Cours Test de Logiciels

Bruno Legeard

Laboratoire d'Informatique
de l'Université de Franche-Comté

Plan du cours Test de logiciels

1 - Introduction au test de logiciels

- Définition du test
- Petite auto-évaluation / difficultés du test
- Le test dans le cycle de vie

2- Le test fonctionnel

- Test aux limites, test statistique
- Test à partir de spécifications

3- Le test structurel dynamique

- Critères de test
- Analyse du flot de contrôle

4- Les outils pour l'automatisation du test

1- Introduction au test de logiciels

- Motivations du test :
  - Coûts d ’un Bug (Ariane 5, Réseau ATT bloqué, an 2000, …)

Erreur

Spécification, conception, programmation

Défaut dans le logiciel

Anomalie de fonctionnement

➔ **Qualité du logiciel : Fiabilité, Conformité aux spécifications, Robustesse**

Méthodes de Validation & Vérification du logiciel

- V & V
  - Validation : Est-ce que le logiciel réalise les fonctions attendues ?
  - Vérification : Est-ce que le logiciel fonctionne correctement ?
- Méthodes de V & V
  - Test statique : Review de code, de spécifications, de documents de design
  - Test dynamique : Exécuter le code pour s'assurer d'un fonctionnement correct
  - Vérification symbolique : Run-time checking, Execution symbolique, …
  - Vérification formelle : Preuve ou model-checking d'un modèle formel, raffinement et génération de code

➔ Actuellement, le test dynamique est la méthode la plus diffusée et représente jusqu'à 60 % de l'effort complet de développement d'un produit logiciel

Définitions du test

- « Le test est l ’exécution ou l ’évaluation d ’un système ou d ’un composant par des moyens automatiques ou manuels, pour vérifier qu ’il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus » - IEEE (Standard Glossary of Software Engineering Terminology)

- « Tester, c ’est exécuter le programme dans l ’intention d ’y trouver des anomalies ou des défauts » - G. Myers (The Art of Software testing)

- “Testing can reveal the presence of errors but never their absence” – Edsgar W. Dijkstra. *Notes on structured programming*. Academic Press, 1972.

Définitions

- Tester c’est réaliser l ’exécution du programme
- Notion d’Oracle : résultats attendus d’une exécution du logiciel
- Coût du test : 30 % à 60 % du coût de développement total
- Deux grandes familles de tests
  - Test fonctionnel (ou test boîte noire)
  - Test structurel (ou test boîte de verre)

Test de logiciels – une auto-évaluation

- Soit la spécification suivante :

  Un programme prend en entrée trois entiers. Ces trois entiers sont interprétés comme représentant les longueurs des cotés d'un triangle. Le programme rend un résultat précisant si il s'agit d'un triangle scalène, isocèle ou équilatéral.

- Produire une suite de cas de tests pour ce programme

Exemple du triangle

- 14 cas de test – GJ Myers – « The Art of Software Testing » - 1979

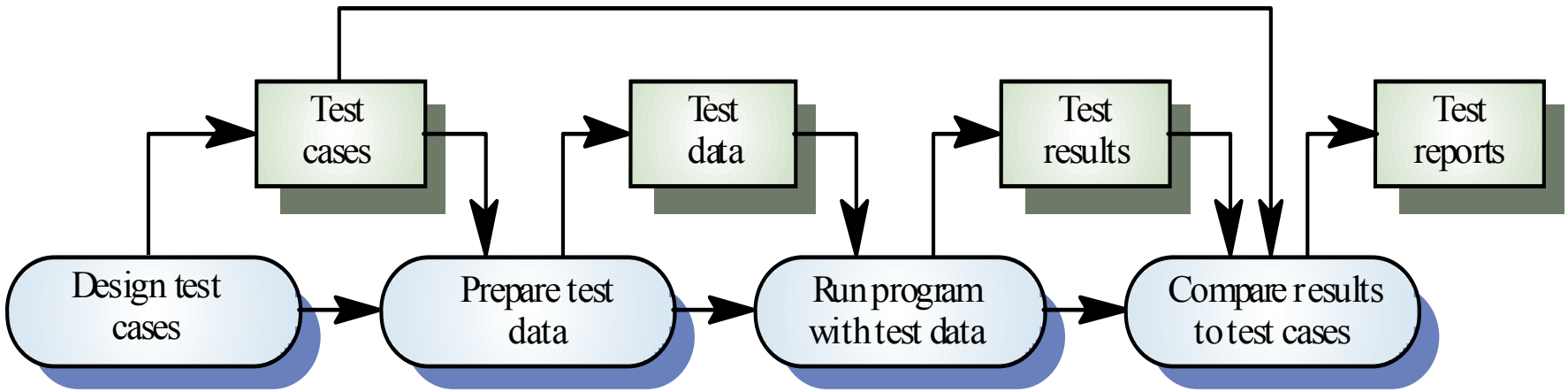
1. Cas scalène valide (1,2,3 et 2,5,10 ne sont pas valides)
2. Cas équilatéral valide
3. Cas isocèle valide (2,2,4 n'est pas valide)
4. Cas isocèle valide avec les trois permutations (e.g. 3,3,4; 3,4,3; 4,3,3)
5. Cas avec une valeur à 0
6. Cas avec une valeur négative
7. Cas ou la somme de deux entrées est égale à la troisième entrée
8. 3 cas pour le test 7 avec les trois permutations
9. Cas ou la somme de deux entrées est inférieur à la troisième entrée
10. 3 cas pour le test 9 avec les trois permutations
11. Cas avec les trois entrées à 0
12. Cas avec une entrée non entière
13. Cas avec un nombre erroné de valeur (e.g. 2 entrées, ou 4)
14. Pour chaque cas de test, avez-vous défini le résultat attendu ?

Exemple du triangle – Evaluation

- Chacun de ces 14 tests correspond à un défaut constaté dans des implantations de cet exemple triangle
- La moyenne des résultats obtenus par un ensemble de développeurs expérimentés est de 7.8 sur 14.

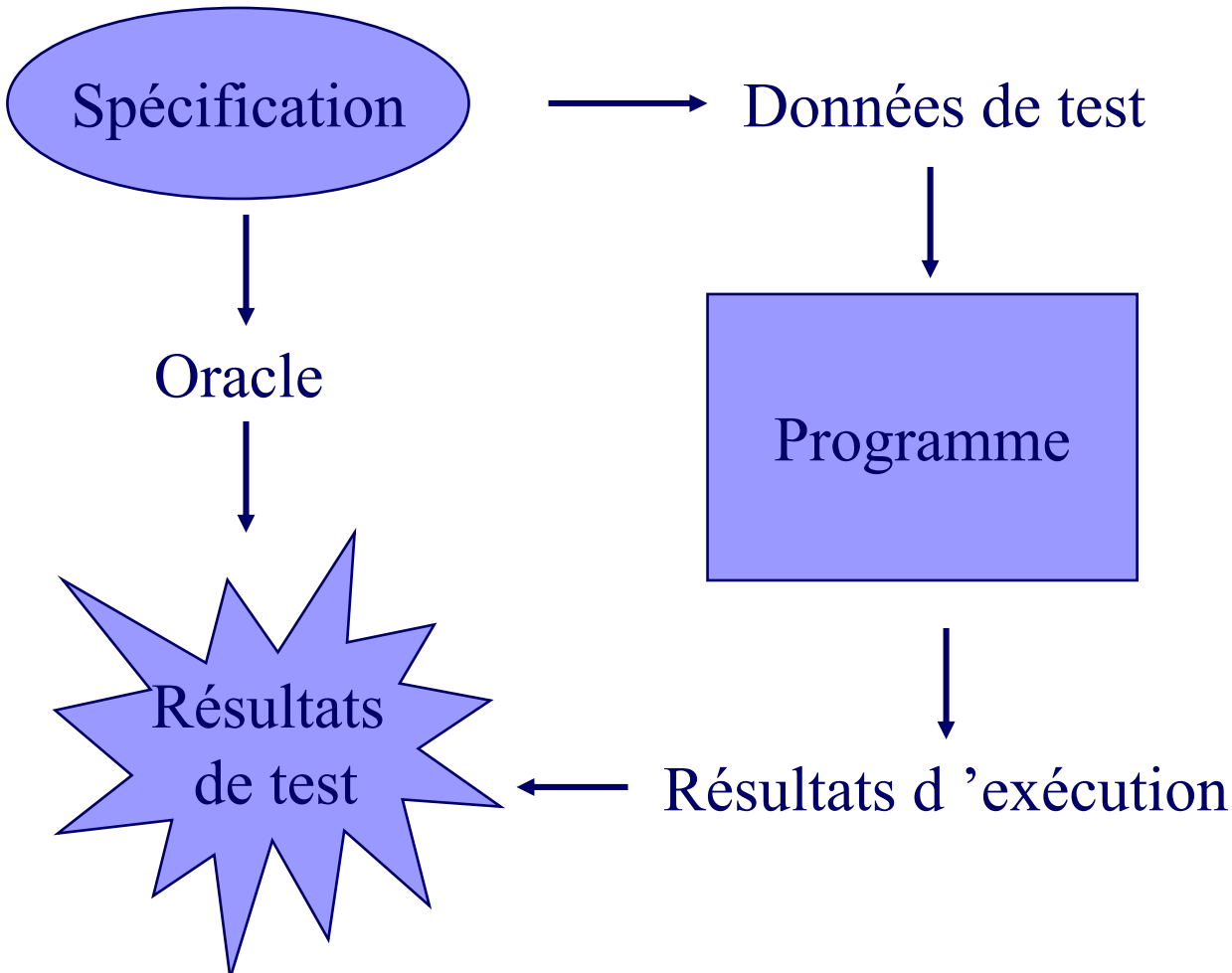
=> La conception de tests est une activité complexe, à fortiori sur de grandes applications

Test et cycle de vie : Le process de test

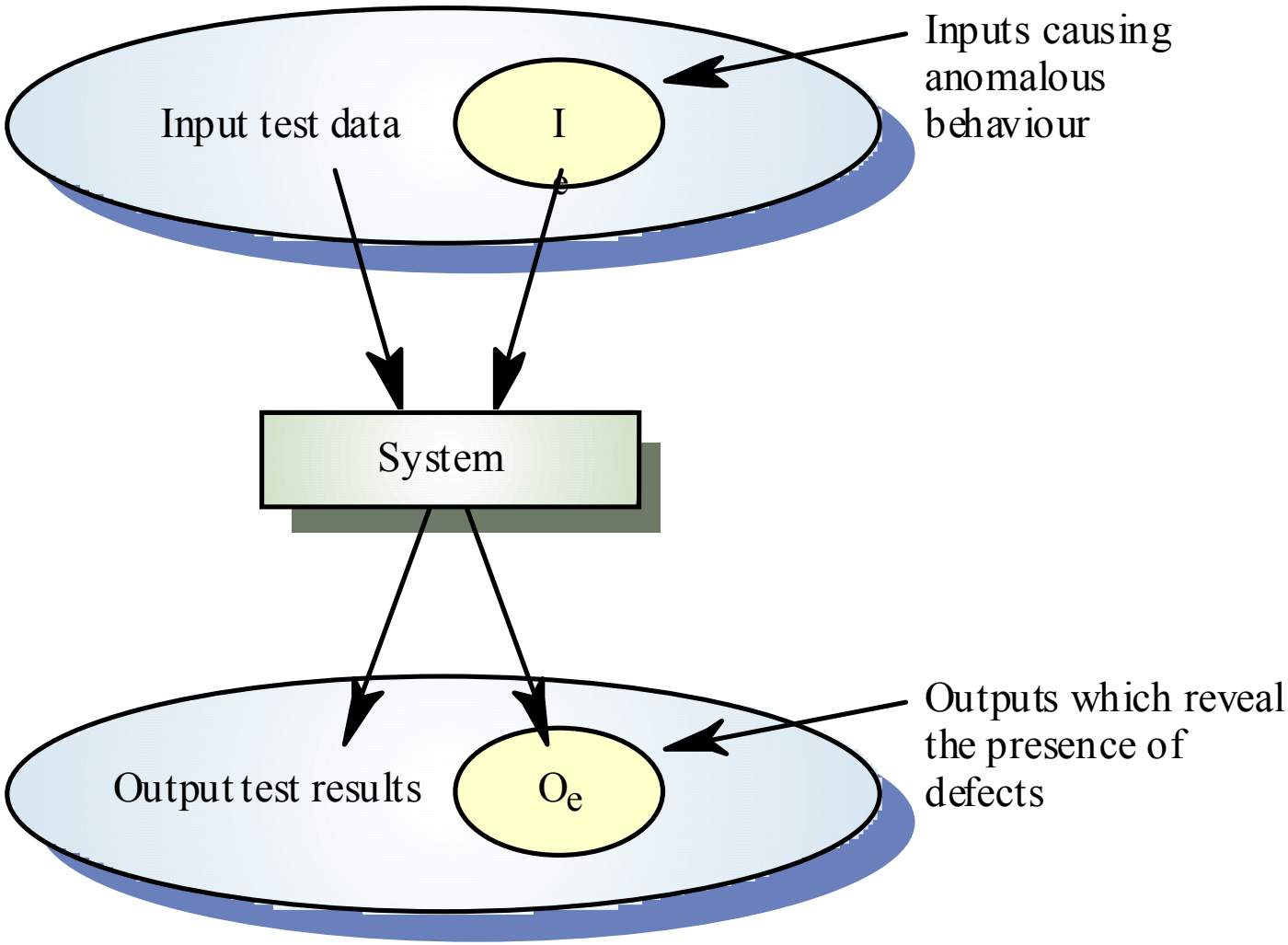


Test fonctionnel

- Test de conformité par rapport à la spécification



Black-box testing



Test structurel – « White Box Testing »

- Les données de test sont produites à partir d'une analyse du code source

Critères de test :
- tous les chemins,
- toutes les branches,
- toutes les instructions

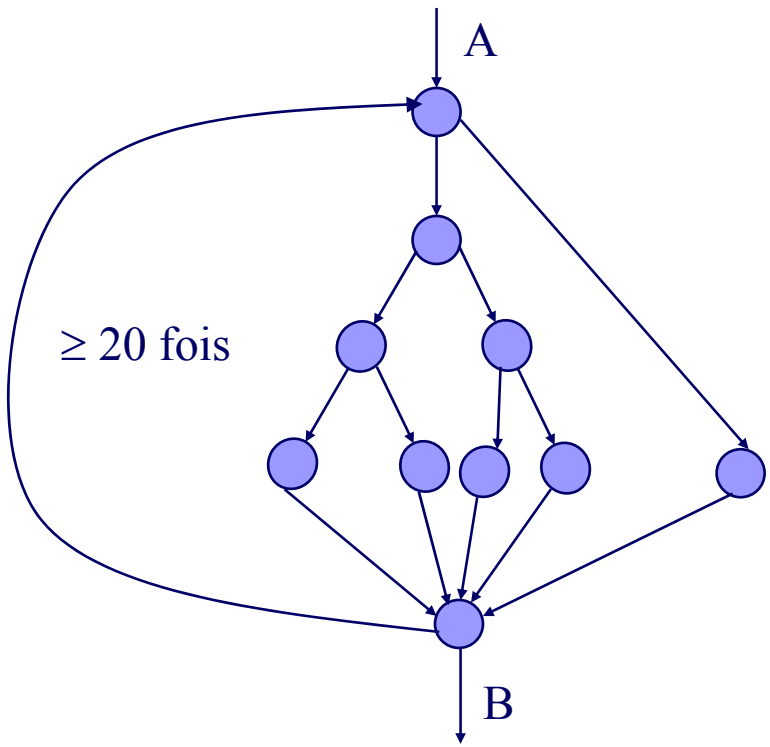


Fig 1 : Flot de contrôle d'un petit programme

Complémentarité test fonctionnel - structurel (1)

- Les techniques fonctionnelles et structurelles sont utilisées de façon complémentaire

Exemple : Soit le programme suivant censé calculer la somme de deux entiers

```
function sum (x,y : integer) : integer;
begin
if (x = 600) and (y = 500) then sum := x-y
else sum := x+y;
end
```

Une approche fonctionnelle détectera difficilement le défaut alors qu ’une approche par analyse de code pourra produire la DT : x = 600, y = 500

Complémentarité test fonctionnel - structurel (2)

- En examinant ce qui à été réalisé, on ne prend pas forcément en compte ce qui aurait du être fait :

⇒ Les approches structurelles détectent plus facilement les erreurs commises

⇒ Les approches fonctionnelles détectent plus facilement les erreurs d'omission et de spécification

Une difficulté du test structurel consiste dans la définition de l'Oracle de test.

Difficultés du test (1)

- Le test exhaustif est en général impossible à réaliser
  - En test fonctionnel, l 'ensemble des données d 'entrée est en général infini ou très grande taille

    Exemple : un logiciel avec 5 entrées analogiques sur 8 bits admet 2^{40} valeurs différentes en entrée
  - En test structurel, le parcours du graphe de flot de contrôle conduit à une forte explosion combinatoire

    Exemple : le nombre de chemin logique dans le graphe de la figure 1 est supérieur à $10^{14} \approx 5^{20} + 5^{19} + \ldots + 5^{1}$

  => le test est une méthode de vérification partielle de logiciels

  => la qualité du test dépend de la pertinence du choix des données de test

Difficultés du test (2)

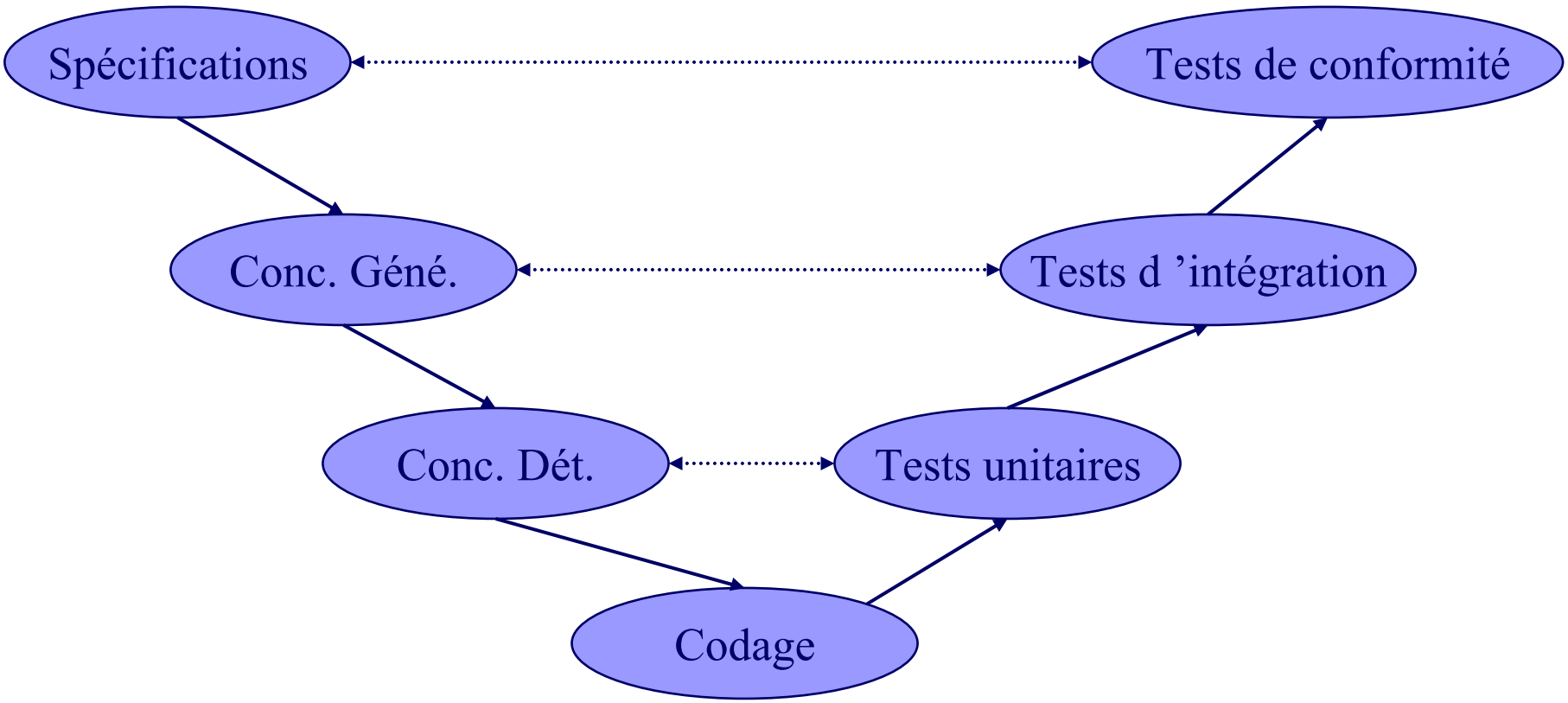
- Difficultés d ’ordre psychologique ou « culturel »
  - Le test est un processus destructif : un bon test est un test qui trouve une erreur

  alors que l ’activité de programmation est un processus constructif - on cherche à établir des résultats corrects
  - Les erreurs peuvent être dues à des incompréhensions de spécifications ou de mauvais choix d ’implantation

  => L ’activité de test s ’inscrit dans le contrôle qualité, indépendant du développement

Le test dans le cycle de vie (1)

- 1- Cycle en V



Types de test (1)

- Tests unitaires :

  Test de procédures, de modules, de composants

- Tests d ’intégration :

  Test de bon comportement lors de la composition de procédures et modules

- Tests de conformité ou test système :

  Validation de l ’adéquation aux spécifications

- Tests de non-régression :

  Vérification que les corrections ou évolution dans le code n ’ont pas créées d ’anomalies nouvelles

Types de test (2)

- Tests nominal ou test de bon fonctionnement :

  Les cas de test correspondent à des données d'entrée valide.

  => Test-to-pass

- Tests de robustesse :

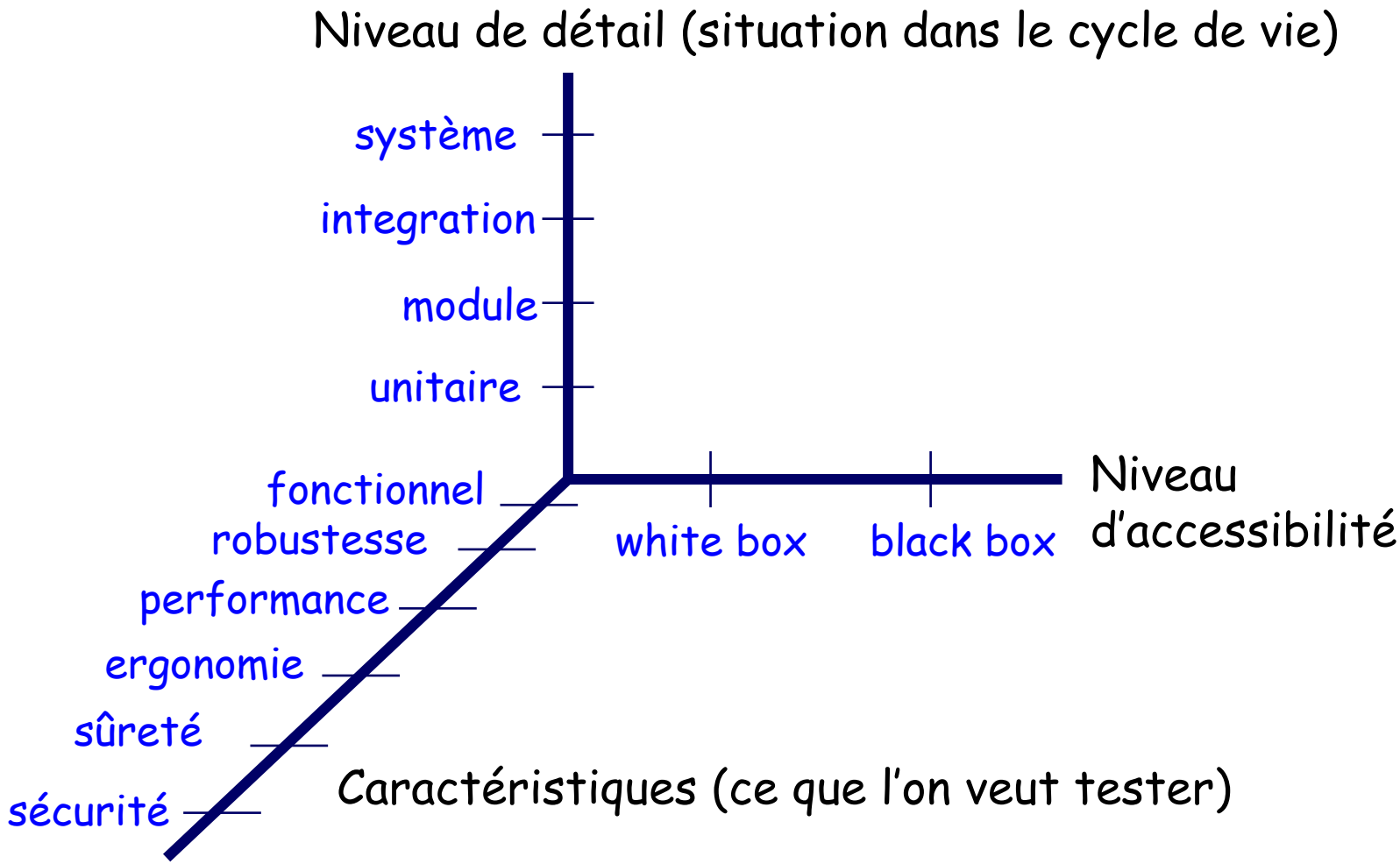
  Les cas de test correspondent à des données d'entrée invalide

  => Test-to-fail

Règle : Les tests nominaux sont passés avant les tests de robustesse.

- Test de performance
  - Load testing (test avec montée en charge)
  - Stress testing (soumis à des demandes de ressources anormales)

Types de test



D'après J. Tretmans – Univ. Nijmegen

Test et démarche d'assurance qualité

- En début de projet, définition d'un Plan de Test et Validation - PTV

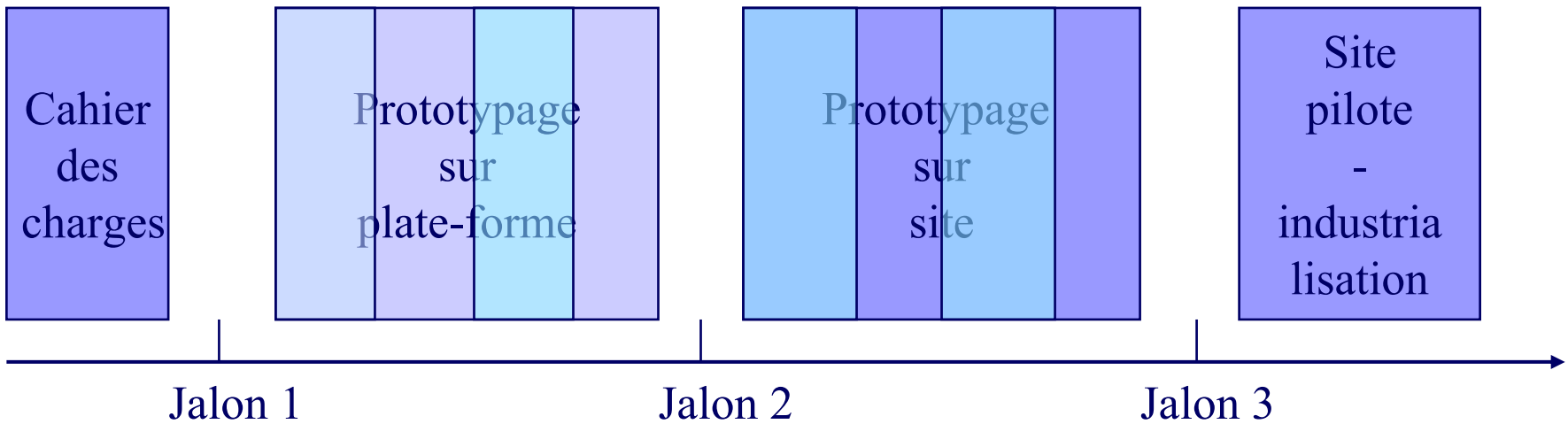
→ Objectifs de test

→ Techniques de test

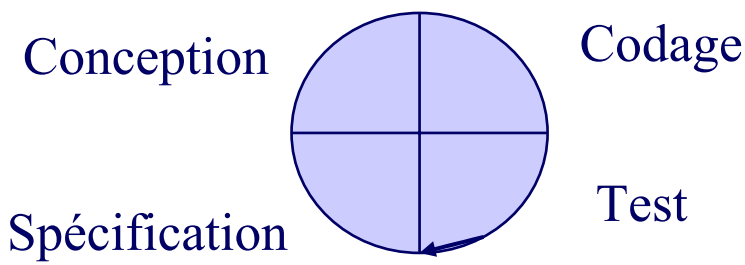
→ Phases et planning

Le test dans le cycle de vie (2)

- 2- Cycle par Prototypage



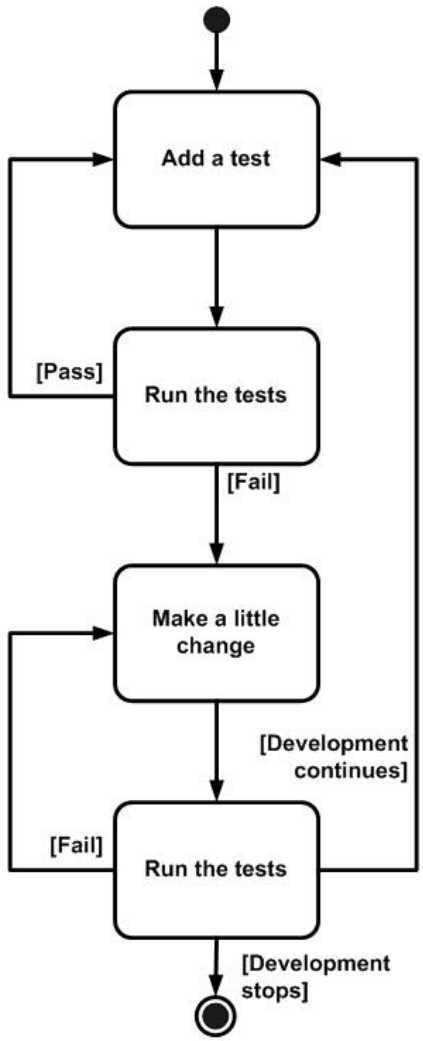
Itération du prototypage



Test et méthodes agiles (1)

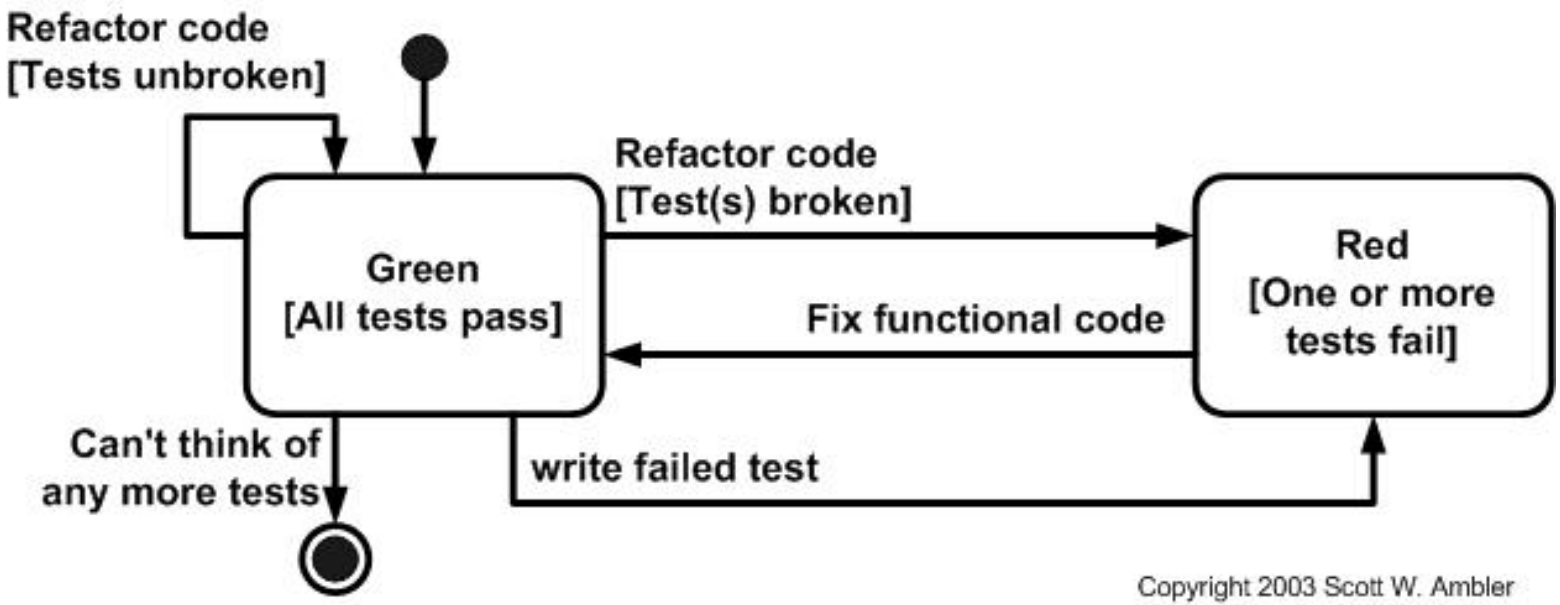
- Test driven development
  1. Développer les tests en premier
  2. Développer le code correspondant
  3. Refactoriser

- Utilisation des outils d'automatisation d'exécution
  - JUnit, TestNG, ….
  - Domage Control, Beetle Juice



Copyright 2003 Scott W. Ambler

Test Driven Development



http://www.agiledata.org/essays/tdd.html

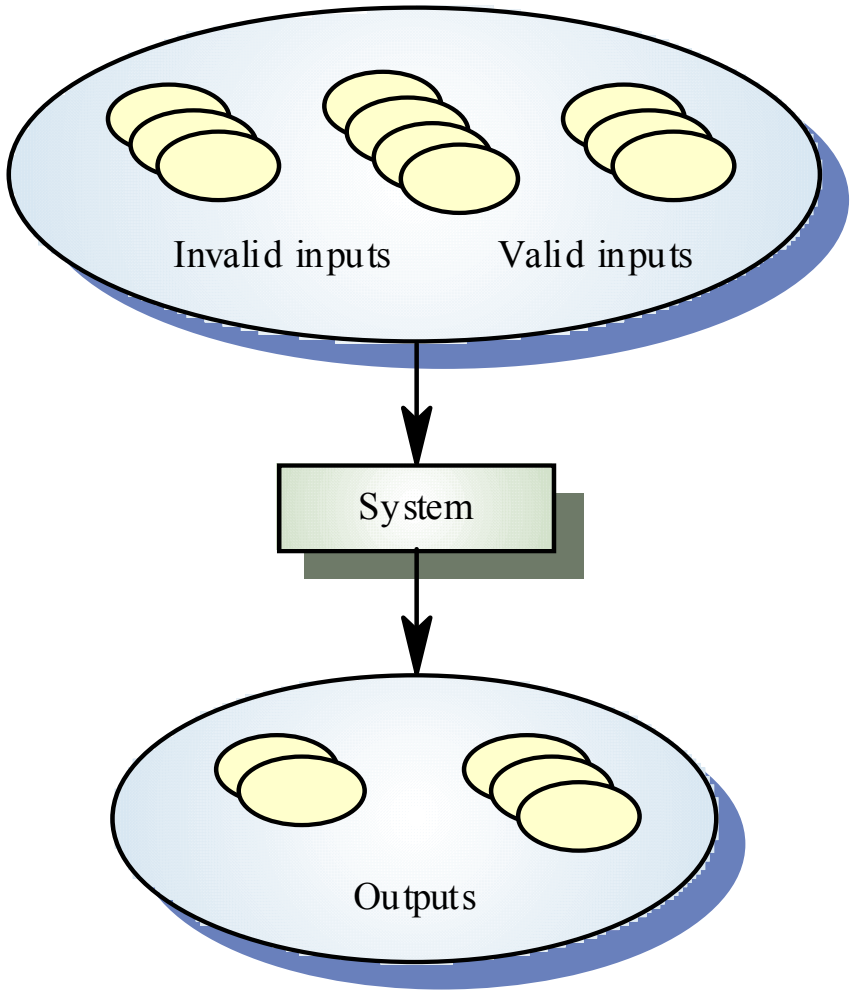
Quelques ressources sur le test de logiciels

- Livre de référence du cours
  - « Le test des logiciels » - Hermès 2000 - S. Xanthakis, P. Régnier, C. Karapoulios
- Autres ouvrages
  - « Software Engineering » -Addison-Wesley – 6th ed. 2001 – I. Sommerville
  - « Test logiciel en pratique » - Vuibert Informatique – John Watkins – 2002
- Plus d'une centaine d'ouvrages sur le test de logiciels
- Lien web
  - http://www.faqs.org/faqs/software-eng/testing-faq/

2- Méthodes de test fonctionnel

- Le test fonctionnel vise à examiner le comportement fonctionnel du logiciel et sa conformité avec la spécification du logiciel
  - ⇨ Sélection des Données de Tests (DT)
- Méthodes du test fonctionnel
  - Analyse partitionnelle des domaines des données d'entrée et test aux limites ➔ test déterministe
  - Test combinatoire – Algorithmes Pairwise
  - Test aléatoire
  - Génération automatique de tests à partir d'une spécification

2.1 Analyse partitionnelle des domaines des données d ’entrée et test aux limites



Une *classe d’équivalence* correspond à un ensemble de données de tests supposées tester le même comportement, c’est-à-dire activer le même défaut.

