

# Modélisation & Vérification

## Test Basé sur les Modèles

Asma BERRIRI

asma.berriri@universite-paris-saclay.fr

Page web du cours :

<https://sites.google.com/view/asmaberriri/home>

université  
PARIS-SACLAY



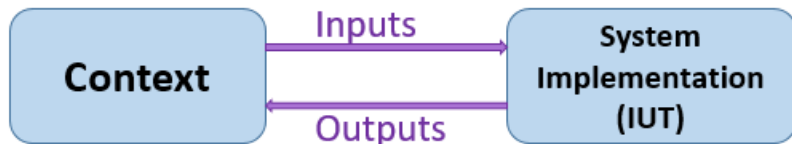
POLYTECH<sup>+</sup>  
PARIS-SACLAY

## 1 Test Fonctionnel: Model Based Testing

## 2 Méthodes de Dérivation des Tests (FSM based)

- Tour de Transition (TT)
- Méthodes d'identification d'états
  - W-Method
  - Distinguishing Sequence (*DS*) Method
  - Unique Input Output (UIO) Method

# MBT: L'idée derrière les tests de conformité



## Context peut être...

- Tout ce avec quoi le système logiciel est censé interagir
- Un matériel spécifique, un logiciel, un environnement...

## L'idée...

- Simuler le 'Context'
- 'Context' envoie des données «cruciales»  
// à partir de **suites de tests**

# MBT: Mettons-nous d'accord!

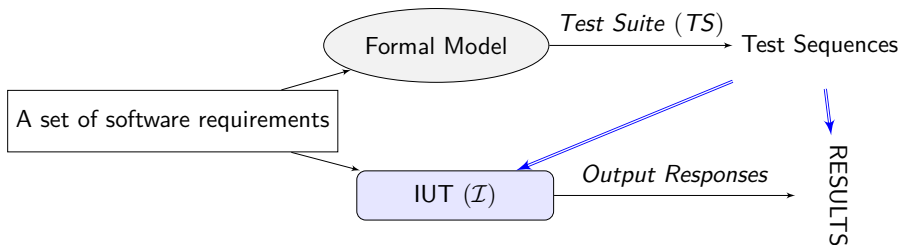
- **Testing** est le processus d'application des entrées (inputs), d'observation des réponses (outputs) et de tirer des conclusions
- **Conclusions** sont faites sur la présence ou l'absence de bugs (de types spécifiques) dans une implémentation sous test (*IUT*)
- **Test suite** est formé de séquences de test ou de cas de test
- **Test sequence** est une séquence d'entrées (inputs) à appliquer à une *IUT*

# MBT: Mettons-nous d'accord!

- Nous verrons dans ce cours les tests fonctionnels  
Un ensemble d'exigences pour le système testé est donné
- Nous verrons les fautes qui peuvent être détectés par une suite de tests  
Un ensemble de tels défauts forme le fault coverage (couverture de fautes)
- Notre objectif  
Est d'augmenter le fault coverage
- Nous nous intéressons à la génération de tests à partir de FSM  
En supposant que le test généré peut toujours être exécuté

# Tests basés sur des modèles formels

Modèle = Bon Formalisme Mathématique



## Testing steps:

- ↪ Deriving test sequences
- ↪ Applying the test sequences against a given *IUT*
- ↪ Drawing a conclusion about the correctness of the *IUT*

we need high quality test suites!

# Pour garantir la couverture des fautes

## Fault model $\mathcal{FM} = \langle \mathcal{S}, @, \mathcal{FD} \rangle$

- ↪  $\mathcal{S}$  is the specification
- ↪  $@$  is the conformance relation
- ↪  $\mathcal{FD}$  is the fault domain that contains all possibly faulty implementations

## Estimate the fault coverage of a $TS$ ?

- ↪ Mutation analysis:
  - Inject 'artificial' faults, called mutations, into the code or the specification model yielding mutants
  - Support test generation
  - Evaluate  $TS$ s effectiveness

# Pour garantir la couverture des fautes

**Objectif: w.r.t.  $\mathcal{FM}$ , une suite de tests  $TS$  tel que**

- *Sound:*

- ↪ Le verdict 'pass' est renvoyé pour chaque implémentation  $\mathcal{I}$  conforme à  $\mathcal{S}$
- ↪  $\forall \mathcal{I} \in \mathcal{FD}, \mathcal{I} @ \mathcal{S}$  passes  $TS$

- *Exhaustive:*

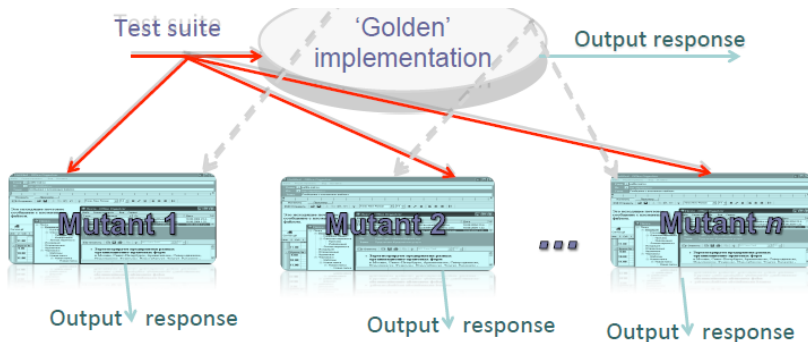
- ↪ Le verdict 'fail' est renvoyé pour chaque implémentation  $\mathcal{I}$  qui n'est pas conforme à  $\mathcal{S}$
- ↪  $\forall \mathcal{I} \in \mathcal{FD}, \mathcal{I} @ \mathcal{S}$  is detected by  $TS$

- *Complete:*

- ↪ Le verdict 'pass' est renvoyé pour chaque implémentation conforme à  $\mathcal{S}$  et Le verdict 'fail' est renvoyé pour chaque implémentation qui n'est pas conforme à  $\mathcal{S}$
- ↪  $\forall \mathcal{I}_1, \mathcal{I}_2 \in \mathcal{FD}, \mathcal{I}_1 @ \mathcal{S}$  passes  $TS$  while  $\mathcal{I}_2 @ \mathcal{S}$  fails  $TS$



# Comment estimer la couverture de fautes d'une suite de tests ?



Output response différents de ceux attendus ? Ou du "Golden" ?

Si oui, on dit que le mutant est tué

# Mutants pour estimer la couverture de fautes

- Les mutants tués démontrent la qualité de la suite de tests

$$|\#\_mutants\_killed| / |\#\_mutants| * 100 \%$$

Montre quelque chose mais quoi exactement ?

- Les mutants peuvent avoir un ordre différent - en fonction du nombre de défauts introduits
  - Souvent, les mutants du premier ordre (une seule faute) sont considérés
- Penser à
    - Quels mutants générer ?
    - Comment les générer ? Tous ou quelque uns ?
    - Et si un mutant est équivalent ?

# Les mutants différents ...

## Code transformation

Les mutants concernent généralement des changements d'instruction

Changement « classiques » :

- un opérateur
- un opérande
- une fonction appelée
- suppression d'instructions

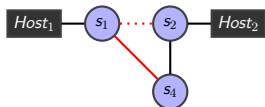
## Model transformation

Les mutants concernent généralement des changements dans le modèle formel décrivant les exigences du système

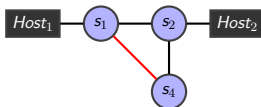
Si le modèle est “graph-like”, les mutants peuvent changer :

- supprimer une arête (lien)
- rediriger une arête
- créer une arête inexistante

Types of Faults: an edge in the implementation



is wrongly directed



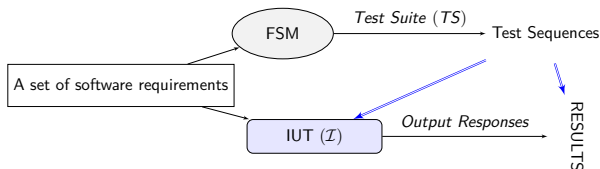
appear



disappear

# Tests basés sur des FSM maintenant ...

modèle = Bon Formalisme pour les systèmes réactifs = **choisissons la machine à états finis FSM**

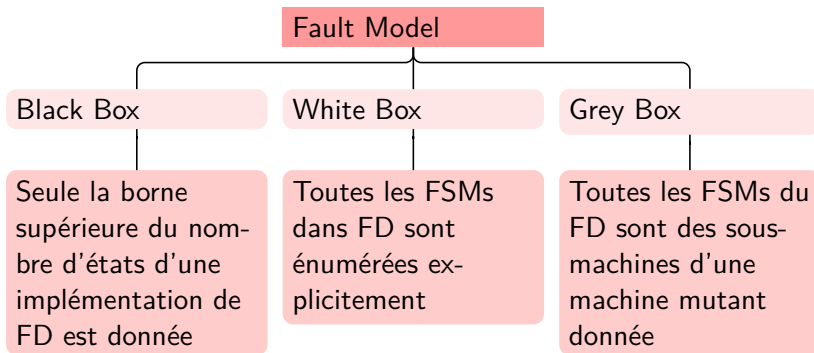


- ↪ **Deriving test sequences**
- ↪ Applying the test sequences against a given *IUT*
- ↪ Drawing a conclusion about the correctness of the *IUT*

**The quality of test suites is its fault coverage**

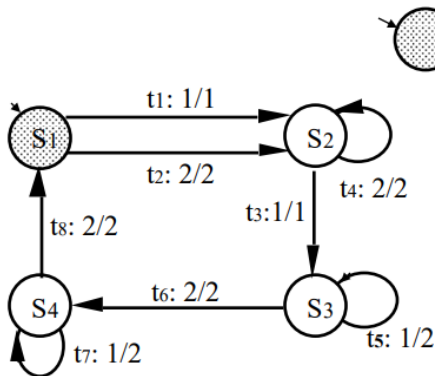
// the set of mutants that can be killed by an application of TS


# Les domaines de fautes "Fault models" pour les FSMs

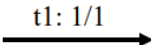


- Les méthodes de génération de tests ( $DS$ ,  $W$ ,  $W_p$ ,  $UIO$ , ...) sont pour ce modèle

# FSM: Rappel (1)



 S1 is an initial state

 Is a transition:

It has a starting state S1, and an ending state S2.

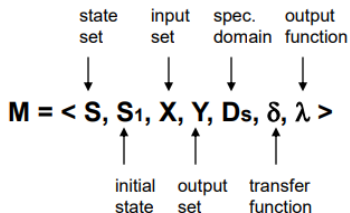
Its label is t1.

The input is 1 and the output is 1.

It separates the input from the output

# FSM: Rappel (2)

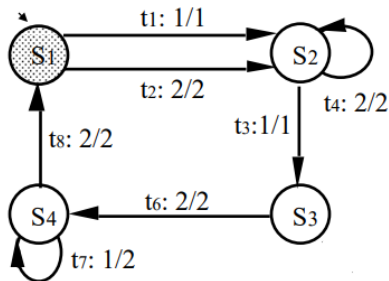
## Mealy Machine



$$D_s \subseteq S \times X$$

$$\delta: D_s \rightarrow S$$

$$\lambda: D_s \rightarrow Y$$



$$S = \{S_1, S_2, S_3, S_4\}$$

$$X = \{1, 2\}$$

$$Y = \{1, 2\}$$

$$D_s = S \times X - \{ \langle S_3, 1 \rangle \}$$

partially defined (specified), deterministic, initialized

- Un FSM sert à spécifier l'exigence ou la conception correcte d'un système (application). Par conséquent, les tests générés à partir d'un FSM ciblent les fautes liés au FSM lui-même
- Quelles sont les fautes visées par les tests générés à l'aide d'un FSM ?
  - 1 Output fault
  - 2 Transfer fault
  - 3 Transfer fault with additional states
  - 4 Additional or missing transitions
  - 5 Additional or missing states



# Fautes pour FSM (1)

## Output Fault on transition $t_1$

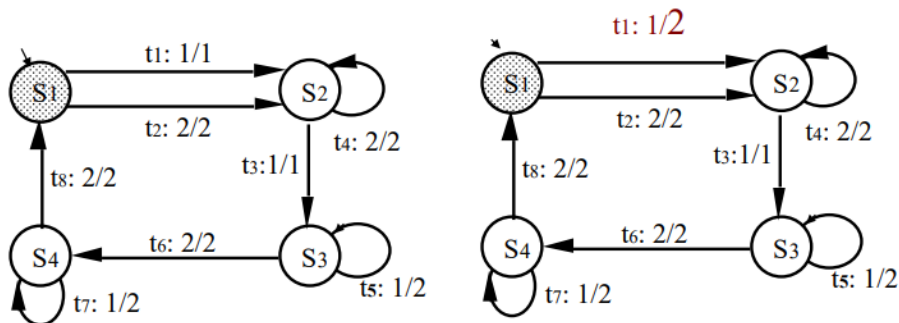


Figure: Specification  $S$  (à gauche), IUT  $I_1$  (à droite)

