
On a $\exists y \in Out(\Delta(S)$ after $\sigma)$, (car $Out(\Delta(S)$ after $\sigma) \neq \emptyset$).
Posons $\sigma' = \sigma.y (\in STraces(S))$ et $TP = \sigma'$ terminé par $Accept$.
Par def, CTG after $\sigma.x \in$ **Fail**. Elaguer CTG en TC tq $\sigma.x \in Traces(TC)$
$\Longrightarrow$ TC after $\sigma.x \in$ **Fail**. Donc TC fails IUT.

Résumé des opérations nécessaire à la synthèse

Sans actions internes : TP défini sur $\Lambda^{\delta}_{\text{VIS}}$

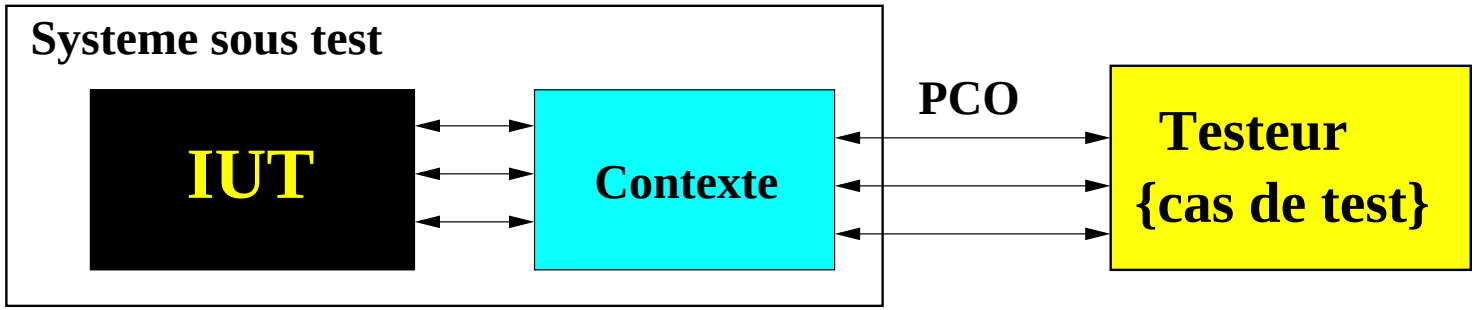


Avec actions internes : TP défini sur Λ



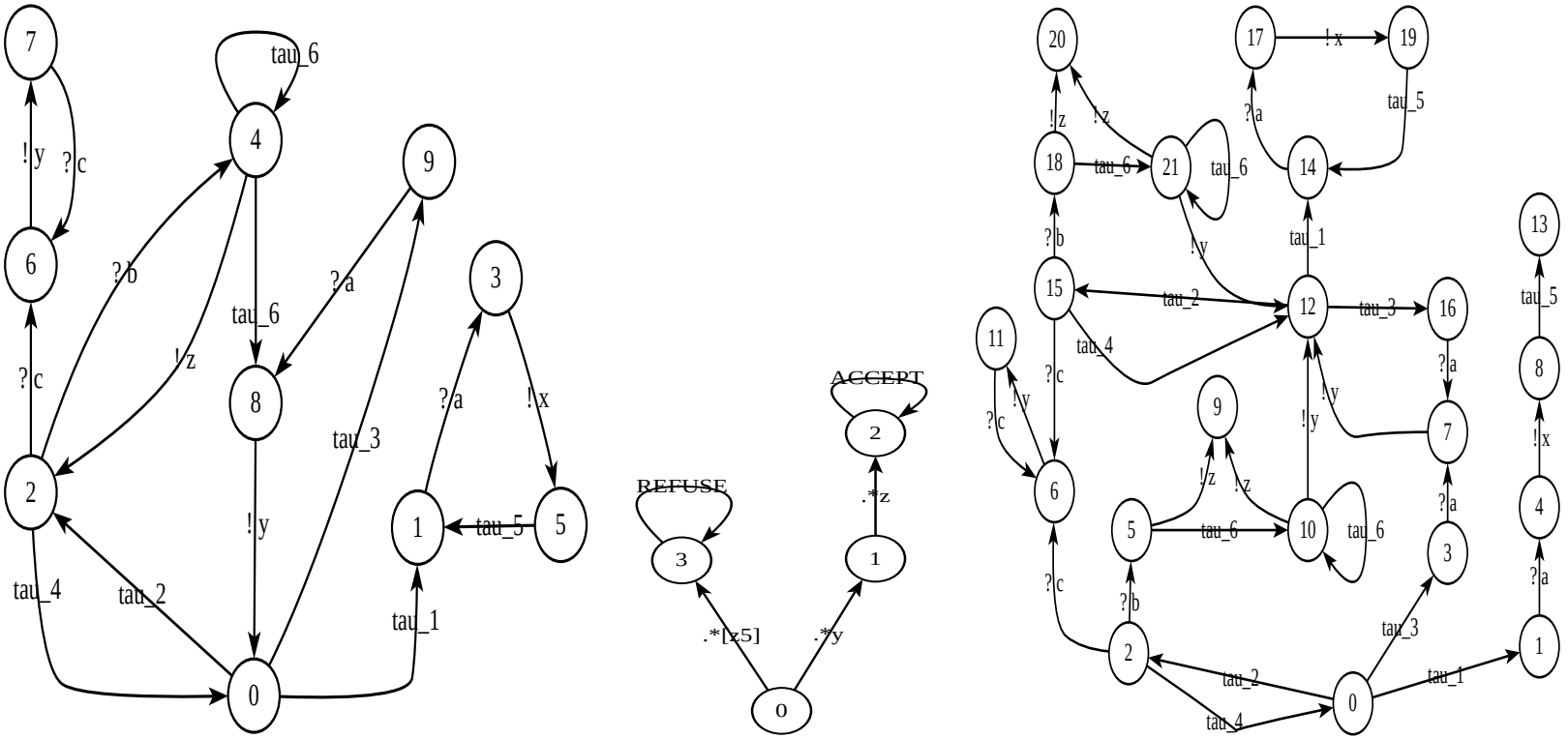
Problèmatique des actions internes dans les objectifs

Important pour le test dans un contexte :
permet une sélection basée sur des comportements internes.