Partitionnement de domaines : exemple

- Soit le programme suivant :

  Un programme lit trois nombres réels qui correspondent à la longueur des cotés d ’un triangle. Si ces trois nombres ne correspondent pas à un triangle, imprimer le message approprié. Dans le cas d ’un triangle, le programme examine s ’il s ’agit d ’un triangle isocèle, équilatéral ou scalène et si son plus grand angle est aigu, droit ou obtus (< 90°, = 90°, > 90 °) et renvoie la réponse correspondante.

- Classes d ’équivalence et Données de Test

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

+ non triangle - 1,2,8

Analyse partitionnelle - Méthode

- Trois phases :
  - Pour chaque donnée d ’entrée, calcul de classes d ’équivalence sur les domaines de valeurs,
  - Choix d’un représentant de chaque classe d ’équivalence,
  - Composition par produit cartésien sur l ’ensemble des données d ’entrée pour établir les DT.

Soit Ci, une classe d ’équivalence,

$$\cup C_i = E \ \wedge \forall i,j \ \ C_i \cap \ C_j = \varnothing$$

Règles de partitionnement des domaines

- Si la valeur appartient à un intervalle, construire :
  - une classe pour les valeurs inférieures,
  - une classe pour les valeurs supérieures,
  - n classes valides.
- Si la donnée est un ensemble de valeurs, construire :
  - une classe avec l ’ensemble vide,
  - une classe avec trop de valeurs,
  - n classes valides.
- Si la donnée est une obligation ou une contrainte (forme, sens, syntaxe), construire :
  - une classe avec la contrainte respectée,
  - une classe avec la contrainte non-respectée

Test aux limites

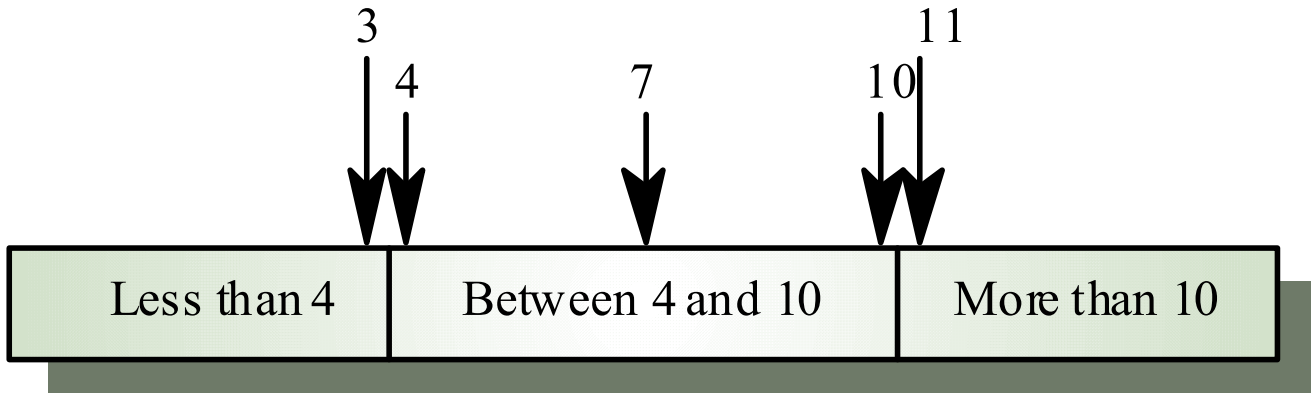
- Principe : on s'intéresse aux bornes des intervalles partitionnant les domaines des variables d 'entrées :
  - pour chaque intervalle, on garde les 2 valeurs correspondant aux 2 limites, et les 4 valeurs correspondant aux valeurs des limites ± le plus petit delta possible

    $n \in 3 .. 15 \Rightarrow v1 = 3, v2 = 15, v3 = 2, v4 = 4, v5 = 14, v6 = 16$
  - si la variable appartient à un ensemble ordonnés de valeurs, on choisit le premier, le second, l 'avant dernier et le dernier

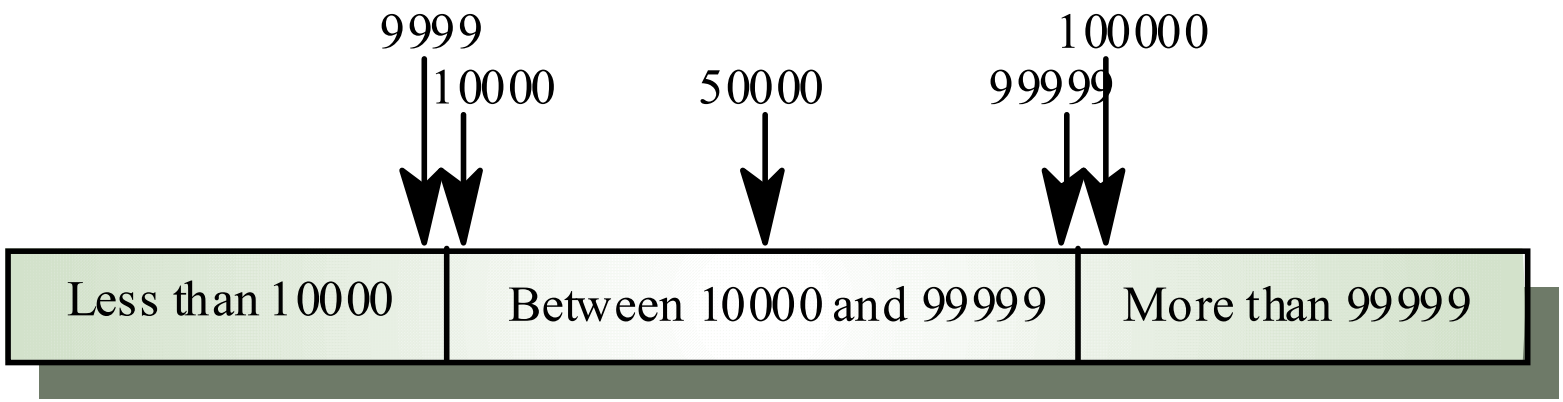
    $n \in \{-7, 2, 3, 157, 200\} \Rightarrow v1 = -7, v2 = 2, v3 = 157, v4 = 200$
  - si une *condition d'entrée* spécifie un *nombre de valeurs*, définir les cas de test à partir du nombre *minimum* et *maximum* de valeurs, et des test pour des nombres de valeurs hors limites invalides.

    Un fichier d'entrée contient 1-255 records, produire un cas de test pour 0, 1, 255 et 256.

Analyse des valeurs aux limites



Number of input values



Input values

Test aux limites - Exemple

- Calcul des limites sur l ’exemple du programme de classification des triangles

1, 1, 2	Non triangle
0, 0, 0	Un seul point
4, 0, 3	Une des longueurs est nulle
1, 2, 3.00001	Presque un triangle sans en être un
0.001, 0.001, 0.001	Très petit triangle
99999, 99999, 99999	Très grand triangle
3.00001, 3, 3	Presque équilatéral
2.99999, 3, 4	Presque isocèle
3, 4, 5.00001	Presque droit
3, 4, 5, 6	Quatre données
3	Une seule donnée
	Entrée vide
-3, -3, 5	Entrée négative

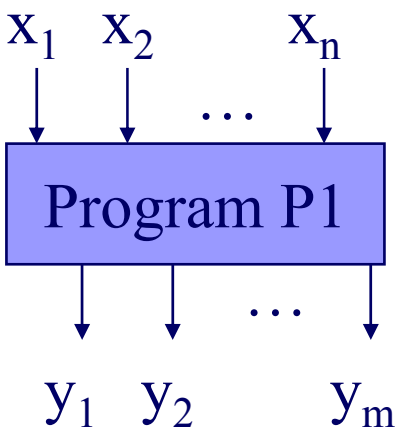
Test aux limites – Types de données

- les données d'entrée ne sont pas seulement des valeurs numériques : caractères, booléens, images, son, …
- Ces catégories peuvent, en général, se prêter à une analyse partitionnelle et à l'examen des conditions aux limites :
  - True / False
  - Fichier plein / Fichier vide
  - Trame pleine / Trame vide
  - Nuances de couleur
  - Plus grand / plus petit
  - ….

Analyse partitionnelle et test aux limites – synthèse

- L' analyse partitionnelle est une méthode qui vise à diminuer le nombre de cas de tests par calcul de classes d'équivalence
  - importance du choix de classes d'équivalence : risque de ne pas révéler un défaut
- Le choix de conditions d'entrée aux limites est une heuristique solide de choix de données d'entrée au sein des classes d' équivalence
  - cette heuristique n'est utilisable qu'en présence de relation d'ordre sur la donnée d'entrée considérée.
- Le test aux limites produit à la fois des cas de test nominaux (dans l'intervalle) et de robustesse (hors intervalle)

Analyse partitionnelle et test aux limites – Combinaison des valeurs d'entrée



- Pb de la combinaison des valeurs des données d'entrée

Données de test pour la variable Xi :

$DTx_i = \{di_1, \ldots, di_n\}$

- Produit cartésien :

$DTX_1 \times DTx_2 \times \ldots \times DTx_n$

=> Explosion du nombre de cas de tests

- Choix de classes d'équivalence portant sur l'ensemble des données d'entrée

Test aux limites - Evaluation

- Méthode de test fonctionnel très productive :
  - le comportement du programme aux valeurs limites est souvent pas ou insuffisamment examiné
- Couvre l ’ensemble des phases de test (unitaires, d ’intégration, de conformité et de régression)
- Inconvénient : caractère parfois intuitif ou subjectif de la notion de limite

  $\Rightarrow$ Difficulté pour caractériser la couverture de test.

Méthodes de test fonctionnel - Exercice 1

- Supposons que nous élaborions un compilateur pour le langage BASIC. Un extrait des spécifications précise :

« *L ’instruction FOR n ’accepte qu* ***’un seul paramètre*** *en tant que variable auxiliaire. Son nom ne doit pas dépasser* ***deux caractères non blancs****; Après le signe = est précisée aussi une borne supérieure et une borne inférieure. Les bornes sont des* ***entiers positifs*** *et on place entre eux le mot-clé TO.* »

- Déterminer par **analyse partitionnelle** des domaines des données d ’entrée les cas de test à produire pour l ’instruction *FOR*.

Méthodes de test fonctionnel - Correction Exercice 1

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

Méthodes de test fonctionnel- Exercice 2

- Considérons les spécifications suivantes :

« *Ecrire un programme statistique analysant un fichier comprenant les noms et les notes des étudiants d ’une année universitaire. Ce fichier se compose au maximum de 100 champs. Chaque champ comprend le nom de chaque étudiant (20 caractères), son sexe (1 caractère) et ses notes dans 5 matières (entiers compris entre 0 et 20). Le but du programme est de :*

  - *calculer la moyenne pour chaque étudiant,*
  - *calculer la moyenne générale (par sexe et par matière),*
  - *calculer le nombre d ’étudiants qui ont réussi (moyenne supérieure à 10)* »

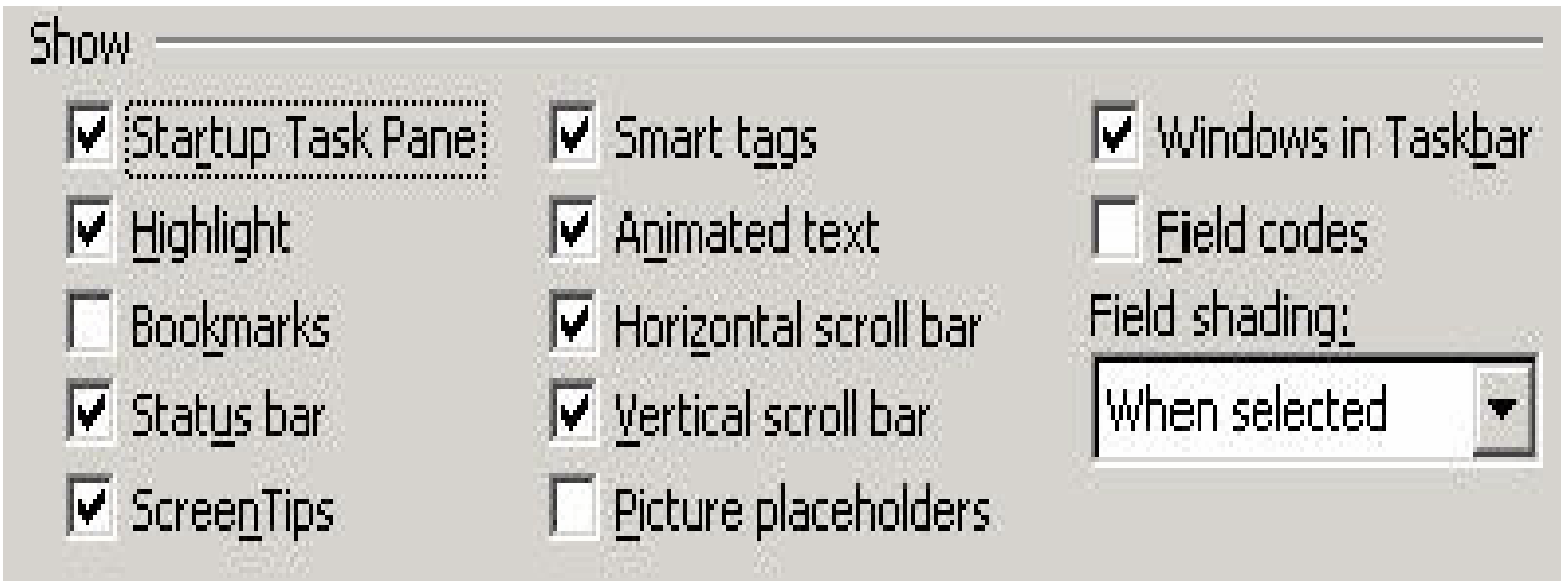
- Déterminer par **une approche aux limites** les cas de test à produire pour cette spécification

Méthodes de test fonctionnel - Correction Exercice 2

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

2.2 Méthode pour le test combinatoire

- Les combinaisons de valeurs de domaines d'entrée donne lieu a explosion combinatoire
- Exemple : Options d'une boite de dialogue MS Word



➔ $2^{12} * 3$ (nombre d'item dans le menu déroulant) = 12 288

Test combinatoire : approche Pairwise

- Tester un fragment des combinaisons de valeurs qui garantissent que chaque combinaison de 2 variables est testé
- Exemple : 4 variables avec 3 valeurs possibles chacune

OS	Réseau	Imprimante	Application
XP	IP	HP35	Word
Linux	Wifi	Canon900	Excel
Mac X	Bluetooth	Canon-EX	Pwpoint

Toutes les combinaisons : 81

Toutes les paires : 9

Pairwise testing

- 9 cas de test : chaque combinaison de 2 valeurs est testée

	OS	Réseau	Imprimante	Application
Case 1	XP	ATM	Canon-EX	Pwpoint
Case 2	Mac X	IP	HP35	Pwpoint
Case 3	Mac X	Wifi	Canon-EX	Word
Case 4	XP	IP	Canon900	Word
Case 5	XP	Wifi	HP35	Excel
Case 6	Linux	ATM	HP35	Word
Case 7	Linux	IP	Canon-EX	Excel
Case 8	Mac X	ATM	Canon900	Excel
Case 9	Linux	Wifi	Canon900	Pwpoint

L'idée sous-jacente : la majorité des fautes sont détectées par des combinaisons de 2 valeurs de variables

Test combinatoire

- L'approche Pairwise se décline avec des triplets, des quadruplets, …. mais le nombre de tests augmente très vite
- Une dizaine d'outils permette de calculer les combinaisons en Pairwise (ou n-valeurs) :
  - http://www.pairwise.org/default.html
  - Prise en charge des exclusions entre valeurs des domaines et des combinaison déjà testée
  - Exemple : outil Pro Test
- Problème du Pairwise :
  - le choix de la combinaison de valeurs n'est peut-être pas celle qui détecte le bug …
  - Le résultat attendu de chaque test doit être fournis manuellement

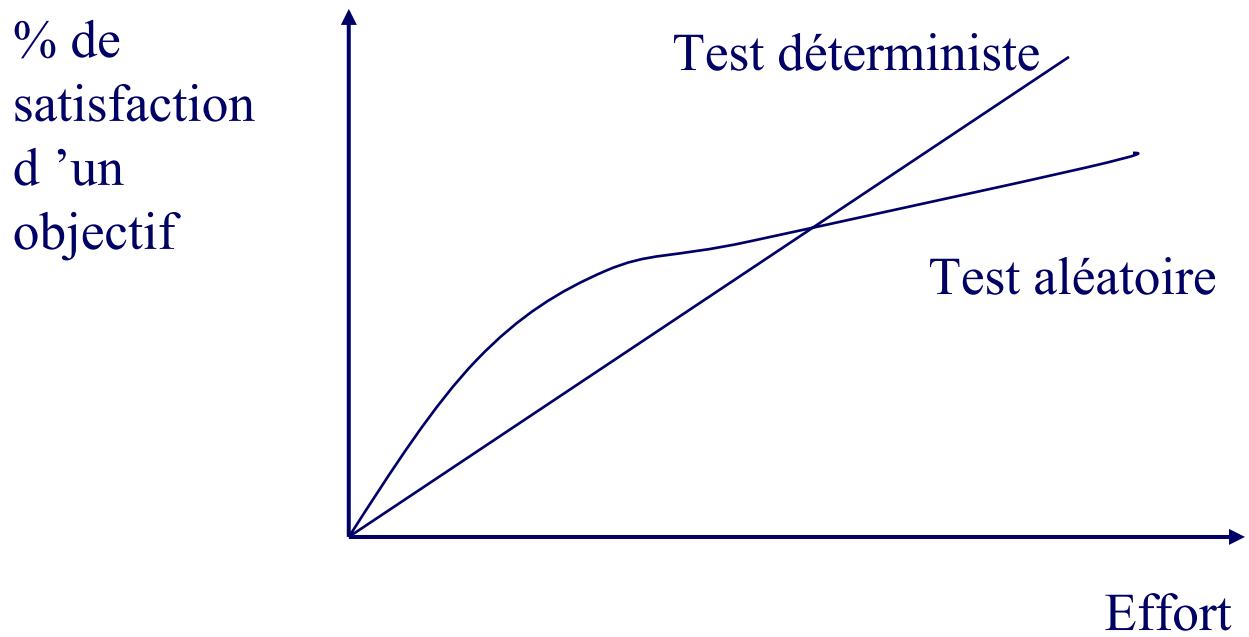
2.3 Test aléatoire ou statistique

- Principe : utilisation d ’une fonction de calcul pour sélectionner les DT :
  - fonction aléatoire : choix aléatoire dans le domaine de la donnée d ’entrée,
  - utilisation d ’une loi statistique sur le domaine.
- Exemples :
  - Echantillonnage de 5 en 5 pour une donnée d ’entrée représentant une distance,
  - Utilisation d ’une loi de Gauss pour une donnée représentant la taille des individus,
  - ...

Evaluation du test aléatoire

- Intérêts de l ’approche statistique :
  - facilement automatisable pour la sélection des cas de test, (plus difficile pour le résultat attendu)
  - objectivité des DT.
- Inconvénients :
  - fonctionnement en aveugle,
  - difficultés de produire des comportements très spécifiques.

Productivité du test aléatoire ou statistique



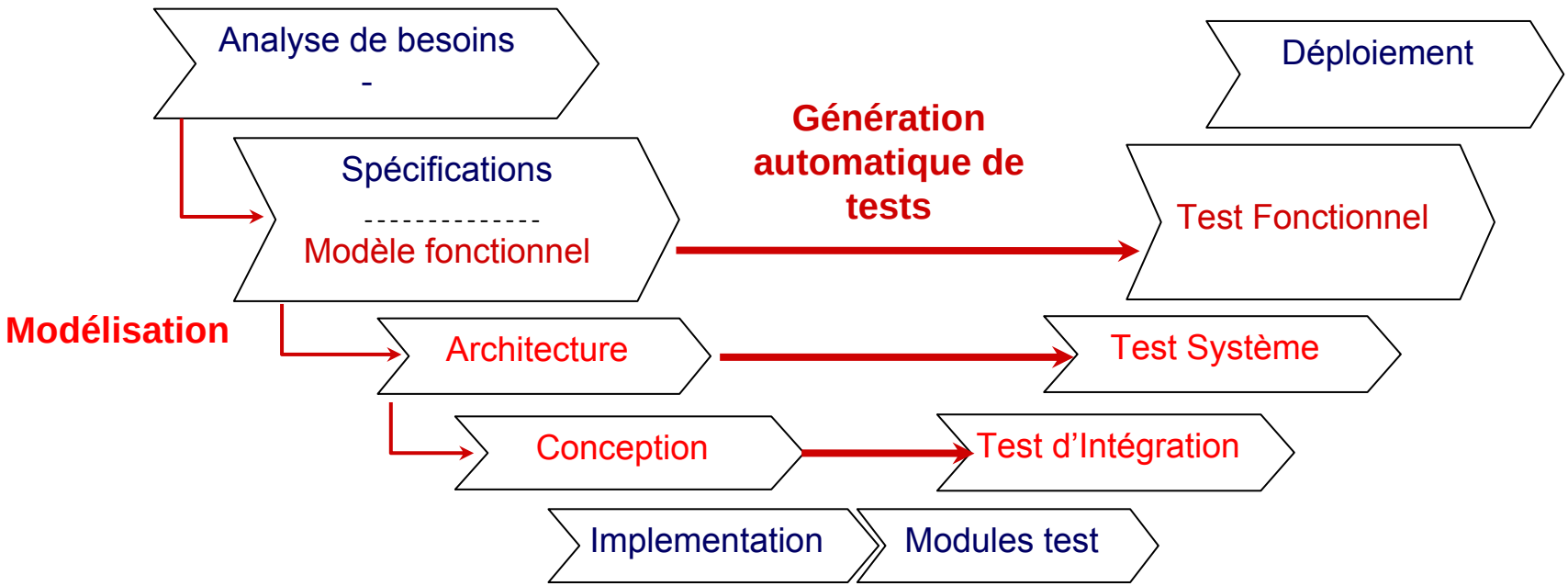
Les études montrent que le test statistique permet d 'atteindre rapidement 50 % de l 'objectif de test mais qu 'il a tendance à plafonner ensuite.

2-4– Génération automatique de tests à partir de spécifications

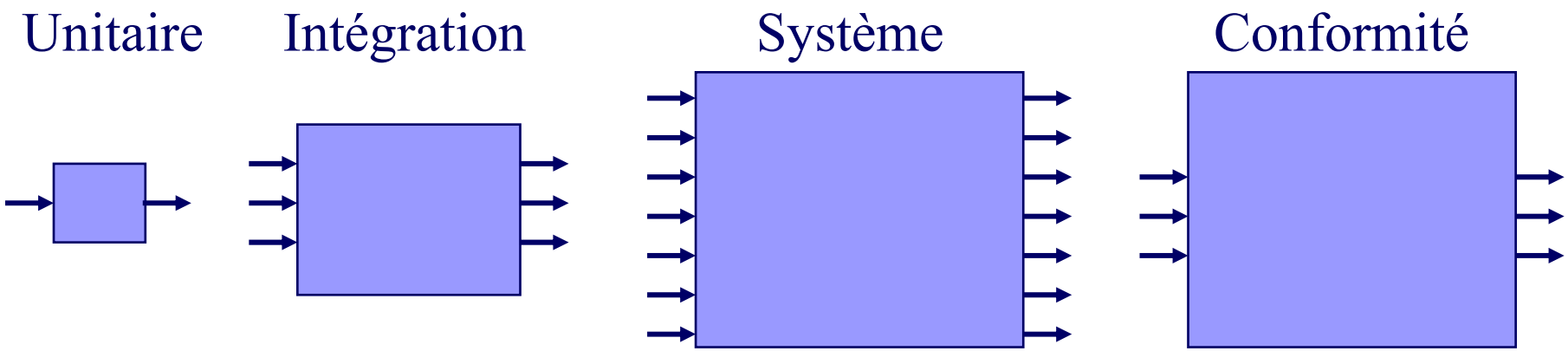
- Test à partir de modèles (Model-Based Testing)
- Modéliser pour tester
- Stratégies de génération
- Sélection des tests – Critères de couverture
- Exemple d'outils : LEIRIOS Test Generator
- Etudes de cas

Test à partir de modèle dans le cycle de vie

- Test boîte noire

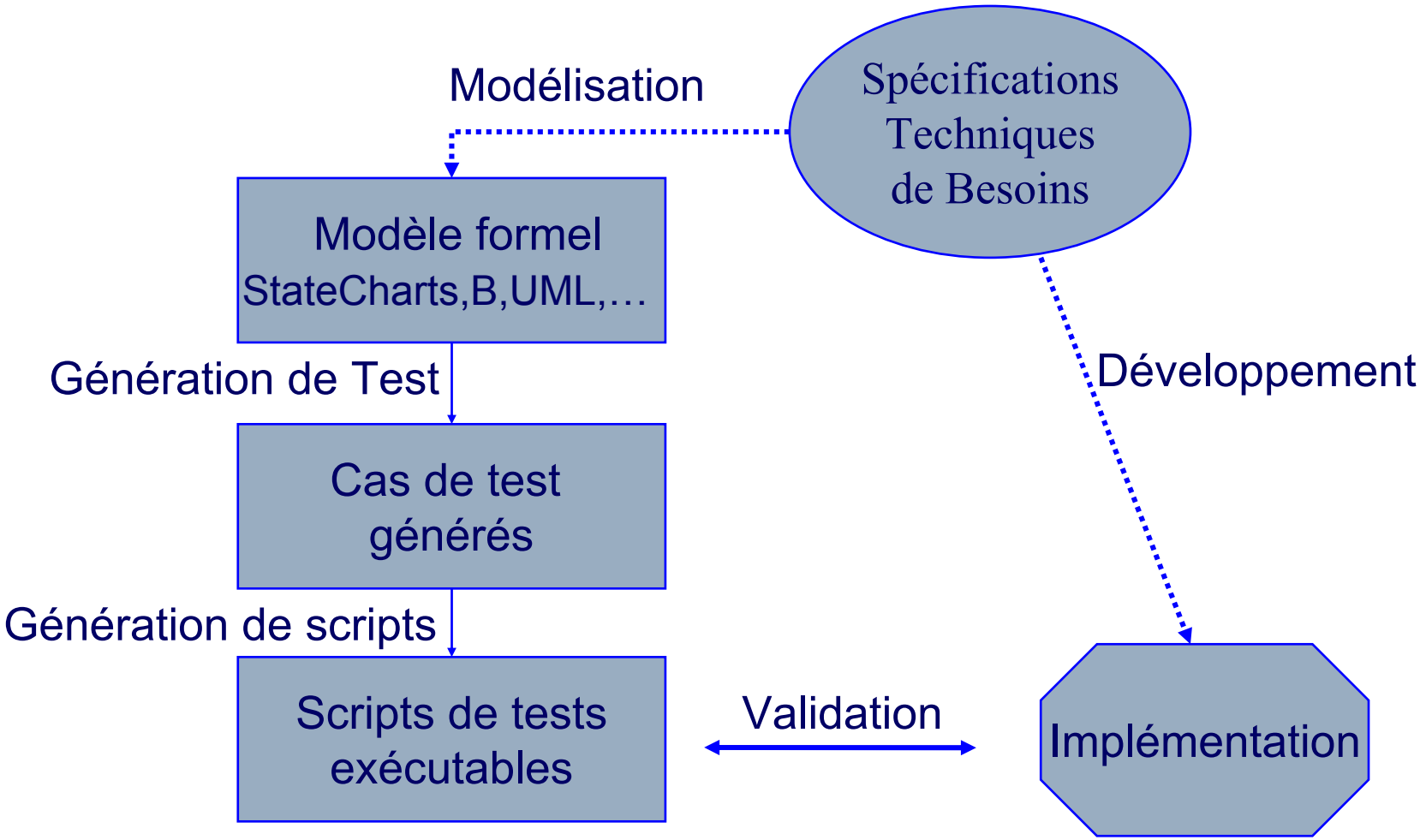


Application du test boite-noire

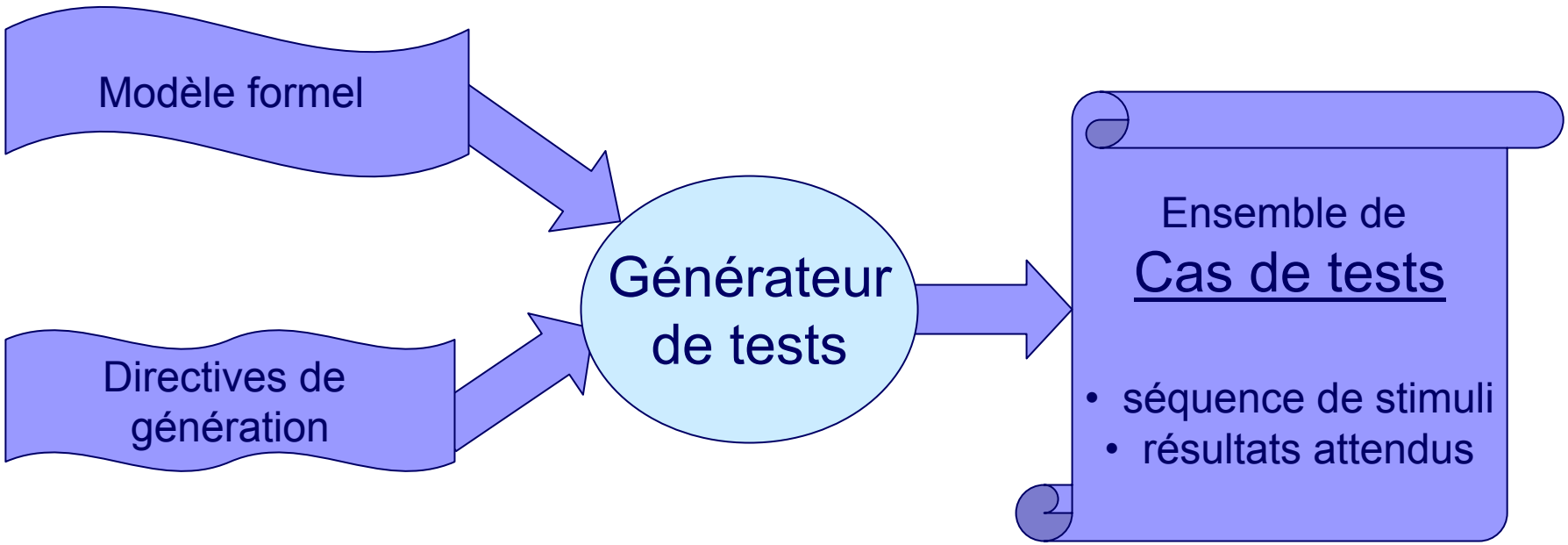


- Le test boite-noire peut s'appliquer au différent niveau de remontée du cycle en V
  - ➔ le modèle utilisé pour la génération sera adapté au niveau testé (module, sous-système ou système) et aux objectifs de test