# Fautes pour FSM (2)

Transfer Fault on transition  $t_2$ , the ending state is now  $S_3$

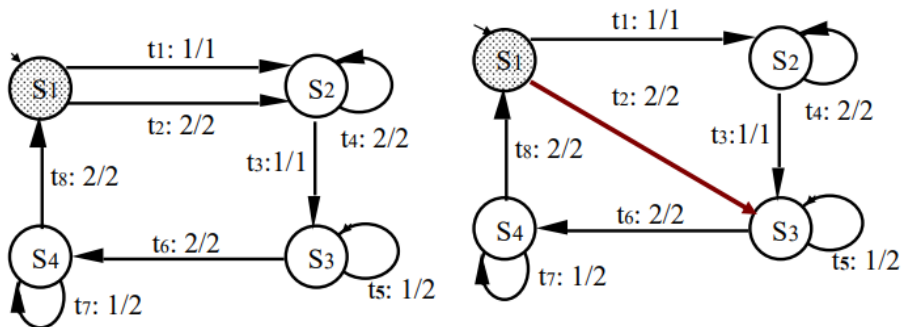


Figure: Specification  $S$  (à gauche),  $IUT I_2$  (à droite)

# Fautes pour FSM (3)

Transfer Fault on transition  $t_5$ , with additional state

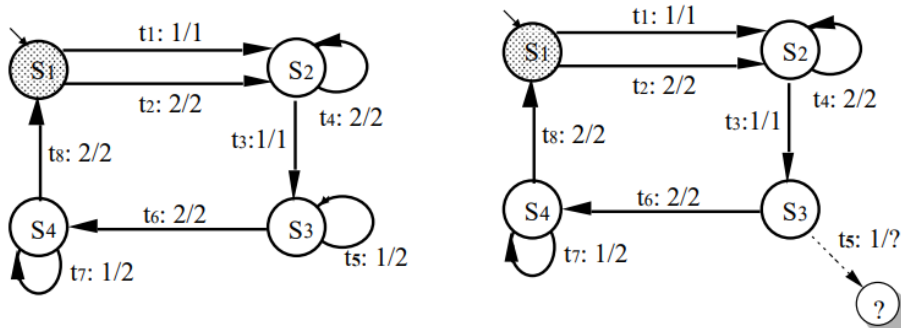
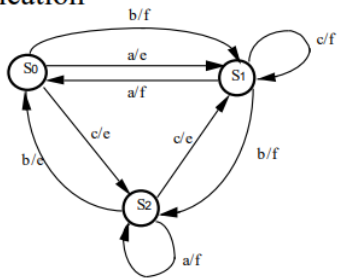


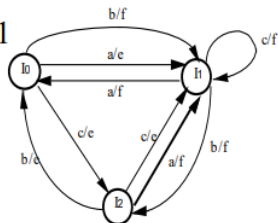
Figure: Specification  $S$  (à gauche), IUT  $I_3$  (à droite)

# Fautes pour FSM (4): Implémentation avec état supplémentaire

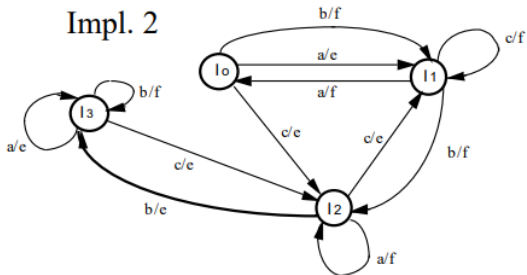
Specification



Impl. 1



Impl. 2

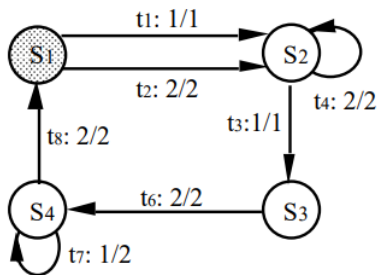


Test de conformité: Garantir au moyen de test qu'une implémentation  $I$  est conforme à sa spécification  $S$

## Une procédure générale pour les tests de conformité

- ↪ Dériver des séquences de "états-transitions" à partir de  $S$ 
  - Transformez chaque état-transition en une séquence de test
  - Testez  $I$  avec une séquence de test pour observer si  $I$  possède ou non la séquence de transitions correspondante
  - **!NB!** Nous supposons l'existence d'une transition de réinitialisation sans sortie ( $r/-$ ) conduisant à l'état initial pour chaque état de  $S$
- ↪ La conformité de  $I$  avec  $S$  peut être vérifiée en choisissant suffisamment de séquences de "états-transitions"

# Exemple d'un TS



**A test suite is a set of input sequences starting from the initial state of the machine**

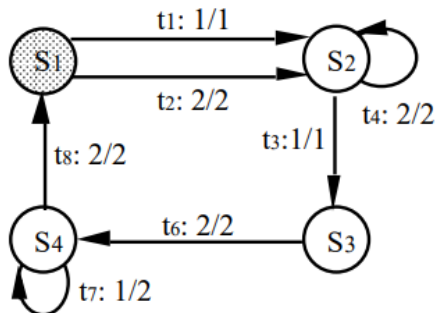
**TS = { r.1.1.2.1, r.2.2.1.2.2 }**

# Exemple d'application d'un $TS$ (1/2)

- $TS = \{r.1.1.2.1, r.2.2.1.2.2\}$

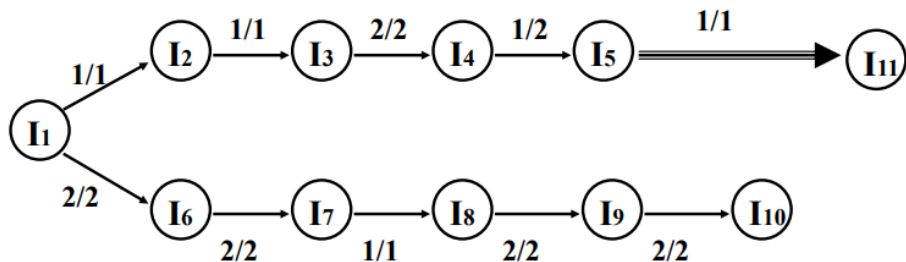
Test Case	Expected output
r.1.1.2.1	1.1.2.2
r.2.2.1.2.2	2.2.1.2.2

- Expected output sequences:  
1.1.2.2, 2.2.1.2.2



## Exemple d'application d'un $TS$ (2/2)

- $TS = \{r.1.1.2.1, r.2.2.1.2.2\}$
- Expected output sequences: 1.1.2.2.2, 2.2.1.2.2



---

Consider input sequence  $r.1.1.2.1.1$

Expected output sequence 1.1.2.2.2

Observed output sequence 1.1.2.2.1



## 1 Tour de Transition

- Garantit la couverture uniquement pour les fautes de sortie

## 2 Méthodes utilisant l'identification d'état

- Avec couverture garantie pour les fautes de sortie et de transfert
  - Le nombre d'états est le même pour l'implémentation  $I$  et la spécification  $S$
  - Le nombre d'états de  $I$  est peut-être plus grand que de  $S$ , mais borné

## 3 Méthodes sans garantie de couverture

- Suite de test faite à la main sans procédure de dérivation de test

## 1 Test Fonctionnel: Model Based Testing

## 2 Méthodes de Dérivation des Tests (FSM based)

- Tour de Transition (TT)
- Méthodes d'identification d'états
  - W-Method
  - Distinguishing Sequence (*DS*) Method
  - Unique Input Output (UIO) Method

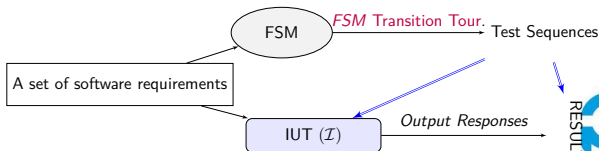
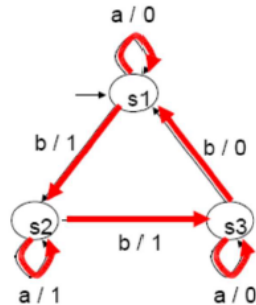
## *Méthodes de dérivation des tests à partir de FSM*

### *1. Transition Tour TT*

# Tour de Transition ( $TT$ )

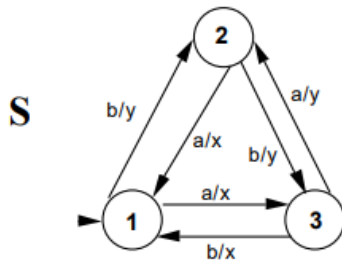
Pour un FSM  $\mathcal{S}$  donné, un tour de transition est une séquence qui prend  $\mathcal{S}$  à **partir de l'état initial**, traverse chaque transition **au moins une fois** et revient à l'état initial

- **Objectif** : dériver un ensemble de séquences de test (input sequences)
- **Idée**: couvrir chaque transition au moins une fois
- **Couverture de fautes**: détecte toutes les fautes de sortie (output faults). Aucune garantie sur les erreurs de transfert



# Tour de Transition: Example-1 (1)

Soit la spécification *FSM* suivante:

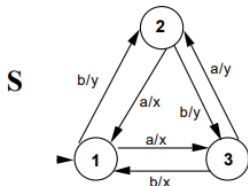


**The specification S**

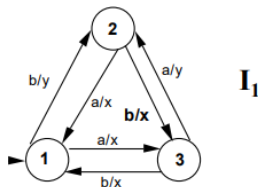
- 1 Dérivez (manuellement) un Tour de Transition pour  $\mathcal{S}$
- 2 Ecrire un algorithme qui simule cet *FSM*, puis une procédure qui dérive automatiquement un Tour de Transition

# Tour de Transition: Example-1 (2)

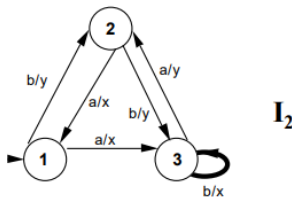
- Les implémentations  $FSMs$   $I_1$  et  $I_2$  suivantes contiennent des fautes
- Le Tour de Transition dérivé peut-il détecter ces fautes?



The specification **S**



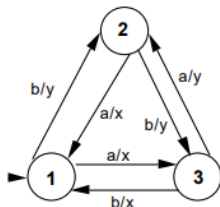
The implementation  **$I_1$**  contains an output error.



The implementation  **$I_2$**  contains a transfer error

# Tour de Transition: Example-1 (3)

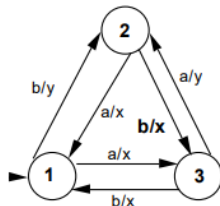
**S**



**The specification S**

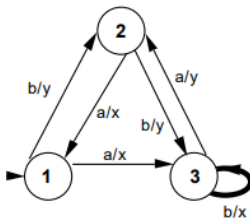
**A transition tour is :  
a.a.a.b.b.b**

**I<sub>1</sub>**



**The implementation I<sub>1</sub> contains an output error. Our transition tour will detect it.**

**I<sub>2</sub>**



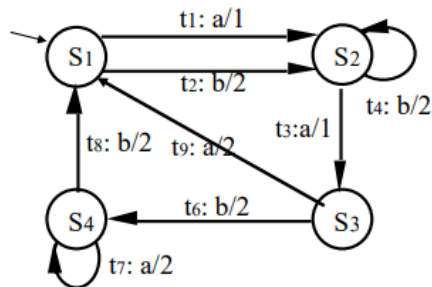
**The implementation I<sub>2</sub> contains a transfer error. Our transition tour will not detect it.**

I\*

# Tour de Transition: Exemple-2 (1)

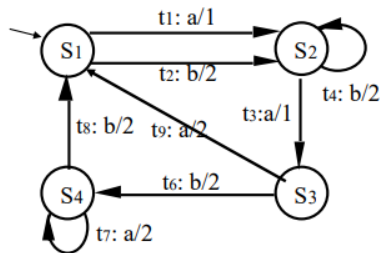
Soit la spécification *FSM* suivante:

- 1 Dérivez un Tour de Transition *TT* pour  $\mathcal{S}$
- 2 Donnez la séquence (input/expected output)