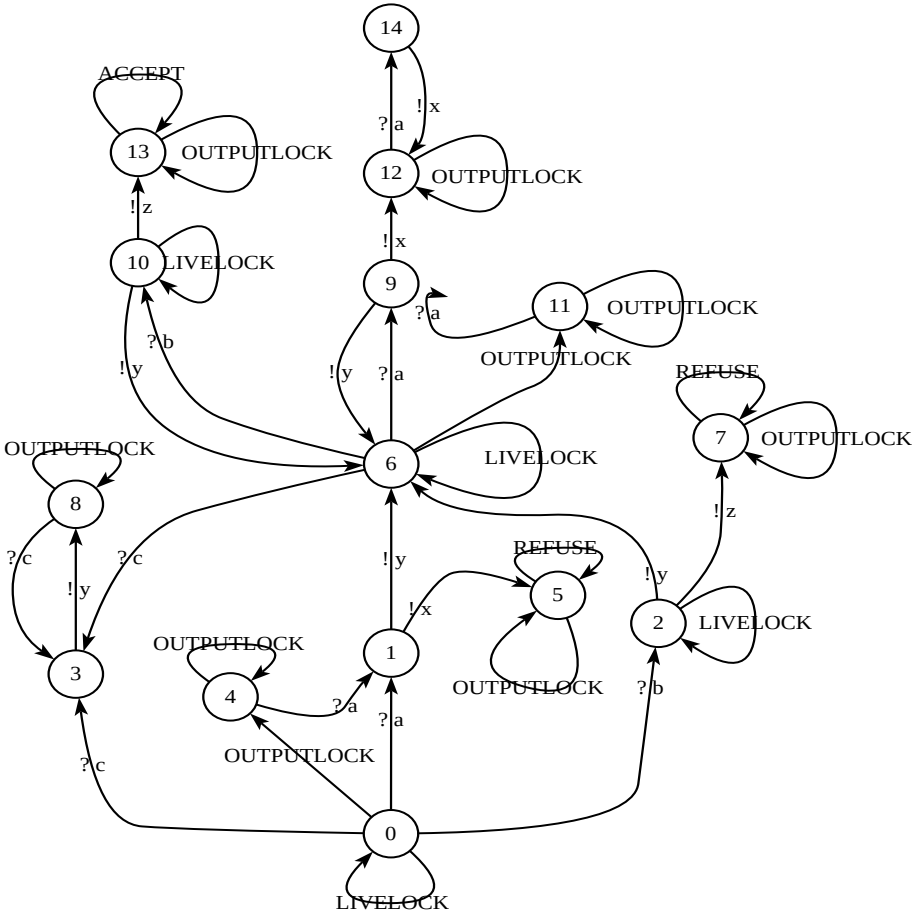


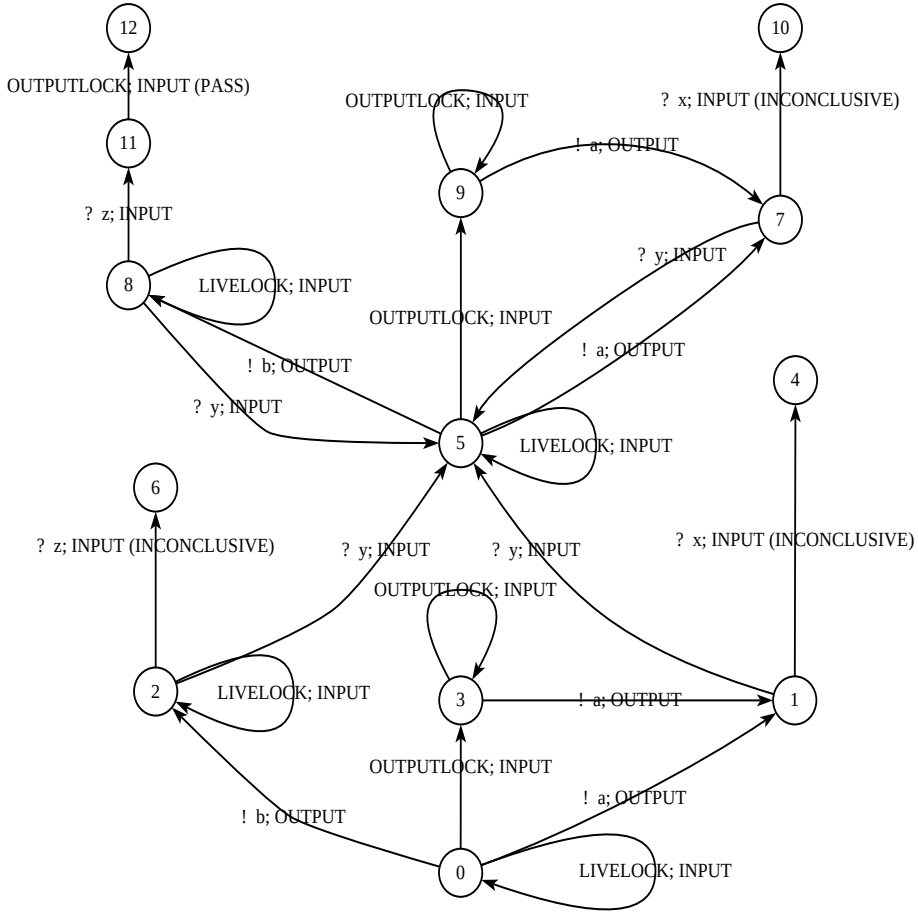
Modifications : TP reconnait le langage $L_{\text{Accept}}(TP)$
$PS = S \times TP \rightarrow L_{\text{Accept}}(PS) = L(S) \cap L_{\text{Accept}}(TP)$
$PS^{\text{VIS}} = det(\Delta(PS)) \rightarrow Traces_{\text{Accept}}(PS^{\text{VIS}}) = STraces_{\text{Accept}}(PS)$
CTG légèrement différent pour atteindre l'exhaustivité (pb de δ).