Process du test à partir de modèles



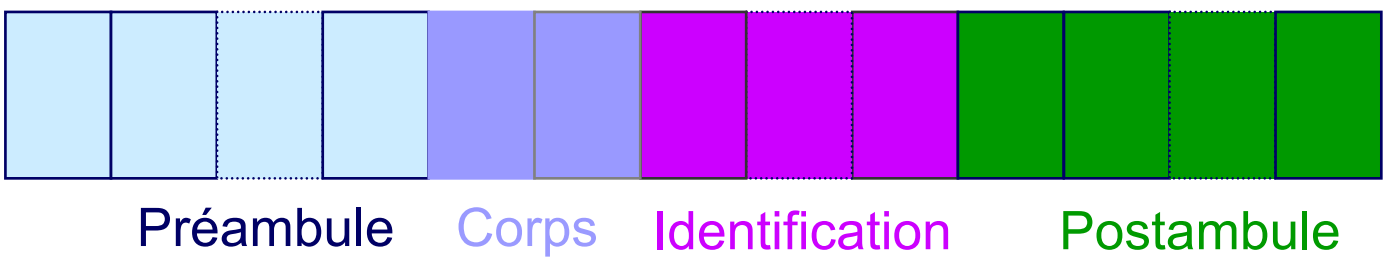
Model Based Test Generation



- Directives de génération (définition d'un scénario de tests) :
  - Critères de couverture du modèle
  - Critères de sélection sur le modèle

Composition d'un cas de test

- Un cas de test généré se décompose en quatre parties:
  - Préambule : séquence des stimuli depuis l'état initial jusqu'à un état recherché pour réalisé le test,
  - Corps : appel des stimuli testé
  - Identification : appel d'opérations d'observation pour consolider l'oracle (facultatif)
  - Postambule : chemin de retour vers l'état initial ou un état connu permettant d'enchaîner automatiquement les tests



Model-Based Testing : Technologie émergente

- De nombreux travaux de recherche depuis 10 ans :
  - Comment simuler l'exécution des spécifications ?
  - Quels critères de couverture du modèle ?
  - Comment maîtriser l'explosion combinatoire ?
  - Comment assurer la liaison en les tests générés à partir du modèle et l'environnement d'exécution des tests ?
  - Quelle modélisation des spécifications fonctionnelles dans le contexte de la génération de tests ?
- De nombreux centres actifs

IBM Haifa
LIFC
Microsoft Research
CEA
LSR
Univ. Twente/Nijmegen
Ac. Sc. St Petersbourg
IRISA
Univ. Munich
Verimag
MIT

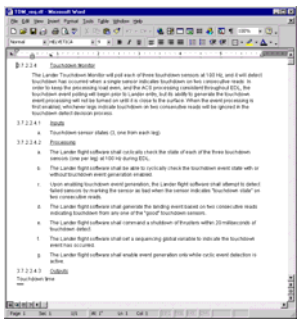
2-4- Génération automatique de tests à partir de spécifications

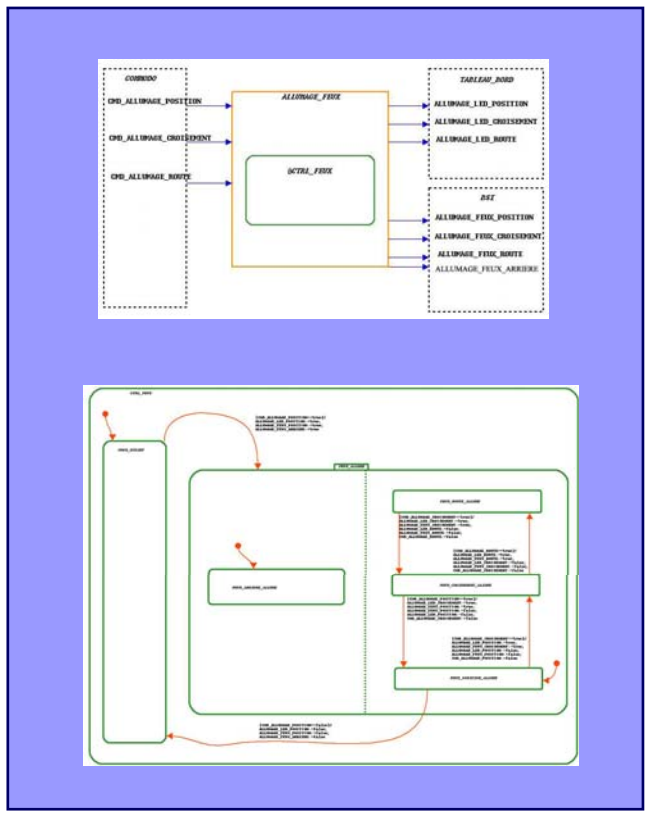
- Test à partir de modèles (Model-Based Testing)
- Modéliser pour tester
  - Modéliser les spécifications fonctionnelles
  - Besoins en modélisation dans le contexte de la génération de tests
  - Choix de notation
- Stratégies de génération
- Sélection des tests – Critères de couverture
- LEIRIOS Test Generator
- Etudes de cas

Modéliser les spécifications fonctionnelles

- Modéliser les objets et les comportements attendus du système

Modélisation





Spécifications fonctionnelles

Diagramme d'activités

Diagramme d'états

Modèle fonctionnel

La modélisation des spécifications est une tendance lourde du développement de logiciels

- Modéliser les spécifications :
  - Valider l'expression de besoins
    - » Inspecter le modèle
    - » Vérifier le modèle
    - » Animer / simuler le modèle
  - Documenter
  - Générer les tests
  - Générer le code

} Besoins ≠

⇒ Model-Driven Development

Modéliser pour tester : le besoin (1)

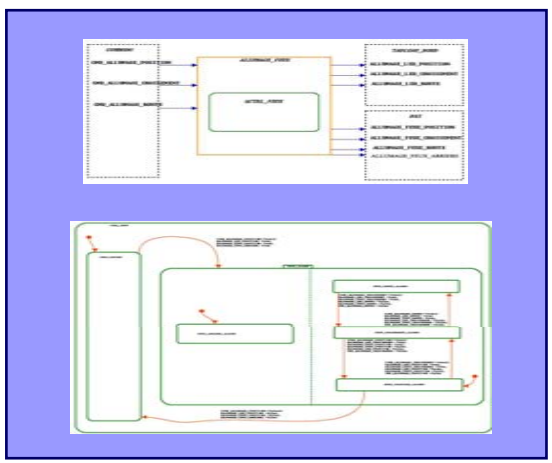
- Modèle abstrait
  - modèle fonctionnel (et non modèle d'architecture et/ou d'implémentation)
- Modèle détaillé et précis
  - Le modèle doit permettre de calculer les cas de test et les résultats attendus

  ➔ établir automatiquement le verdict de test
- Modèle validé et vérifié
  - Si le verdict de test « Fail » : l'erreur est-elle dans l'implantation ou dans le modèle ?

  Le test à partir de modèle confronte deux points de vue

Modéliser pour tester : le besoin (2)

- Modèle adapté aux objectifs de test
  - Plus le modèle intègre d'informations inutiles par rapport aux objectifs de tests, plus cela génère de « bruit » en génération de tests
- Modèle tenant compte des points de contrôle et d'observation du système sous test



Modèle fonctionnel

Génération des tests

Génération des scripts

Banc de test

Notations de modélisation

- De nombreux paradigmes
  - Systèmes de transitions (Etats / Transitions / Evénements)
    - » Flow Charts
    - » Data Flow Diagrams
    - » Diagramme d'état
  - Diagrammes objet & association (Entité-Relation, héritage, …)
    - » Diagramme de classe (UML)
    - » Entité-Relation
  - Représentation Pré-Post conditions
    - » OCL (UML)
    - » Machine Abstraite B
- Représentation :
    - » Graphique (plus « intuitive »)
    - » Textuelle (plus précise)

Quelles notations de modélisation en génération de tests à partir de modèles ?

1. Pouvoir calculer les cas de test et les résultats attendus
2. Pouvoir se situer à un niveau abstrait
3. Pouvoir capter le modèle de données (variables/valeur)
4. Pouvoir capter les caractéristiques spécifiques de l'application (ex. Temps réel, …)

```
Invariant
  x ∈ -1000 .. 1000
Pre_op
  x ≤ 100
Post_op
  IF  x ≤ 0 THEN  y := default
        ELSE IF  x ≤ 40
                 THEN  y := low
                 ELSE  y:= high
        END
  END
```

Exemple en notation B

Choix de notations

- Evalutation pour quelques notations

Notation	**Présentation**	+++	---
UML 2.0 (Diag. Classe + OCL + Diag. État)	Unified Modeling Language	. Grande diffusion . Variété de diagrammes	. Peu précise . Pas de sémantique
Statecharts	Diag. Etat	. Bien adapté contrôleur système embarqué	. Faible sur les données
Notation B	Méthode formelle	. Sémantique claire . Précision	. Apprentissage

Exemple 1 – Le régulateur de vitesse

- Le régulateur de vitesse permet de régler la vitesse du véhicule suivant la vitesse demandée par le conducteur du véhicule.
- Un bouton "*on/off*" permet d'activer le système. Si le conducteur accélère, le système de régulation est désactivé pendant le temps de l'accélération. Si le conducteur freine, le système est mis à *off*. Ce système est modélisé en *Statecharts Statemate* (I-Logix) au travers d'un diagramme d'activités et d'un diagramme d'états.

Diagramme d'activités du contrôleur de vitesse

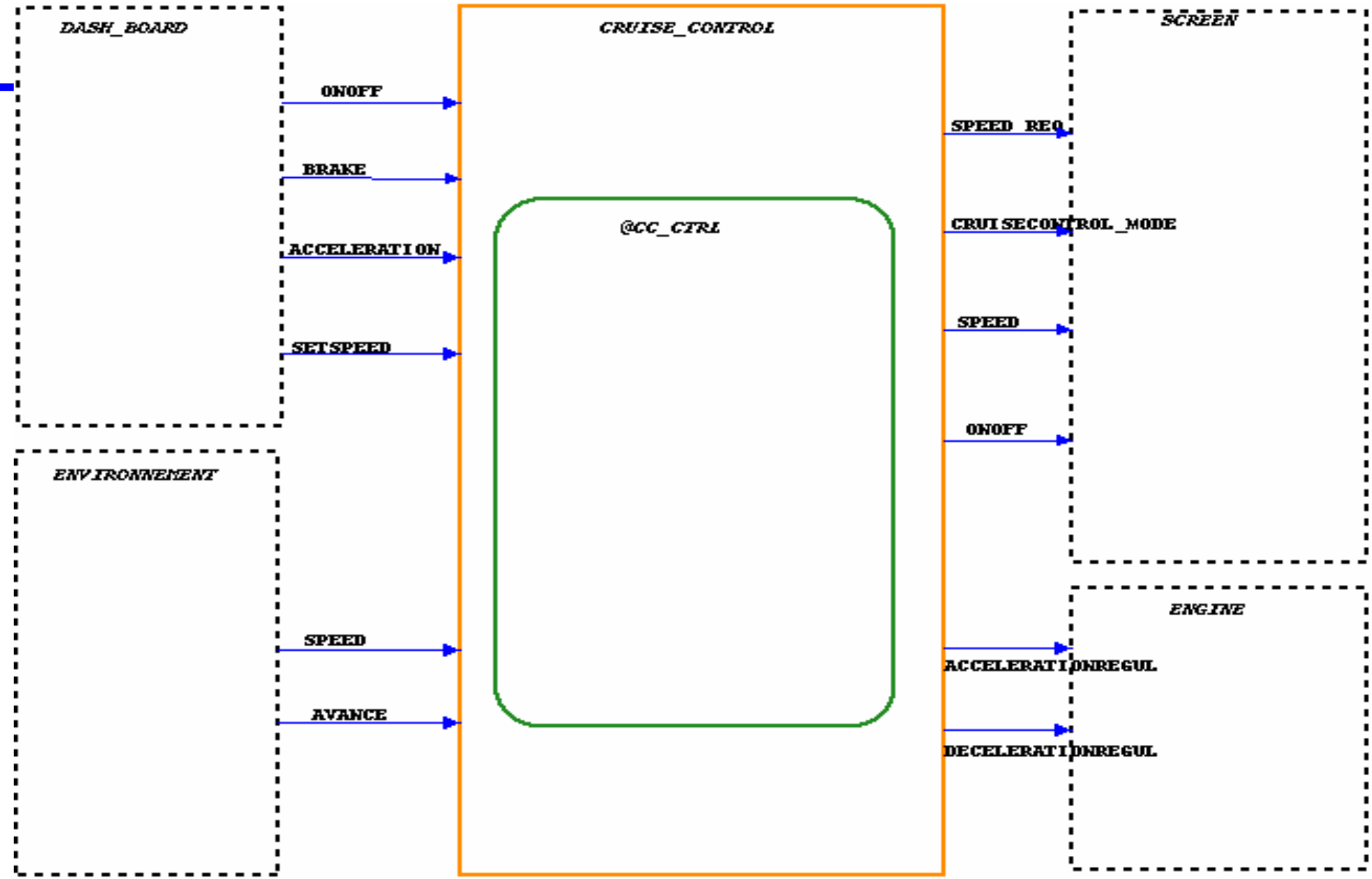
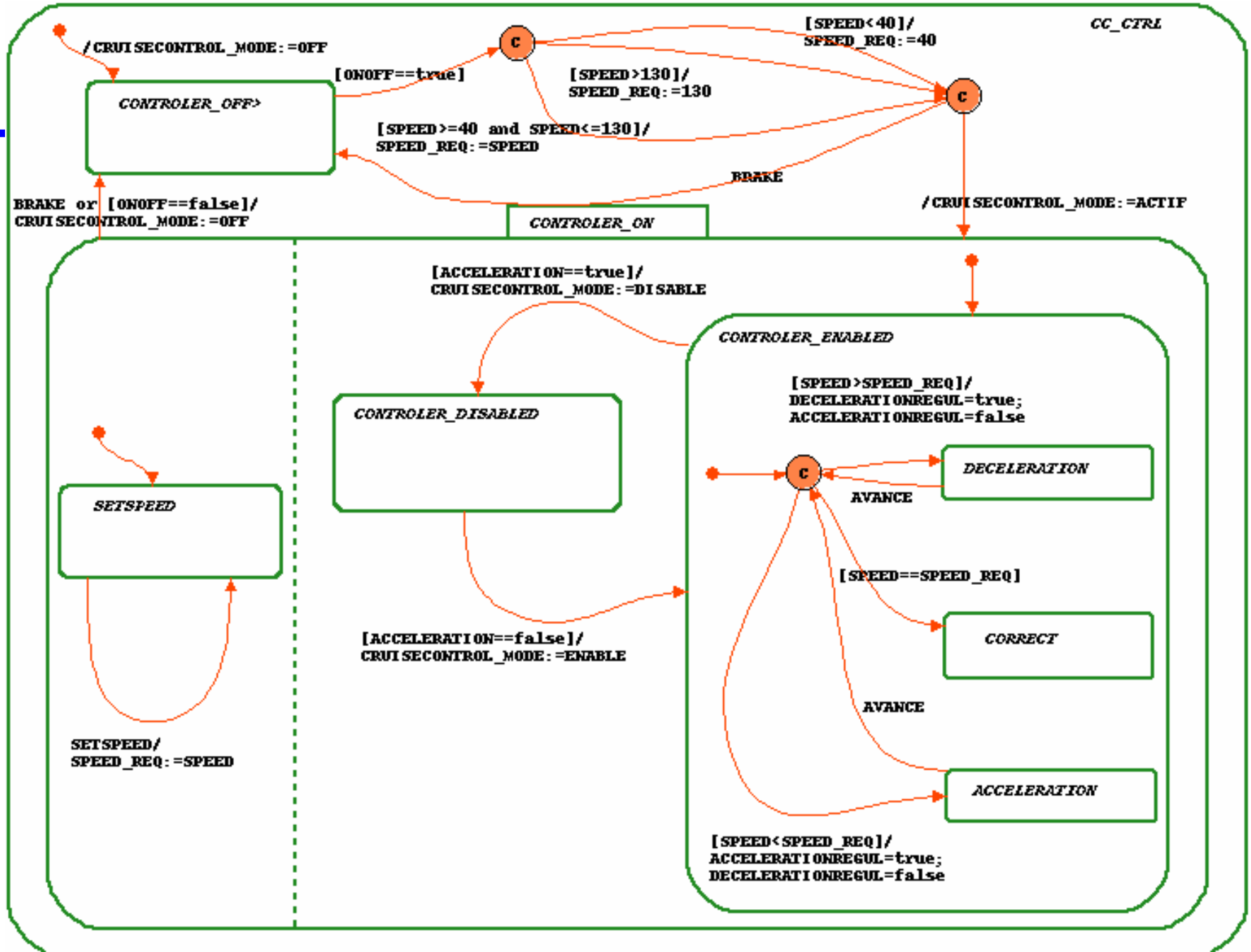


Diagramme d'états du contrôleur de vitesse



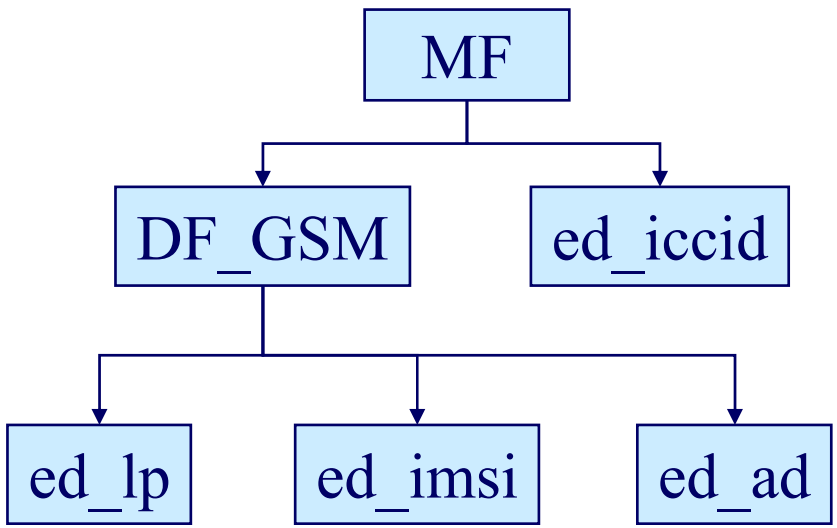
Exemple 2 – La norme carte à puces GSM 11-11

- La norme GSM 11-11 défini l'interface entre le Mobile (téléphone GSM) et la puce SIM – Subscriber Identifier Module
- La puce SIM est une carte à puces qui contient toutes les données utilisateurs et gére la protection de l'accès
- Le standard GSM 11-11 propose 21 API incluant la gestion du PIN – Personal Identifier Number – et du PUK – Personal Unblocking Key – et la lecture et mise à jour de données.

Noyau de la norme GSM 11.11

- Commandes :
  - Verify_Chv(chv,code),
  - Change_Chv(chv,code1,code2),
  - Disable_Chv(code),
  - Enable_Chv(code),
  - Unblock_Chv(chv,code_unblock 1, code_unblock2),
  - Select_File(ff),
  - Read_Binary,
  - Update_Binary(data),
  - Reset,

- Arborescence de fichiers



Modèle en B de la norme GSM 11-11 (fragment) – 1/4

MACHINE GSM_11-11

SETS

FILES = {mf,df_gsm,ef_iccid,ef_lp,ef_imsi,ef_ad};
PERMISSION = {always,chv,never,adm};
VALUE = {true, false};
BLOCKED_STATUS = {blocked, unblocked};
CODE = {a_1,a_2,a_3,a_4};
DATA = {d_1,d_2,d_3,d_4}

CONSTANTS

FILES_CHILDREN,
PERMISSION_READ,
MAX_CHV,
MAX_UNBLOCK,
PUK

DEFINITIONS

MF == {mf};
DF == {df_gsm};
EF == {ef_iccid,ef_lp,ef_imsi,ef_ad};
COUNTER_CHV == 0..MAX_CHV;
COUNTER_UNBLOCK_CHV == 0..MAX_UNBLOCK

Modèle en B de la norme GSM 11-11 (fragment) – 2/4

PROPERTIES

FILES_CHILDREN : FILES <-> FILES &
FILES_CHILDREN = {(mf|->df_gsm), (mf|->ef_iccid),(df_gsm|->ef_lp),
(df_gsm|->ef_imsi),(df_gsm|->ef_ad)} &
PERMISSION_READ : EF --> PERMISSION &
PERMISSION_READ = {(ef_imsi|->never),(ef_lp|->always),
(ef_iccid|->chv),(ef_ad|->adm)} &
MAX_CHV = 3 &
MAX_UNBLOCK = 10 &
PUK : CODE &
PUK = a_3

VARIABLES

current_file,
current_directory,
counter_chv,
counter_unblock_chv,
blocked_chv_status,
blocked_status,
permission_session,
pin,
data

Modèle en B de la norme GSM 11-11 (fragment) – 3/4

INVARIANT

…

((blocked_chv_status = blocked) => ((chv|->false) : permission_session)) &
((counter_chv = 0) <=> (blocked_chv_status = blocked)) &
((counter_unblock_chv = 0) <=> (blocked_status = blocked))

INITIALISATION

current_file := {} ||
current_directory := mf ||
counter_chv := MAX_CHV ||
counter_unblock_chv := MAX_UNBLOCK ||
blocked_chv_status := unblocked ||
blocked_status := unblocked ||
permission_session := {(always|->true),(chv|->false),(adm|->false),(never|->false)} ||
pin := a_1 ||
data := {(ef_iccid|->d_1),(ef_lp|->d_2),(ef_imsi|->d_3),(ef_ad|->d_4)}

Modèle en B de la norme GSM 11-11 (fragment) – 4/4

```
sw <-- VERIFY_CHV(code) =
        PRE
            code : CODE
        THEN
            IF (blocked_chv_status = blocked)
            THEN
                sw := 9840
            ELSE
                IF (pin = code)
                THEN
                    counter_chv := MAX_CHV || permission_session(chv) := true || sw := 9000
                ELSE
                    IF (counter_chv = 1)
                    THEN
                        counter_chv := 0 || blocked_chv_status := blocked ||
                        permission_session(chv) := false || sw := 9840
                    ELSE
                        counter_chv := counter_chv - 1 || sw := 9804
                    END
                END
            END

        END;
```

Modélisation en B pour la génération de tests

- Niveau Machine

Abstraite (sans raffinement, mono-machine)

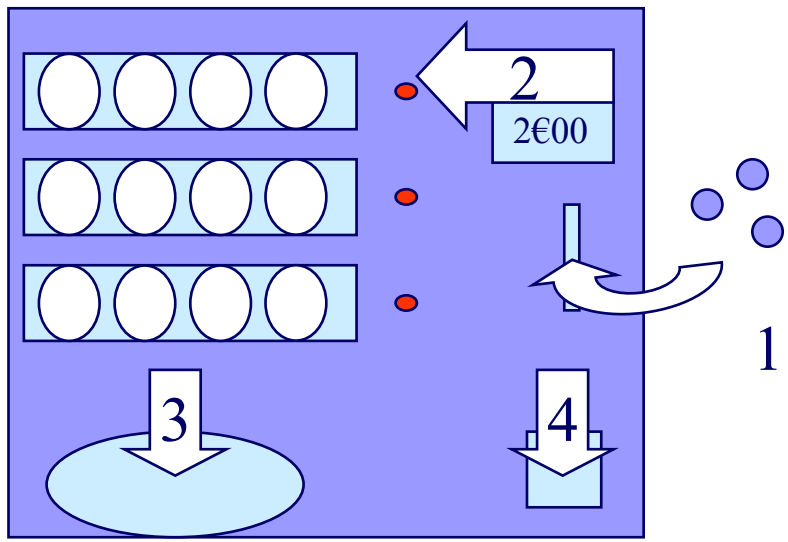
- Permet une bonne prise en compte des données et des traitements
- Bon niveau d'abstraction

```
sw ← CHANGE_Pin(old_code, new_code) =
    PRE
    old_code ∈ Nat ∧ new_code ∈ Nat
    THEN
        IF counter_pin = 0
            THEN
            sw := 9840
            ELSE
               IF code_pin = old_code
                  THEN
                  code_pin := new_code          ||
                  counter_pin := 3              ||
                  permission_session := true    ||
                  sw := 9000
                  ELSE IF counter_pin =1
                           THEN
                           counter_pin := 0                 ||
                           permission_session := false      ||
                           sw := 9840
                           ELSE
                           counter_pin := counter_pin – 1   ||
                           sw := 9804
END END END END END ;
```

Exemple 3 – Distributeur de boisson

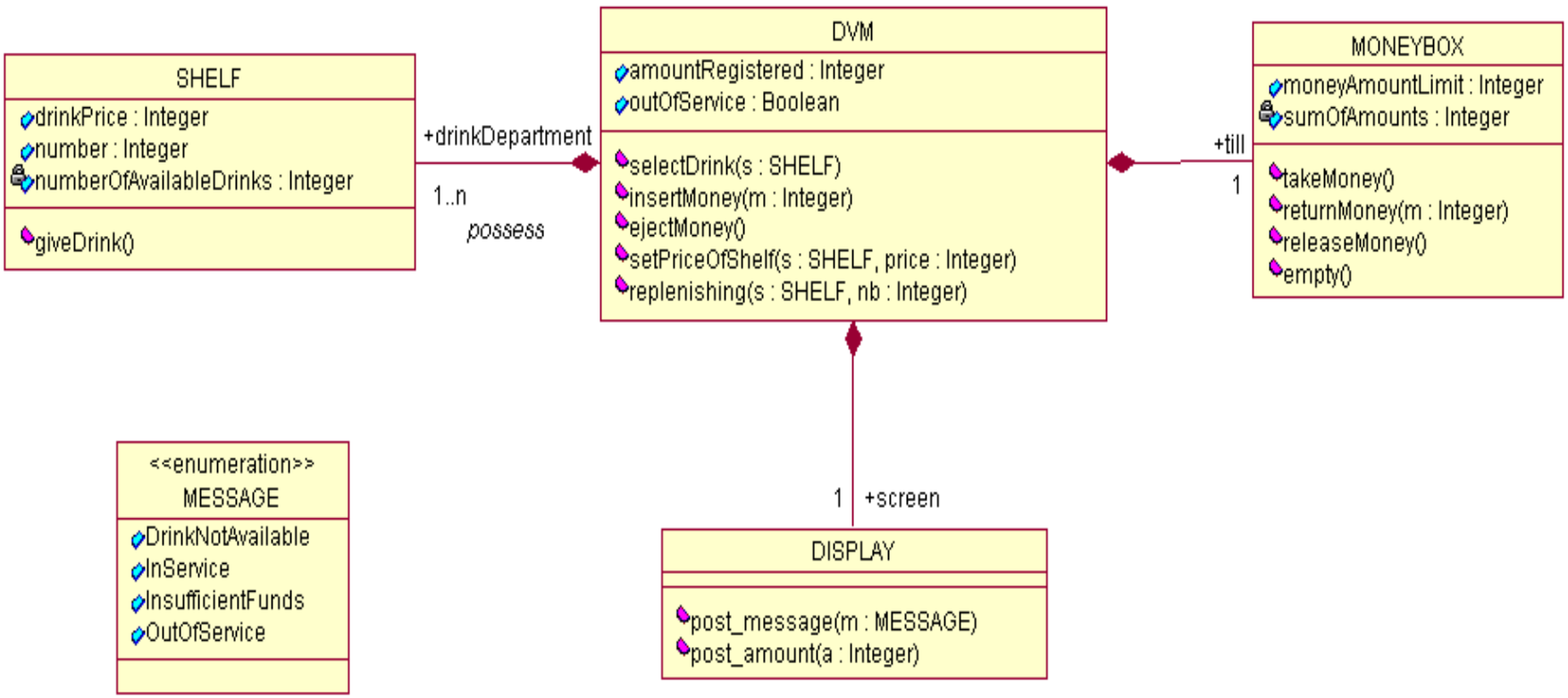
Scénario d'utilisation :

1. Le client introduit les pièces
2. Il choisit une boisson
3. Le distributeur remet la boisson choisie
4. Le cas échéant, la monnaie est rendue

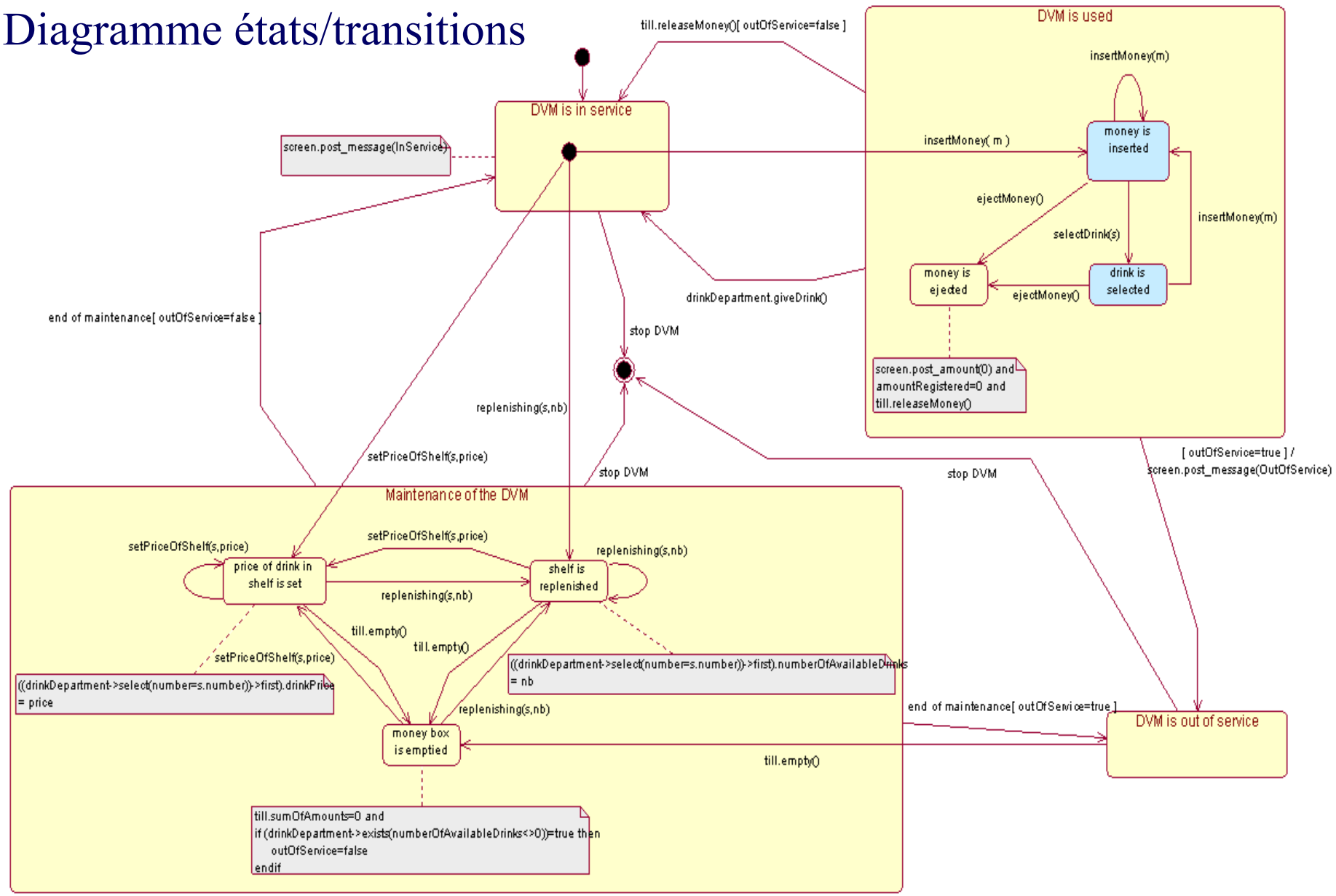


- Ce système est modélisé en *UML* au travers d'un diagramme de classe et d'un diagramme états / transitions.