## Tour de Transition: Exemple-2 (2)

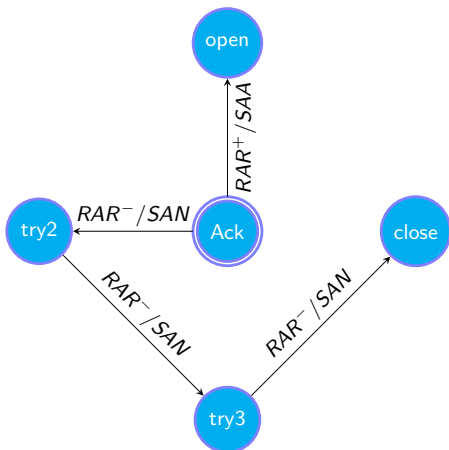


Transition tour

TT:  $t_1, t_4, t_3, t_9, t_2, t_3, t_6, t_7, t_8$

TT (input/expected output):  $a/1.b/2.a/1.a/2.b/2.a/1.b/2.a/2.b/2$

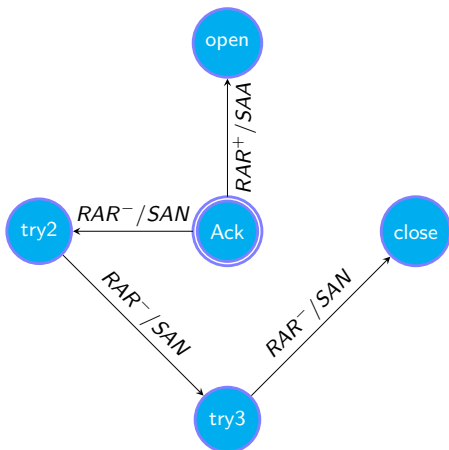
# Tour de Transition: Exemple-3 (PAP)



$RAR^+$ :	"good" login	$SAA$ :	Ack
$RAR^-$ :	"bad" login	$SAN$ :	Nack

- Specification  $\mathcal{S}$
- Test Suite  $TT$ :  
à vous de jouer!
- Expected output responses:  
à vous de jouer!
- Observed output responses:  
à vous de jouer!

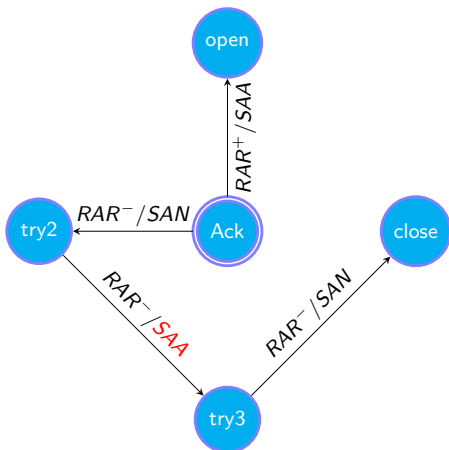
# Tour de Transition: Exemple-3 (PAP)



$RAR^+$ :	"good" login		SAA:	Ack
$RAR^-$ :	"bad" login		SAN:	Nack

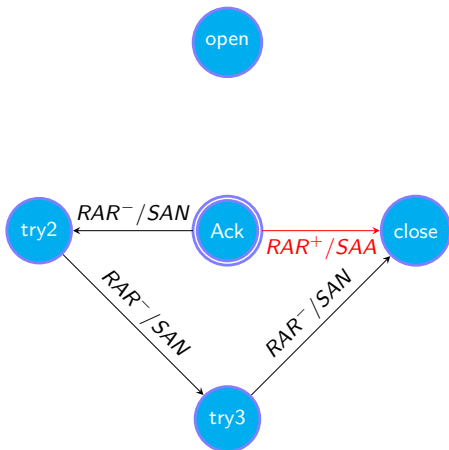
- Specification  $\mathcal{S}$
- Test Suite  
 $RAR^+ RAR^- RAR^- RAR^-$
- Expected output responses  
SAA SAN SAN SAN
- Observed output responses  
SAA SAN SAN SAN

# Tour de Transition: Exemple-3 (PAP)



- Implementation  $I_1$
- Test Suite  
 $RAR^+ RAR^- RAR^- RAR^-$
- Expected output responses  
SAA SAN **SAN** SAN
- Observed output responses  
SAA SAN **SAA** SAN

# Tour de Transition: Exemple-3 (PAP)



- Test Suite  
 $RAR^+ RAR^- RAR^- RAR^-$
- Expected output responses  
SAA SAN SAN SAN
- Observed output responses  
SAA SAN SAN SAN

!! Une faute de transfert ne peut pas être détectée par un TT!

- Que faire alors ? Comment détecter chaque faute de transfert ?

*Méthodes de dérivation des tests à partir de FSM*

*2. Méthodes basée sur l'identification d'état*

- $\mathcal{S} = \langle S, I, O, f, g, S_0 \rangle$  est un *FSM*
- Deux fonctions associées à une transition
  - ↪  $f : S \times I \rightarrow S$  is a behaviour relation that maps a state/input pair to the next state  $\Rightarrow$  Fonction de l'état suivant  $\Rightarrow$  Ignoré dans la méthode TT (inconvenient de la méthode TT)
  - ↪  $g : S \times I \rightarrow O$  is an output function that maps a state/input pair to an output  $\Rightarrow$  Fonction de sortie  $\Rightarrow$  facile à vérifier dans la méthode TT

# Méthodologie générale pour les méthodes d'identification d'état

- Génération de test basée sur la spécification
  - Trouver l'ensemble de couverture d'état : entrées minimales qui atteignent un état à partir de l'état initial
  - Trouver l'ensemble de couverture de transitions: qui couvrira toutes les transitions restantes

## Générer des suites de tests à l'aide de ces ensembles

- Détection de fautes
  - Appliquer les suites de tests générées à la spécification pour obtenir les **résultats attendus**
  - Appliquer les suites de tests générées à l'implémentation pour obtenir les **résultats observés**

Comparer les résultats attendus et observés (résultats des tests)

S'ils sont différents, le verdict est un échec ; sinon c'est une réussite pour les *TSs* appliqués



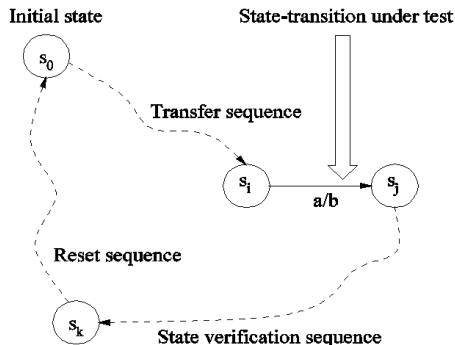


# Méthodologie générale pour les méthodes d'identification d'état

(1) Reaching each *FSM* state  $s_i$

(2) Traversing a single transition to check the output and final state  $s_j$

(3) Distinguishing  $s_j$  from any other *FSM* state



- Non seulement chaque transition doit être couverte, mais aussi une séquence distinguant chaque paire d'états doit être appliquée
- Cette hypothèse est utilisée dans les méthodes *DS*, *W*, *UIO*, lorsque le nombre d'états de l'*IUT* ne dépasse pas celui de la *Spec*

# Méthodes d'identification d'état

## Hypothèses

- H1)  $\mathcal{S}$  est **complètement** spécifié, **minimal**, **connexe** et **déterministe**
- H2)  $\mathcal{S}$  démarre dans un **état initial fixe**
- H3)  $\mathcal{S}$  peut être **réinitialisé avec précision à l'état initial**. Une sortie **null** est générée pendant l'opération de réinitialisation
- H4)  $\mathcal{S}$  et l'implémentation testée  $\mathcal{I}$  ont le **même alphabet d'entrée**
- H5) Le nombre d'états est le même pour  $\mathcal{S}$  et  $\mathcal{I}$  (Le nombre d'états de  $\mathcal{I}$  peut-être plus grand, mais borné)
  - La longueur de test est polynomiale uniquement lorsque  $|Imp\_States| \leq |Spec\_States|$
  - Sinon, la longueur de test est exponentielle w.r.t la différence  $|Imp\_States| - |Spec\_States|$

Méthodes appliquées en deux phases à partir de l'état initial

- phase-1: *Sequence* pour vérifier que chaque état défini par la spécification existe aussi dans l'implémentation
- phase-2: *Sequence* pour vérifier toutes les transitions individuelles dans la spécification pour une sortie et un transfert corrects dans l'implémentation

*Méthodes de dérivation de tests à partir de FSM*

*2. Méthodes basée sur l'identification d'état*

*2.1 W-Method*

## Etapes de la méthode $W$

- ① Etape 1 : Estimer le nombre maximal d'états ( $m$ ) dans l'implémentation  $\mathcal{I}$  du FSM donné  $\mathcal{S}$
- ② Etape 2 : Construire l'ensemble de caractérisation  $W$  pour  $\mathcal{S}$
- ③ Etape 3 :
  - ③.1 Construire l'arbre de test pour  $\mathcal{S}$  et
  - ③.2 Générer l'ensemble de couvertures de transition  $P$  à partir de l'arbre de test
- ④ Etape 4 : Construire l'ensemble  $Z$  à partir de  $W$  et  $m$
- ⑤ Etape 5 :
  - ⑤.1 Génération des tests
  - ⑤.2 Exécution des tests

# Etape 1: Estimation de $m$

- Ceci est basé sur une connaissance de l'implémentation
- En l'absence d'une telle connaissance, soit  $m = |S|$  (nombre d'états dans le FSM  $\mathcal{S}$ )

## Etape 2: Construction de $W$

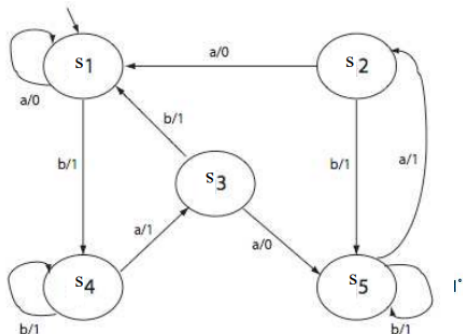
- ① Etape 2.1 : Construction des *partitions  $K$  – Equivalence*
- ② Etape 2.2 : Dérivation de l'*ensemble de caractérisation*

# Qu'est-ce qu'un ensemble de caractérisation ?

- Soit  $\mathcal{S} = \langle I, O, S, S_0, f, g \rangle$  un FSM minimal et complet
- $W$  est un ensemble fini de séquences d'entrée qui distinguent le comportement de tout paire d'états dans  $\mathcal{S}$ .
  - Chaque séquence d'entrée dans  $W$  est de longueur finie
- Etant donnés  $s_i$  et  $s_j \in S$ ,  $W$  contient un mot  $\alpha$  tel que  $g(s_i, \alpha) \neq g(s_j, \alpha)$

## Exemple d'ensemble de caractérisation $W$

- $W = \{baaa, aa, aaa\}$
- E.g.,  $baaa$  distingue l'état  $s_1$  de  $s_2$  comme  $g(s_1, baaa) \neq g(s_2, baaa)$
- Plus précisément,  
 $g(s_1, baaa) = 1101$  et  
 $g(s_2, baaa) = 1100$



# Qu'est-ce qu'une partition $k$ -équivalence de $S$ ?

- Étant donné  $\mathcal{S} = \langle I, O, S, S_0, f, g \rangle$  un FSM
- Partition  $k$  – *equivalence* de  $S$  notée  $P_k$ , est une collection de  $n$  ensembles finis

$$\sum_{k1}, \sum_{k2}, \dots, \sum_{kn}$$

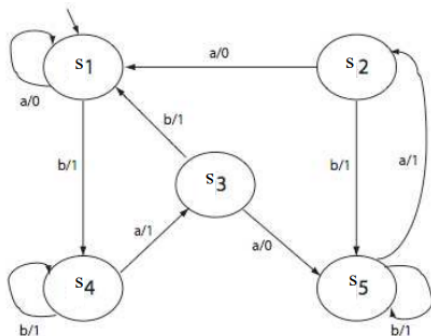
tels que

- $\bigcup_{i=1}^n \sum_{ki} = S$
- les états dans  $\sum_{ki}$  pour  $1 \leq i \leq n$  sont  $k$  – *equivalents*
- Si l'état  $u$  est dans  $\sum_{ki}$  et  $v$  dans  $\sum_{kj}$  pour  $i \neq j$ , alors  $u$  et  $v$  sont  $k$  – *distinguishables*



# Construction de 1 – *equivalence* partition (1)

- Construction de 1 – *equivalence* partition,  $P_1$
- Commencer avec une représentation tabulaire de  $\mathcal{S}$