Produit synchrone avec actions internes : exemple

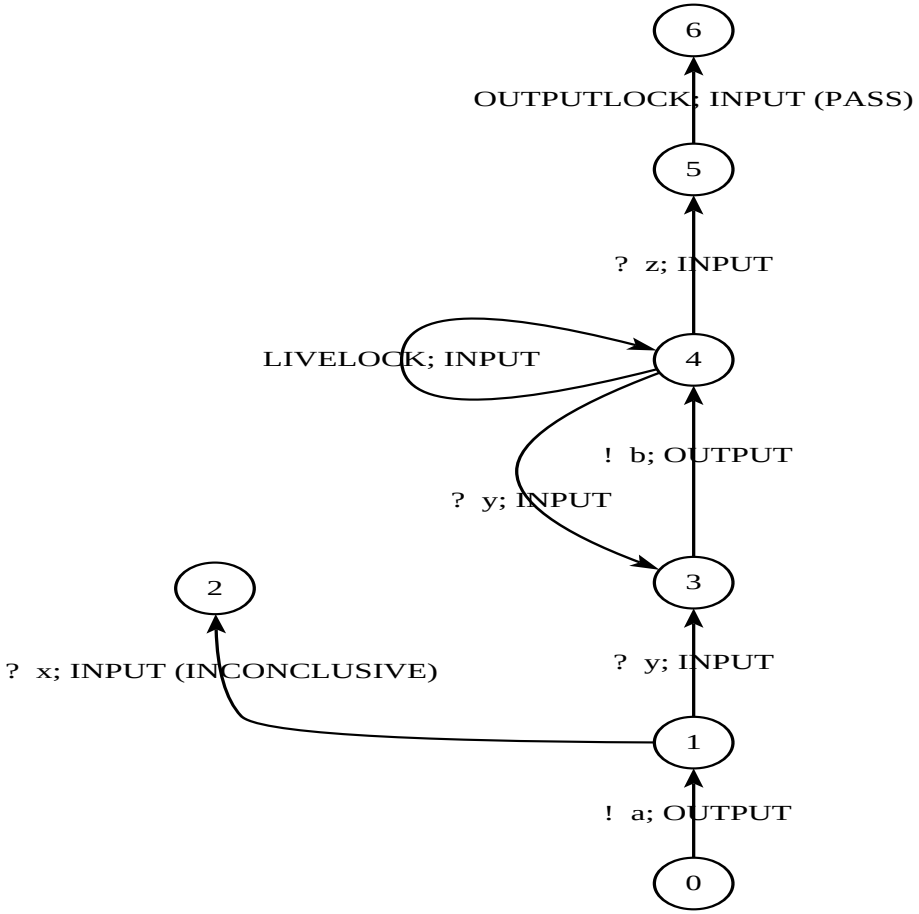


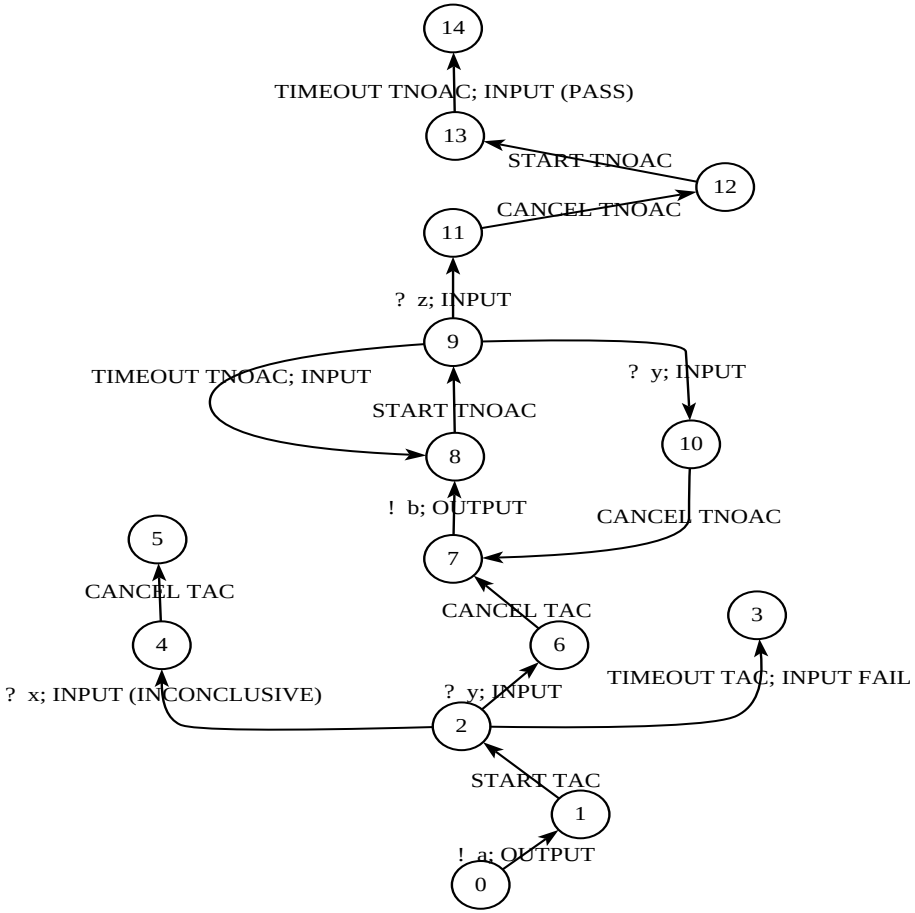
$PS^{\text{VIS}} = det(\Delta(PS))$ et graphe de test complet CTG





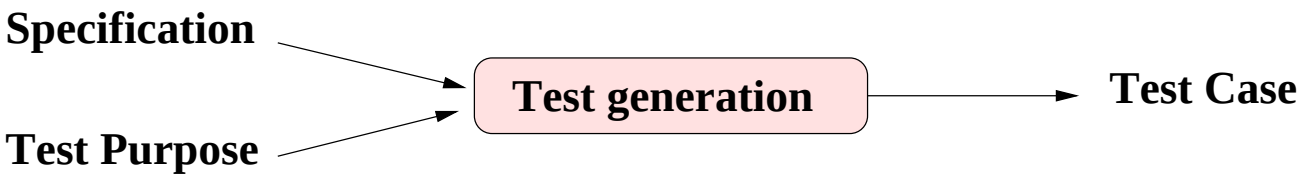
cas de test sans et avec temporisateurs



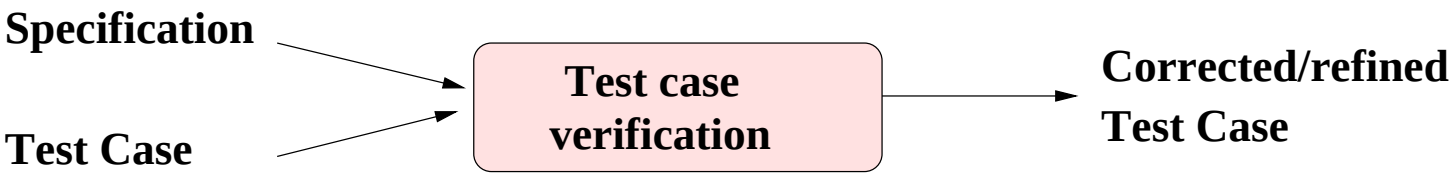


L'outil TGV

Synthèse de tests



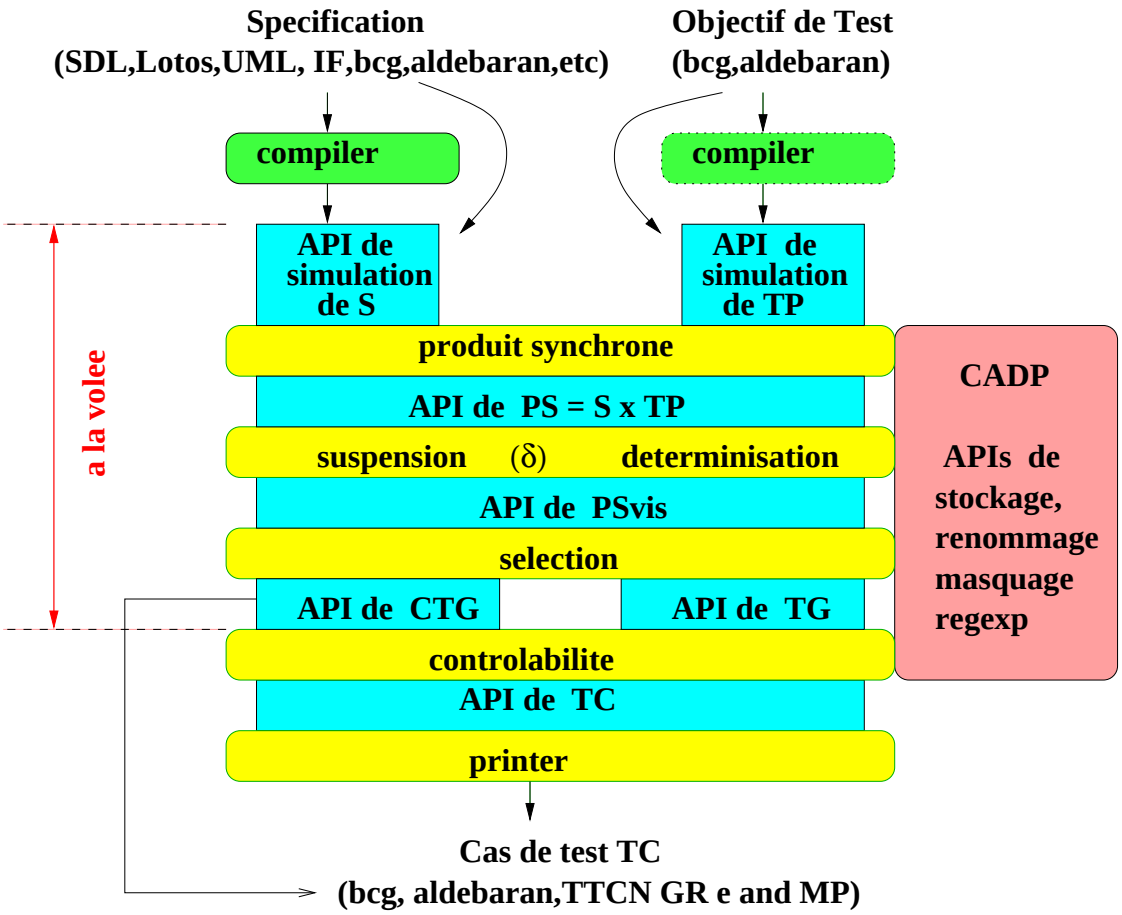
Vérification de cas de test :