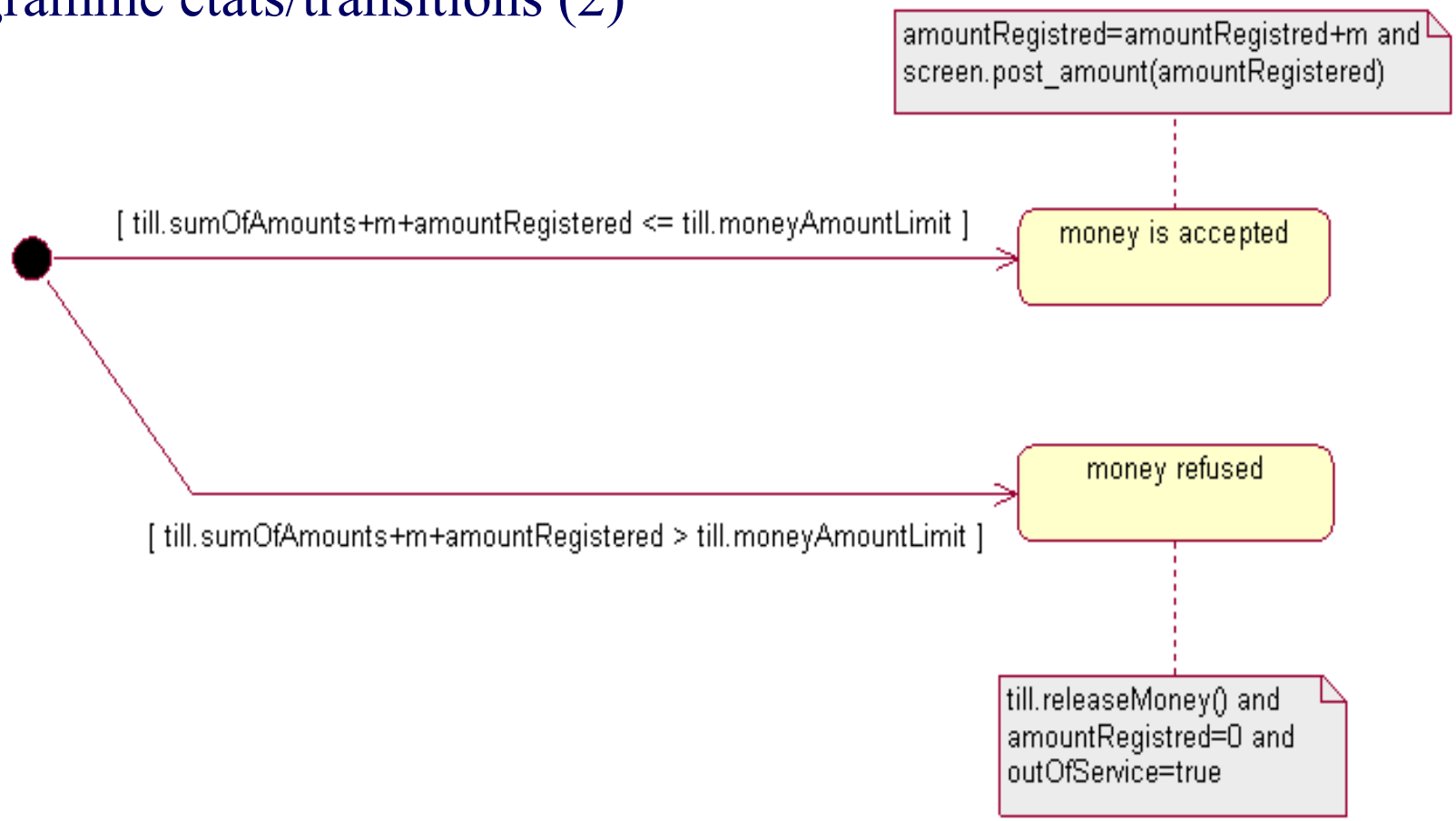
Distributeur de boisson – Diagramme de classes



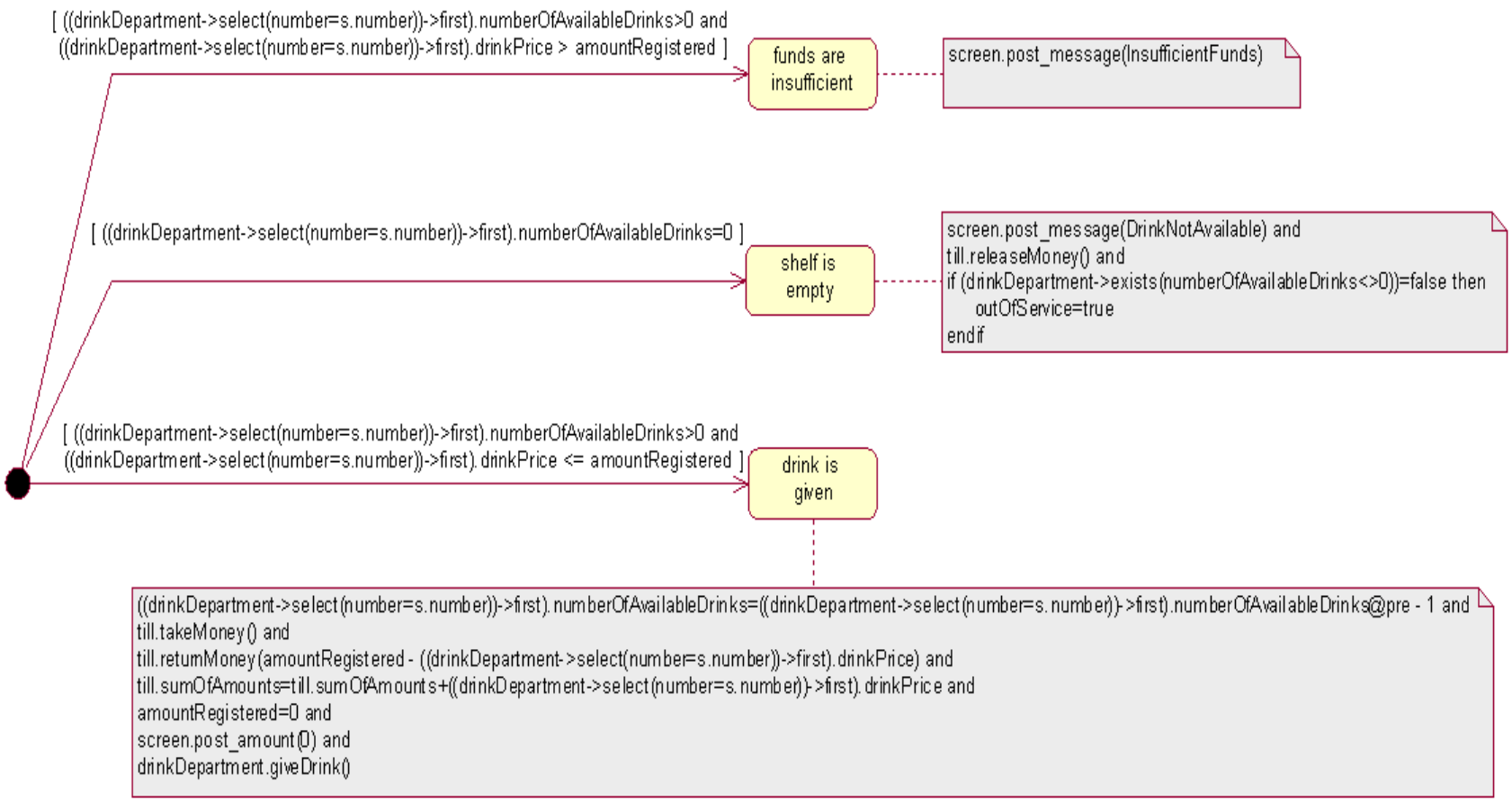
Distributeur de boisson : Diagramme états/transitions



Distributeur de boisson : Diagramme états/transitions (2)



Distributeur de boisson : Diagramme états/transitions (3)



Critères de choix d'une notation / type d'application

- Applications type contrôle-commande (automobile, énergie, aéronautique, …)
  - ➔ Statecharts, Lustre
- Applications mobiles et logiciels enfouis (télécom, électronique grand public, carte à puces, monétique, …)
  - ➔ Pré/Post conditions (UML/OCL, B, …)
- Systèmes d'information
  - ➔ Modélisation objet (UML)

Stratégie de modélisation

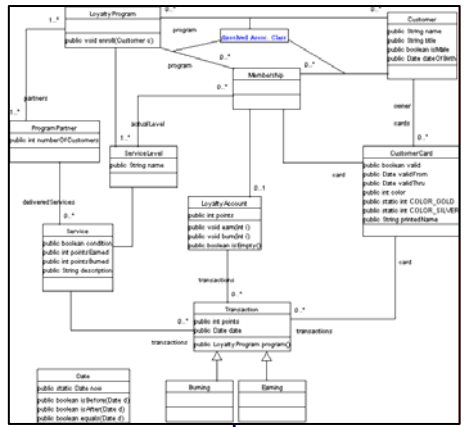
"Partial formal models focused on a specific feature or component of the software under test, for the sole purpose of test generation, is feasible and provide good results in a realistic industrial setting"

Farchi – Hartman - Pinter - IBM Systems Journal 41:1 - 2002

Processus Model-Based Testing – Schéma 1

Ingénieur modélisation

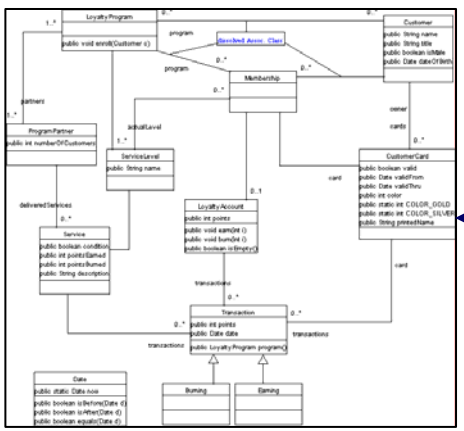
Modèle existant



Analyse de besoins

Documentation

Support du développement



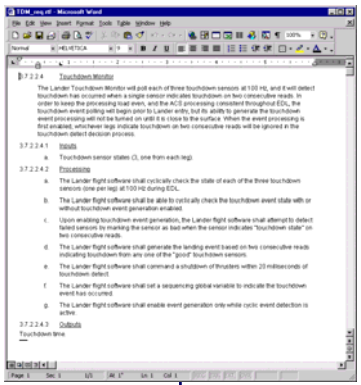
Adaptation

Ingénieur validation

Génération des tests

Cas de test

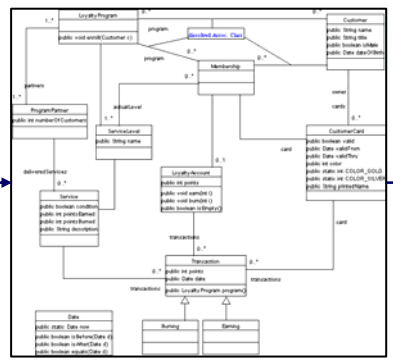
Processus Model-Based Testing – Schéma 2



Spécifications fonctionnelles

Développement

Modèle pour le test fonctionnel



Modélisation

Ingénieur validation

Pilotage

Génération des tests

Cas de test

Génération automatique de tests à partir de spécifications

- Test à partir de modèles (Model-Based Testing)
- Modéliser pour tester
- Stratégies de génération
  - Classes d'équivalence et test aux bornes
  - Maîtriser la combinatoire
- Sélection des tests – Critères de couverture
- LEIRIOS Test Generator
- Etudes de cas

Génération de tests fonctionnels

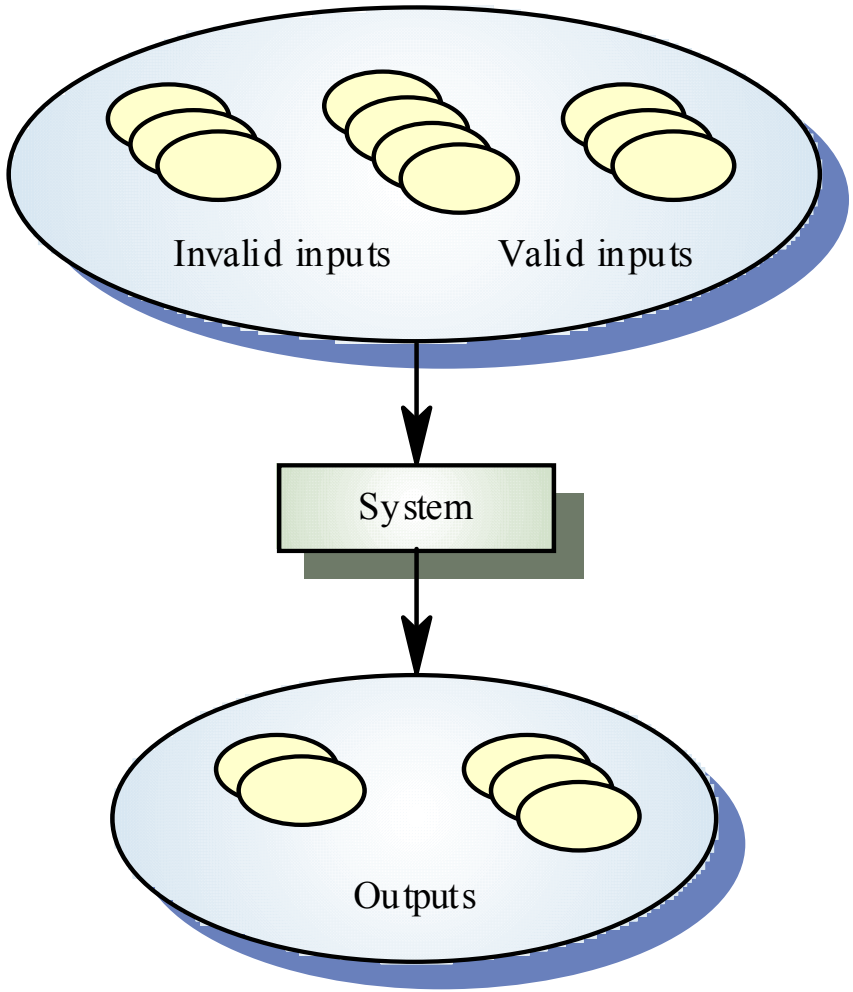
Automatise les stratégies classiques :

- Analyse partitionnelle des données d'entrée
- Test aux bornes
- Couverture de toutes les décisions

⇨ Sélection des Données de Tests

Objectif : Optimiser le ratio couverture du modèle / nombre de tests réalisés

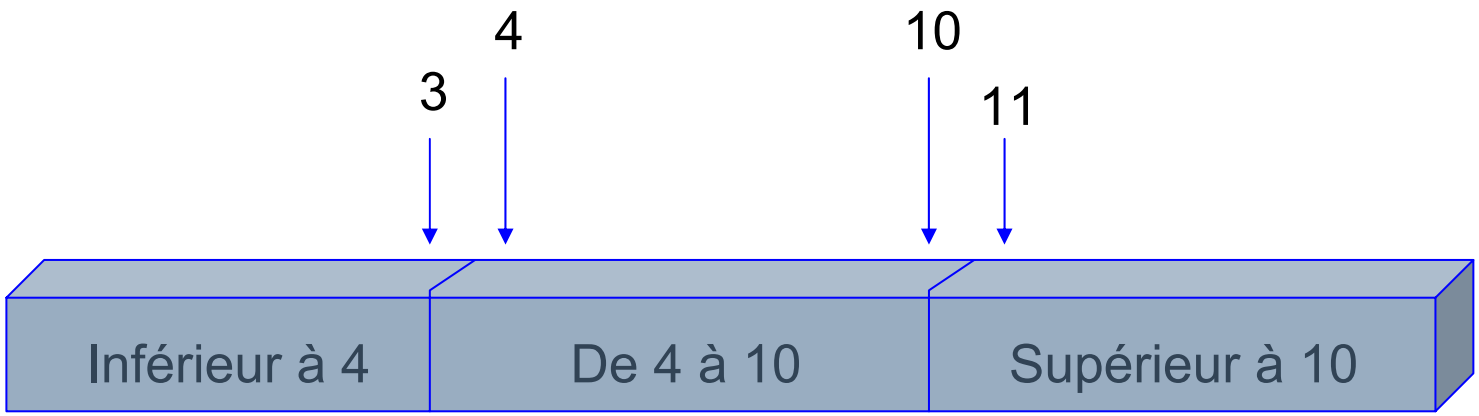
Analyse partitionnelle des domaines des données d'entrée



Une *classe d'équivalence* correspond à un ensemble de données de tests qui activent le même comportement.

La définition des classes d'équivalence permet de passer d'un nombre infinis de données d'entrée à un nombre fini et limité de données de test.

Test aux bornes des domaines des variables du modèle

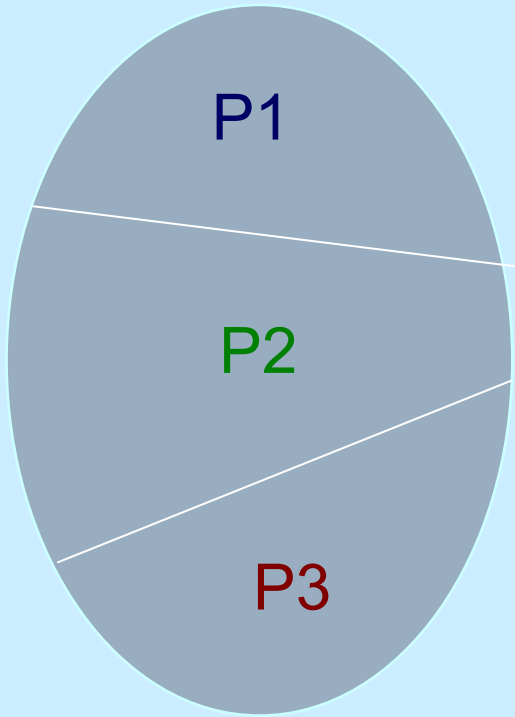


Le test aux bornes produit à la fois des cas de test nominaux (dans l'intervalle) et de robustesse (hors intervalle)

Analyse partitionnelle - Exemple

Invariant
$x \in -1000 .. 1000$
Pre_{op}
$x \leq 100$
$Post_{op}$
IF $x \leq 0$ **THEN** y := default
ELSE IF $x \leq 40$
THEN y := low
ELSE y:= high
END END

Classes de comportements
P1: $x \leq 0$
P2: $x > 0 \wedge x \leq 40$
P3: $x > 40 \wedge x \leq 100$



Ensemble de tous les états du système qui permettent d'activer l'opération

Calcul des données de test aux bornes

Prédicats après partitionnement

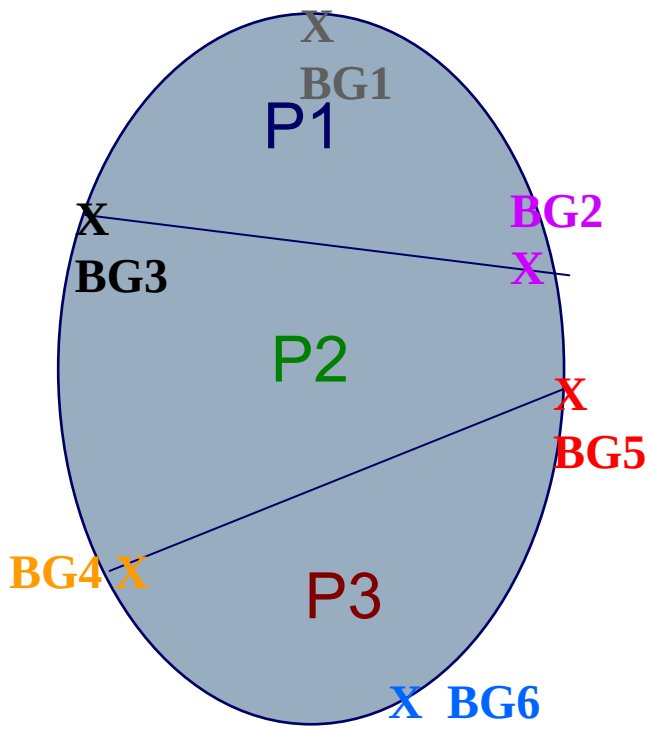
P1: $x \leq 0$

P2: $x > 0 \wedge x \leq 40$

P3: $x > 40 \wedge x \leq 100$

Tests aux bornes

BG1	x = -1000
BG2	x = 0
BG3	x = 1
BG4	x = 40
BG5	x = 41
BG6	x = 100



Calcul des données de test aux bornes

Prédicats après partitionnement

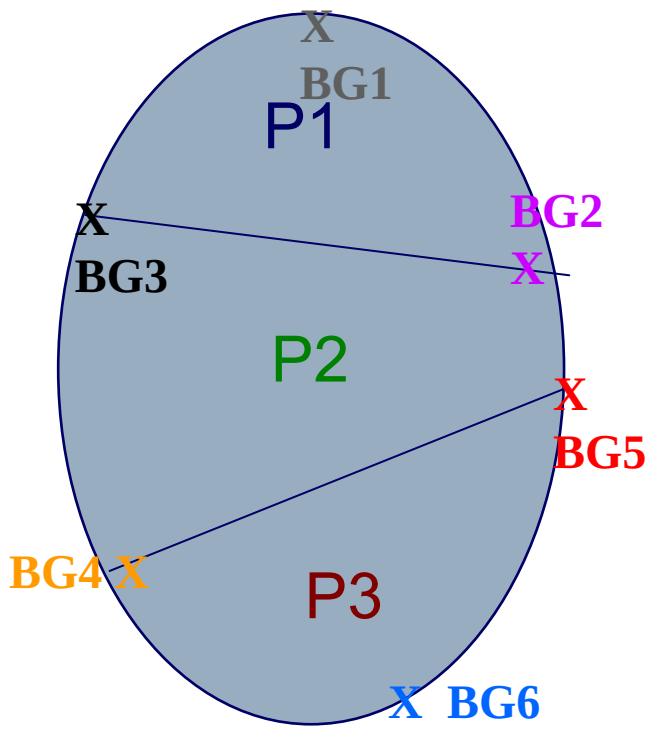
P1: $x \leq 0$

P2: $x > 0 \wedge x \leq 40$

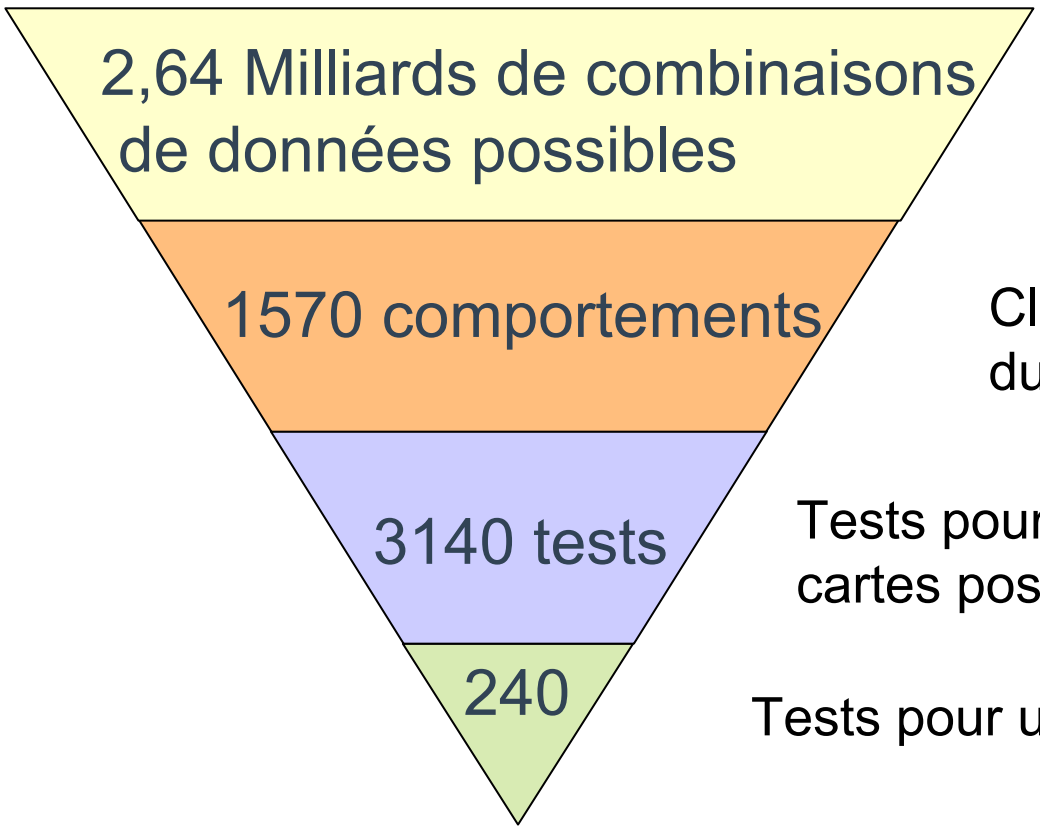
P3: $x > 40 \wedge x \leq 100$

Tests aux bornes

BG1	x = -1000
BG2	x = 0
BG3	x = 1
BG4	x = 40
BG5	x = 41
BG6	x = 100



Maîtriser la combinatoire : exemple de la validation terminal Carte Bancaire



MagIC 500

Schlumberger device

Classes de comportements du système

Tests pour toutes les configurations cartes possibles et les valeurs aux bornes

Tests pour une configuration particulière

2-4 - Génération automatique de tests à partir de spécifications

- Test à partir de modèles (Model-Based Testing)
- Modéliser pour tester
- Stratégies de génération
- Sélection des tests – Critères de couverture
  - critères de couverture du modèle
  - critères de sélection sur le modèle
- LEIRIOS Test Generator
- Etudes de cas

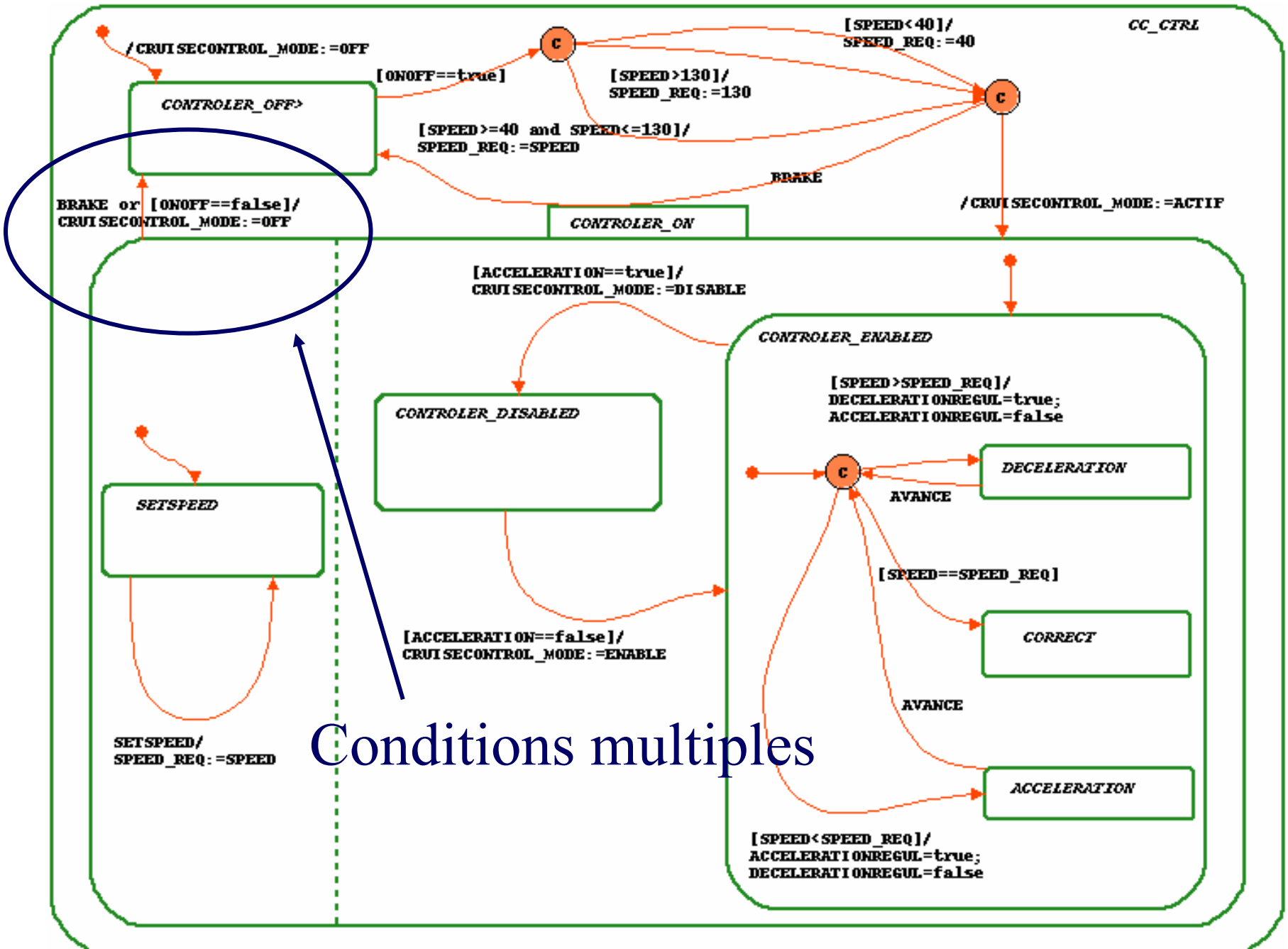
Génération des tests à partir de critères

- Critères de couverture du modèle
  - Obtenir le niveau de dépliage et de couverture souhaité à partir du modèle
- Critères de sélection
  - Focaliser la génération sur certains comportements
- Evolution du modèle
  - Sélectionner les tests par différentiel de comportements entre deux versions du modèle
- Cas utilisateur
  - Animer le modèle pour définir des cas de test

Critères de couverture

- Critères de conditions multiples dans les décisions
  - Toutes les décisions (DC)
  - Toutes les conditions/décisions (D/CC)
  - Toutes les décisions / conditions modifiées (MC/DC)
  - Toutes les conditions multiples (MCC)

Diagramme d'états du contrôleur de vitesse

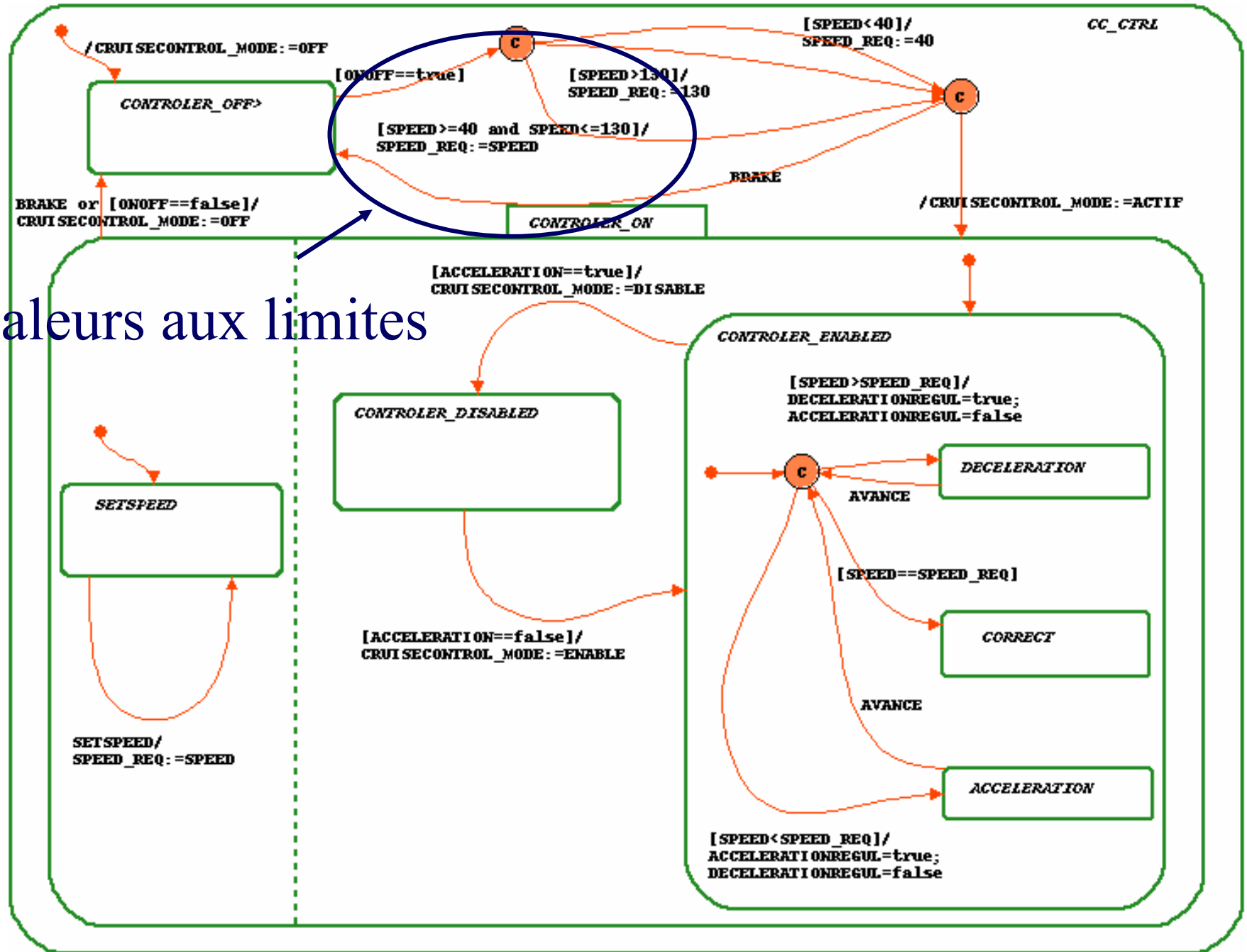


Conditions multiples

Critères de couverture

- Critères de conditions multiples dans les décisions
  - Toutes les décisions (DC)
  - Toutes les conditions/décisions (D/CC)
  - Toutes les décisions / conditions modifiées (MC/DC)
  - Toutes les conditions multiples (MCC)
- Critères de couvertures des valeurs aux bornes équivalentes
  - Une valeur
  - Toutes les valeurs