Current state	Output		Next state	
	a	b	a	b
s1	0	1	s1	s4
s2	0	1	s1	s5
s3	0	1	s5	s1
s4	1	1	s3	s4
s5	1	1	s2	s5

# Construction de 1 – equivalence partition (2)

- Grouper les états identiques dans leurs **sorties**. Cela nous donne une partition  $P_1$  composée de  $\sum_1 = \{s_1, s_2, s_3\}$  et  $\sum_2 = \{s_4, s_5\}$
- Nous avons la partition à 1 – equivalence comme suit
  - $P_1 = \{1, 2\}$
  - $Groupe_1 = \sum_{11} = \{s_1, s_2, s_3\}$
  - $Groupe_2 = \sum_{12} = \{s_4, s_5\}$

$\Sigma$	Current state	Output		Next state	
		a	b	a	b
1	s1	0	1	s1	s4
	s2	0	1	s1	s5
	s3	0	1	s5	s1
2	s4	1	1	s3	s4
	s5	1	1	s2	s5

## Construction de 2 – equivalence partition (1)

- **Réécrire le tableau  $P_1$ .** Supprimez les colonnes de sortie. Remplacer une entrée d'état  $s_i$  par  $s_{ij}$  où  $j$  est le numéro de groupe dans lequel se trouve l'état  $s_i$

$\Sigma$	Current state	Next state	
		a	b
1	s1	s11	s42
	s2	s11	s52
	s3	s52	s11
2	s4	s31	s42
	s5	s21	s52

group number

state  $s_5$  is in group 2

state  $s_1$  is in group 1

## Construction de 2 – *equivalence* partition (2)

- **Construire le tableau  $P_2$ .** Regroupez toutes les entrées avec des seconds indices identiques sous la **colonne Next State**. Cela nous donne la table  $P_2$ .
  - Notez le changement dans les seconds indices
  - Nous avons trois groupes dans le tableau  $P_2$

$\Sigma$	Current state	Next state	
		a	b
1	s1	s11	s43
	s2	s11	s53
2	s3	s53	s11
3	s4	s32	s43
	s5	s21	s53

$P_2$  Table

state  $s_5$  is in group 3

# Construction de 3 – *equivalence* partition

- **Construire le tableau  $P_3$ .** Regroupez toutes les entrées avec des seconds indices identiques sous la **colonne Next State**. Cela nous donne la table  $P_3$ .
  - Notez le changement dans les seconds indices
  - Nous avons quatre groupes dans le tableau  $P_3$

$\Sigma$	Current state	Next state	
		a	b
1	s1	s11	s43
	s2	s11	s54
2	s3	s54	s11
3	s4	s32	s43
4	s5	s21	s54

$P_3$  Table

state  $s_5$  is in group 4

# Construction de 4 – *equivalence* partition

- Construire le tableau  $P_4$ . Continuez de regrouper et renommer.
  - Notez le changement dans les seconds indices
  - Nous avons cinq groupes dans le tableau  $P_4$

$\Sigma$	Current state	Next state	
		a	b
1	s1	s11	s44
2	s2	s11	s55
3	s3	s55	s11
4	s4	s33	s44
5	s5	s22	s55

$P_4$  Table

state  $s_5$  is in group 5

# Partition k-équivalence : Convergence

- Le processus est assuré de **converger**
- Lorsque le processus converge et que la machine est minimale, chaque état sera dans un groupe séparé
- L'étape suivante consiste à obtenir les chaînes **distinctives pour chaque état**

## Etape 2.2 :

Utilisation de la procédure  $W$  pour dériver l'ensemble de caractérisation  $W$

Une procédure pour dériver  $W$  d'un ensemble de tables de partition  
construites à l'Etape 2.1



# Dériver l'ensemble de caractérisation $W$

## Procédure pour dériver $W$

- 1 Soit  $\mathcal{S} = \langle S, I, O, S_0, f, g \rangle$  la FSM pour laquelle  $P = \{P_1, P_2, \dots, P_n\}$  est l'ensemble de tables de partition de  $k$  – *equivalence* pour  $k = 1, 2, \dots, n$
- 2 Initialiser  $W = \emptyset$
- 3 Traversez les partitions de  $k$  – *equivalence* dans l'ordre inverse pour obtenir une **séquence distinctive** pour chaque paire d'états

# Trouver les séquences distinctives (1)

A) Trouvons une **séquence distinctive pour les états  $s_1$  et  $s_2$**

- 1 Trouver les tables  $P_i$  et  $P_{i+1}$  telles que  $(s_1, s_2)$  soient dans le même groupe dans  $P_i$  et dans des groupes différents dans  $P_{i+1}$ . On obtient  $P_3$  et  $P_4$
- 2 Initialiser  $z = \epsilon$ . Trouvez le symbole d'entrée qui distingue  $s_1$  et  $s_2$  dans  $P_3$ . Ce symbole est **b**. Nous mettons à jour  $z$  en  $z.b$ . Donc  $z$  devient maintenant **b**
- 3 Les états suivants pour  $s_1$  et  $s_2$  sur  $b$  sont respectivement  $s_4$  et  $s_5$
- 4 Nous constatons que  $s_4$  et  $s_5$  sont dans le même groupe en  $P_2$  et dans des groupes différents en  $P_3$
- 5 Nous passons à la table  $P_2$  et trouvons le symbole d'entrée qui distingue  $s_4$  et  $s_5$ . Choisissons **a** comme symbole distinctif. Mettre à jour  $z$  qui devient maintenant **ba**
- 6 Se référer au tableau  $P_2$ , les états suivants pour les états  $s_4$  et  $s_5$  sur le symbole  $a$  sont, respectivement,  $s_3$  et  $s_2$ . Ces deux états se distinguent dans  $P_1$  par **a** et **b** (c'est-à-dire que  $s_2$  et  $s_3$  sont dans le même groupe dans  $P_1$  et dans des groupes différents dans  $P_2$ ). Choisissons **a**. Nous mettons à jour  $z$  en **baa**

## Trouver les séquences distinctives (2)

- 7 Reportez-vous au tableau  $P_1$ . Les états suivants pour  $s_2$  et  $s_3$  sur  $a$  sont respectivement  $s_1$  et  $s_5$
- 8 Nous constatons que  $s_1$  et  $s_5$  sont dans le même groupe dans le tableau d'origine et dans des groupes différents dans  $P_1$
- 9 En passant à la table de transition d'état d'origine, nous obtenons  $a$  comme symbole distinctif pour  $s_1$  et  $s_5$
- 10 Nous mettons à jour  $z$  en  $baaa$ . C'est le plus loin que l'on puisse remonter dans les différents tableaux.  $baaa$  est la **séquence distinctive souhaitée pour les états  $s_1$  et  $s_2$** . Vérifiez que  $g(s_1, baaa) = 1101$  et  $g(s_2, baaa) = 1100$ . On a  $g(s_1, baaa) \neq g(s_2, baaa)$

B) En utilisant la procédure analogue à celle utilisée pour  $s_1$  et  $s_2$ , nous pouvons trouver la séquence distinctive pour chaque paire d'états. Cela nous amène à l'ensemble de caractérisation suivant pour notre FSM. Nous avons les séquences distinctives comme suit

$s_1, s_2 \rightarrow$	$baaa$	$s_2, s_4 \rightarrow$	$a$
$s_1, s_3 \rightarrow$	$aa$	$s_2, s_5 \rightarrow$	$a$
$s_1, s_4 \rightarrow$	$a$	$s_3, s_4 \rightarrow$	$a$
$s_1, s_5 \rightarrow$	$a$	$s_3, s_5 \rightarrow$	$a$
$s_2, s_3 \rightarrow$	$aa$	$s_4, s_5 \rightarrow$	$aaa$

Cela donne  $W = \{a, aa, aaa, baaa\}$

# Où sommes-nous? :)

## Etapas de la méthode $W$

- ① Etape 1 : Estimer le nombre maximal d'états ( $m$ ) dans l'implémentation  $\mathcal{I}$  du FSM donné  $\mathcal{S}$
- ② Etape 2 : Construire l'ensemble de caractérisation  $W$  pour  $\mathcal{S}$
- ③ Etape 3 :
  - ① Construire l'arbre de test pour  $\mathcal{S}$  et
  - ② Générer l'ensemble de couvertures de transition  $P$  à partir de l'arbre de test
- ④ Etape 4 : Construire l'ensemble  $Z$  à partir de  $W$  et  $m$
- ⑤ Etape 5 : Génération et exécution des tests

## Etape 3.1

### Construire l'arbre de test pour $\mathcal{S}$

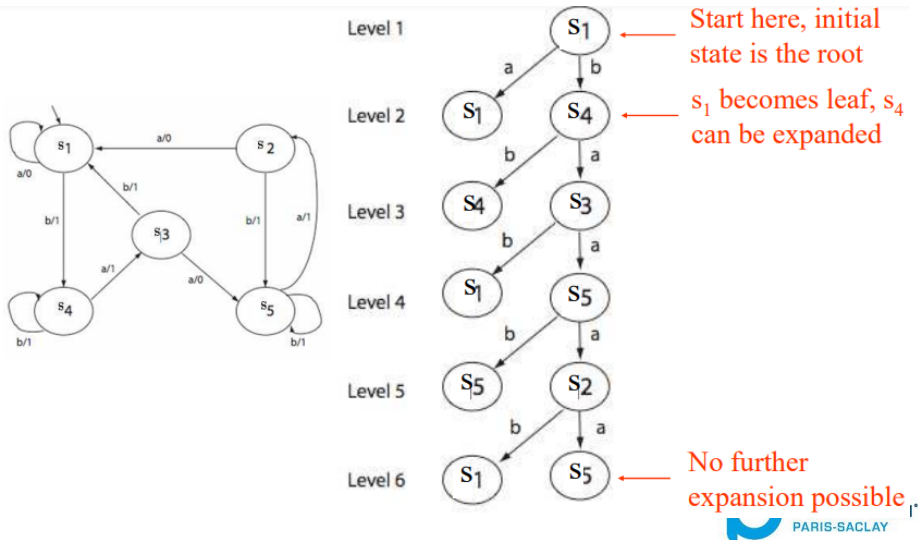
# Construction de l'arbre de test (1)

Un **arbre de test** d'un FSM,  $\mathcal{S} = \langle S, I, O, S_0, f, g \rangle$ , est un arbre dont **la racine est l'état initial**. Il **contient au moins un chemin de l'état initial aux états restants** dans  $\mathcal{S}$

## Construction Arbre de Test

- ❶  $s_0$ , l'état initial, est la racine de l'arbre de test
- ❷ Supposons que l'arbre de test ait été construit jusqu'au niveau  $k$
- ❸ Le  $(k + 1)$ ème niveau est construit comme suit
  - Sélectionnez un noeud  $n$  au niveau  $k$
  - Si  $n$  apparaît à n'importe quel niveau de 1 à  $k$ , alors  $n$  est un noeud feuille et n'est pas développé davantage
  - Si  $n$  n'est pas un noeud feuille alors nous le développons en ajoutant une branche du noeud  $n$  vers un nouveau noeud  $m$  si  $f(n, \alpha) = m$  pour  $\alpha \in I$
  - Cette branche est notée  $\alpha$
  - Cette étape est répétée pour tous les noeuds au niveau  $k$

# Construction de l'arbre de test (2)



# Construction de l'arbre de test (3)

- 1 L'arbre de test est initialisé avec l'état initial  $s_1$  comme noeud racine
- 2 C'est le niveau 1 de l'arbre
- 3 On note que  $g(s_1, a) = s_1$  et  $g(s_1, b) = s_4$ . Par conséquent, nous créons deux noeuds au niveau suivant et les étiquetons  $s_1$  et  $s_4$
- 4 Les branches de  $s_1$  à  $s_1$  et  $s_4$  sont étiquetées, respectivement,  $a$  et  $b$
- 5 Comme  $s_1$  est le seul noeud au niveau 1, nous procédons maintenant au développement de l'arbre pour former le niveau 2
- 6 Au niveau 2, nous considérons d'abord le noeud étiqueté  $s_1$ . Cependant, un autre mode étiqueté  $s_1$  apparaît déjà au niveau 1 ; par conséquent, ce noeud devient un noeud feuille et n'est plus développé.
- 7 Ensuite, nous examinons le noeud étiqueté  $s_4$ . On note que  $g(s_4, a) = s_3$  et  $g(s_4, b) = s_4$ . Nous créons donc deux nouveaux noeuds au niveau 3 et étiquetons-les comme  $s_3$  et  $s_4$  et étiquetons les branches correspondantes comme  $a$  et  $b$ , respectivement