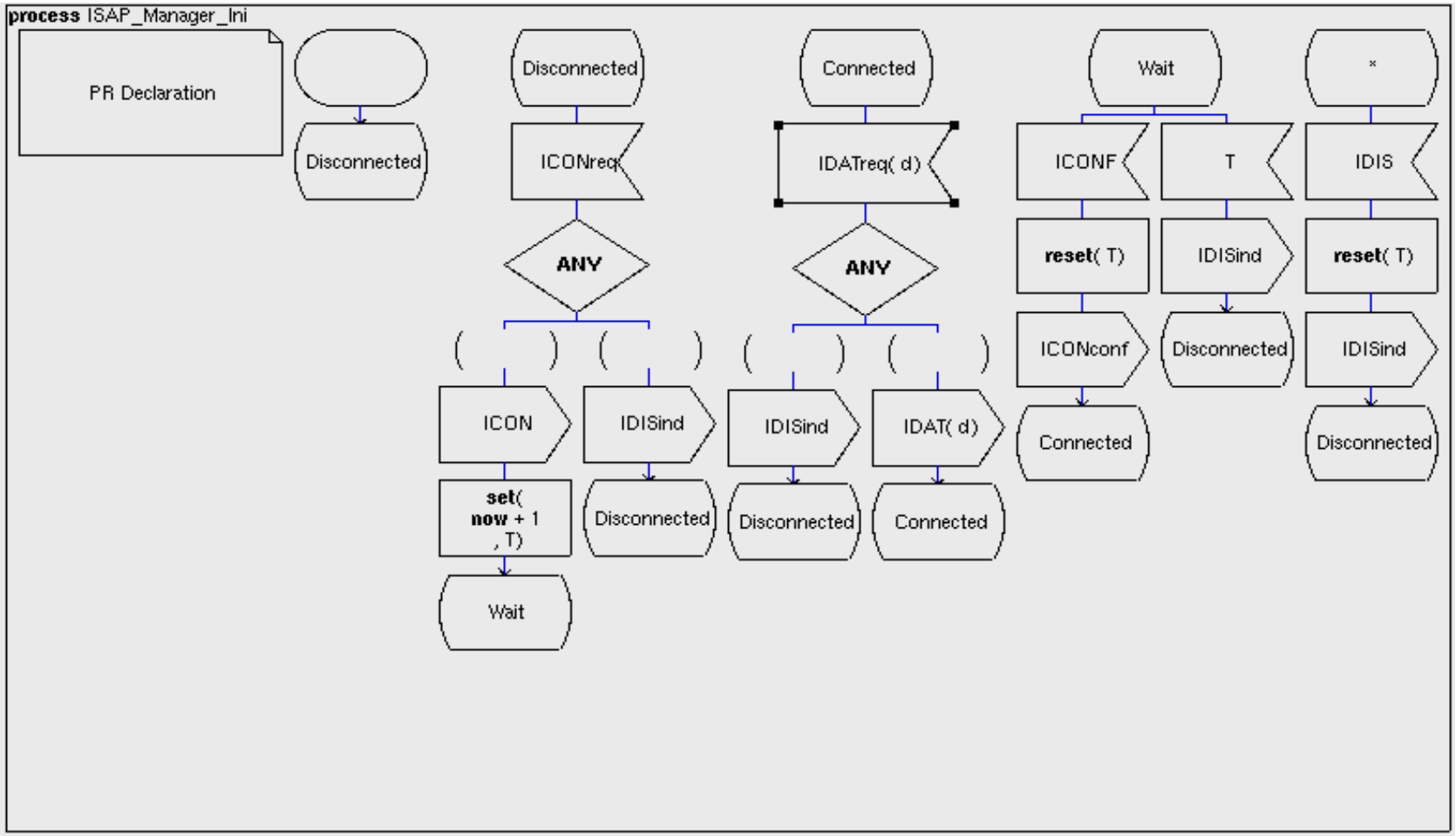
Principes généraux de TGV

- sélection des cas de tests par objectifs de tests
- algorithmes fondés sur le model-checking à la volée
- indépendance / langage de spécification
  mais permet de traiter des spécifications SDL, Lotos, UML
  grâce à des algorithmes fondés sur le modèle IOLTS