Diagramme d'états du contrôleur de vitesse

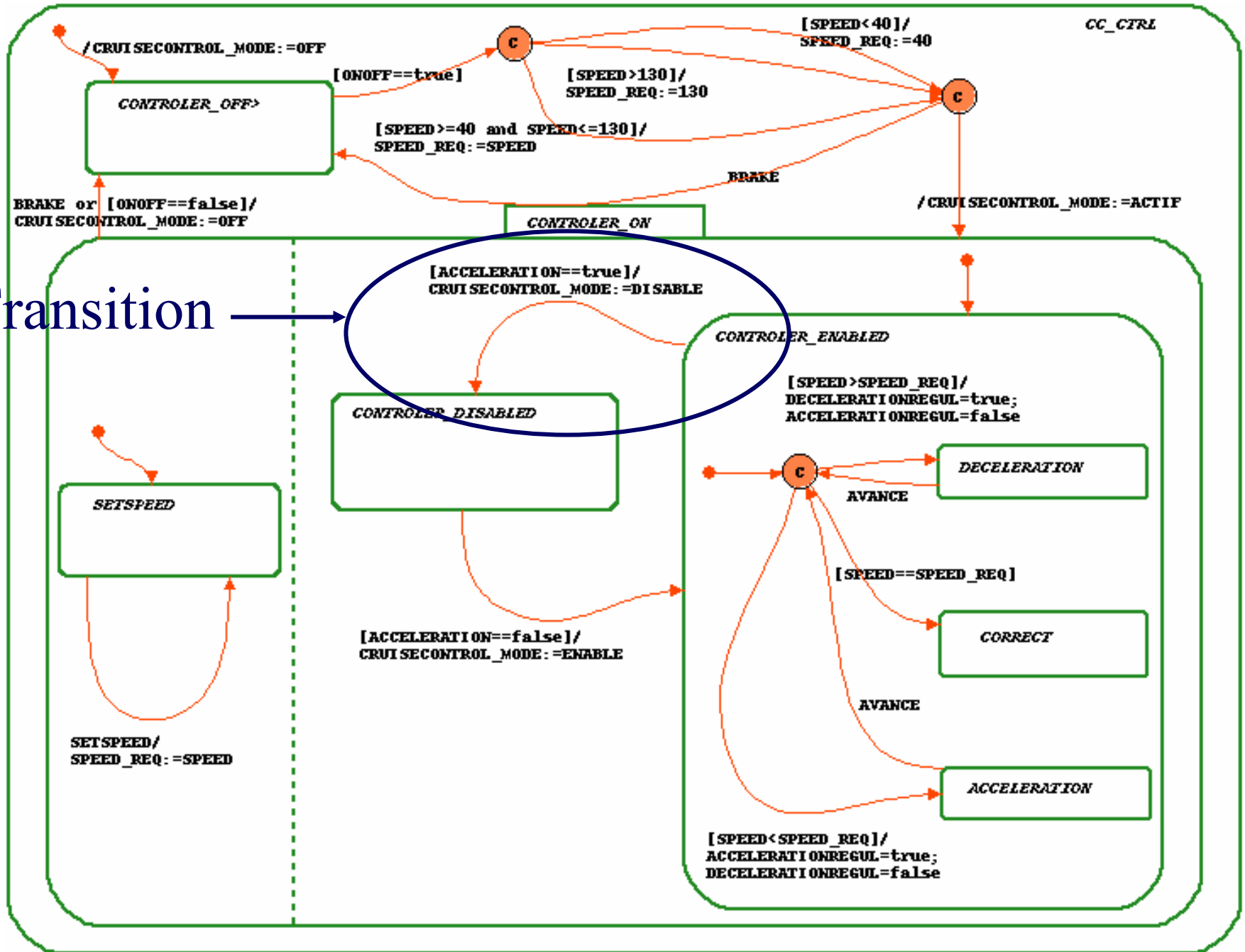


Valeurs aux limites

Critères de couverture

- Critères de conditions multiples dans les décisions
  - Toutes les décisions (DC)
  - Toutes les conditions/décisions (D/CC)
  - Toutes les décisions / conditions modifiées (MC/DC)
  - Toutes les conditions multiples (MCC)
- Critères de couvertures des valeurs limites équivalentes
  - Une valeur
  - Toutes les valeurs
- Critères de couverture des transitions
  - Toutes les transitions
  - Toutes les paires de transitions

Diagramme d'états du contrôleur de vitesse



Critères de sélection

Focus sur le modèle

- Sélection sur le choix des opérations / événements activables
- Sélection par choix sur les variables / valeurs du modèle

Evolution du modèle

- Test de tous les comportements ajoutés ou modifiés
- Test des comportements inchangés

Cas définis par l'utilisateur

- Animation du modèle pour définir des séquences d'activation: le générateur de test permet la production de l'oracle de test

2-4 - Génération automatique de tests à partir de spécifications

- Test à partir de modèles (Model-Based Testing)
- Modéliser pour tester
- Stratégies de génération
- Sélection des tests – Critères de couverture
- LEIRIOS Test Generator
- Etudes de cas

L'outil LEIRIOS Test Generator

- Approche : tester tous les comportements du système définis dans la spécification avec des données d'entrée aux bornes

- Caractéristiques de la technologie :
  - Génération automatique des tests et du résultat attendu
  - Traduction des séquences de tests abstraits en scripts exécutables concrets
  - Génération des tests nominaux (cas de test positifs) et des tests de robustesse (cas de tests négatifs)
  - Génération de la matrice de traçabilité Exigences / cas de tests

LEIRIOS Test Generator

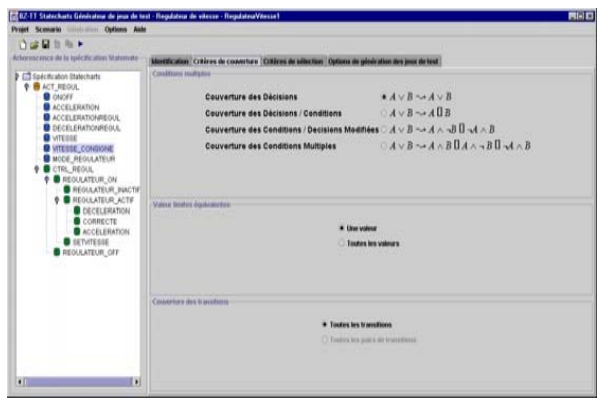
- Technologie issue de travaux de recherche démarré en 1996 à l'Université de Franche-Comté (LIFC) :
  - S'appuie sur une animation symbolique en Programmation en Logique avec Contraintes du modèle formel
- Applications industrielles depuis 1999 :
  - Carte à puces avec Schlumberger/Axalto,
  - Systèmes monétiques (EMV 2000, CB 5.2) avec Parkeon et G. Carte Bancaire
  - Systèmes embarqués dans l'automobile avec PSA Peugeot Citroën
  - Systèmes critiques avec IRSN, ESA
- La société LEIRIOS créée en septembre 2003 à partir de l'équipe BZ-Testing-Tools du LIFC :
  - Développe et commercialise l'outil LEIRIOS Test Generator – LTG

LEIRIOS Test Generator

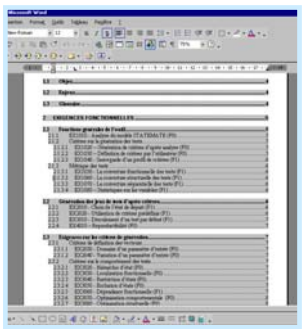
Caractéristiques principales

- LTG supporte actuellement trois notations :
  - Notation B (niveau machine abstraite)
  - Statecharts Statemate
  - UML (diagrammes de classe et états / transitions avec des expressions OCL)

- LTG propose des critères de couverture et de sélection pour piloter la génération de tests



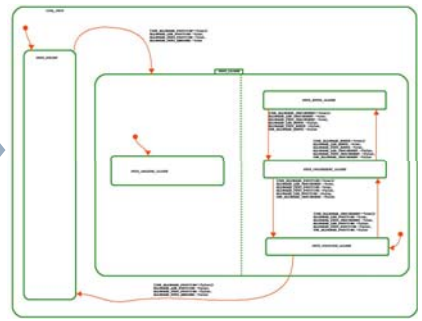
Process de génération



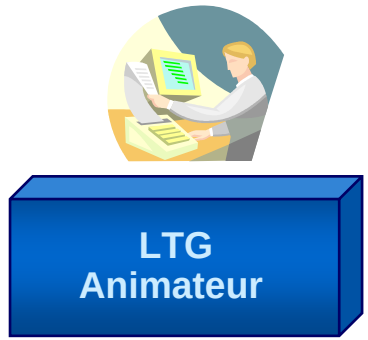
Spécifications
Techniques de Besoins

Modélisation
Fonctionnelle

Modèle fonctionnel
(B, Statecharts, UML, …)



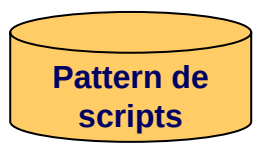
Validation du
modèle



Critères
de génération

Génération des
cas de tests

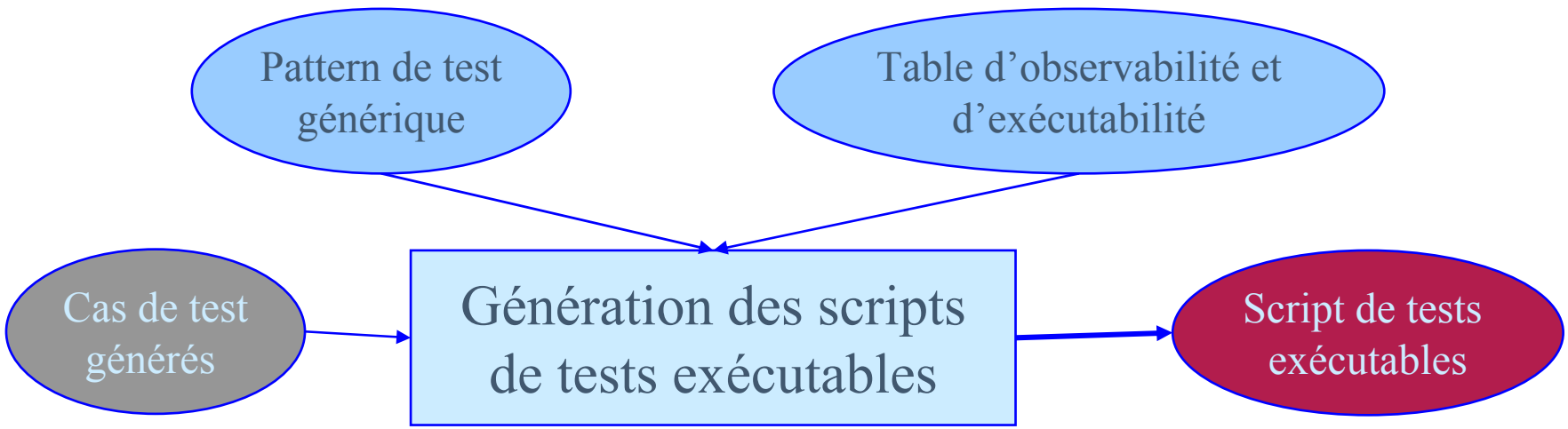
Environnement
d'exécution des tests



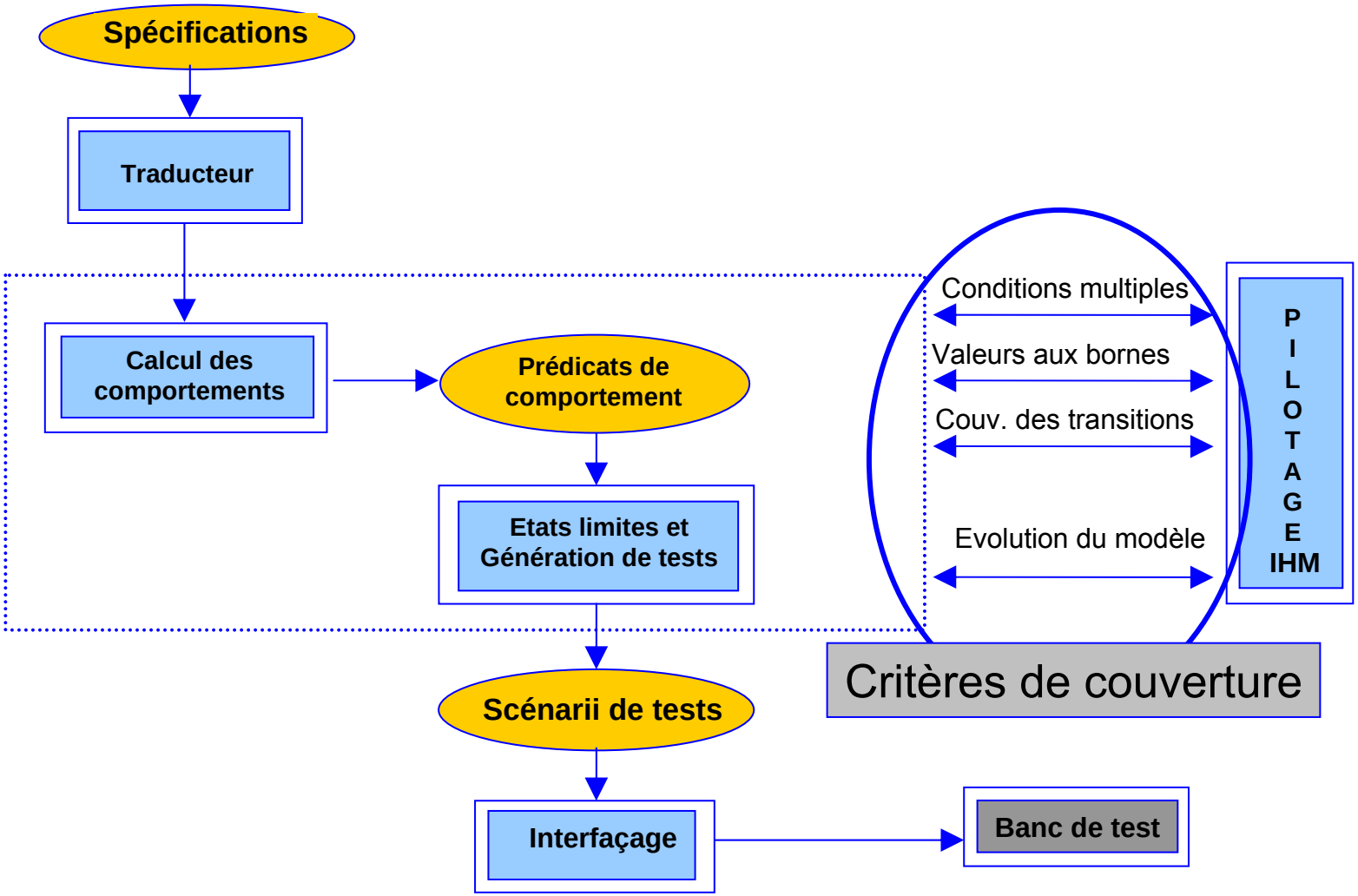
Génération des
scripts de test

Pilotage du banc de test et simulation de l'environnement

- A partir des cas de tests abstraits, d'un source de test générique et d'une table d'observabilité et d'exécutabilité, les **scripts exécutables sur le banc cible** sont générés.
- Le pilotage du banc permet une **exécution des tests et un verdict automatiques.**

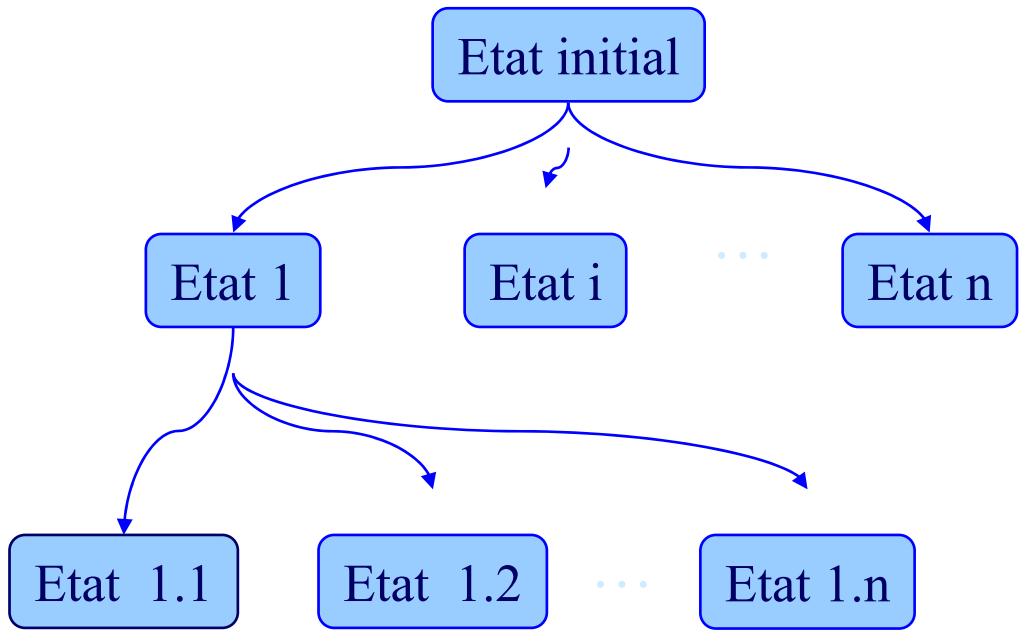


La génération de tests : un processus piloté par l'ingénieur validation



Au cœur de la technologie LEIRIOS Test Generator : moteur d'animation symbolique des spécifications

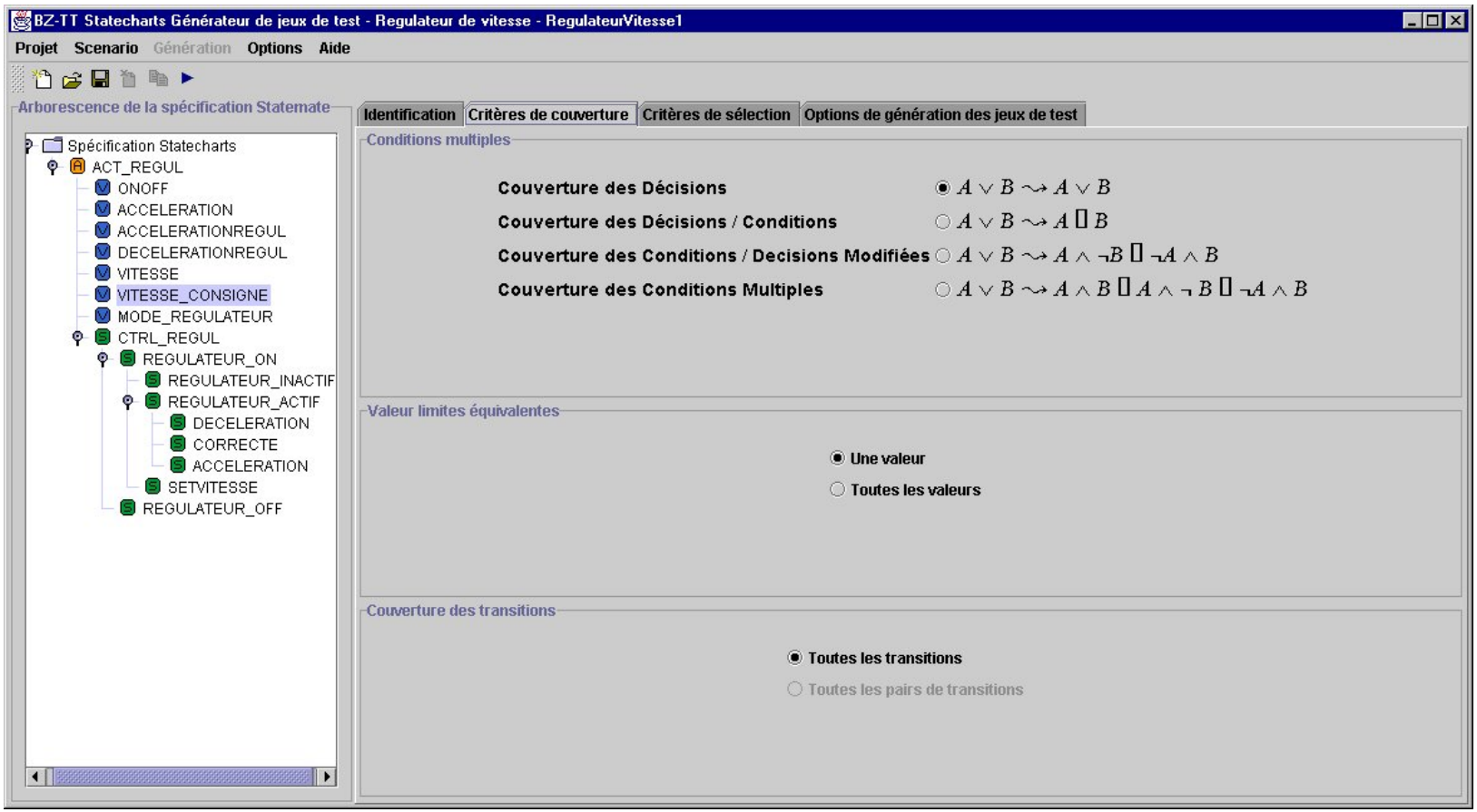
- Fondée sur des résultats originaux en résolution de contraintes
- Permet de calculer le système de contraintes suivant l'activation d'un comportement du système



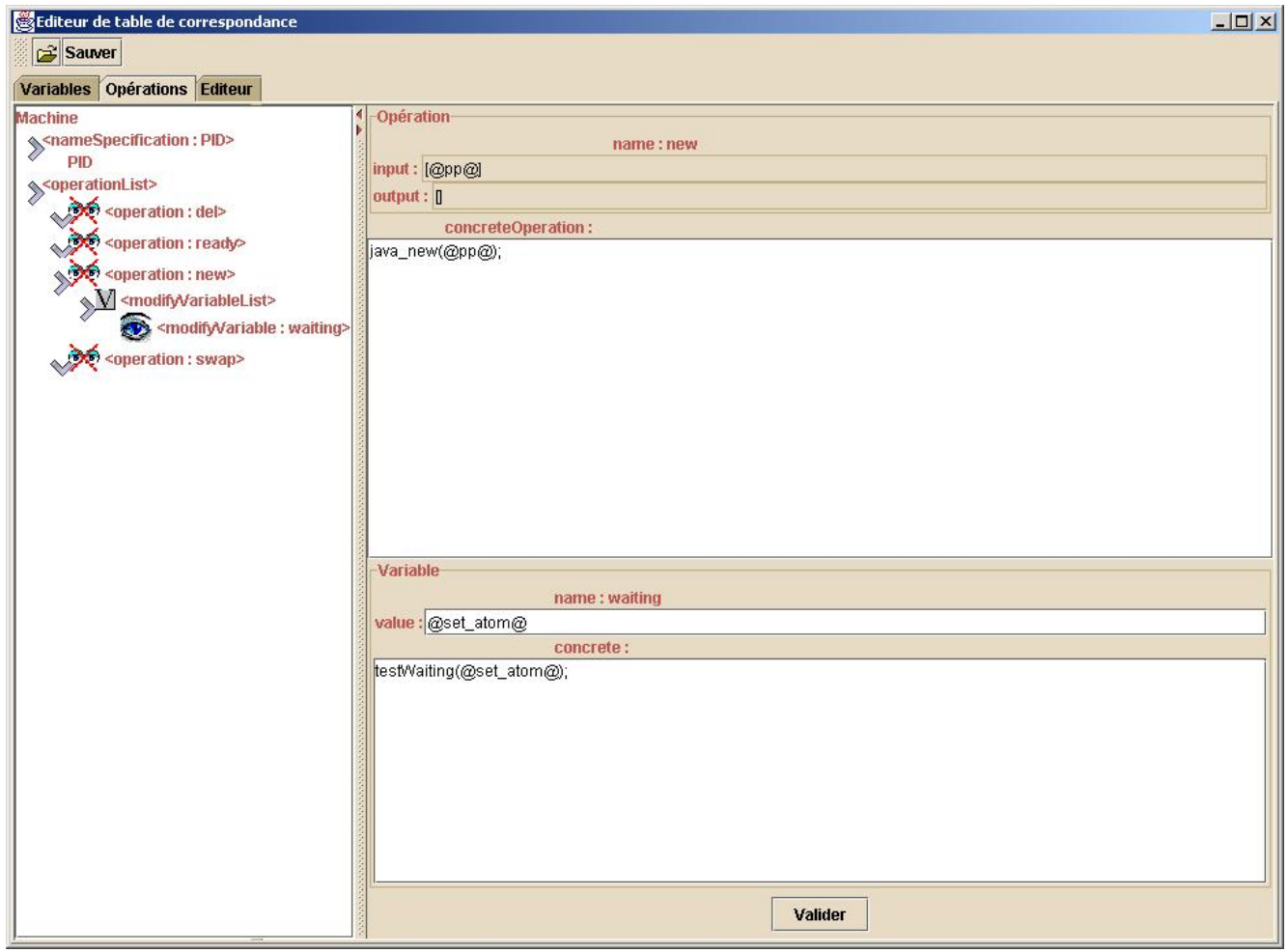
⇒ Maîtrise de l'explosion combinatoire du calcul des cas de test

⇒ Optimise le nombre de cas de test / couverture du modèle

LEIRIOS Test Generator – Interface de pilotage



LEIRIOS Test Generator – Liaison avec le banc



LEIRIOS TEST GENERATOR

Exemple d'applications

- GSM 11-11 – gestion puce SIM
- Java Card – JCVM mécanisme de transaction
- Gestion des clés, des identités et de la sécurité

Groupement des Cartes Bancaires

- Authentification du porteur

- Fonctions visibilité générique – essuyage, dégivrage, lavage
- Contrôleur de climatisation

- Algorithme de validation tickets Metro/RER
- Validation terminal de paiement (CB 5.2)

2-4 - Génération automatique de tests à partir de spécifications

- Test à partir de modèles (Model-Based Testing)
- Modéliser pour tester
- Sélection des tests – Critères de couverture
- Stratégies de génération
- LEIRIOS Test Generator
- Etudes de cas
  - Le régulateur de vitesse
  - La norme carte à puces GSM 11-11
  - Le distributeur de boissons

Diagramme d'activités du contrôleur de vitesse

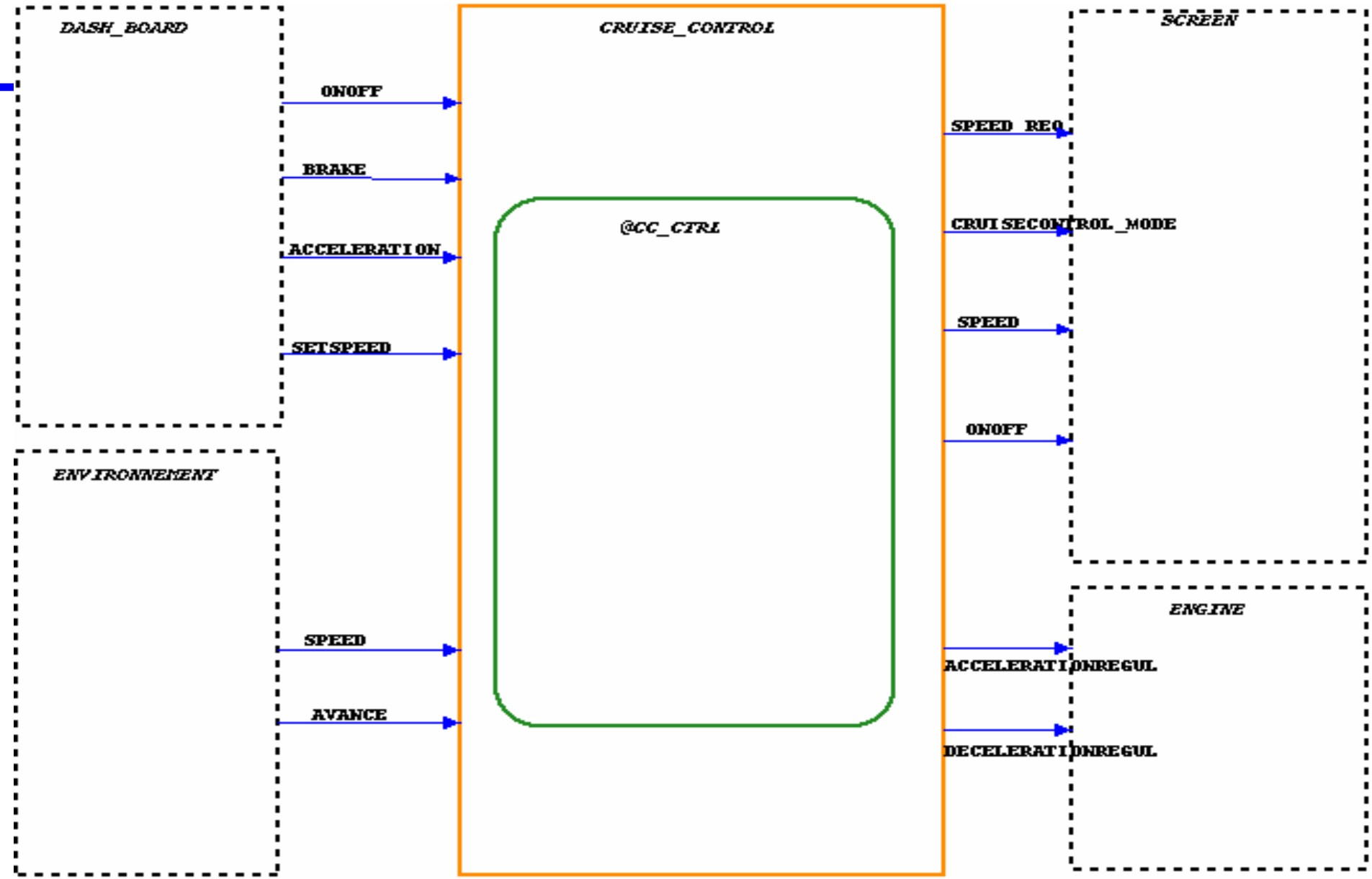
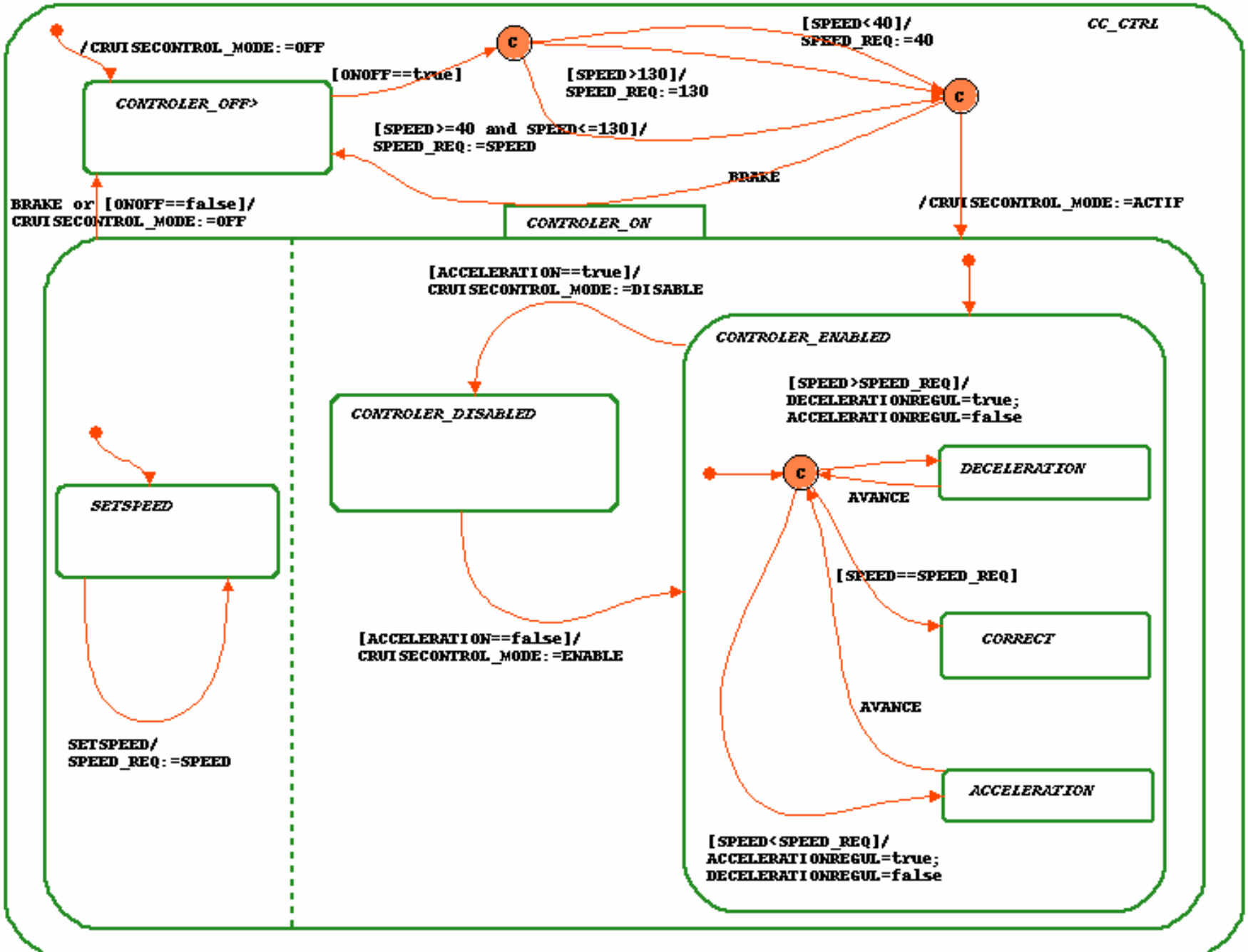


Diagramme d'états du contrôleur de vitesse



Contrôleur de vitesse

- Variables : 12

  Vitesse, régime moteur, état du contrôleur…

  - Observable : 6
  - Interne : 2
- Stimuli : 6
  - Information persistante : Vitesse, moteur…
  - Information non-persistante : activation/désactivation du contrôleur, freinage…
- Activité : 1
- Etat : 9
- Transitions : 18 hiérarchiques

Contrôleur de vitesse

- Réguler la vitesse : vitesse de consigne
- Action sur le contrôleur :
  - Activation / Désactivation
  - Freinage
  - Accélération
- Paramétrage :
  - Vitesse minimal : 40 km/h
  - Vitesse maximal : 130 km/h