## Etape 3.2

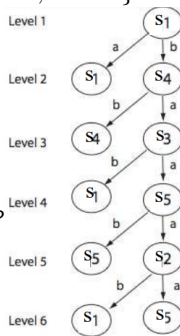
Générer l'ensemble de couvertures de transition  $P$  à partir de l'arbre de test

# Générer l'ensemble de couvertures de transition $P$

Un **ensemble de couverture de transition**  $P$  est un ensemble de toutes les chaînes représentant des sous-chemins, commençant à la racine, dans l'arbre de test.

$$P = \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\}$$

- Un sous-chemin est un chemin partant de la racine de l'arbre de test et aboutissant à n'importe quel noeud de l'arbre
- La concaténation des étiquettes le long des bords d'un sous-chemin est une chaîne qui appartient à  $P$
- Le mot vide ( $\epsilon$ ) appartient aussi à  $P$



# Pourquoi s'appelle-t-il un ensemble de couvertures de transition ?

- Exciter un FSM en  $s_0$ , avec un élément de  $P$ , force le FSM dans un certain état
- Après que le FSM a été excité avec tous les éléments de  $P$ , à chaque fois en commençant à l'état initial, le FSM a atteint chaque état
- Ainsi, exciter un FSM avec des éléments de  $P$  garantit que **tous les états sont atteints** et que **toutes les transitions ont été traversées au moins une fois**
  - La séquence d'entrée vide ne traverse aucune branche mais est utile pour construire une séquence de test souhaitée

## Etape 4

### Construire l'ensemble $Z$ à partir de $W$ et $m$

# Construire l'ensemble $Z$ à partir de $W$ et $m$

- Étant donné que  $I$  est l'alphabet d'entrée et  $W$  l'ensemble de caractérisation. Supposons que le nombre d'états estimés dans l'implémentation testée soit  $m$ , et que le nombre d'états dans la spécification soit  $n$ ,  $m > n$
- Nous calculons  $Z$  comme
$$Z = (I^0.W) \cup (I^1.W) \cup \dots (I^{m-1-n}.W) \cup (I^{m-n}.W)$$
  - Rappelons que  $I^0 = \{\epsilon\}$ ,  $I^1 = I$ ,  $I^2 = I.I$ , et ainsi de suite, où  $(.)$  indique une concaténation de chaînes
- Pour  $m = n$ , on obtient  $Z = I^0.W = W$
- Pour  $m < n$ , on utilise  $Z = IW$

## Etape 5.1

### Génération des tests

# Génération d'une suite de test à partir de $P$ et $Z$

- Les suite (entrées) de test basées sur le FSM  $\mathcal{S}$  peuvent maintenant être dérivées comme  $TS = P.Z$
- Utilisons le même exemple. Supposons  $m = n = 5$ . On a aussi  $I = \{a, b\}$  et  $W$  (l'ensemble de caractérisation)  $= \{a, aa, aaa, baaa\}$ 
  - En concaténant  $P$  avec  $Z$ , on obtient le test désiré
    - $Z = I^0.W = \{a, aa, aaa, baaa\}$  ( $//Rq : I^0 = \{\epsilon\}$ )
    - $TS = P.Z =$   
 $\{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\} \cdot \{a, aa, aaa, baaa\}$
- Si nous supposons que l'implémentation a un état supplémentaire, c'est-à-dire  $m = 6$ , nous avons

$$\begin{aligned} Z &= I^0.W \cup (I^1.W) \\ &= \{a, aa, aaa, baaa, \underline{aa}, \underline{aaa}, aaaa, abaaa, \underline{ba}, \underline{baa}, \underline{baaa}, \underline{bbaaa}\} \\ TS = P.Z &= \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\} \cdot \\ &\quad \{a, aa, aaa, baaa, aa, aaa, aaaa, abaaa, ba, baa, baaa, bbaaa\} \end{aligned}$$



## Etape 5.2

### Exécution des tests



# Exécution du suite de test généré $TS$

Pour tester une implémentation donnée par rapport à sa spécification  $S$ , nous procédons comme suit pour chaque entrée de  $TS$

- Trouver la réponse attendue à chaque élément de  $TS$
- Générer des cas de test pour le système. Note que même si le système est modélisé par  $S$ , il peut y avoir des variables à définir avant de pouvoir l'exercer avec des éléments de  $TS$
- Exécutez le système et vérifiez si la réponse correspond **Réinitialiser le système à l'état initial après chaque test**

NB!

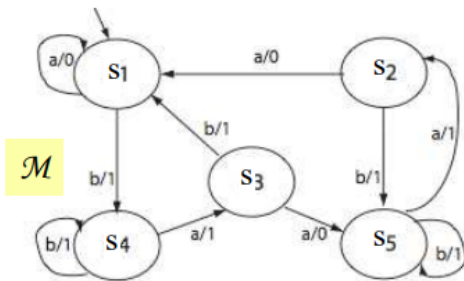
Un mismatch entre la réponse attendue et la réponse réelle n'implique pas nécessairement une erreur dans l'implémentation

- La spécification est-elle exempte d'erreurs ?
- Les réponses attendues et réelles sont-elles déterminées sans aucune erreur ?
- La comparaison entre elles est-elle correcte ?

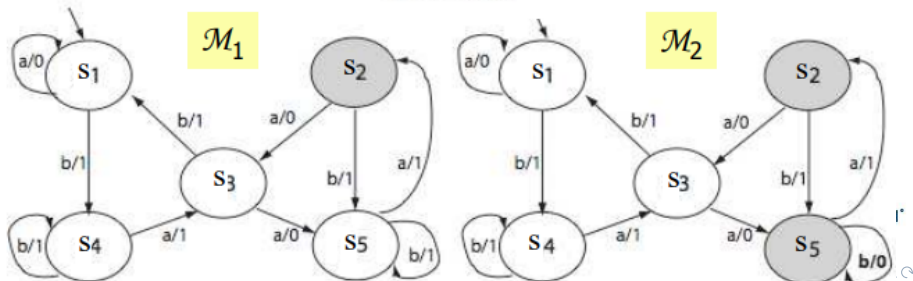
Si la réponse est OUI à toutes ces questions, alors une discordance implique une erreur dans l'implémentation



# W-Method : 1er Exemple: $n = m = 5$



(a) Correct design



# W-Method : 1er Exemple: $n = m = 5$

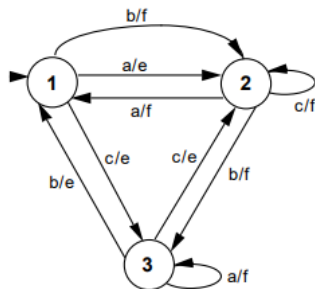
Pour tester  $\mathcal{M}_1$  contre  $\mathcal{M}$ , on applique chaque test  $t$  de l'ensemble  $P.Z$  et on compare  $\mathcal{M}(t)$  avec  $\mathcal{M}_1(t)$

- Nous constatons que lorsque  $t = baaaaa$ ,  $\mathcal{M}(t) = 1101000$  et  $\mathcal{M}_1(t) = 1101001$ . Par conséquent, la séquence d'entrée  $baaaaa$  révèle l'erreur de transfert dans  $\mathcal{M}_1$

De même, pour tester  $\mathcal{M}_2$  contre  $\mathcal{M}$ , on applique chaque test  $t$  de l'ensemble  $P.Z$  et on compare  $\mathcal{M}(t)$  avec  $\mathcal{M}_2(t)$

- Nous constatons que lorsque  $t = baaba$ ,  $\mathcal{M}(t) = 11011$  et  $\mathcal{M}_2(t) = 11001$ . Par conséquent, la séquence d'entrée  $baaba$  révèle l'erreur de transfert dans  $\mathcal{M}_2$

## W-Method : 2ème Exemple: $n = m = 3$ (1/4)



### The specification S

We assume the existence of a reset transition with no output (r/-) leading to the initial state for every state of S

A characterization set is  $W=\{a, b\}$

- W1 state 1 : a/e,
- W2 state 2 : a/f, b/f
- W3 state 3 : b/e

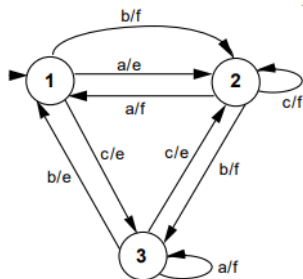
$W = \text{Union of all } W_i$

A transition cover set for the specification S is :

$P=\{e, a, b, c, b.a, b.b, b.c, c.a, c.b, c.c\}$

The W-method generates the following test sequences:  $(P.W) =$   
r.a, r.b, r.a.a, r.a.b, r.b.a, r.b.b, r.c.a,  
r.c.b, r.b.a.a, r.b.a.b, r.b.b.a, r.b.b.b,  
r.b.c.a, r.b.c.b, r.c.a.a, r.c.a.b, r.c.b.a, r.  
r.c.b.b, r.c.c.a, r.c.c.b

# W-Method : 2ème Exemple: $n = m = 3$ (2/4)



## The specification S

We assume the existence of a reset transition with no output (r/-) leading to the initial state for every state of S

Derivation of W

state	1	2	3
a	e	f	f
b	f	f	e

A characterization set is  $W = \{a, b\}$  for W method

- for state 1 : a/e
- for state 2 : a/f, b/f
- for state 3 : b/e

The identification sets are :

- $W_1 = \{a\}$ , distinguishes the state 1 from all other states
- $W_2 = \{a, b\}$ , distinguishes the state 2 from all other states
- $W_3 = \{b\}$ , distinguishes the state 3 from all other states

## W-Method : 2ème Exemple: $n = m = 3$ (3/4)

$$W_1 : \{ a/e \} , \quad W_2 : \{ a/f, b/f \} , \quad W_3 : \{ b/e \}$$

A state cover set for the specification  $S$  is :  $Q = \{\epsilon, b, c\}$

A transition cover set for the specification  $S$  is :

$$P = \{\epsilon, a, b, b.c, b.a, b.b, c, c.a, c.c, c.b\}$$

$$P-Q = \{a, b.c, b.a, b.b, c.a, c.c, c.b\}$$

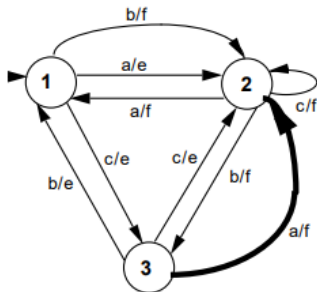
Based on these sets, the Wp-method yields the following test sequences :

- Phase 1:  $Q.W_i = \{r.a.a_1, r.b.a_2, r.b.b_2, r.c.b_3\}$

The ending state  $W_i$  is given in subscript

- Phase 2 :  $(P-Q).W_i = \{r.a.a_2, r.a.b_2, r.b.c.a_2, r.b.c.b_2, r.b.a.a_1, r.b.b.b_3, r.c.a.b_3, r.c.c.a_2, r.c.c.b_2, r.c.b.a_1\}$

## W-Method : 2ème Exemple: $n = m = 3$ (4/4)



A faulty implementation  $I$

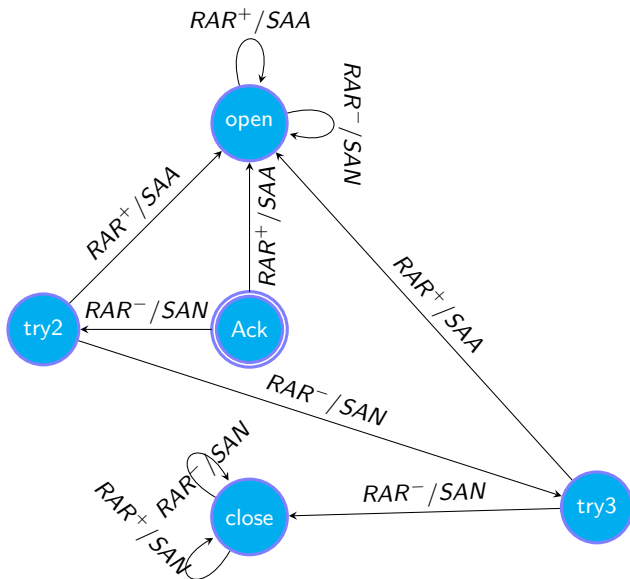
$I$  contains a transfer error 3-a/f->2 (fat arrow) instead of 3-a/f->3 as defined in the specification  $S$

- The application of the test sequences obtained in Phase 2 leads to the following sequences of outputs :

e.f, e.f, f.f.f, f.f.f, f.f.e, f.f.e, **e.f.f**, e.e.f,  
e.e.f, e.e.e

- The output printed in bigger size is different from the one expected according to the specification. Therefore, the transfer error in the implementation is detected by this test sequence.