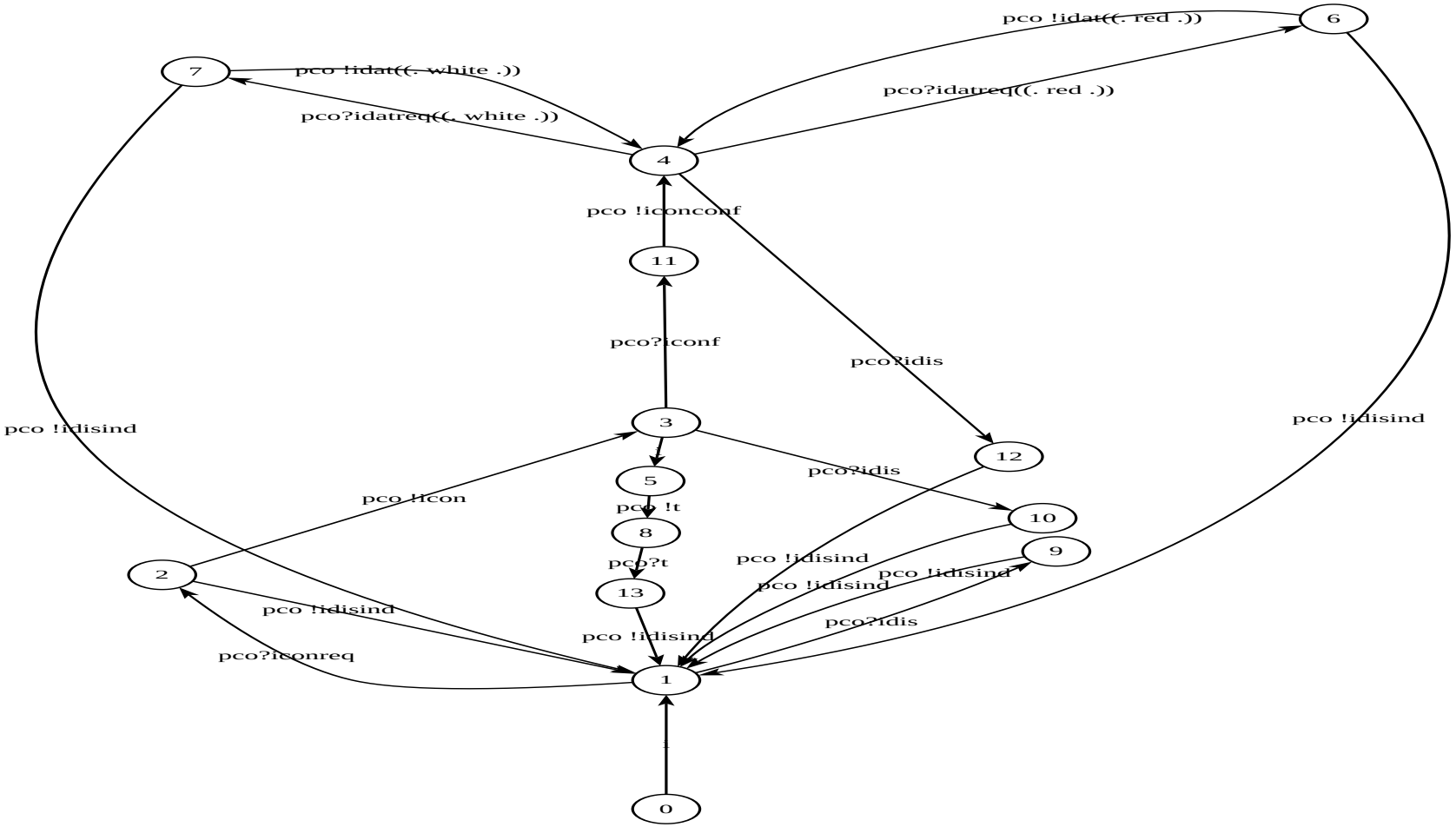
Architecture de TGV



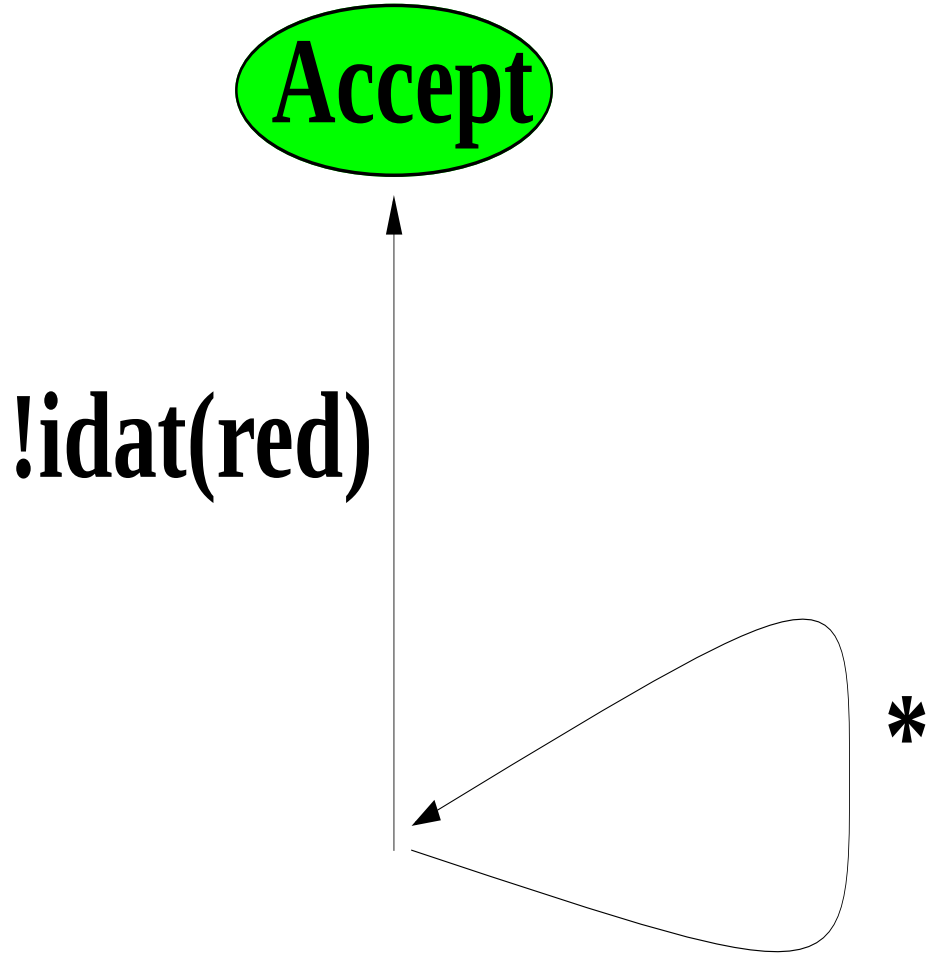
Exemple SDL : processus initiateur de l'Inres