=> 36 cas de test

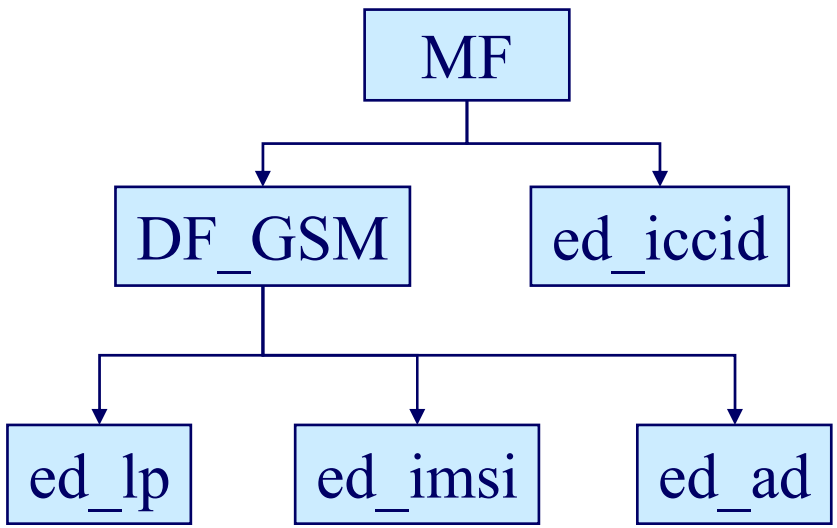
Exemple de cas de test généré

Stimuli	Explication	Variable observable
ON/OFF	Le conducteur active le contrôleur de vitesse	SPEED_REQ = 40 km/h CRUISECONTROL_MODE = ON SPEED = 0 km/h ON/OFF = ON ACCELERATIONREGUL = ACTIVE DECELERATIONREGUL=UNACTIVE
Acceleration Regulator	Le contrôleur fait accélère la voiture jusqu'à la vitesse demandé de 40 km/h	SPEED_REQ = 40 km/h CRUISECONTROL_MODE = ON SPEED = 40 km/h ON/OFF = ON ACCELERATIONREGUL = UNACTIVE DECELERATIONREGUL = UNACTIVE

Génération de tests sur l'application GSM 11.11

- Commandes :
  - Verify_Chv(chv,code),
  - Change_Chv(chv,code1,code2),
  - Disable_Chv(code),
  - Enable_Chv(code),
  - Unblock_Chv(chv,code_unblock 1, code_unblock2),
  - Select_File(ff),
  - Read_Binary,
  - Update_Binary(data),
  - Reset,

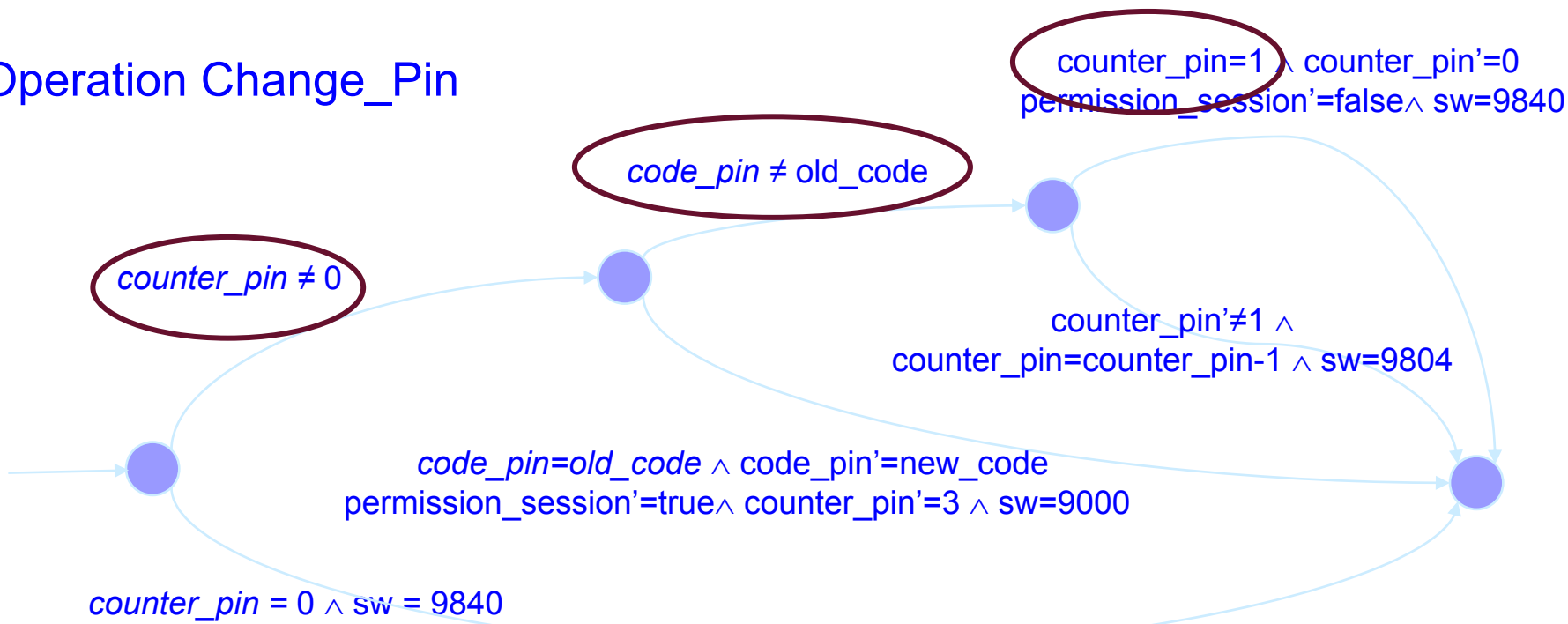
- Arborescence de fichiers



Méthode de génération

Test de chaque comportement : chaque chemin d'exécution pour chaque opération

Operation Change_Pin



Prédicat de comportement == *counter_pin* ≠ 0 ∧ *code_pin* ≠ old_code ∧ counter_pin=1

Valeurs limites pour la spécification GSM 11.11

- Couverture des comportements et des états limites - LTG

1. Extraction des comportements
2. Calcul des états limites
3. Construction des cas de tests
   - Préambule
   - Corps de test
   - Identification
   - Postambule

- Exemples de valeurs limites :
  - counter_chv(chv1) = 0
  - counter_chv(chv1) = 1
  - counter_chv(chv1) = 3
  - counter_unblock(chv1) = 0
  - counter_unblock(chv1) = 1
  - counter_unblock(chv1) = 10
  - enabled_chv1_status = enabled
  - enabled_chv1_status = disabled

Activation d'un comportement de la commande VERIFY_CHV

```
sw <--  VERIFY_CHV(code) =
        PRE
            code : CODE
        THEN
            IF (blocked_chv_status = blocked)
            THEN
                sw := 9840
            ELSE
                IF (pin = code)
                THEN
                    counter_chv := MAX_CHV || permission_session(chv) := true || sw := 9000
                ELSE
                    IF (counter_chv = 1)
                    THEN
                        counter_chv := 0 || blocked_chv_status := blocked ||
                        permission_session(chv) := false || sw := 9840
                    ELSE
                        counter_chv := counter_chv - 1 || sw := 9804
                    END
                END
            END

        END;
```

Séquences de test pour la spécification GSM 11.11

- Séquences de test de l ’état initial counter_chv(chv1) = 3 à la valeur limite counter_chv(chv1) = 0

- Préambule :

1. VERIFY_CHV (chv1) false code
2. VERIFY_CHV (chv1) false code
3. VERIFY_CHV (chv1) false code

Exemples de test à la valeur limite counter_chv(chv1) = 0 (1)

- Test 1 : VERIFY_CHV(1) true code >>> 9840
  - STATUS :

current_file	[]
enabled_chv1-status	enabled
counter_chv	[(chv1,0),(chv2,3)]
counter_unblock	[(chv1,10),(chv2,10)]

- Test 2 : VERIFY_CHV(0) wrong code >>> 9840
  - STATUS :

current_file	[]
enabled_chv1-status	enabled
counter_chv	[(chv1,0),(chv2,3)]
counter_unblock	[(chv1,10),(chv2,10)]

Exemples de test à la valeur limite counter_chv(chv1) = 0 (2)

- Test 3 : UNBLOCK_CHV(chv1,1,0) true code >>> 9000
  - STATUS :

current_file	[]
enabled_chv1-status	enabled
counter_chv	[(chv1,3),(chv2,3)]
counter_unblock	[(chv1,10),(chv2,10)]

- Test 4 : UNBLOCK_CHV(chv1,0,0) wrong code >>> 9804
  - STATUS :

current_file	[]
enabled_chv1-status	enabled
counter_chv	[(chv1,0),(chv2,3)]
counter_unblock	[(chv1,9),(chv2,10)]

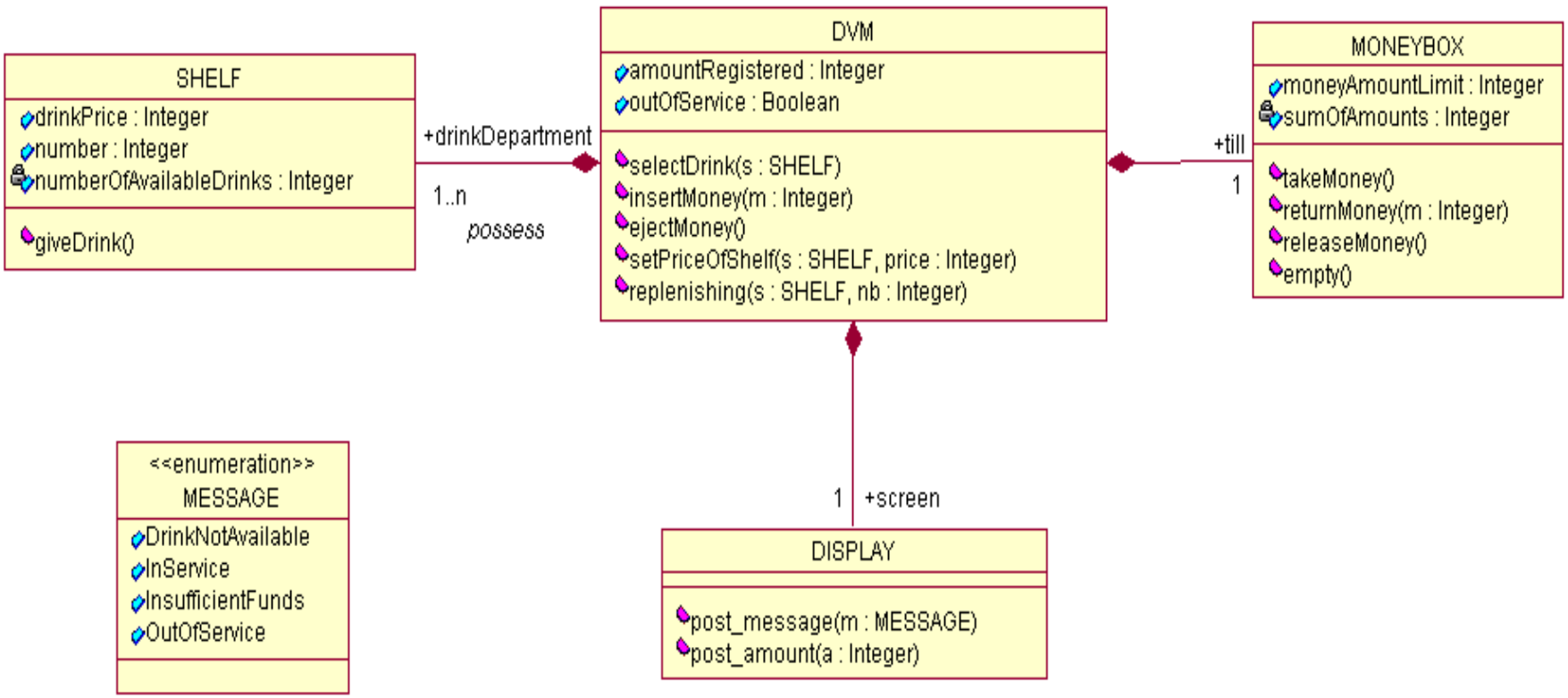
Démonstration de l'outil LEIRIOS Test Generator

Distributeur de boissons – Modèle UML

- Les tests générés correspondent à des séquences d'appel de méthode définies au niveau du diagramme de classes du modèle

- La stratégie de génération explore les chemins d'activation du diagramme états / transitions suivant les paramètres de génération choisis (dépliage des conditions multiples, valeurs des domaines)
- Les cas de test peuvent être visualisé sous la forme de diagramme de séquence

Distributeur de boisson – Diagramme de classes



Distributeur de boisson – Exemple de cas de test

1. InsertMoney(3)	AmountRegistered = 3
2. SelectDrink(s1)	
→ giveDrink	NumberOfAvailableDrink – 1
→ returnMoney(1)	AmountRegistered = 0 SumOfAmounts = SumOfAmounts + AmountRegistered

- Le test est vu depuis la classe DVM qui constitue l'objectif de test

2-4 – Automatisation de l'exécution des tests et de la remontée du verdict

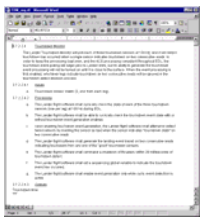
- Problématique
- Traduire des tests abstraits en scripts exécutables
- Environnement d'automatisation de l'exécution
- Exemple du « Contrôleur de vitesse »

Process de génération de tests

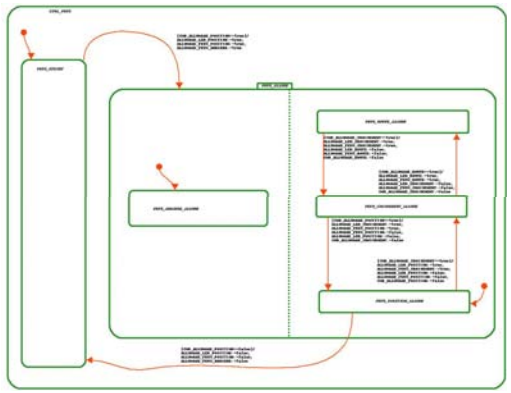
Outils de modélisation (Statecharts, B ou UML)

Validation Du modèle

Modélisation fonctionnelle



Spécifications fonctionnelles



LTG Animateur des spécifications

LTG Générateur de tests

Modèle des spécifications

Cas de test

LTG Générateur de scripts

Banc de test

Pattern de test et table de correspondance

Génération des scripts et exécution de tests

Problématique de la traduction des tests générés en scripts exécutables

- Nomage des appels et variables : Les tests générés sont des séquences d'appel sur les opérations ou stimuli définis au niveau du modèle
  - ➔ Table de correspondance entre noms du modèle (variables et appels) et noms de l'environnement d'exécution des tests
- langage de tests : L'environnement d'exécution des tests prend en entrée un langage de définition des scripts exécutables
  - ➔ Définition d'un pattern générique au sein duquel les appels seront insérés
- Verdict de test : Les tests générés comprennent le résultat attendus
  - ➔ Le verdict est construit par comparaison entre le résultat attendu et le résultat obtenu pendant l'exécution du script de test

Processus de génération de scripts de test

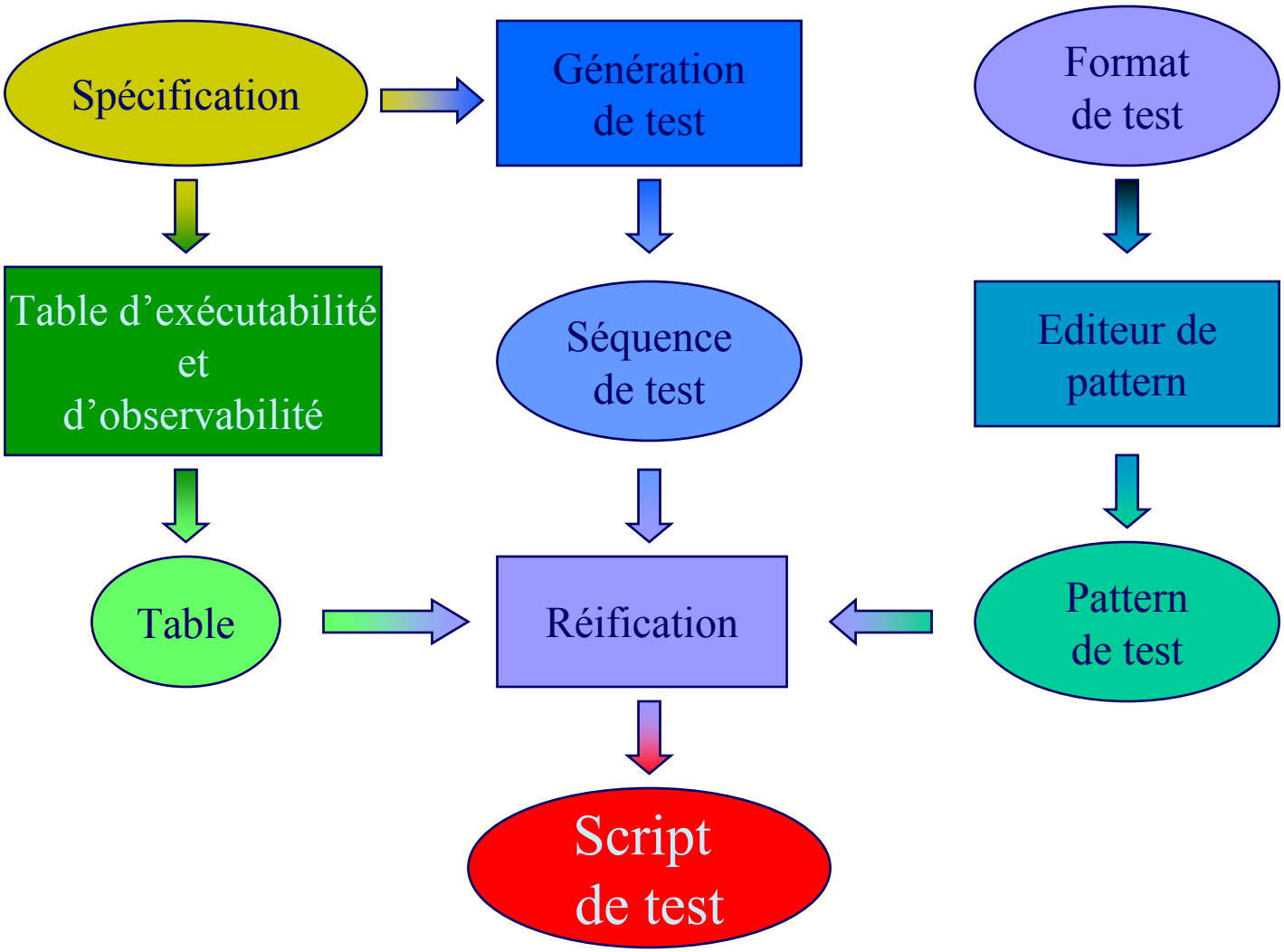


Table d'exécutabilité et d'observabilité

- Relation entre Abstrait et Concret :
  - Stimuli
  - Variables : Interne, Entrée, Sortie
- Observabilité :
  - Définir les éléments spécifiques au modèle
  - Les éléments identifiables
- Exécutabilité :
  - Appel concret
  - Reconstruire les éléments (@)
  - Gestion spécifique au langage destination

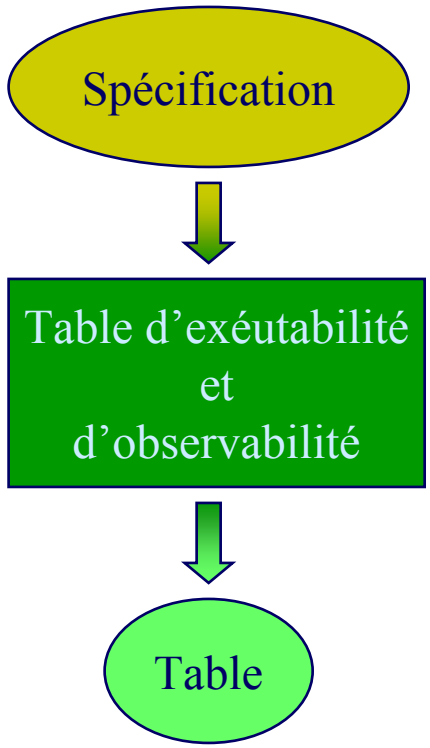
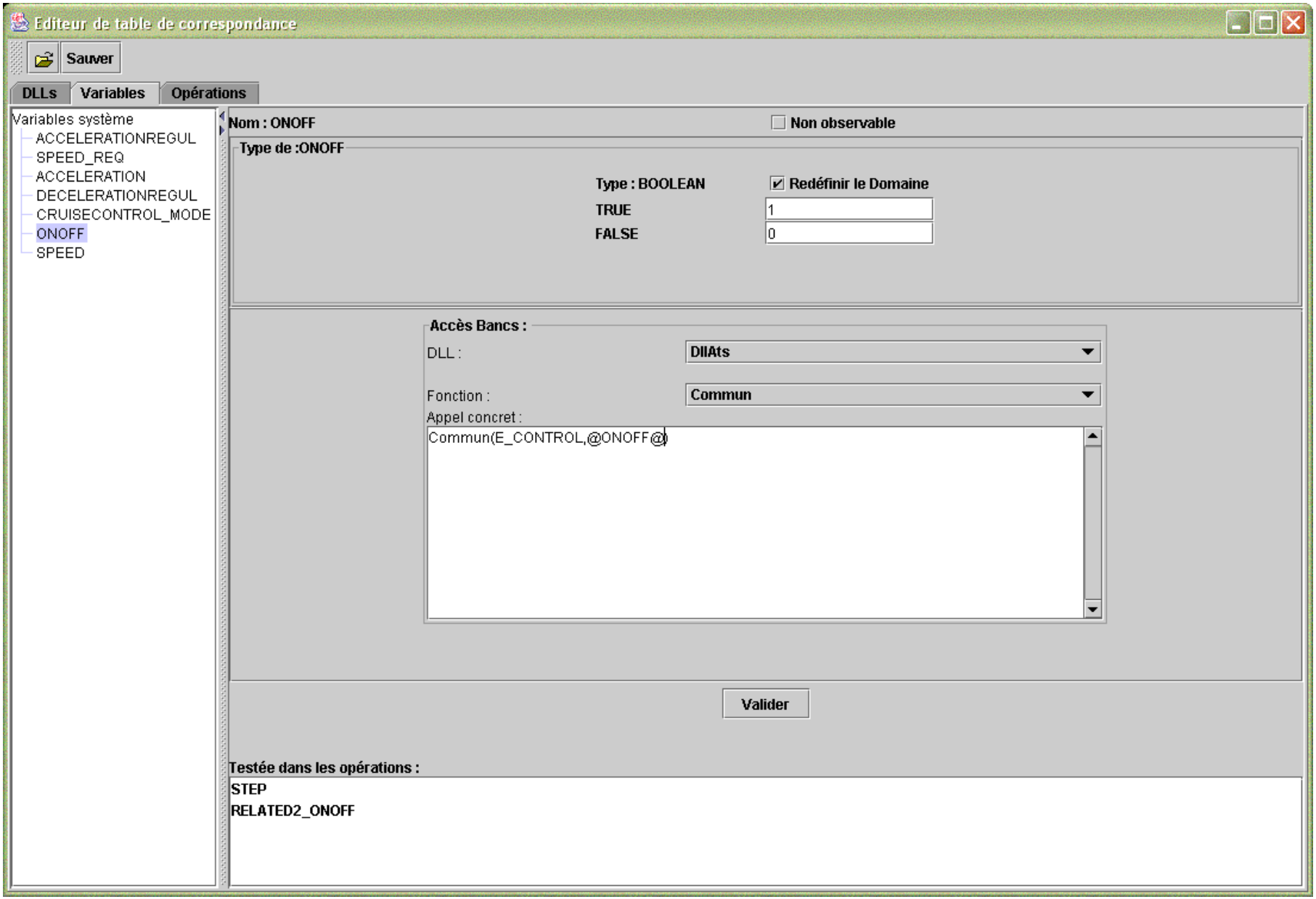
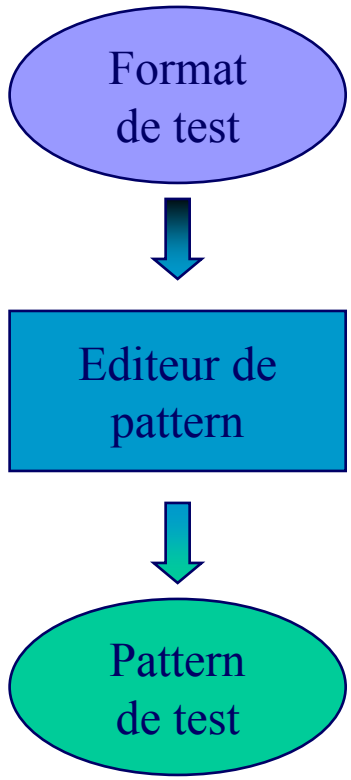


Table d'exécutabilité et d'observabilité

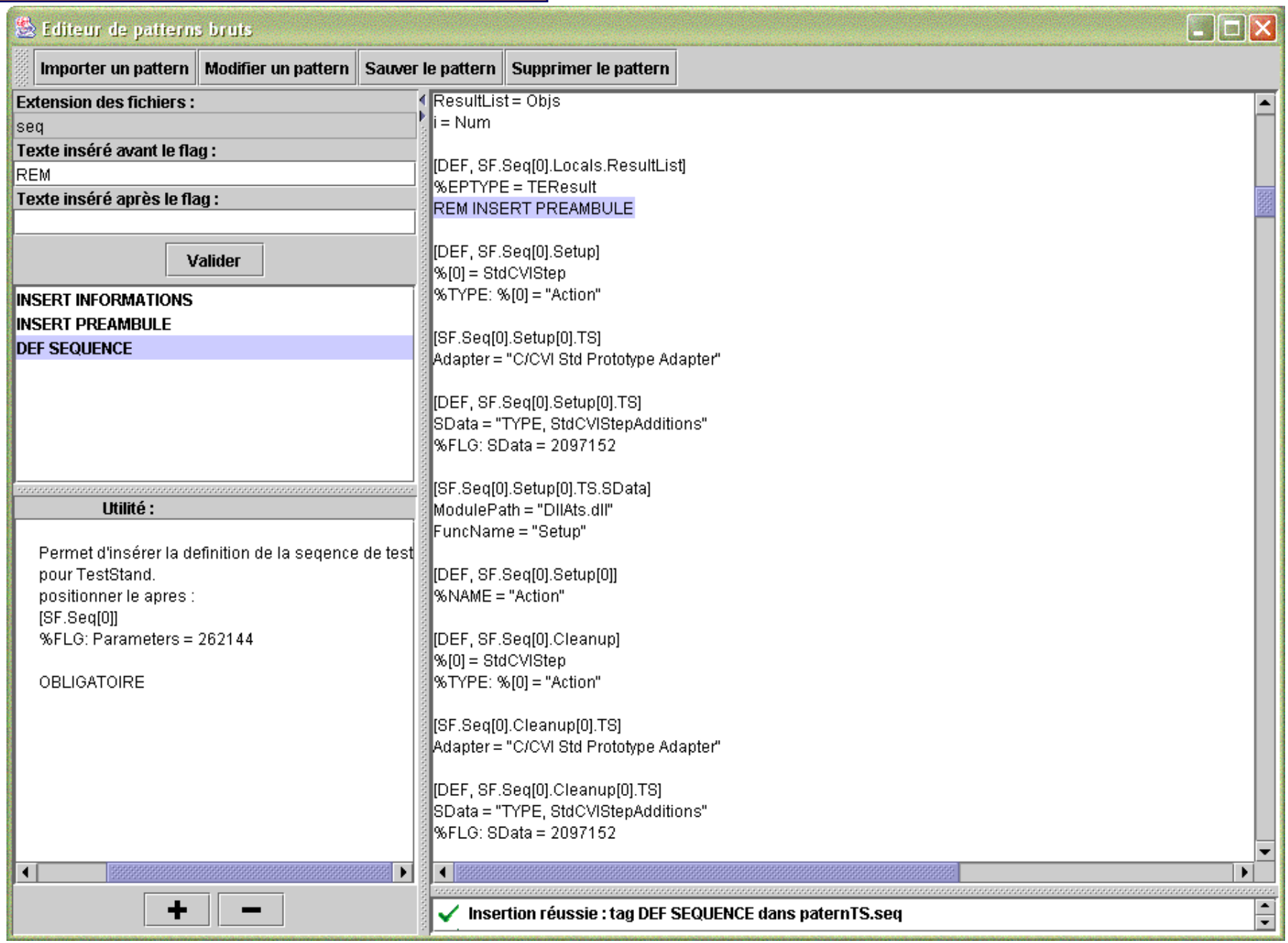


Editeur de patterns

- Automatisation de la génération :
  - Squelette générique : source à trous
  - Tag pour l'insertion des parties du test
- Tag :
  - Information sur le test
  - Préambule
  - Postambule

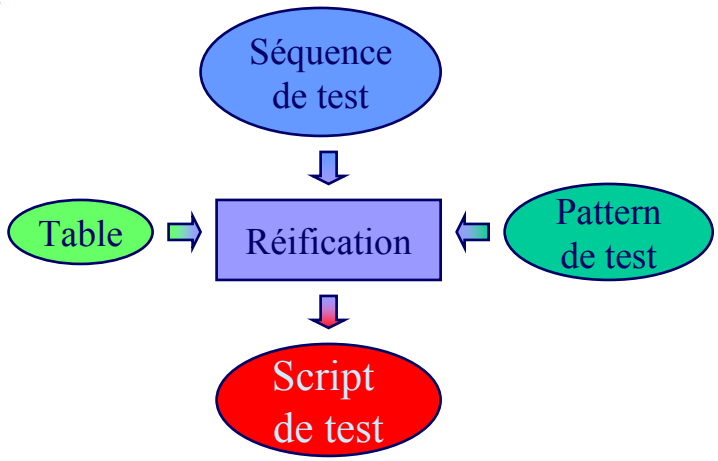


Editeur de patterns



Génération des scripts de test exécutables

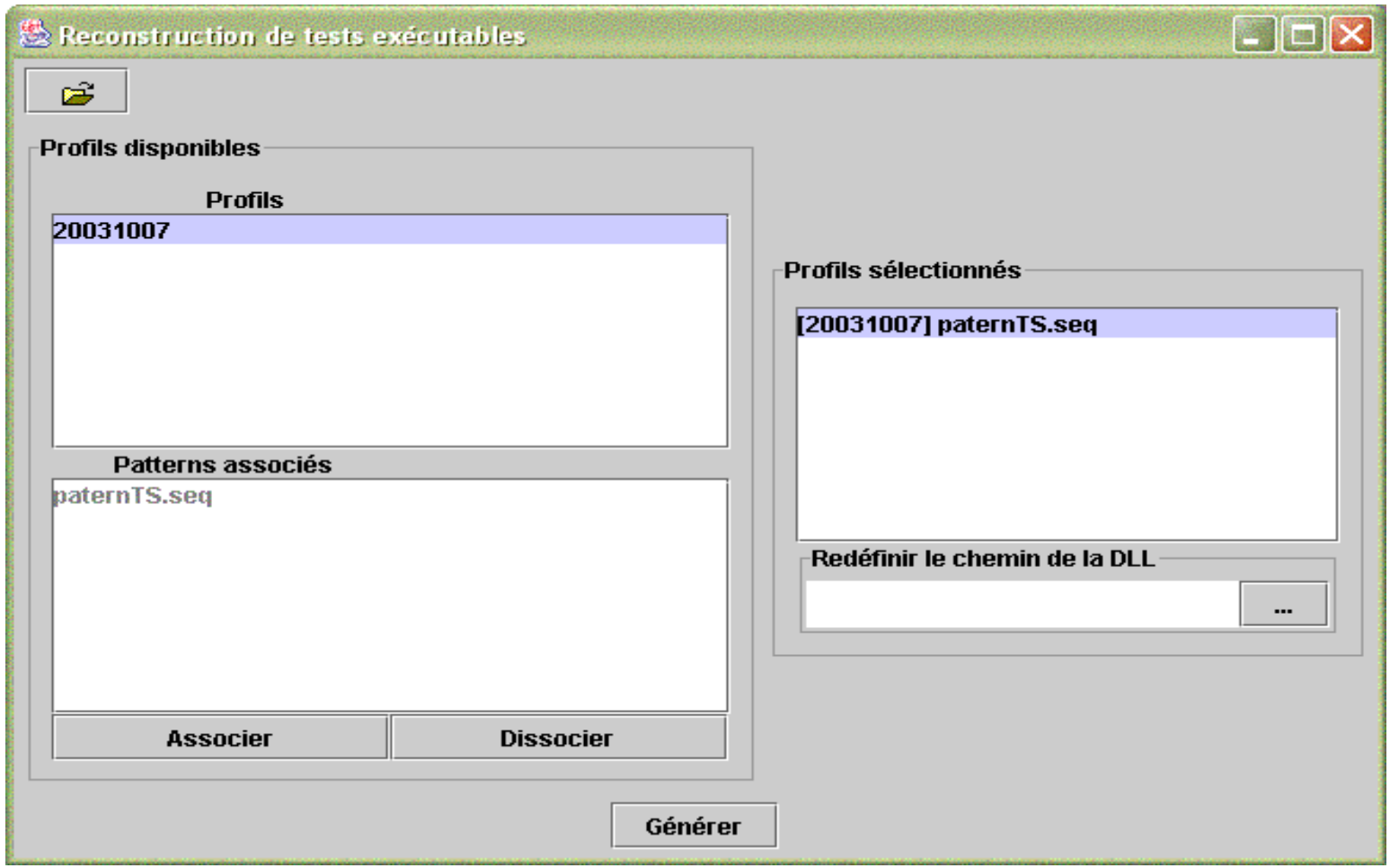
- Générer automatique les scripts à partir :
  - Séquence de test
  - Table d'exécutabilité et observabilité
  - Pattern de test