# W-Method : 3ème Exemple: $n = m = 5$ (1/4)

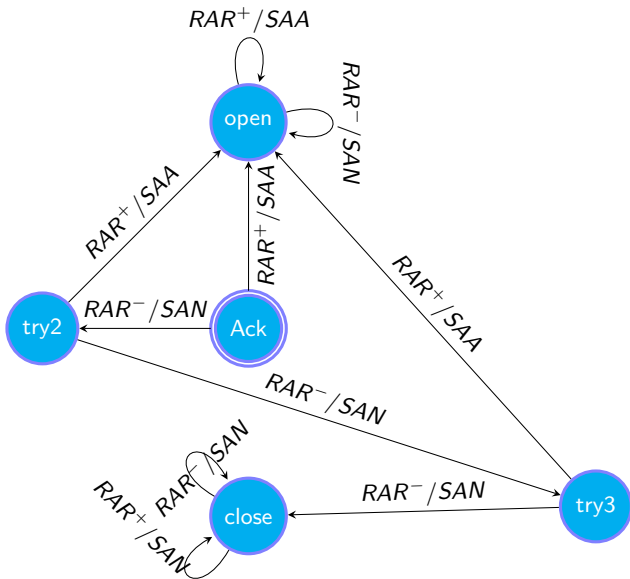


Modèle PAP  
“Complètement  
spécifié” ?  
Définissez les  
transitions indéfinies  
Chaque fois que  
l'accès est

- interdit, la réponse est SAN
- donné, la réponse est SAA



## W-Method : 3ème Exemple: $n = m = 5$ (2/4)



## Distinguishing sequences pour

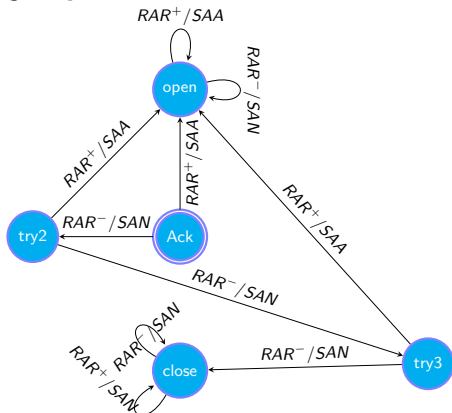
$(Ack, open)$	?
$(Ack, try2)$	?
$(Ack, try3)$	?
$(Ack, close)$	?
$(open, try2)$	?
$(open, try3)$	?
$(open, close)$	?
$(try2, try3)$	?
$(try2, close)$	?
$(try3, close)$	?

## W-Method : 3ème Exemple: $n = m = 5$ (3/4)

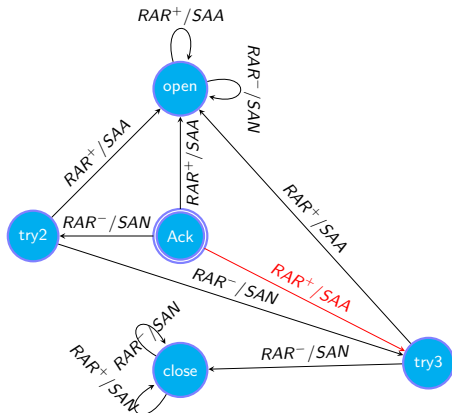
$(Ack, open)$	$RAR^- RAR^- RAR^- RAR^+$
$(Ack, try2)$	$RAR^- RAR^- RAR^+$
$(Ack, try3)$	$RAR^- RAR^+$
$(Ack, close)$	$RAR^+$
$(open, try2)$	$RAR^- RAR^- RAR^+$
$(open, try3)$	$RAR^- RAR^+$
$(open, close)$	$RAR^+$
$(try2, try3)$	$RAR^- RAR^+$
$(try2, close)$	$RAR^+$
$(try3, close)$	$RAR^+$

# W-Method : 3ème Exemple: $n = m = 5$ (4/4)

SPEC



IMP



Test sequence : ? // À vous de jouer !

Spec reaction : ? // À vous de jouer !

Imp reaction : ? // À vous de jouer !

# Ensembles de tests générés par W-Method

- La méthode  $W$  est utilisée pour construire un jeu de test à partir d'une spécification  $FSM$   $\mathcal{S}$  donné
- L'ensemble de test ainsi construit est un ensemble fini de séquences qui peuvent être entrées dans un programme dont la structure de contrôle est modélisée par  $\mathcal{S}$
- Les tests peuvent également être intégrés à une conception pour tester son exactitude par rapport à certaines spécifications
- La plupart des systèmes logiciels ne peuvent pas être modélisés avec précision à 100% à l'aide d'un  $FSM$ . Cependant, la structure de contrôle globale d'un système logiciel peut être modélisée par un  $FSM$

*Méthodes de dérivation des tests à partir de FSM*

*2. Méthodes basée sur l'identification d'état*

*2.2 Distinguishing Sequence (DS) Method*

# DS-methode de distinction (1)

Peut-on remplacer un ensemble  $W$  par une seule séquence distinctive ?

- Si c'est le cas, nous pouvons utiliser **une autre méthode de génération de test -  $DS$** 
  - La raison d'utiliser la méthode  $DS$  au lieu de  $W$  est très naturelle - elle peut réduire la longueur totale du  $TS$
  - Comment vérifier si cela est possible ?
    - > Il devrait y avoir un algorithme efficace pour vérifier s'il existe une séquence distinctive  $\gamma$  pour l'ensemble des états  $S$

## DS

La *DS* est une séquence d'entrée qui distingue deux états quelconques dans *S* en fonction de la sortie observée. Elle peut être représentée sous forme adaptative (ADS) ou prédéfinie (PDS)

- ADS est un arbre où chaque chemin de racine à feuille représente une séquence d'entrée spécifique à l'état représenté par la feuille. Ainsi, pour chaque état, l'entrée suivante dépend de la sortie de l'entrée actuelle
- Par contre PDS, la séquence est fixée d'avance et sa séquence de sortie est différente pour chaque état initial, PDS est une séquence d'entrée unique pouvant être utilisée pour distinguer chaque état des autres

- + Détecte toutes les fautes de sortie
- + Détecte toutes les fautes de transfert
- Il y a deux inconvénients majeurs; Le premier est que, dans la pratique, très peu de *FSM* possèdent en réalité une *DS*. Deuxièmement, même si un *FSM* a une *DS*, la limite supérieure de la longueur de la *DS* sera trop grande pour être utile en général

# DS-methode de distinction (3)

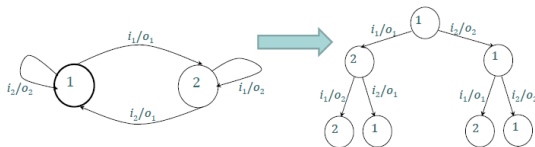
Une séquence d'entrée est une séquence de distinction (distinguishing sequence *DS*) pour un *FSM*  $S$ , si la sortie produite par le *FSM*  $S$  est différente lorsque la séquence d'entrée est appliquée à chaque état différent. Une *DS* est utilisée comme séquence d'identification d'état

## DS

Let  $X$  be an input sequence.  $X$  is a *DS* for *FSM*  $S$ , if each state produces a **unique** (i.e. **different**) **output sequence** in response to  $X$

### Arbre de successeurs

- Simuler le comportement du *FSM*
- Représente le comportement *FSM* sur toutes les séquences d'entrée
- Est infini, bien sûr





# DS-methode de distinction (4)

Derive a truncated successor tree (*TST*)

$h_S : Input \times CurrentState \rightarrow Output \times NextState$

$\exists o$

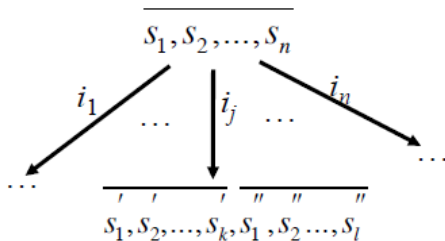
$((s_1, i_j, o, s'_1) \in h_S \ \&$

$(s_2, i_j, o, s'_2) \in h_S \ \&$

$(s_3, i_j, o, s'_3) \in h_S \ \& \dots$

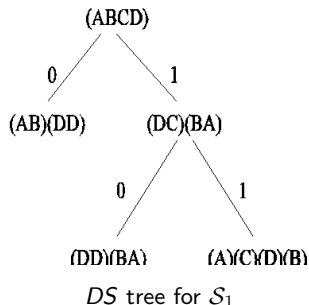
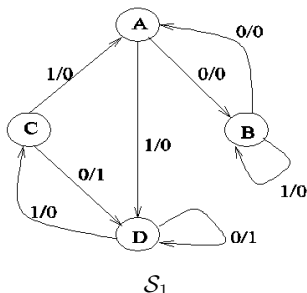
Truncating rules

- Rule 1:  $P$  has only singletons
- Rule 2: Set  $P$  coincides with the set  $P$  that is met 'upper' in the tree
- Rule 3:  $P$  contains a multiset



- $DS$  is a distinguishing sequence iff it labels the path truncated by Rule 1

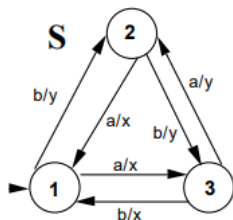
# DS-methode de distinction (5)



State	Output Seq
A	00
B	11
C	10
D	01

Output of  $S_1$  in response to input sequence 11 in different states

# DS-method: Exemple-1

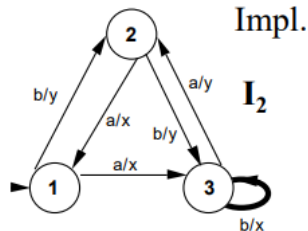


The specification S

A distinguishing sequence is :  
b.b

If we apply it from :

- state 1 we obtain y.y
- state 2 we obtain y.x
- state 3 we obtain x.y

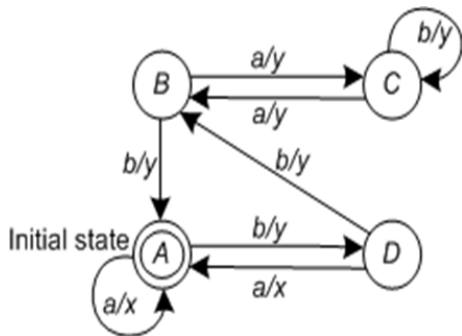


A test case, which allows the  
detection of the transfer error is :  
a.b.b

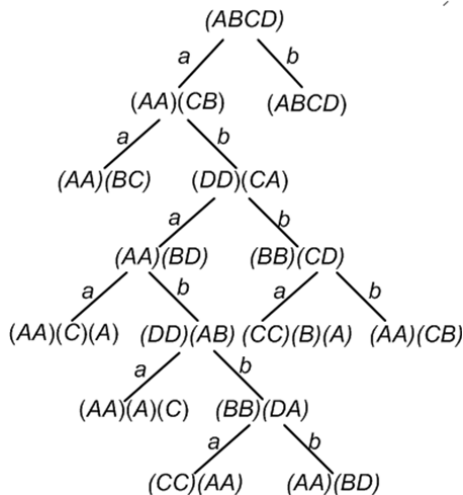
If we apply it from the initial state of:

- the specification we obtain x.x.y
- the implementation we obtain x.x.x

# DS-method: Exemple-2



$S_2$  that does not possess a DS



DS tree for  $S_2$

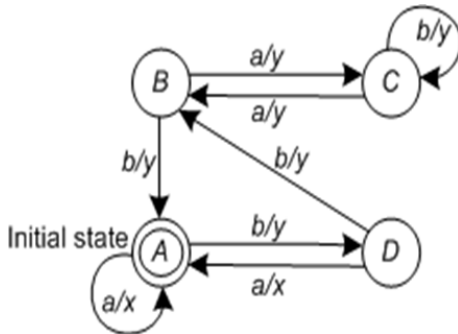
# DS-method: Characterizing Sequence

- Some *FSMs* do not have a *DS*
- State verification is still possible
- Use characterizing sets (We have seen in *W – method*)

## Characterizing Sets (*W*)

A *P* for a state *s* is a set of input sequences such that, when each sequence is applied to the *FSM* in *s*, the set of output sequences is unique. Thus *s* is uniquely identified.