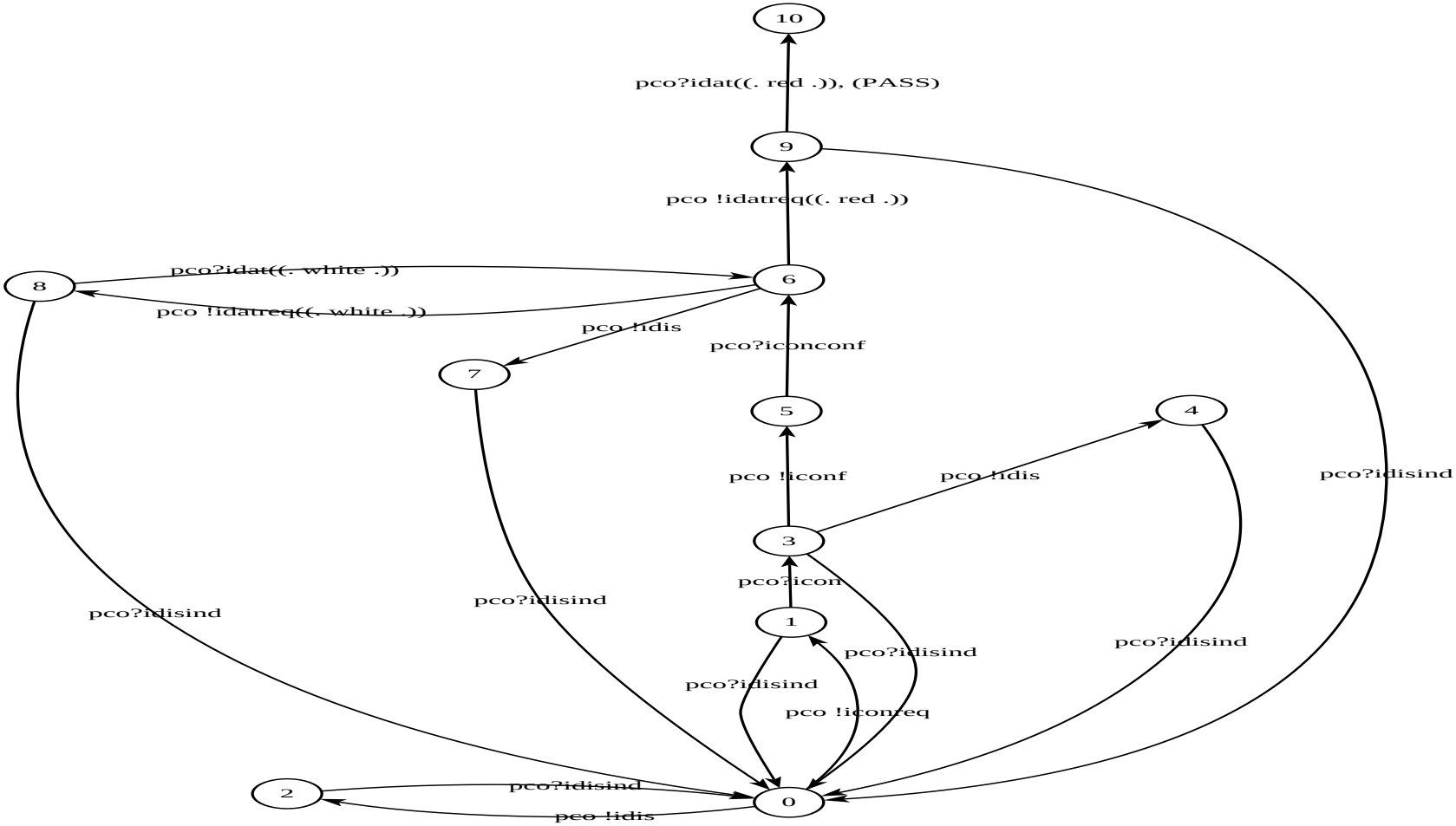
Graphe d'états complet



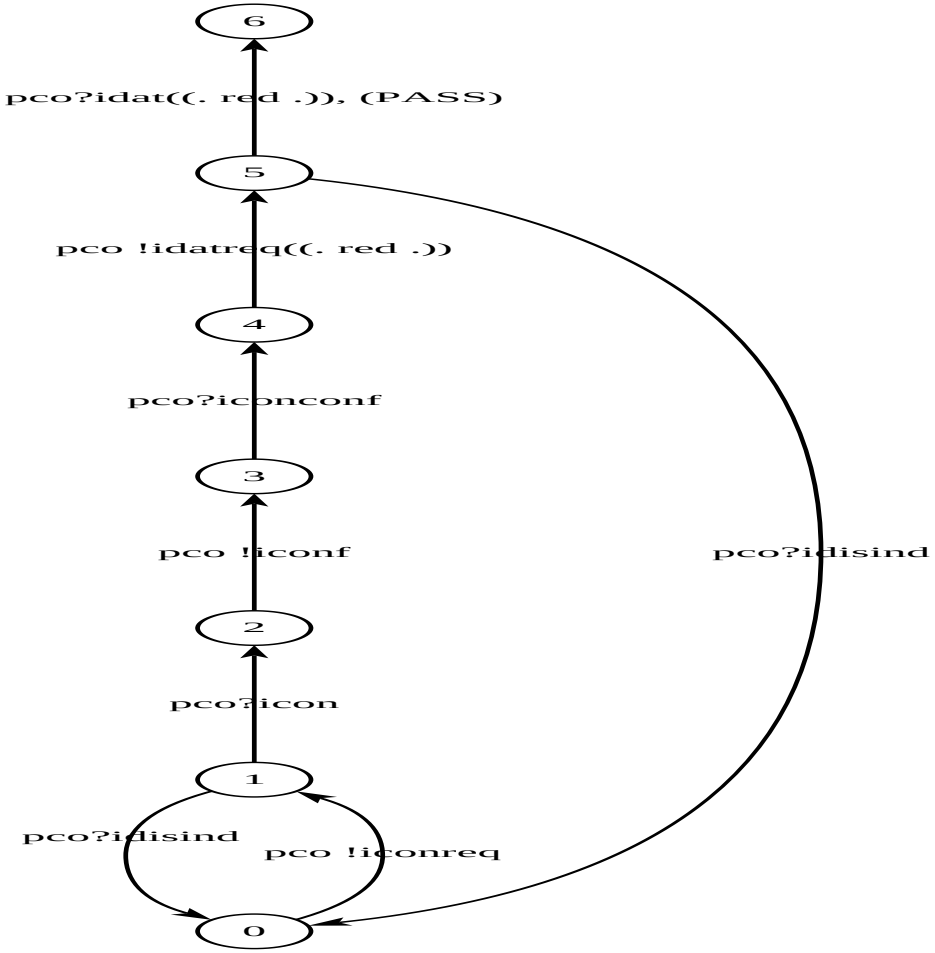
Un objectif simple



Graphe de test complet produit par TGV



Cas de test produit à la volée



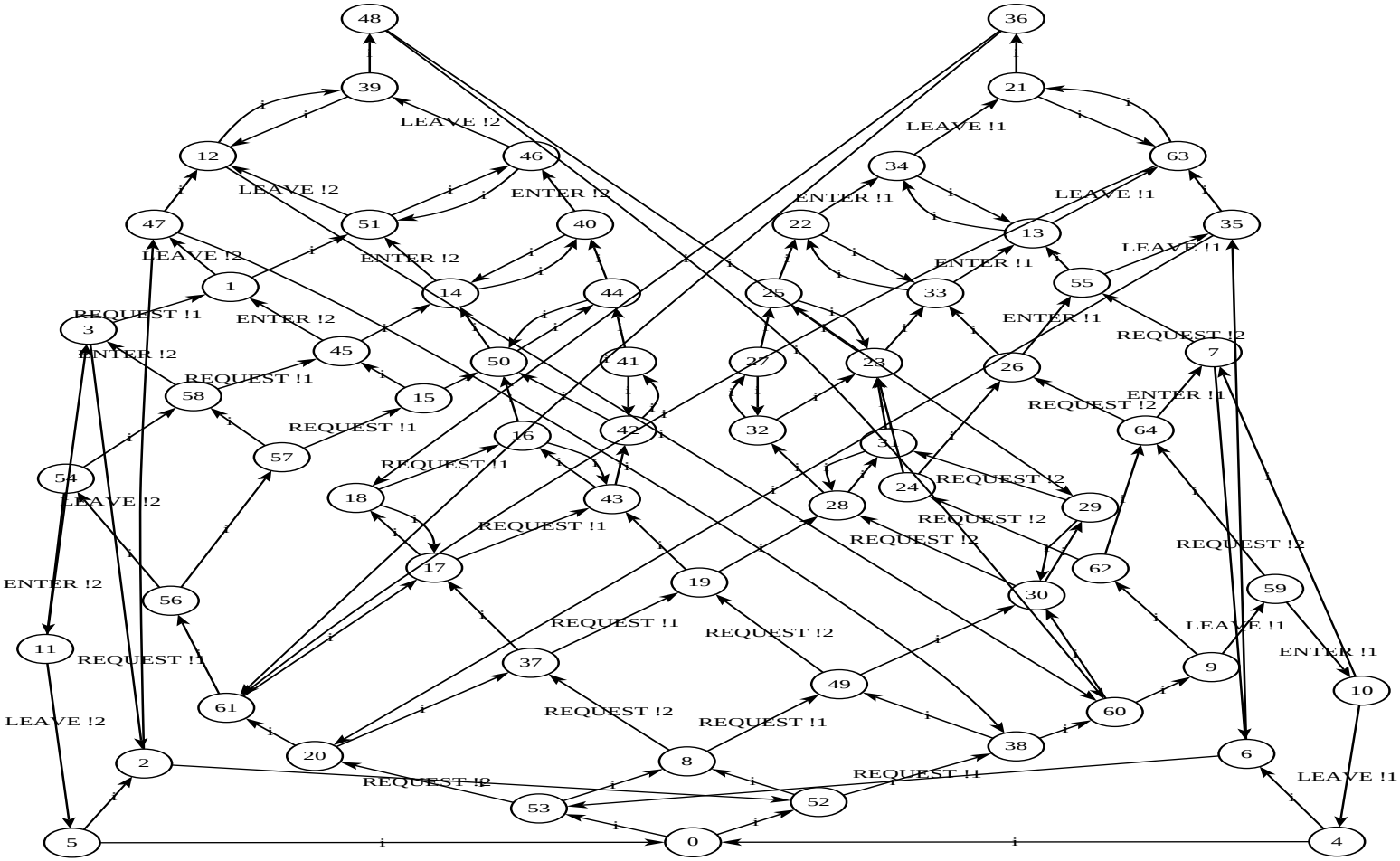
Cas de test TTCN

Test Case Dynamic Behaviour

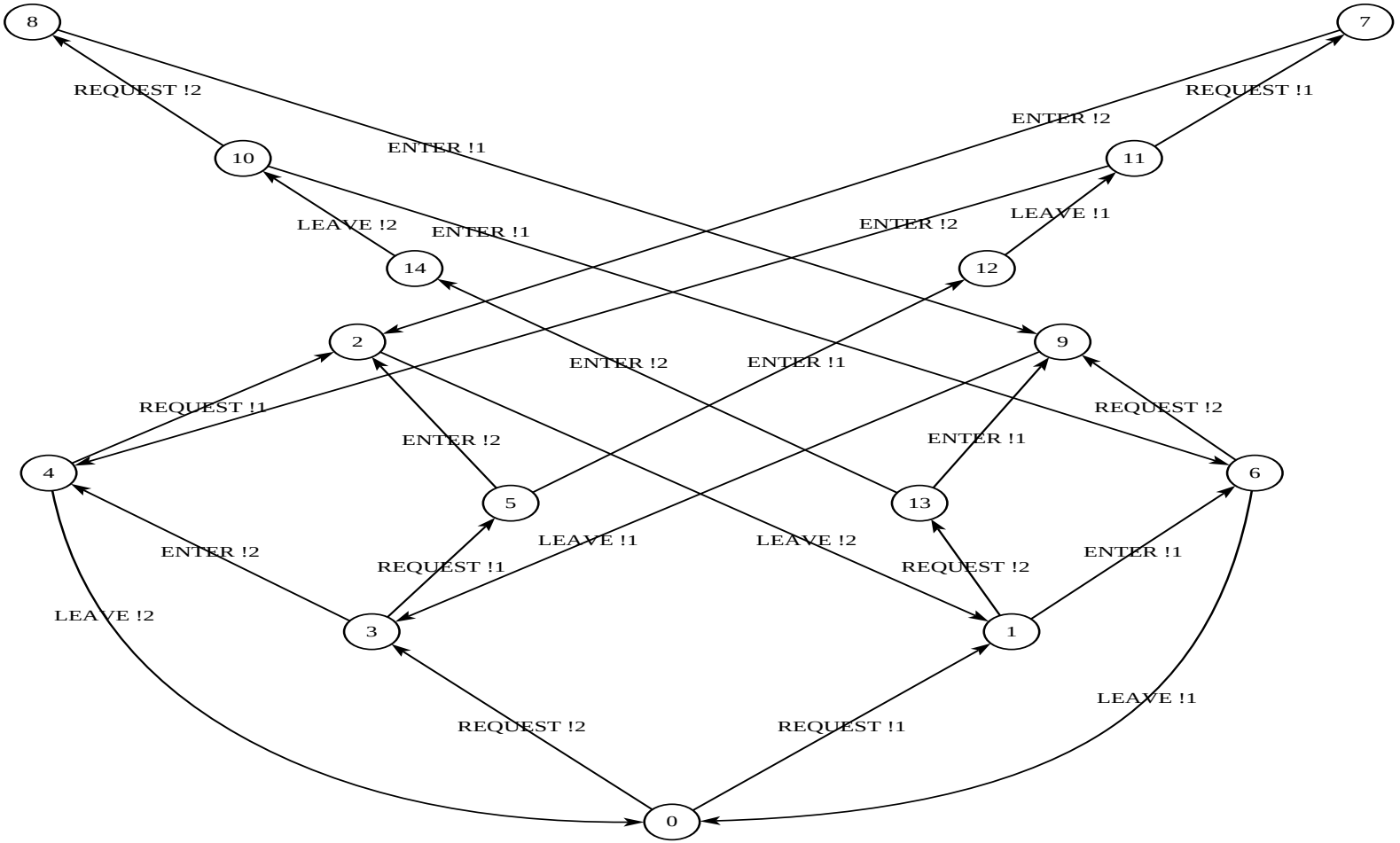
Test Case Name	: graphes/t01_fly_obs_2
Group	:
Purpose	:
Default	:
Comments	:

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1	L1	pco! iconreq, St tidisind, St ticon	iconreq0		
2		pco? icon, Cl ticon, Cl tidisind	icon2		
3		pco! iconf, St ticonconf	iconf3		
4		pco? iconconf, Cl ticonconf	iconconf4		
5		pco! idatreq, St tidisind, St tidat	idatreq5		
6		pco? idat, Cl tidat, Cl tidisind	idat6	(PASS)	
7		pco? idisind, Cl tidat, Cl tidisind	idisind1		
8		GOTO L1	idisind1		
9		? tidat		FAIL	
10		? tidisind		FAIL	
11		? ticonconf		FAIL	
12		pco? idisind, Cl ticon, Cl tidisind	idisind1		
13		GOTO L1	idisind1		
14		? ticon		FAIL	
15		? tidisind		FAIL	