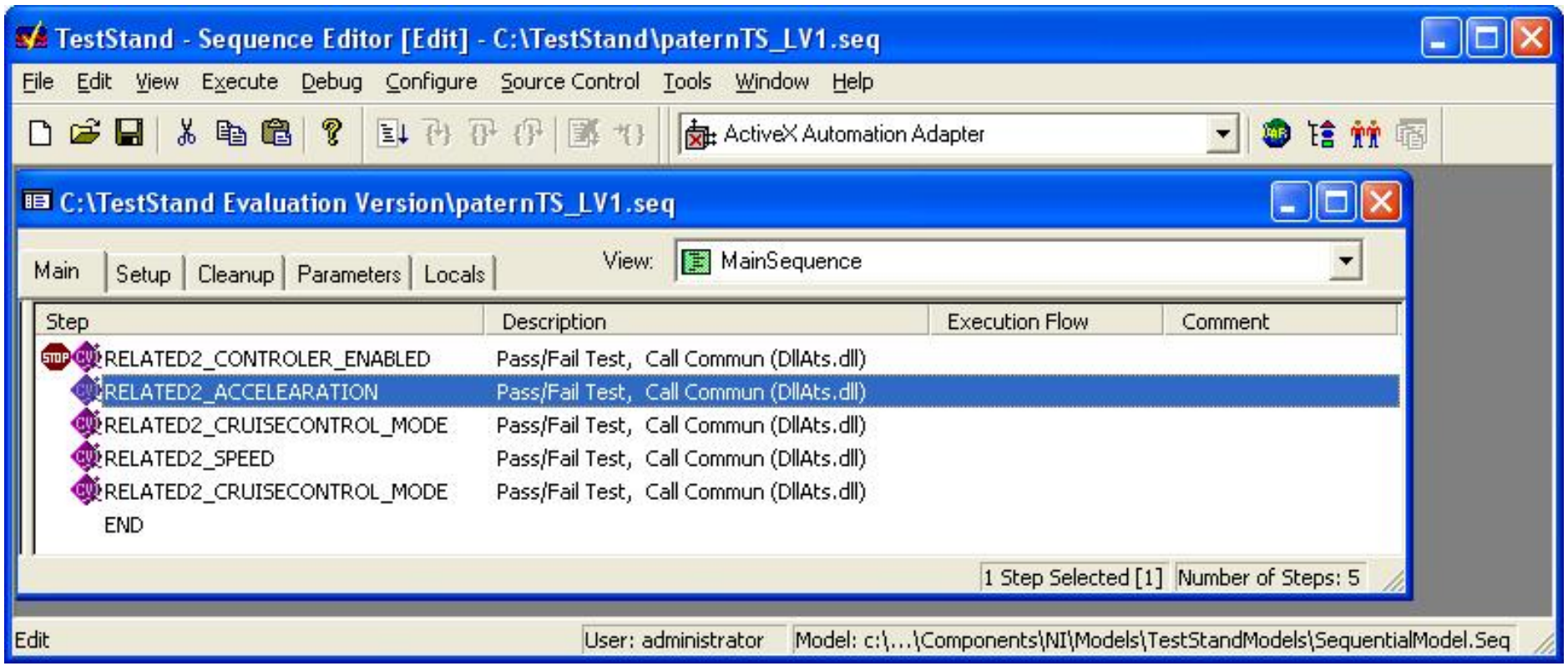
- Optimisation sur les scripts :
  - Regrouper des séquences dans le même script
  - Activation d'observation en dehors de l'identification
  - Associer une séquence à différents patterns

=> Source à utiliser dans les environnement

Interface de réification



Exécution sur le banc de test



Problèmes

- Interprétation du cahier des charges :
  - Non dit
  - Optimisation
  - Ordre des traitements

  => Expérimentation
- Niveau d'observation (qui, où, quoi)
  - Information directe : variable ou méthode
  - Reconstructible : calcul ou appel
  - Non-observable

  => Réalisation du modèle

2-4 - Caractéristiques des outils de génération de tests à partir de modèles

1. Nature du modèle
2. Pilotage de la génération
   - Spécifications de tests
   - Critères de couvertures
3. Techniques de calcul pour la génération
4. Contrôle de l'exécution des tests générés

1- Nature du modèle

- Spécification formelle du système sous test
  - Comportements attendus du système dans son environnement
    - » Ex.: LTG
- Modèle du système + Modèle de l'environnement
  - Le modèle d'environnement permet de restreindre l'espace de calcul
    - » Ex. : GATEL (CEA)
- Modèle des exécutions attendues (Observateur)
  - Définition des traces à tester
    - » Ex. : Conformiq

2- Notation de modélisation

- Systèmes de transitions :
  - IOLTS (STG – IRISA, TORX – Univ. Nijmegen)
  - Statecharts (AGATHA – CEA)
- Pré/post conditions – Machine abstraites
  - ASML (Microsoft Research)
  - B (LTG)
  - UML/OCL (LTG, UML-Casting, …)
- Langages synchrones
  - Lustre (GATEL – CEA)

3- Pilotage de la génération (1)

1. Spécifications de tests
   - Même notation que le modèle
     - Propriétés LUSTRE dans l'outil GATEL
     - IOLTS pour définir l'objectif de test dans STG
   - Notation différentes
     - FlowGraph pour spécifier l'objectif de test en Z (P. Strooper – Univ. Queensland Australie)
     - Propriétés PLTL pour spécifier l'objectif de tests

3- Pilotage de la génération (2)

2. Critères de couverture
   - Couverture structurelle du modèle
     - Flot de contrôle
       - Couverture des états, des transitions ou paires de transitions (LTS, Statecharts)
       - Couverture d'un partitionnement des post-conditions, DNF (B, Z, VDM, OCL)
       - Couverture des règles (ASM)
       - Couverture des décisions multiples : DC, D/CC, FPC, MC/DC, MCC
     - Flot de données
       - All-Definitions, All-Uses, All-Definitions-Uses
   - Couvertures des données (variables d'entrée, d'états)
     - Données aux bornes (boundary testing)
     - Choix Random, stochastiques
   - Couverture des exigences (traçabilité)

4- Techniques de calcul pour la génération

- Model-Checking
  - Techniques de parcours du modèle éprouvée
  - La négation de l'objectif de tests est vue comme une propriété à démontrer (un contre-exemple ≈ cas de test)
  - Problème d'explosion combinatoire si méthode valuée
    - » Model-Checking symbolique
- Interprétation abstraite
  - Propagation des équations sur le modèle à partir d'entrée symbolique
  - Nécéssite de pouvoir instancier des cas de test concrets
    - » Résolution de contraintes
- Programmation en Logique avec Contraintes
  - Calcul des conditions de chemin par résolution de contraintes
  - Le système de contraintes courant représente un ensemble d'état possible du système

5- Contrôle de l'exécution des tests générés

- Calcul et passage des tests à la volée
  - Ex. STG : calcul d'un graphe complet de test à partir du modèle et de la spécification de test ; ce graphe est exploité dynamiquement pour l'exécution des tests
  - Avantages : bonne gestion du non déterminisme (choix à la volée en fonction des réponses du système sous test)
  - Inconvénients : Problème du temps de calcul des tests à la volée
- Traduction des tests abstraits en scripts exécutables et exécution en mode batch
  - Ex. LTG – Réification des tests générés en scripts exécutables
  - Avantages : S'inscrit dans le processus de test (tests de non régression)
  - Inconvénients : Prise en compte du non-déterminisme de contrôle plus difficile

Apports et limites de la génération automatique de tests

- Couverture fonctionnelle des spécifications
- Cohérence des spécifications
- Maturité du process de test

Apports de la génération automatique – Couverture du modèle

- Couverture systématique des spécifications suivant
  - tous les comportements
  - Traçabilité entre les exigences et les tests

  ⇒ Maîtrise de la couverture de tests

- Génération du nombre minimal de données de tests
  - En fonction des critères de couverture sélectionnés

  ⇒ Maîtrise de l'explosion combinatoire du nombre de cas de tests

Apports de la génération automatique – Cohérence des spécifications

- Le modèle est
  - validé par animation
  - vérifié par preuve de consistance (les comportements sont cohérents)

  ⇒ Validation des spécifications

- En cas d'évolutions fonctionnelles
  - génération des tests correspondant aux nouveaux comportements
  - tests des comportements inchangés

  ⇒ Automatise le test des changements fonctionnels et des effets de bords (test de non-regression)

Apports de la génération automatique – Maturité du process de validation

- L'automatisation de la génération des scripts de tests, permet à l'ingénieur validation de se consacrer à l'essentiel
  - le pilotage de la génération (objectif de tests)
  - l'analyse des anomalies détectées

  ⇒ Meilleure maturité du process de validation
- Une fois la chaîne automatisée
  - reproductibilité de la génération
  - suppression de l'écriture des scripts

  ⇒ Réduction de l'effort de test et du « time to market »

Réduction du temps de test - Source K. Stobie

Test Architect - Microsoft Corporation – SASQAG – Janvier 2003

"In a typical application of this approach, test engineer **productivity has increased by a factor of five to ten** over conventional manual approaches".

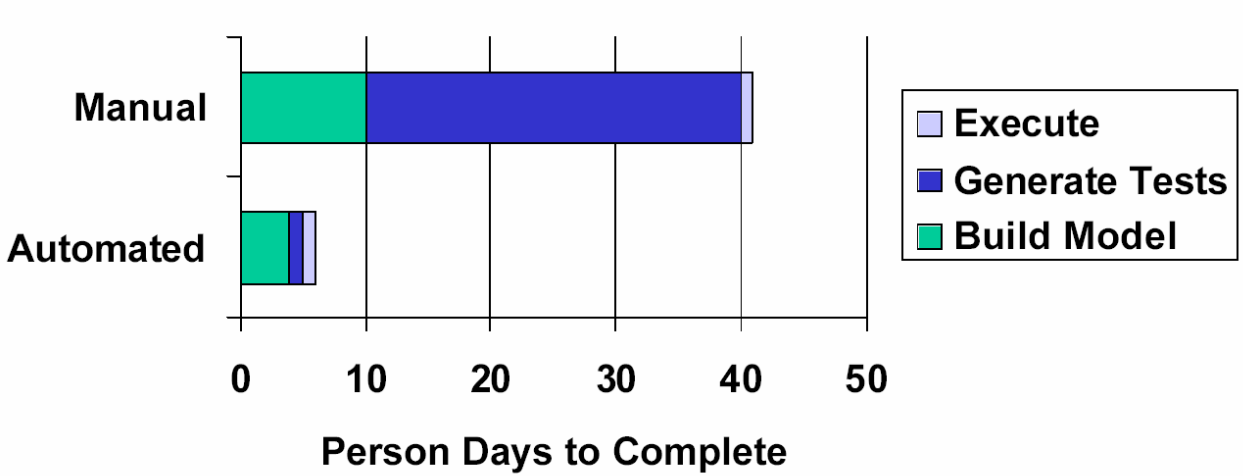


Figure 5. Tests for Call Waiting Feature was completed in 12% of the time

SASQAG : Seattle Area Software Quality Assurance Group

Exemples de résultats obtenus avec LTG

- Sur l'application GSM 11-11: comparaison par rapport à une suite manuelle très mature
  - 50% de couverture fonctionnelle en plus
  - Réduction de 30% de l'effort de conception des tests
- Sur l'application Validation Terminal de Paiement sur Automate – Transaction de débit CB 5.2
  - Couverture 3 fois supérieure aux pratiques antérieures
  - Temps moyen de spécification de test très inférieur

Quand mettre en œuvre la génération automatique à partir des spécifications ?

Critères décisifs :

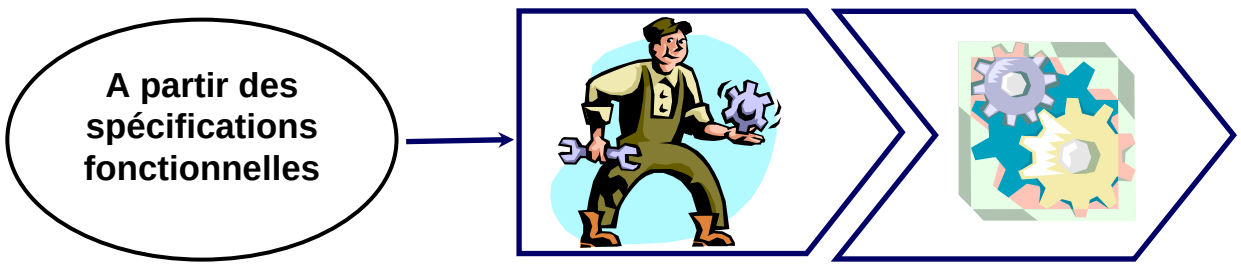
- Fort besoin en validation
- Croissance forte des fonctionnalités
- Besoin de réduire l'effort et les temps de test

Systèmes embarqués, logiciels enfouis

Control Systems
Consumer Electronics
Autonomous Systems
Automobile Industry
Transport
Smart cards
Telecommunication
Energy, Medical,
….

Automatiser le test pour mieux valider !

- L'automatisation du test fonctionnel s'appuie sur la mise en place d'une chaîne continue :



Modélisation

Génération de tests et de scripts

Environnement D'exécution des tests

3- Méthodes de test structurel

- Le test structurel s 'appuie sur l 'analyse du code source de l'application (ou d'un modèle de celui-ci) pour établir les tests en fonction de critères de couverture

⇨ Basés sur le graphe de flot de contrôle (toutes les instructions, toutes les branches, tous les chemins, …)

⇨ Basés sur la couverture du flot de données (toutes les définitions de variable, toutes les utilisations, …)

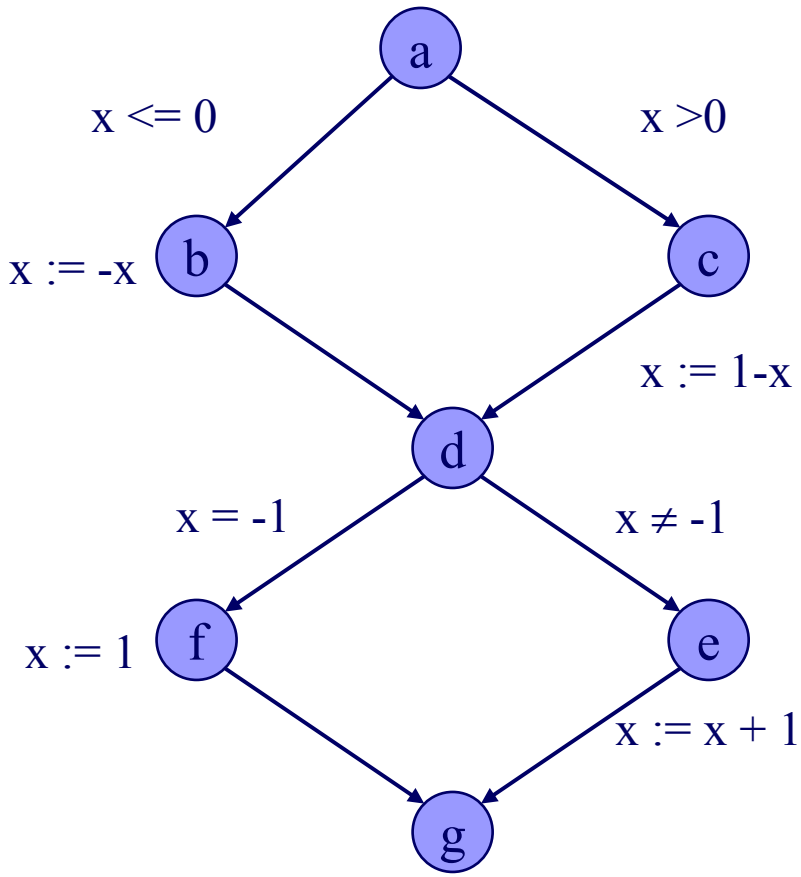
⇨ Basés sur les fautes (test par mutants)

3-1 Graphe de flot de contrôle

- Soit le programme P1 suivant :

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

Ce programme admet le graphe de contrôle G1.



G1

Chemins dans le graphe de contrôle

- Le graphe G1 est un graphe de contrôle qui admet une entrée - le nœud a - , une sortie - le nœud g.
  - le chemin *[a, c, d, e, g]* est un chemin de contrôle,
  - le chemin *[b, d, f, g]* n ’est pas un chemin de contrôle.

- Le graphe G1 comprend 4 chemins de contrôle :
  - β1 = *[a, b, d, f, g]*
  - β2 = *[a, b, d, e, g]*
  - β3 = *[a, c, d, f, g]*
  - β4 = *[a, c, d, e, g]*

Expression des chemins de contrôle

- Le graphe G1 peut-être exprimé sous forme algébrique sous la forme suivante :

  G1 = abdfg + abdeg + acdfg + acdeg

  le signe + désigne le « ou » logique entre chemins.

- Simplification de l ’expression de chemins

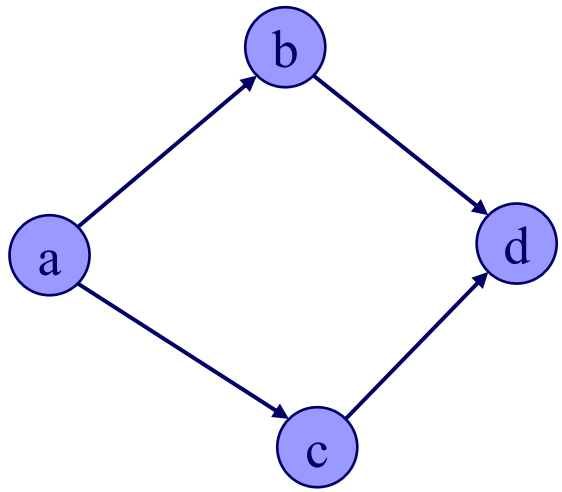
  G1 = a (bdf + bde + cdf + cde) g

  G1 = a (b + c) d (e + f) g

Calcul de l ’expression des chemins de contrôle

- On associe une opération d’addition ou de multiplication à toutes les structures primitives apparaissant dans le graphe de flot de contrôle

Forme séquentielle : ab



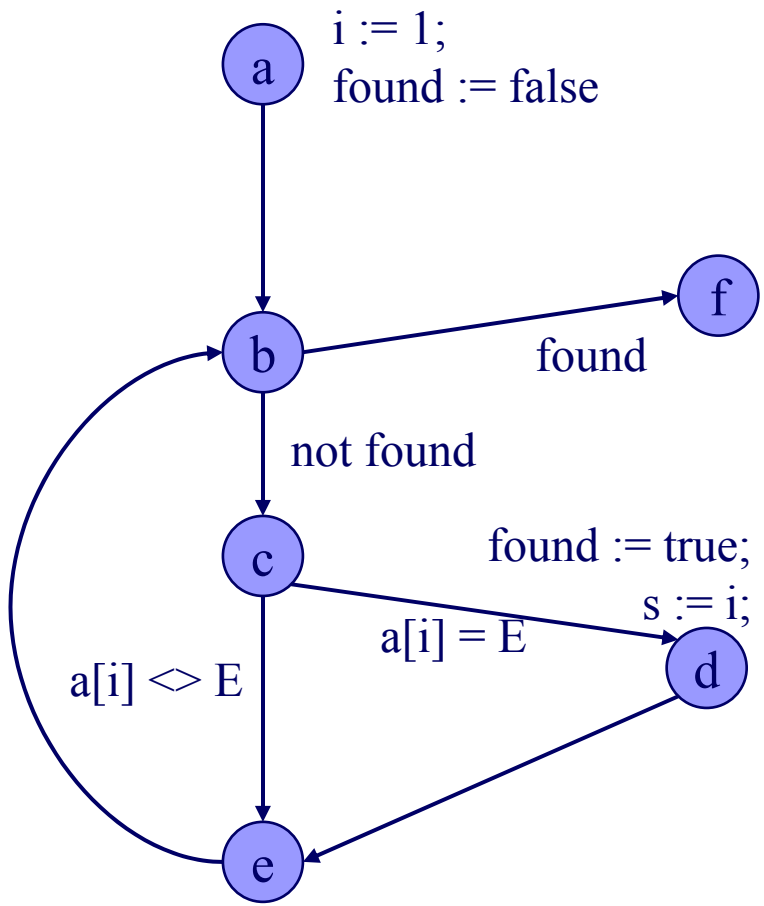
Forme alternative :
a (b + c) d

Forme itérative : ab (cb)* d

Chemins de contrôle avec boucles

- Soit le programme P2 suivant :

```
 i := 1;
found := false;
while (not found) do
      begin
      if (a[i] = E) then
         begin
         found := true;
         s := i;
         end;
      i := i + 1;
      end;
```



G2 = ab [c (1 + d) eb]* f

G2

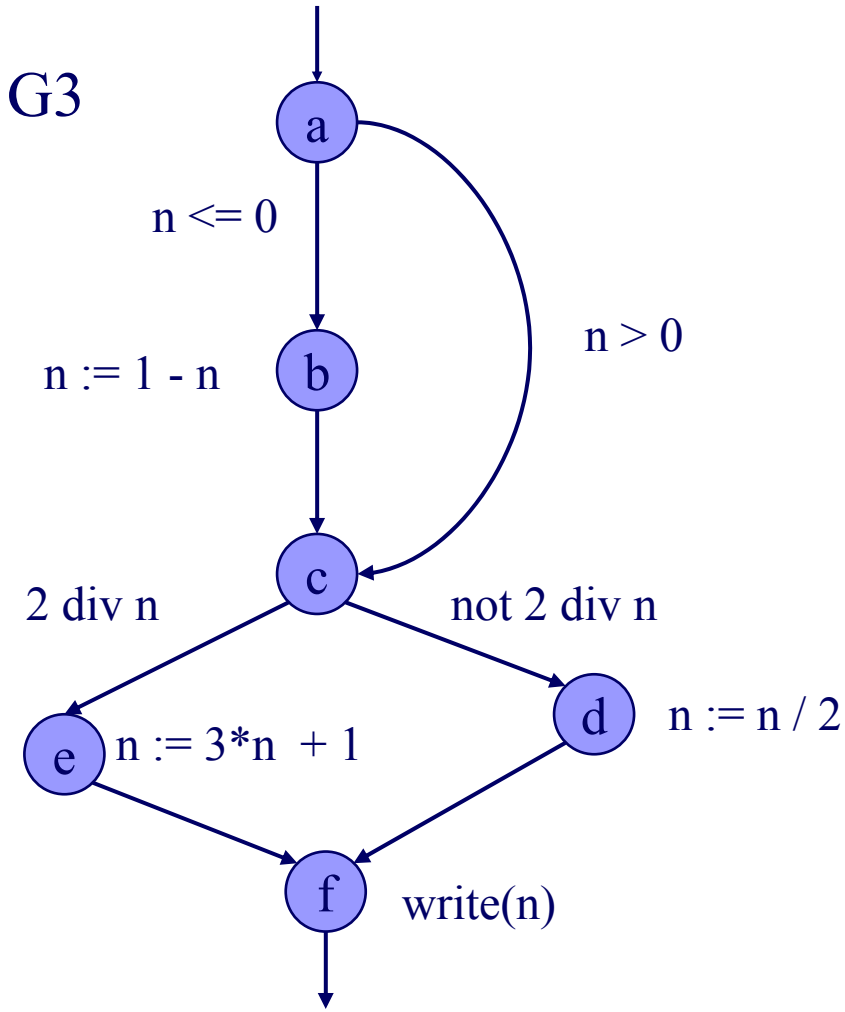
Expressions de chemins - Exercice

- Soit le programme P3 suivant :

```
if n <= 0 then n := 1-n
    end;
if 2 div n
       then n := n / 2
       else n := 3*n + 1
       end ;
write(n);
```

- Etablir le graphe de flot de contrôle de ce programme
- Fournir l ’expression des chemins

Graphe de flot de contrôle du programme P3



G3 = a (1 + b) c (e + d) f

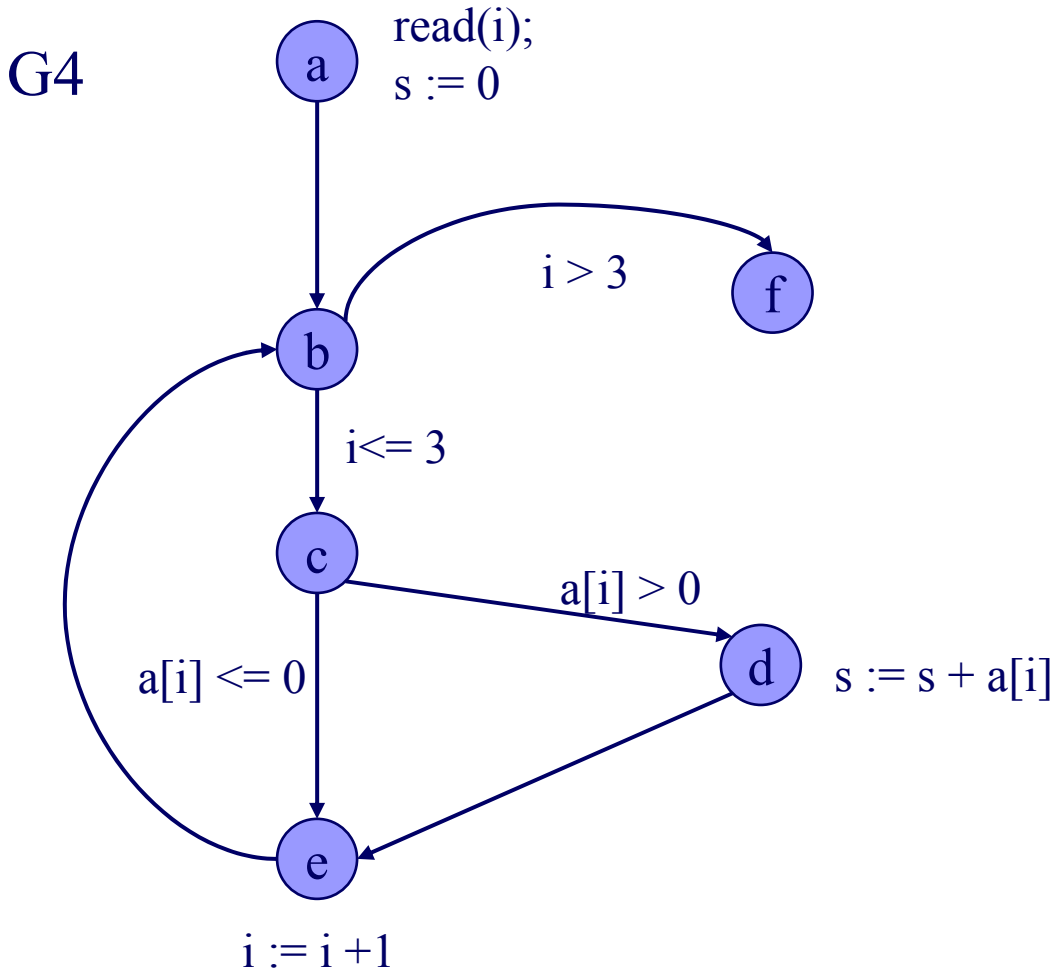
Expressions de chemins - Exercice

- Soit le programme P4 suivant :

```
read(i);
s := 0;
while (i <= 3) do
        begin
        if a[i] > 0 then s := s + a[i];
        i := i + 1;
        end
end;
```

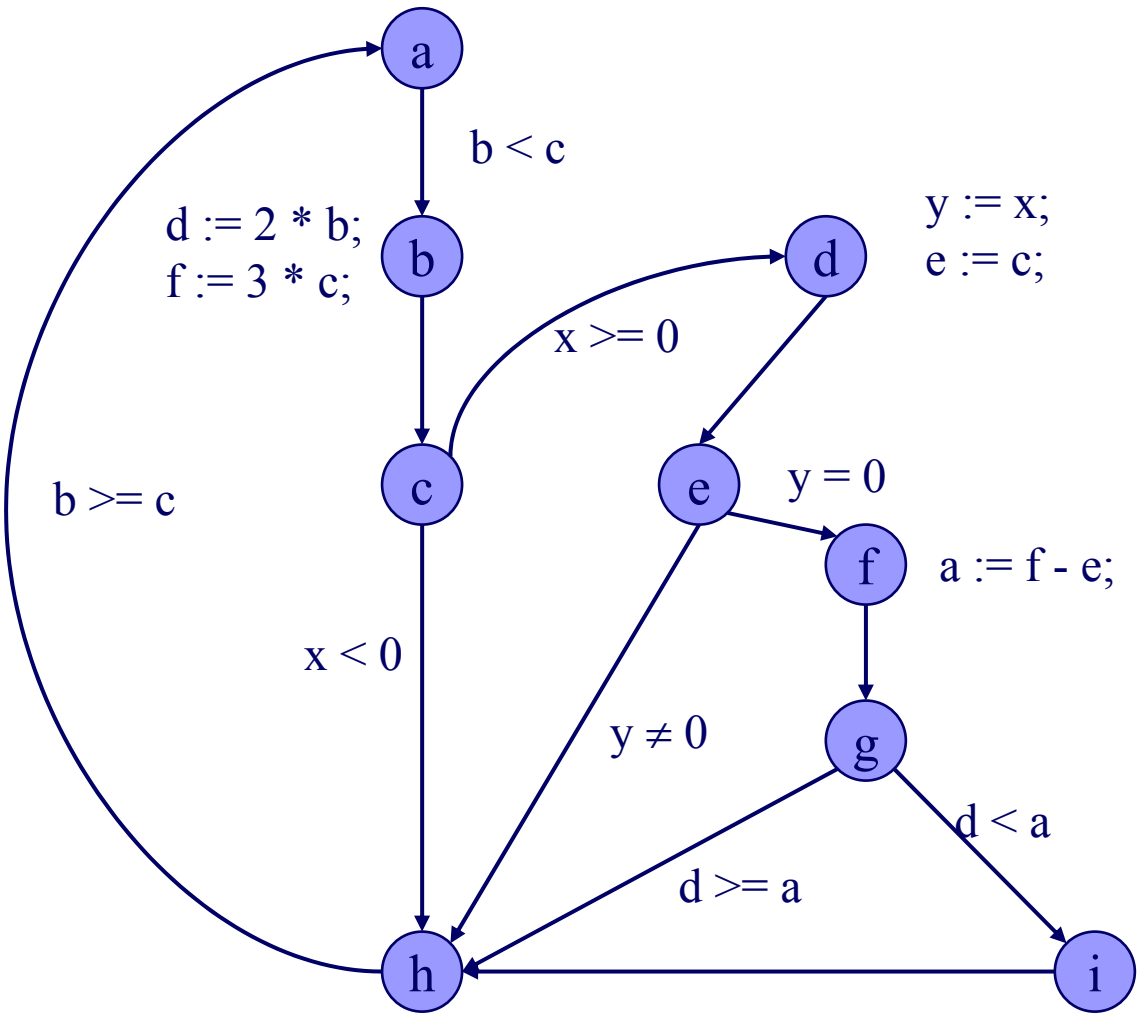
- Etablir le graphe de flot de contrôle de ce programme
- Fournir l ’expression des chemins

Graphe de flot de contrôle du programme P4



G4 = ab [c (1 + d) eb]* f

Problème des chemins non exécutables



- L 'arc **g-h** n'est pas activable :
2b < 2c donc d < a

- Le problème de la détection de chemin non exécutable est un problème difficile, indécidable dans le cas général

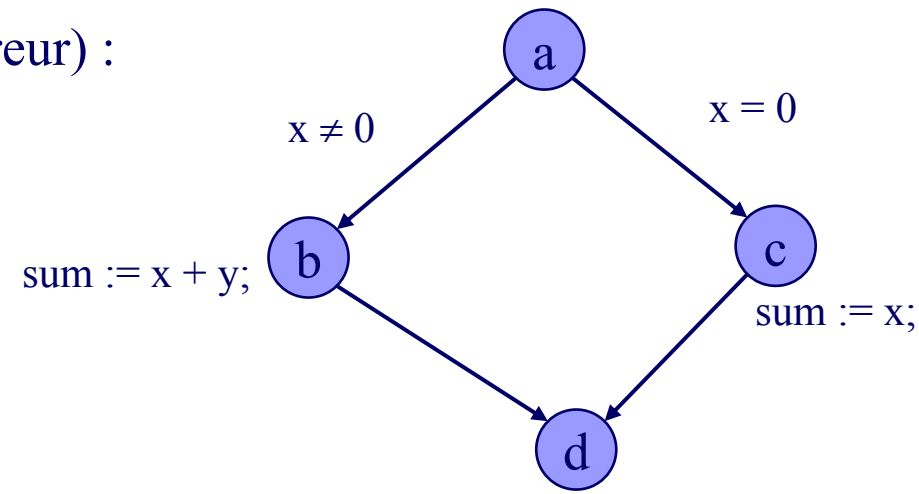
Méthodes de test structurel : couverture de *tous-les-nœuds*

- Taux de couverture :

$$\frac{\text{nb de nœuds couverts}}{\text{nb total de nœuds}}$$

Soit le programme P5 (somme avec erreur) :

```
function sum (x,y : integer) : integer;
begin
if (x = 0) then sum := x
else sum := x + y
end;
```

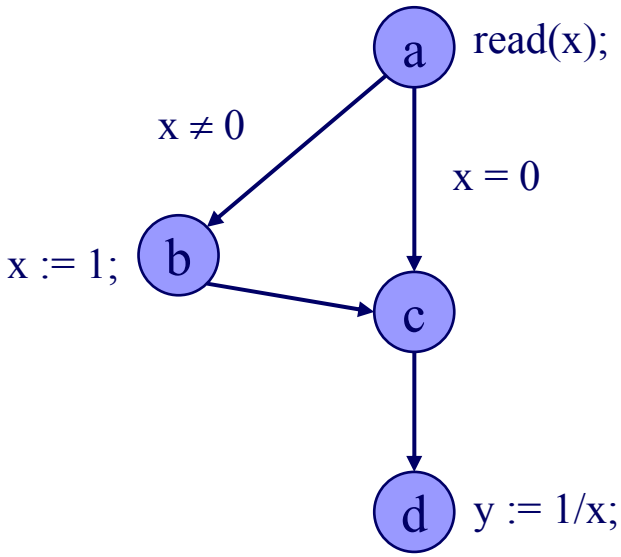


⇒ L ’erreur est détectée par l’exécution du chemin [acd]

Limites du critère *tous-les-nœuds*

Soit le programme P6 (avec erreur) :

```
read(x);
…
if (x <> 0) then x := 1;
…
y  := 1/x;
```



⇒ Le critère tous-les-nœuds est satisfait par le chemin [abcd] sans que l ’erreur ne soit détectée.