# DS-method: Characterizing Sequence-Example-1



Starting States	Output generated by $W1 = aba$
A	xyx
B	yyy
C	yyx
D	xyx

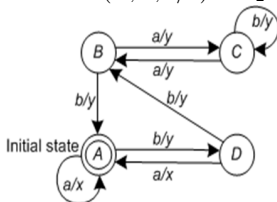
Output sequences generated by  $S_2$  as a response to  $W1$

Starting States	Output generated by $W2 = ba$
A	xy
B	yx
C	yy
D	yy

Output sequences generated by the FSM  $S_2$  as a response to  $W2$

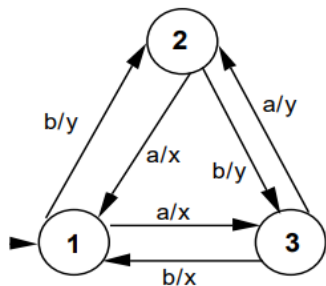
# DS-method: Characterizing Sequence-Example-1

Test sequences for the transition  $(D, A, a/x)$  of  $S_2$



Test Sequence Table				
Step	Current State	Next State	Message to SUT	Message from SUT
Apply $T(D)$ :				
1	A	D	b	y
Test the Transition $(D, A, a/x)$ :				
2	D	A	a	x
Apply $W_1$ :				
3	A	A	a	x
4	A	D	b	y
5	D	A	a	x
Apply $RI$ :				
6	A	D	b	y
7	D	A	a	x
8	A	D	b	y
9	D	A	a	x
10	A	D	b	y
11	D	A	a	x
Apply $T(D)$ :				
12	A	D	b	y
Test the Transition $(D, A, a/x)$ :				
13	D	A	a	x
Apply $W_2$ :				
14	A	D	b	y
15	D	A	a	x
Apply $RI$ :				
16	A	D	b	y
17	D	A	a	x
18	A	D	b	y
19	D	A	a	x
20	A	D	b	y
21	D	A	a	x

# DS-method: Characterizing Sequence-Example-2



## Phase 1: Identification of all states/ State cover

From state 1, we can reach state 2 with b/y  
and state 3 with a/x

The Q set:

$Q = \{ , a, b \}$

DS = b.b

Test suite = {r.b.b, r.a.b.b, r.b.b.b}

## Phase 2: to cover all transitions for output faults and transfer faults

The P set:

$P = \{ , a, b, a.b, a.a, b.b, b.a \}$

Test suite: {r.b.b, r.a.b.b, r.b.b.b, r.a.b.b.b, r.a.a.b.b,  
r.b.b.b.b, r.b.a.b.b}



# DS-method: Characterizing Sequence

Les cas de test sont :

state 1:	<b>a</b> .b.b
	<b>b</b> .b.b
Reaching state 3:	a. <b>a</b> .b.b
	a. <b>b</b> .b.b
Reaching state 2:	b. <b>a</b> .b.b
	b. <b>b</b> .b.b

La structure de cas de test:

preamble.**tested transition**.state identification

*Méthodes de dérivation des tests à partir de FSM*

*2. Méthodes basée sur l'identification d'état*

*2.3 Unique Input Output (UIO) Method*

# Unique Input/Output Sequence UIO-Method

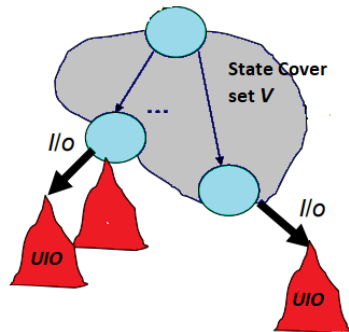
- La méthode *UIO* peut être appliquée si pour chaque état de la spécification, il existe une séquence d'entrée telle que la sortie produite par la machine, lorsqu'elle est initialement dans l'état donné, est différente de celle de tous les autres états

## *UIO*

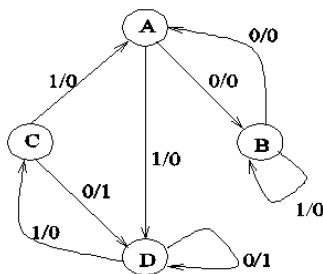
Let  $X$  be an input sequence applied in a state  $s$ , and  $Y$  be the corresponding output sequence.  $X/Y$  is a *UIO* sequence for  $s$  if no other state produces output sequence  $Y$  in response to input  $X$ . Thus,  $X/Y$  is unique to  $s$ .

- L'ensemble de caractérisation  $W$  est modifié avec l' $UIO$  correspondant pour un état approprié
  - $\gamma$  est une séquence d'entrée-sortie unique ( $UIO$ ) pour l'état  $s$  si elle distingue  $s$  de tout autre état
  - Une suite est  $UIO$  pour un état  $s$  si:
- \*\*** Après application de la suite d'inputs, on peut déterminer l'état initial par la suite d'outputs observée

$$TS = V.I.UIO$$

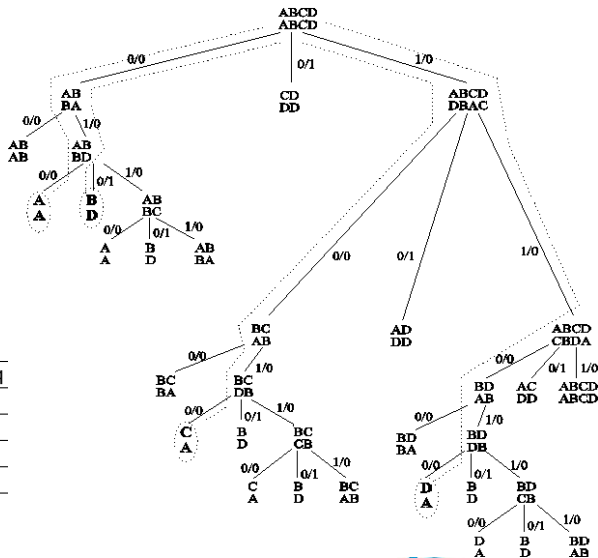


# UIO-Method: Exemple-1

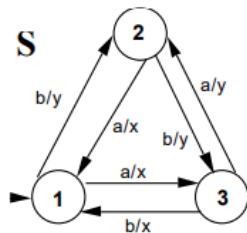


State	Input Seq	Output Seq
A	010	000
B	010	001
C	1010	0000
D	11010	00000

Identification of *UIO* sequences on the *UIO* tree



# UIO-Method: Exemple-2



**The specification S**

We assume the existence of a reset transition with no output (r/-) leading to the initial state for every state of S

**UIO sequences are:**

- state 1: a.b
- state 2: a.a
- state 3: a

**A transition cover set is:**

**$P = \{e, a, a.b, a.a, b, b.a, b.b\}$**

**The test sequences generated by the UIO-method are:**

**r.a.b, r.a.a, r.a.b.a.b, r.a.a.a.a, r.b.a.a, r.b.a.a.b, r.b.b.a**

- Le comportement d'une grande variété de systèmes peut être modélisé à l'aide de machines à états finis (*FSM*)
- La méthode *W* est une méthode théorique pour générer des tests à partir d'un modèle *FSM* donné
- Les tests générés à l'aide des méthode basées sur l'identification des états donnent le garantis de détecter toutes les erreurs de sortie, les erreurs de transfert et les erreurs d'état manquants/supplémentaires dans l'implémentation, étant donné que le *FSM* représentant l'implémentation est complètement spécifié, connecté et minimal
  - Que se passe-t-il si ce n'est pas le cas ?

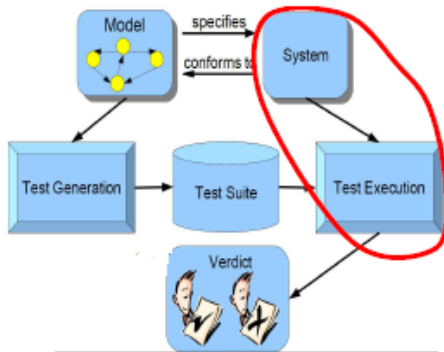
- Objectif: Lier les valeurs abstraites de la spécification aux valeurs concrètes de l'implantation
- Les cas de test synthétisés décrivent des séquences d'actions qui ont une interprétation au niveau abstrait de la spécification
- Pour pouvoir exécuter ces test sur l'implantation, on doit concrétiser ces tests en termes d'exécution à travers l'interface I/O du système.



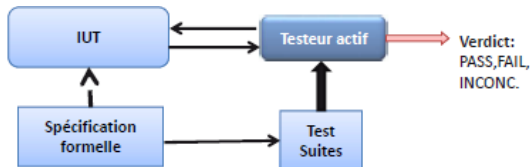
# L'Exécution des Tests

Objectif: forcer l'(IUT) d'exécuter la séquence spécifique d'événements (TSs) qui a été sélectionnée Deux exigences:

- Mettre le système dans un état à partir duquel les tests spécifiés peuvent être lancés (précondition)
- Reproduire la séquence désirée (connu sous le nom de Replay problem): problème très difficile, notamment en présence de processus concurrents et de non-déterminisme (i.e. même input, différents outputs!).

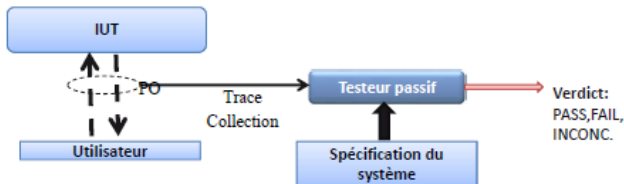


# Test actif Vs Test passif



## pros et cons :

- Possibilité de s'intéresser à une partie précise de la spécification
- ⊗ Génération automatique des tests
- ⊗ Peut altérer (crash) le fonctionnement de l'IUT

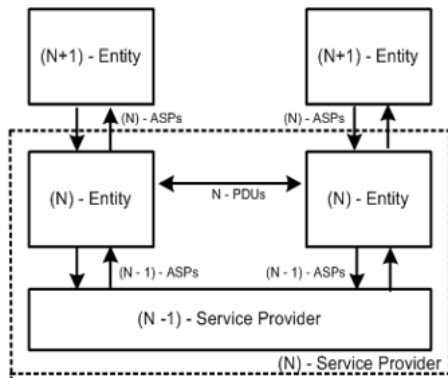


## pros et cons :

- pas d'interférences avec l'IUT
- pas de génération des tests
- ⊗ efficacité des algorithmes

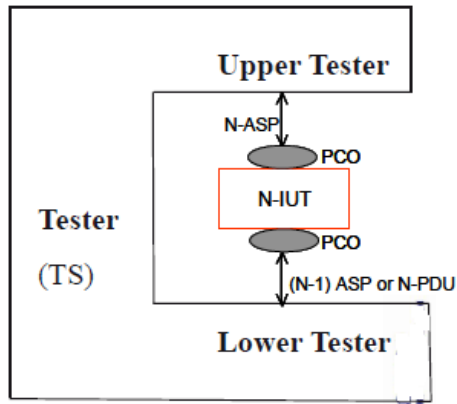
# Architectures de Test

- Points of Control and Observation (*PCO*): est un point d'interaction entre un système et ses utilisateurs
- Une architecture de test est une certaine configuration de
  - une *IUT*
  - une/plusieurs entités de test
  - un ou deux *PCO*
  - un service de communication
- Architectures de test communes
  - > Architecture locale
  - > Architecture distribuée
  - > Architecture coordonnée
  - > Architecture à distance



# Architectures de Test Actif

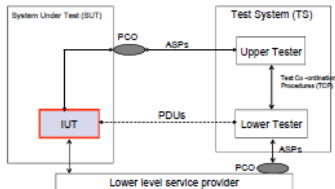
- définies par l'ISO 9646
- Conceptuellement:
  - Le testeur est directement connecté à l'IUT et contrôle son comportement
  - utilisées uniquement lorsque le test est effectué localement par le testeur humain: Optimal pour détecter les fautes!
  - Mais pas directement utilisable pour les tests de conformité, car la communication entre les testeurs supérieur et inférieur doit être effectuée via "l'environnement" (couches inférieures)



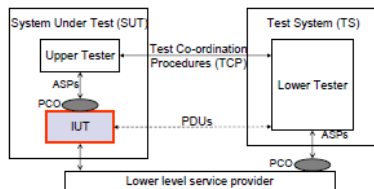
ISO 9646 décrit quatre architectures principales:

- Locale
  - Les testeurs supérieurs et inférieurs sont dans le SUT
  - Le testeur supérieur est directement contrôlé par le testeur et son interface avec l'IUT est un *PCO*
- Distribuée
  - Le testeur supérieur est dans le *SUT*
  - Il est directement contrôlé par le testeur et son interface avec l'IUT est un *PCO*
- Coordonnée
  - Le testeur supérieur est dans le *SUT* mais est mis en oeuvre par le testeur humain
  - Il est directement contrôlé par le testeur et son interface avec l'*IUT* n'est pas directement observable
- Remote (à distance)
  - Aucun testeur supérieur