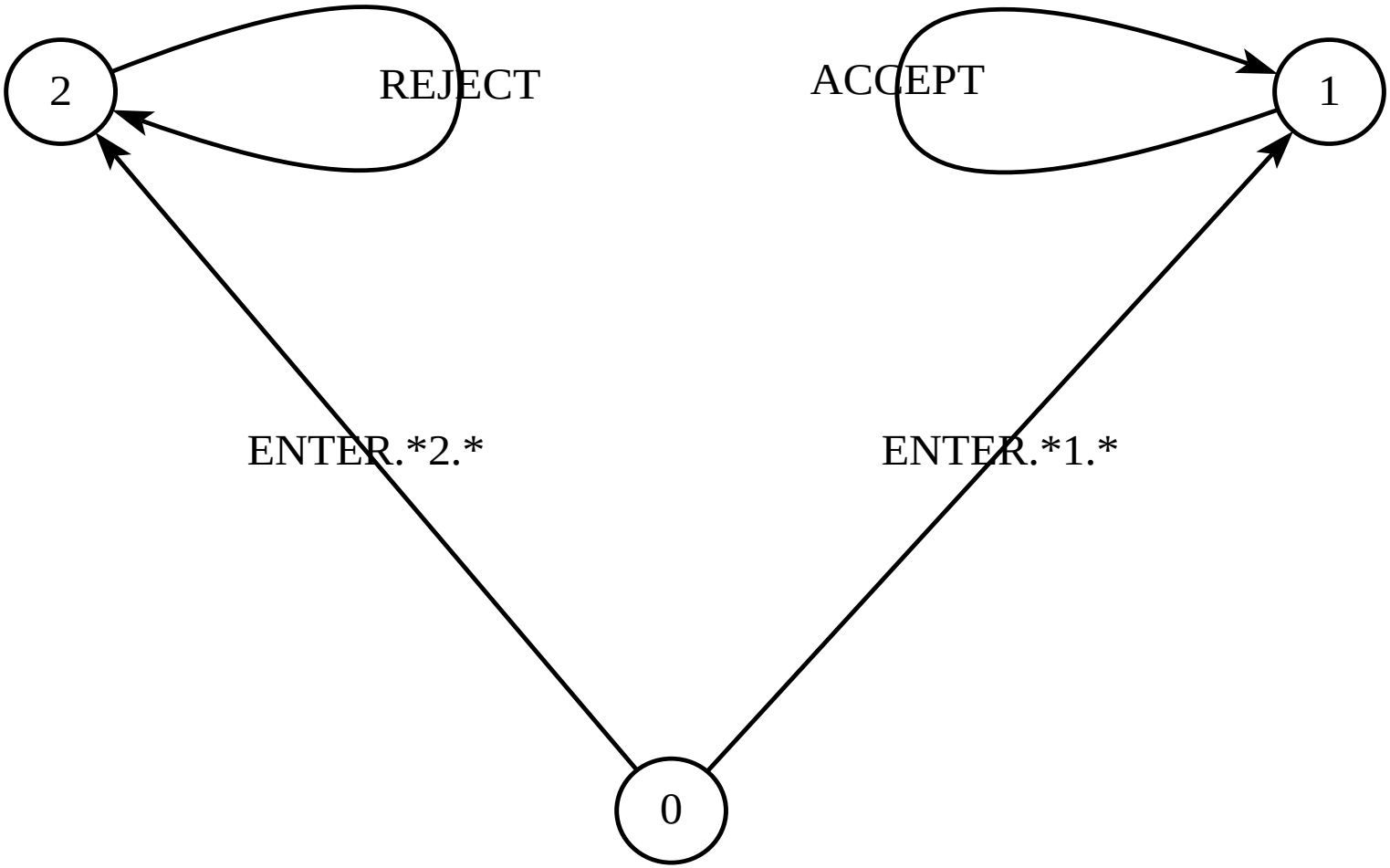
Mutex : graphe d'états modulo équivalence forte



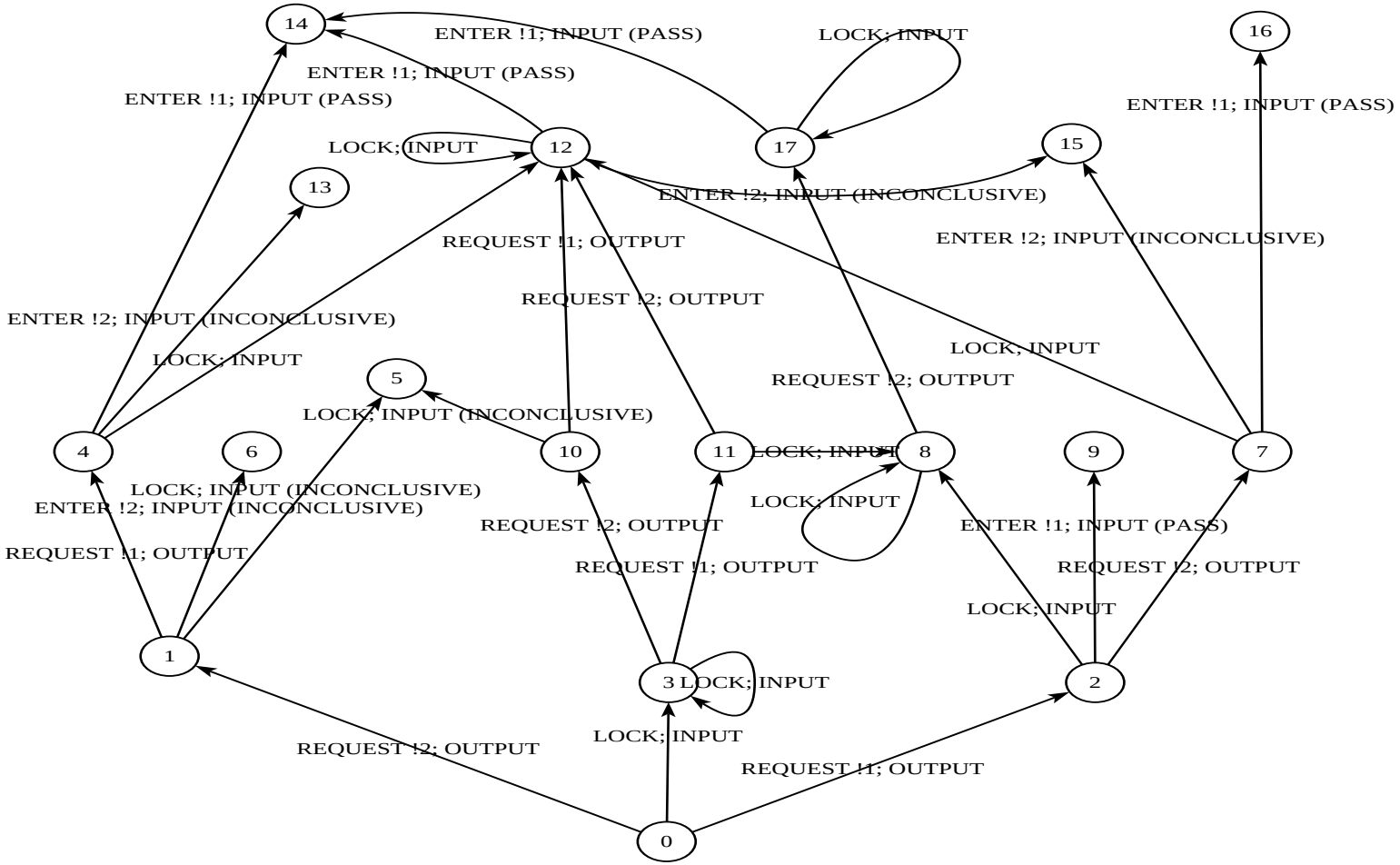
Mutex : graphe d'états modulo $\tau^*.a$



Objectif de test



Graphe de test complet



Cas de test (sans et avec timers)