# Architectures de Test Actif

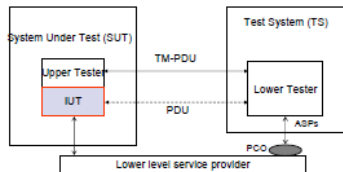


Local

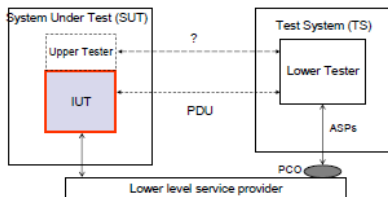


Distributed

IUT: Implementation under Test  
PCO: Point of Control and Observation  
ASP: Abstract Service Primitive  
PDU: Protocol Data Units



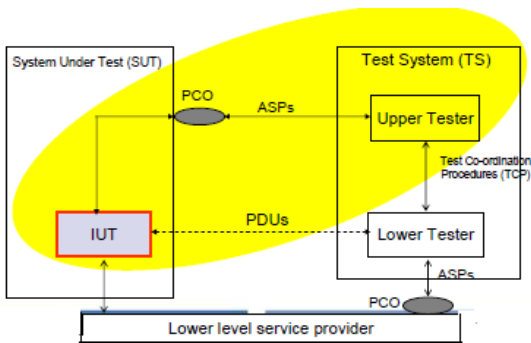
Coordinated



Remote

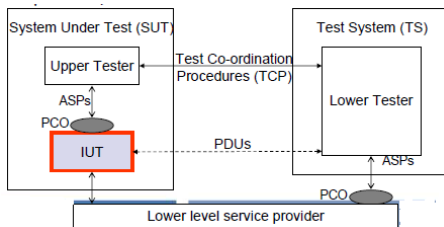
# Contrôlabilité dans l'architecture locale

- Dans un système réel, la couche supérieure, ici illustrée comme le testeur supérieur, communique directement avec l'*IUT*
- Pour être efficace, la communication entre l'*IUT* et l'UT (Upper Tester) doit être synchronisée, les deux entités devraient fonctionner car elles seraient directement connectées
  - La zone jaune de la figure représente cette synchronisation
- Cette architecture est couramment utilisée pour tester
  - Ainsi *SUT*, *TS*, *PCO* seront des éléments physiques



# Architecture distribuée

- Le testeur supérieur est mis en oeuvre par les testeurs humains
- Le *TCP* peut être manuel ou automatisé
- La coordination entre l'*UT* et le *LT* est un protocole développé par les testeurs humains
- Les suites de test sont les mêmes que dans une architecture locale
- Approprié pour tester une couche complète de pile de protocole (étant plus proche des composants *SUT*)



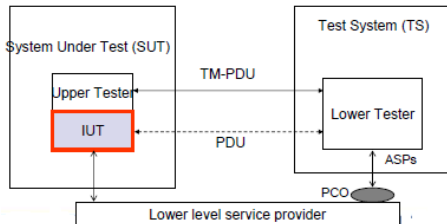


Pour tester un commutateur de téléphone:

- L'UT pourrait simuler l'utilisateur (directement connecté)
- Le LT pourrait
  - Simuler le commutateur à distance
  - Donner des instructions à l'UT (par exemple, décrocher le téléphone)
  - Et contrôle la réponse sur le PCO avec lequel il est directement connecté

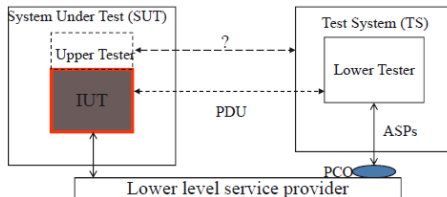
# Architecture distribuée

- Cette architecture a comme principal inconvénient que l'IUT doit intégrer un *UT* directement contrôlé par le testeur
  - Le testeur supérieur est directement et normalement connecté à l'IUT, développé par le développeur de l'IUT
  - Pas de *PCO* du côté *SUT*!
  - Il communique avec le testeur par un protocole de gestion des tests qui échange de la *TM – PDUS*
- > Le protocole de gestion des tests doit être normalisé car le testeur pourrait être tout type d'entité
- La coordination entre *LT* et *UT* (*TM – PDUS*) doit faire une partie des suites de test
  - Les messages détaillant cette coordination peuvent être:
- > Soit inclus dans les parties de données du *N – PDU* (puis passer par le *LLSP*)
- > ou transmis par une connexion séparée
- Approprié pour tester une couche intermédiaire



# Architecture à distance

- L'*UT* n'est pas nécessaire, cela peut être exploité en suivant les instructions informelles
- Le *LT* peut envoyer des *PDU* qui contiennent des données qui seront interprétées par l'*IUT* comme des primitives à appliquer à son interface supérieure (ligne pointillée)
- Les possibilités de détecter les fautes sont limitées car il n'est pas possible de contrôler ou d'observer directement l'interface supérieure
- Cependant, cette architecture est simple et facilement développée
- Approprié pour tester les protocoles dont le rôle de l'interface supérieure du SUT est limité (par exemple, *FTP*)

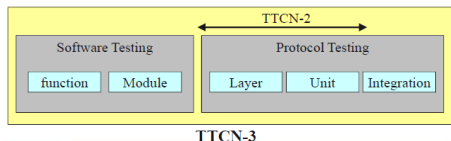


# Lien testeur supérieur (UT) / système de test (ST)

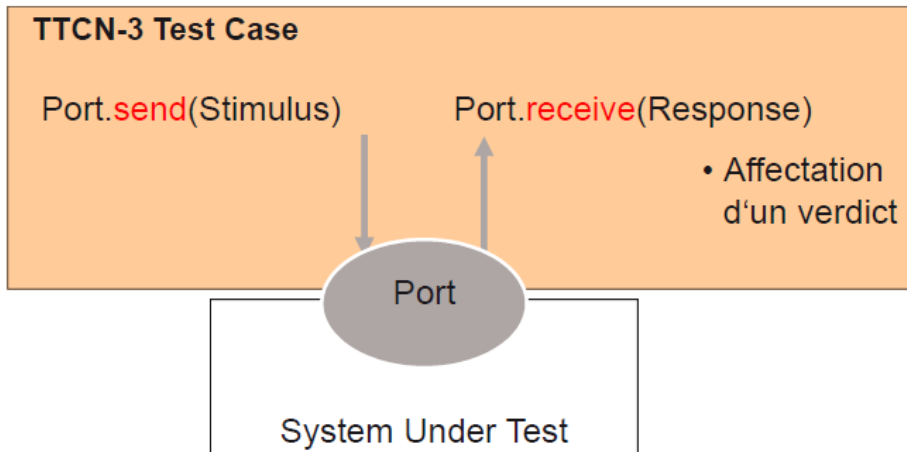
- Toutes les architectures (à l'exception de l'architecture distante) planifient un lien entre l'*UT* et le *TS*
- Ce lien est réel et doit être implémenté séparément du *LLSP*
- Possibilités:
  - Un lien implémenté indépendant et fiable?
  - Deux personnes communiquant par un autre médium?

# Testing and Test Control Notation 3 (TTCN-3)

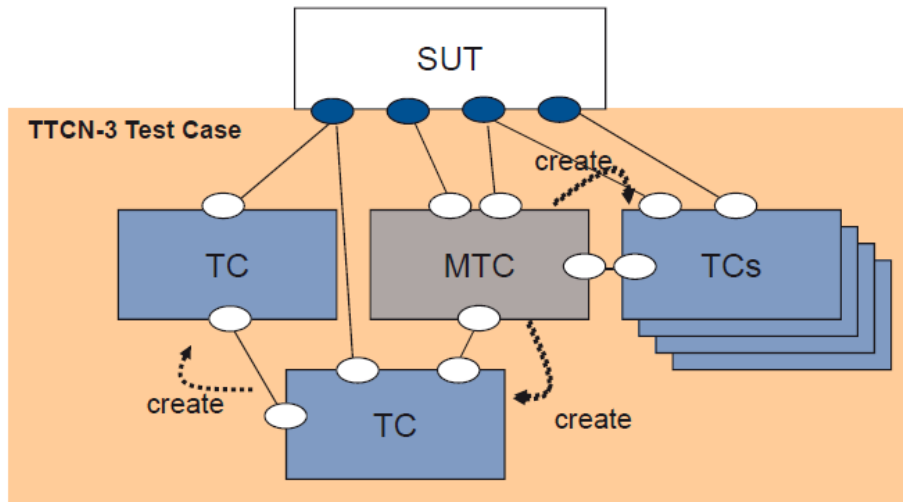
- Langage standardisé (par *ETSI* (European Telecom Standards Institute)) afin de définir formellement les cas de test
- Notation textuelle efficace
- Outil indépendant
- Black-Box Testing with *TTCN* – 3
- Configuration Tests
- Components of Tests
- Communication ports
- Tests Verdicts
- *TTCN* – 3 elements
- *TTCN* – 3 specification structures
- Existing code reuse



# Black-box testing with TTCN-3



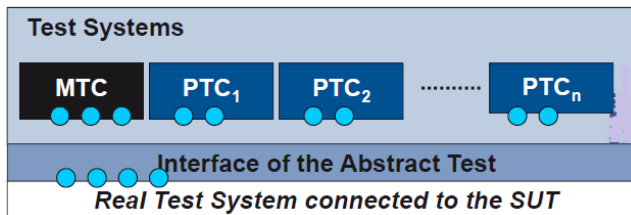
# Tests Configuration



# Tests components

## Trois types de composants

- Interface système du système de test abstrait
- MTC (Main Test Component)
- PTC (Parallel Test Component)

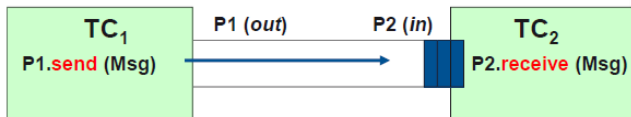




# Communication Ports

Les composants de tests communiquent via des ports

- Un port est défini comme un *FIFO* infini (mais peut être configuré)
- Les ports ont des directions (in, out, inout)
- Trois types de ports: basés sur des messages, basés sur les procédures et mixtes

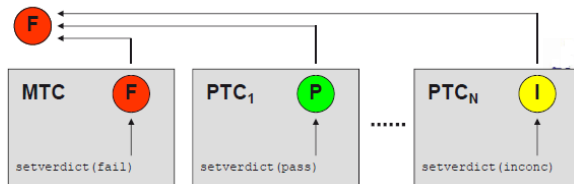


# Test verdicts

Verdicts: aucun < pass < inconc < échec < Erreur

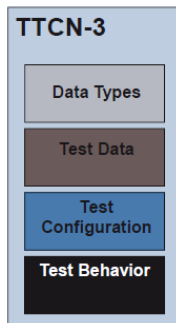
- Chaque composant de test a son propre verdict local qui peut être défini (setVerdict) et lu (getVerdict)
- Un cas de test fournit un verdict global

*Verdict returned by the test case  
when it terminates*



# TTCN-3 elements

- Types de données génériques (par exemple, pour définir des messages, des primitives de service, des informations, des *PDU*)
- Les données de test transmises / reçues pendant le processus de test
- Définition des composants et des ports de communication utilisés pour configurer les tests
- Spécification du comportement du système de test



# Structure of the TTCN-3 Modules specs - Control

- Les modules sont les blocs modélisant toutes les spécifications TTCN-3
- Une suite de test est un module
- Un module est constitué d'une définition et d'une pièce de contrôle (optional)
- Les modules peuvent avoir des paramètres
- Un module peut importer des définitions à partir d'autres modules
- Les définitions d'un module sont globales
- Les définitions des types de données sont basées sur des types structurés et prédéfinis
- Les modèles définissent les données de test
- Ports et composants utilisés dans les configurations de test
- Les fonctions, les altsts et les cas de test définissent le comportement
- La partie de contrôle est la partie dynamique de la spécification *TTCN3*, où les cas de test sont exécutés
- Déclaration locale telle que les variables
- Les éléments de codes de base peuvent être utilisés pour sélectionner et contrôler l'exécution des cas de test

```
module ... {  
  ...  
  control(  
    var integer count;  
    if (execute(SIP_UA_REC_V_001()) == pass) {  
      // Execute test case 10 times  
      count := 0;  
      while (count <= 10) {  
        execute(SIP_UA_REC_V_002());  
        count := count + 1;  
      } // end while  
    } // end if  
  } // end control  
} // end module
```