Méthodes de test structurel : couverture de *tous-les-arcs*

- Taux de couverture :

$$\frac{\text{nb des arcs couverts}}{\text{nb total des arcs}}$$

- La couverture de tous les arcs équivaut à la couverture de toutes les valeurs de vérité pour chaque nœud de décision.

⇒ Lorsque le critère *tous-les-arcs* est totalement réalisé, cela implique que le critère *tous-les-nœuds* est satisfait

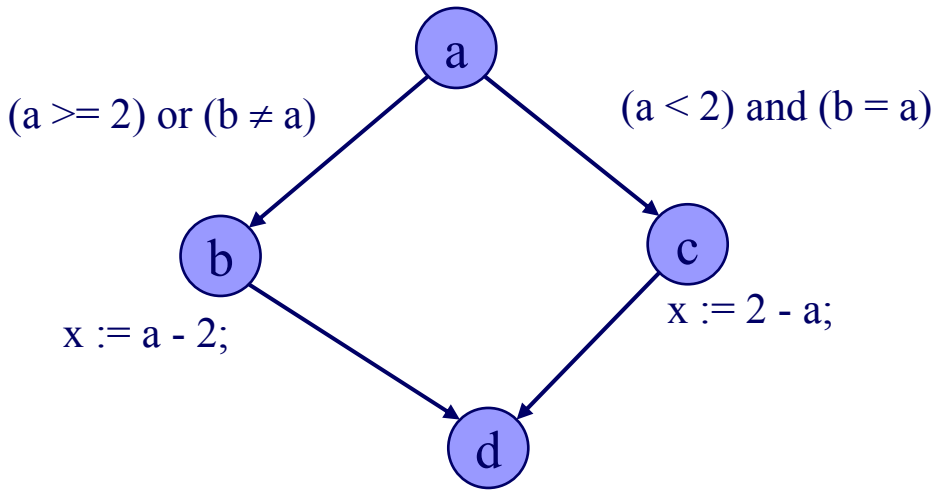
Cas des conditionnelles composées (1)

- Exemple :

  if ((a < 2) and (b = a))

  then x := 2 - a

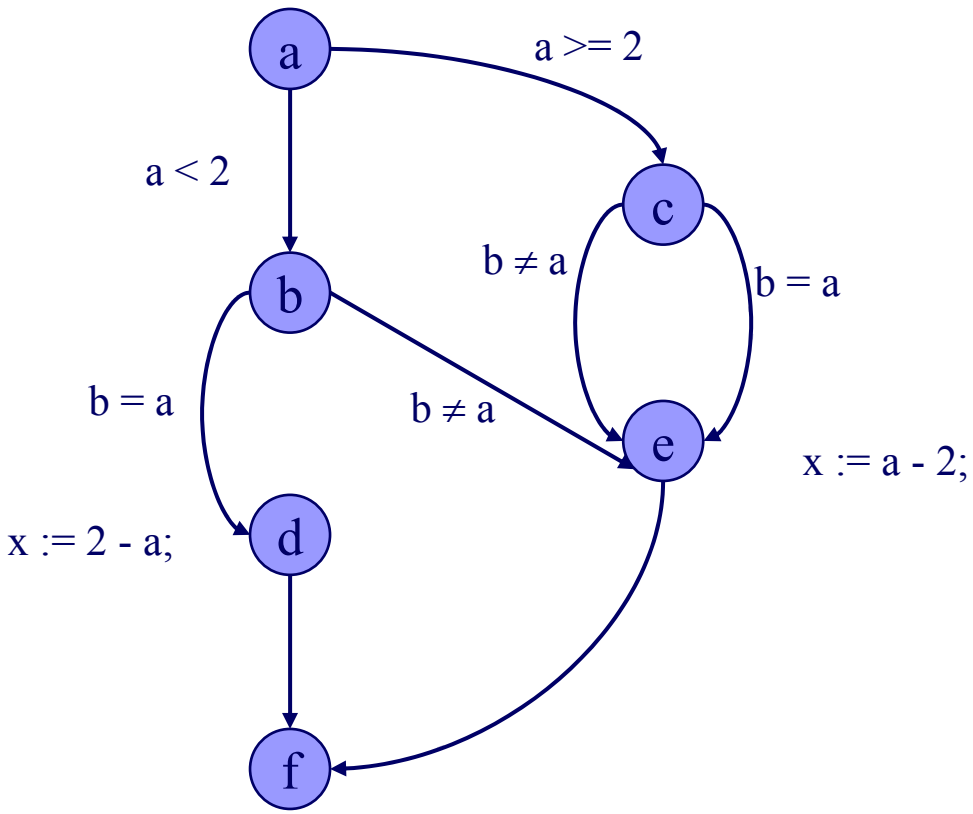
  else x := a - 2



- Le jeu de test DT1 = {a=b=1}, DT2 = {a=b=3} satisfait le critère *tous-les-arcs* sans couvrir toutes les décisions possibles - ex. DT3 = {a=3, b=2}.

Cas des conditionnelles composées (2)

- Le graphe de flot de contrôle doit décomposer les conditionnelles :



Données de test :

- DT1 = {a=b=1}
- DT2 = {a=1, b=0}
- DT3 = {a=3, b=2}
- DT4 = {a=b=3}

Critère de couverture de *condition-décision multiple*

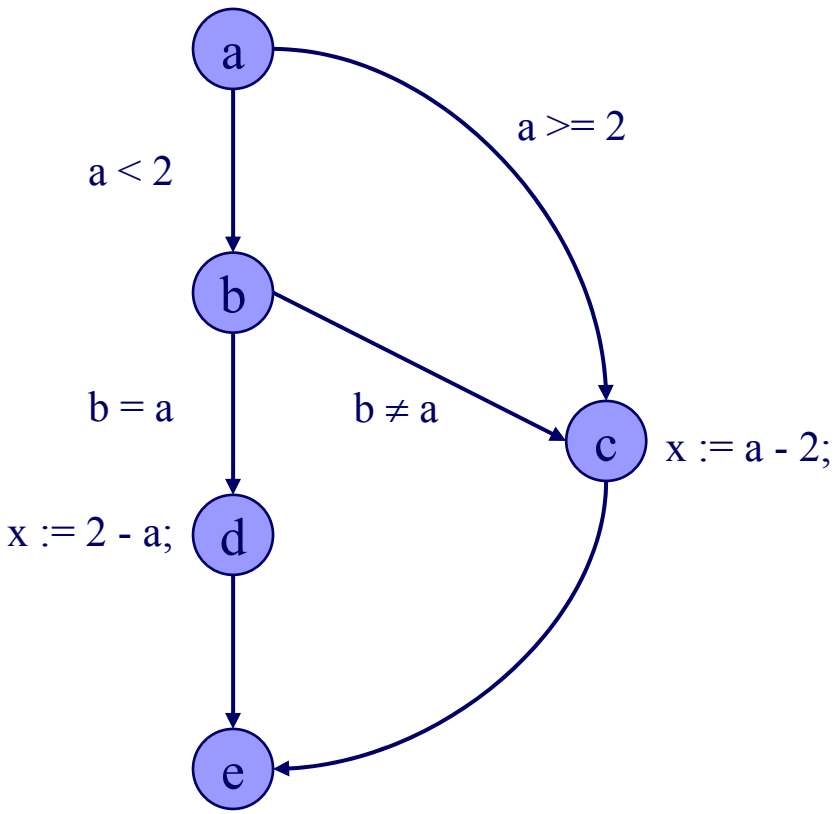
- Le critère de condition-décision multiple est satisfait si :
  - Le critère *tous-les-arcs* est satisfait
  - En plus, chaque sous-expression dans les conditions prend toutes les combinaisons de valeurs possibles

  - Si A & B Then …….. Nécessite :
    - A = B = vrai
    - A = B = faux
    - A = vrai, B = faux
    - A = faux, B = vrai

  - Problème de la combinatoire lors d'expression logique complexe

Sensibilité au compilateur

- Problème : sensibilité au traitement des conditionnelles par le compilateur



Le nombre d 'arcs dépend de la manière dont l 'algorithme est codé et compilé.

Limites des critères *tous-les-arcs* et *condition-décision multiple*

- Il n ’y a pas détection d ’erreurs en cas de non-exécution d ’une boucle

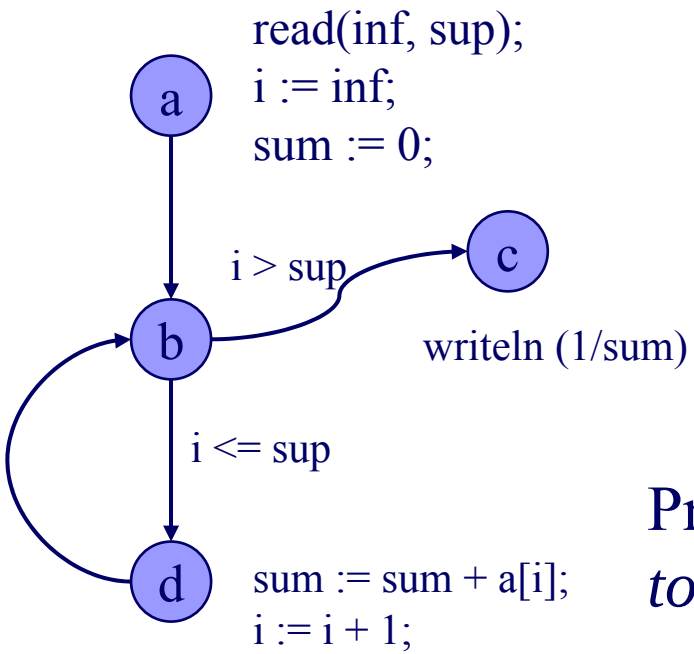
Soit le programme P7 (avec erreur) :

```
read(inf, sup);
i := inf;
sum := 0;

while (i <= sup) do
begin
sum := sum + a[i];
i := i + 1;
end;
writeln (1/sum);
```

Limites des critères *tous-les-arcs* et *condition-décision multiple*

- Il n ’y a pas détection d ’erreurs en cas de non-exécution d ’une boucle



La donnée de test DT1 :

DT1= {a[1]=50, a[2]=60, a[3]=80, inf=1, sup=3}

couvre le *critère tous-les-arcs*

Problème non détecté par le critère *tous-les-arcs* : si inf >sup erreur sur 1/sum

Méthodes de test structurel : couverture de *tous-les-chemins-indépendants*

- Le critère *tous-les-chemins-indépendants* vise à parcourir tous les arcs dans chaque configuration possible (et non pas au moins une fois comme dans le critère *tous-les-arcs*)
- Lorsque le critère *tous-les-chemins-indépendants* est satisfait, cela implique :
  - le critère *tous-les-arcs* est satisfait,
  - le critère *tous-les-nœuds* est satisfait.
- Sur le programme P7, l 'arc [c-e] est sensibilisé lorsque $i > sup$ à la fois dans le contexte sum = 0 (la boucle n 'est pas activée) et sum ≠ 0 (la boucle est activée)

Procédure : *tous-les-chemins-indépendants*

- 1 - Evaluer le nombre de décisions par analyse des conditionnelles
- 2 - Produire une DT couvrant le maximum de nœuds de décisions du graphe
- 3 - Produire la DT qui modifie la valeur de vérité de la première instruction de décision de la dernière DT calculée.

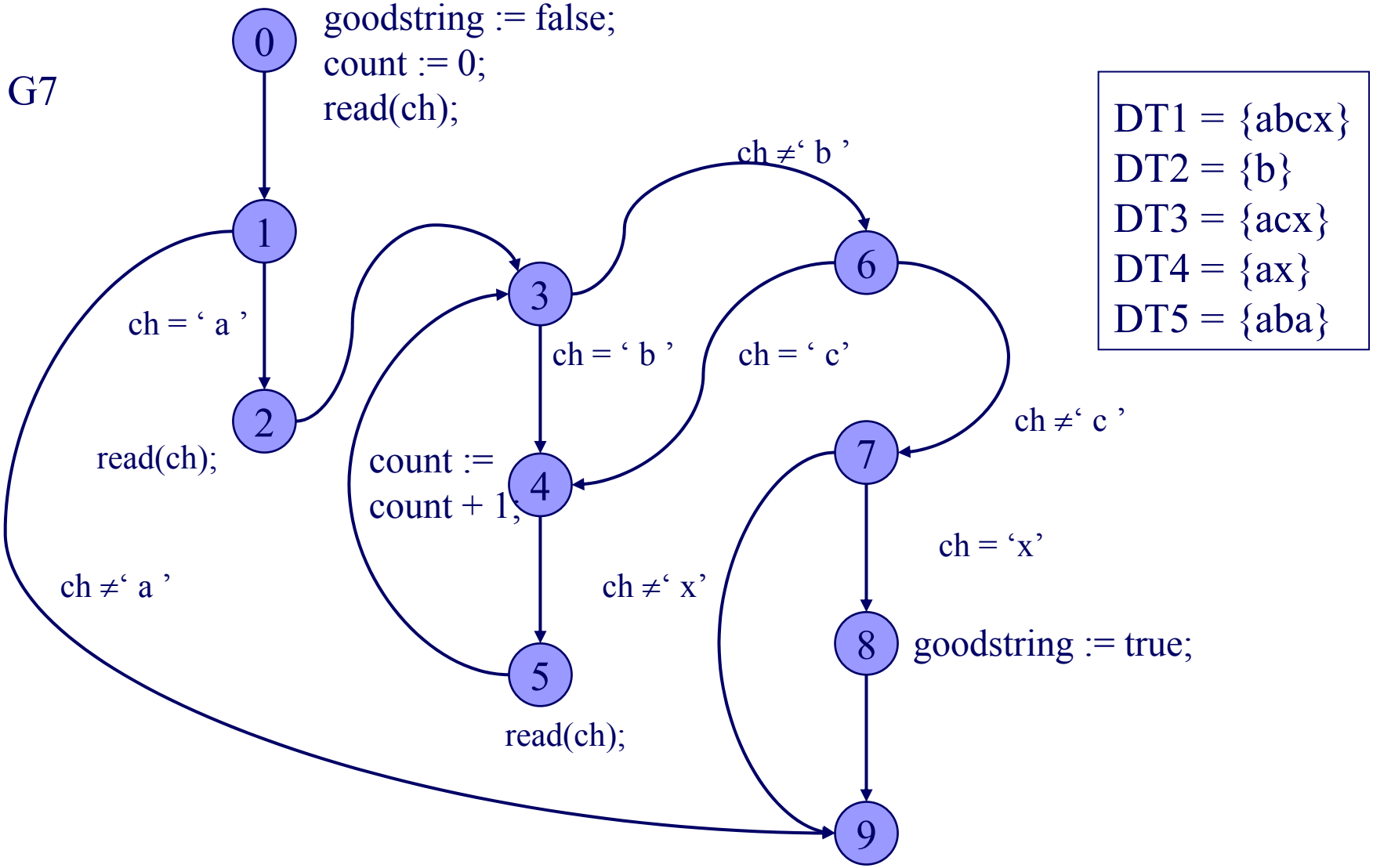
  Recommencer l ’étape 3 jusqu ’a la couverture de toutes les décisions.

Critère *tous-les-chemins-indépendants* - Exemple

- Soit le programme P8 suivant :

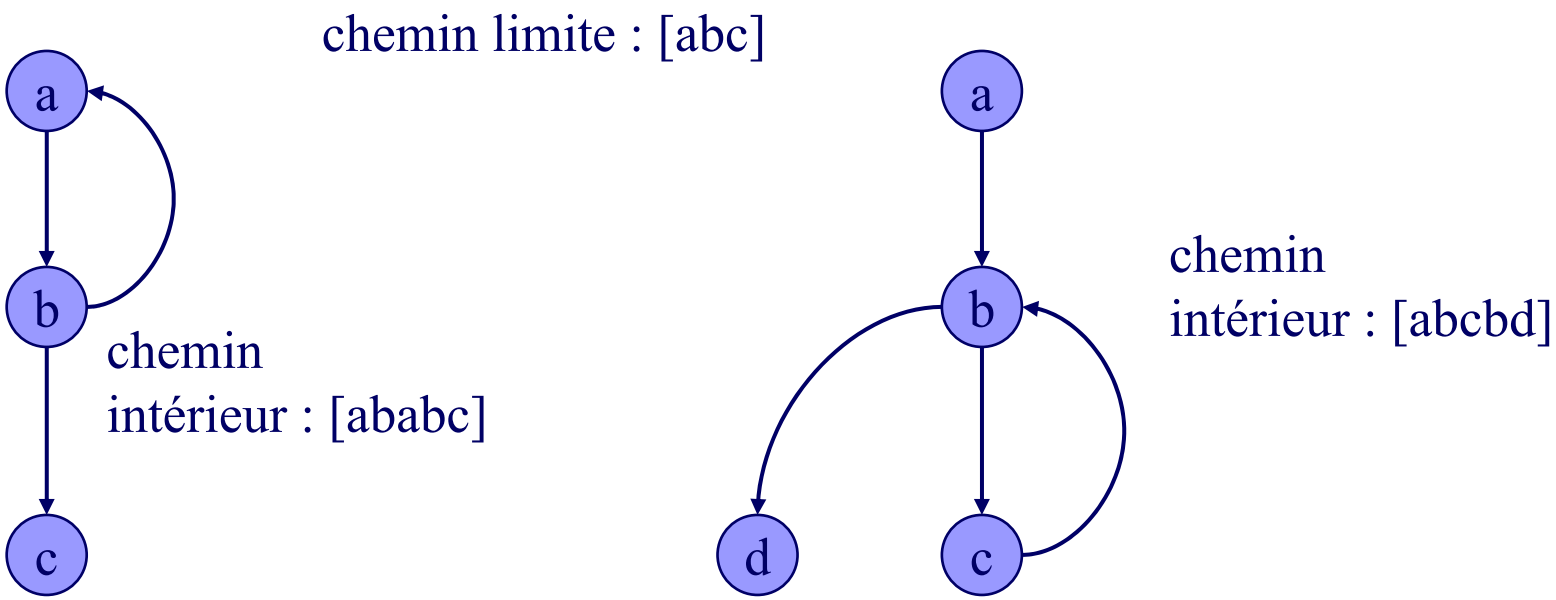
```
function goodstring (var count : integer) :
    boolean;
var ch : char;
begin
    goodstring := false;
    count := 0;
    read(ch);
    if ch = ‘ a ’ then
     begin
     read(ch)
  while (ch=‘ b ’) or (ch=‘ c ’) do
    begin
    count := count + 1;
    read(ch);
    end;
  if ch = ‘ x ’ then
            goodstring = true;
  end;
end;
```

Critère *tous-les-chemins-indépendants* - Exemple

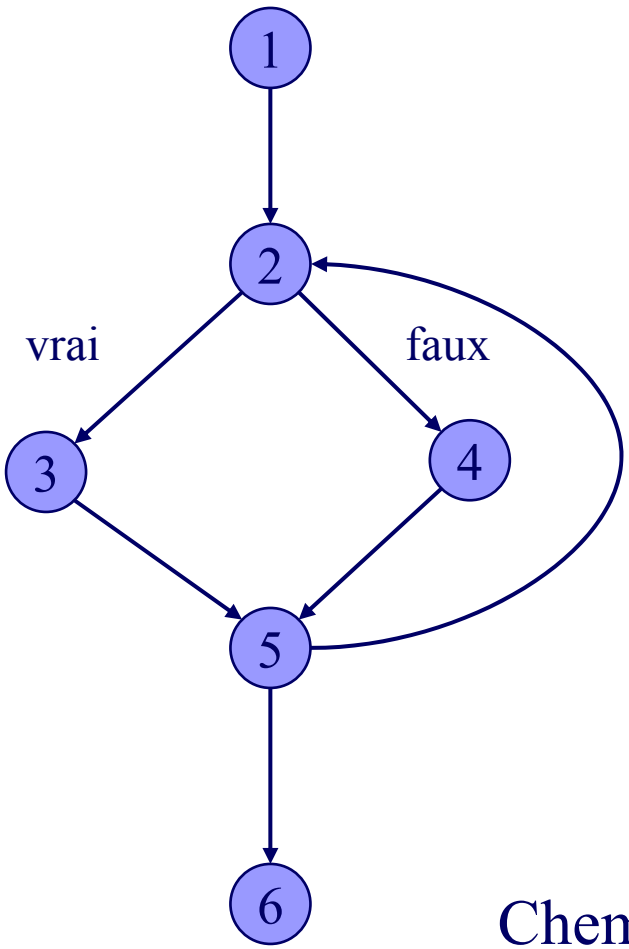


Couverture des chemins limites et intérieurs (1)

- Les chemins limites traversent la boucle mais ne l ’itèrent pas;
- Les chemins intérieurs itèrent la boucle une seule fois;



Couverture des chemins limites et intérieurs (2)



Chemins limites :

- [1,2,3,5,6]
- [1,2,4,5,6]

Chemins intérieurs :

- [1,2,3,5,2,3,5,6]
- [1,2,3,5,2,4,5,6]
- [1,2,4,5,2,3,5,6]
- [1,2,4,5,2,4,5,6]

Chemins intérieurs : exécute n fois la boucle

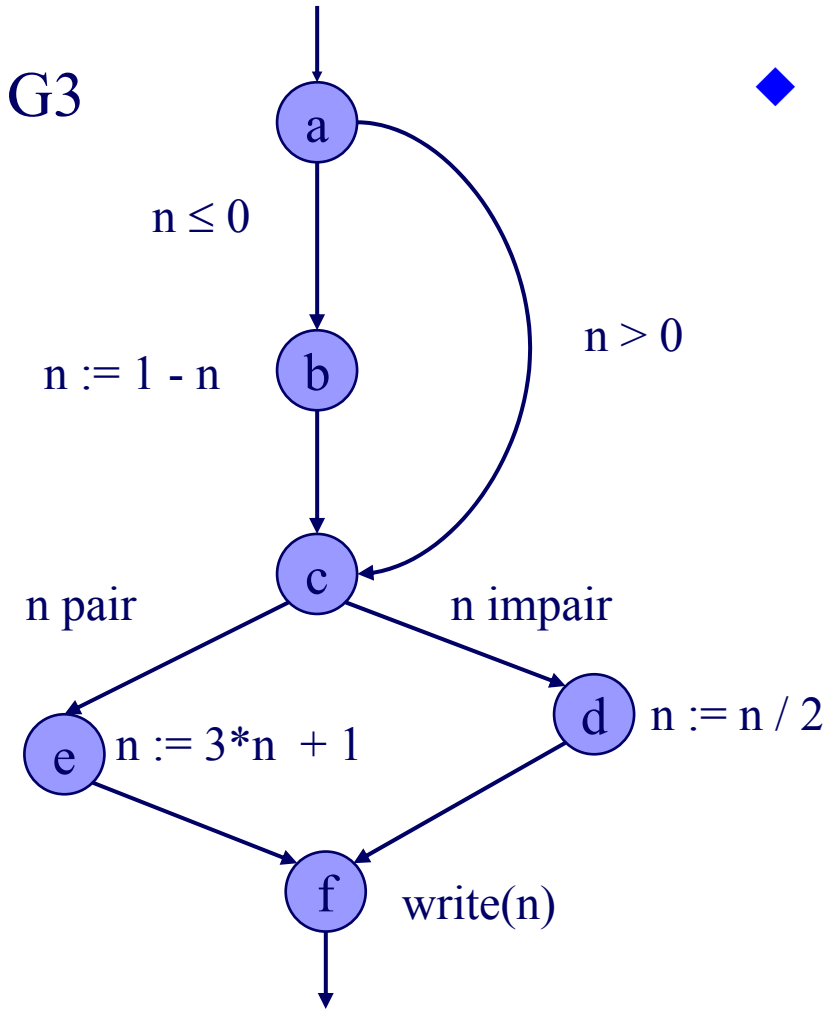
Méthodes de test structurel - Exercice

Soit le programme P3 suivant :

```
If n ≤ 0 then n := 1-n
      end;
If n pair
                then n := n / 2
                else n := 3*n + 1
                end ;
Write(n);
```

- Calculer les DT suivant les critères :
  - *tous-les-nœuds,*
  - *tous-les-arcs,*
  - *tous-les-chemins-indépendants.*

Méthodes de test structurel - Exercice



- Critères de test :
  - *tous-les-nœuds*
    - » *n = 0, n = -1*
  - *tous-les-arcs*
    - » *n = 2, n = -2*
  - *tous-les-chemins-indépendants*
    - » n = -1, n = -2, n = 1, n = 2

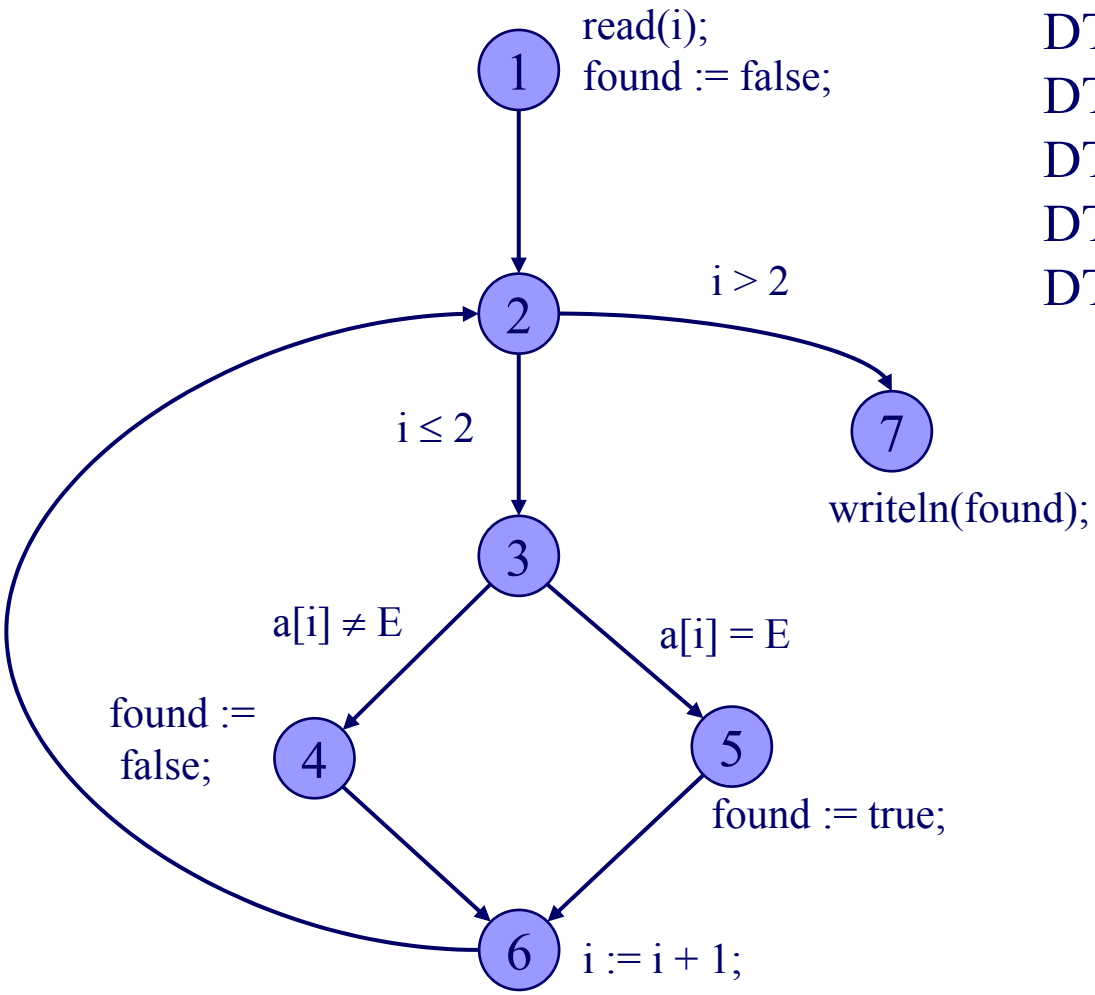
Méthodes de test structurel - Exercice

- Soit le programme P9 suivant :

```
program p (input, output) ;
var a : array[1..2] of integer;
E, i : integer;
begin
    read(i, E, a[1], a[2]);
    found := false;
while i <= 2 do
    begin
     if (a[i] = E) then found := true;
    else found := false;
    i := i + 1;
    end;
writeln(found);
end;
```

- Calculer les DT suivant les critères :
  - *tous-les-nœuds*,
  - *tous-les-arcs*,
  - *tous-les-chemins-indépendants*.

Méthodes de test structurel - Exercice



DT1 = {i=3}
DT2 = {i=1, E=10, a[1]=20, a[2]=10}
DT3 = {i=1, E=10, a[1]=10, a[2]=20}
DT4 = {i=1, E=10, a[1]=10, a[2]=10}
DT5 = {i=1, E=10, a[1]=20, a[2]=30}

Couverture du critère Portion Linéaire de Code suivie d'un Saut - PLCS

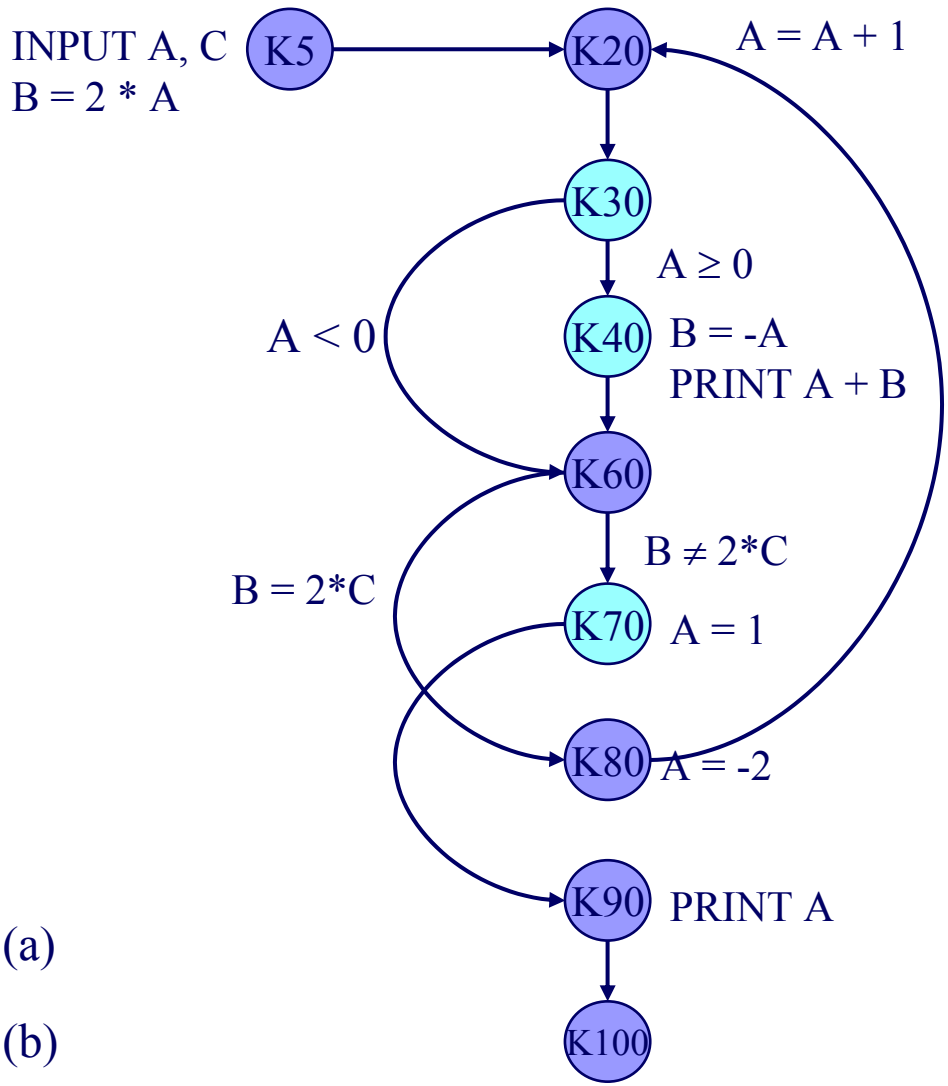
- Principes :
  - On considère, dans le graphe de flot de contrôle, l'entrée, la sortie et les nœuds qui constituent l'arrivée d'un branchement – type (a) – , et les autres nœuds – type (b).
  - On appelle PLCS un chemin partant d'un nœud de type (a) et aboutissant à nouveau à un nœud de type (a); l'avant dernier et le dernier nœud doivent constituer le seul saut du chemin.
  - On définit le critère TER3 :

$$\frac{\text{PLCS couvertes}}{\text{Total des PLCS}}$$

Couverture du critère PLCS – Exemple

Soit le programme suivant :

```
005 INPUT A, C
010 B = 2 * A
020 A = A + 1
030 IF A < 0 THEN GOTO 60
040 B = -A
050 PRINT A + B
060 IF B = 2 * C THEN GOTO 80
070 A = 1 : GOTO 90
080 A = -2 : GOTO 20
090 PRINT A
100 END
```



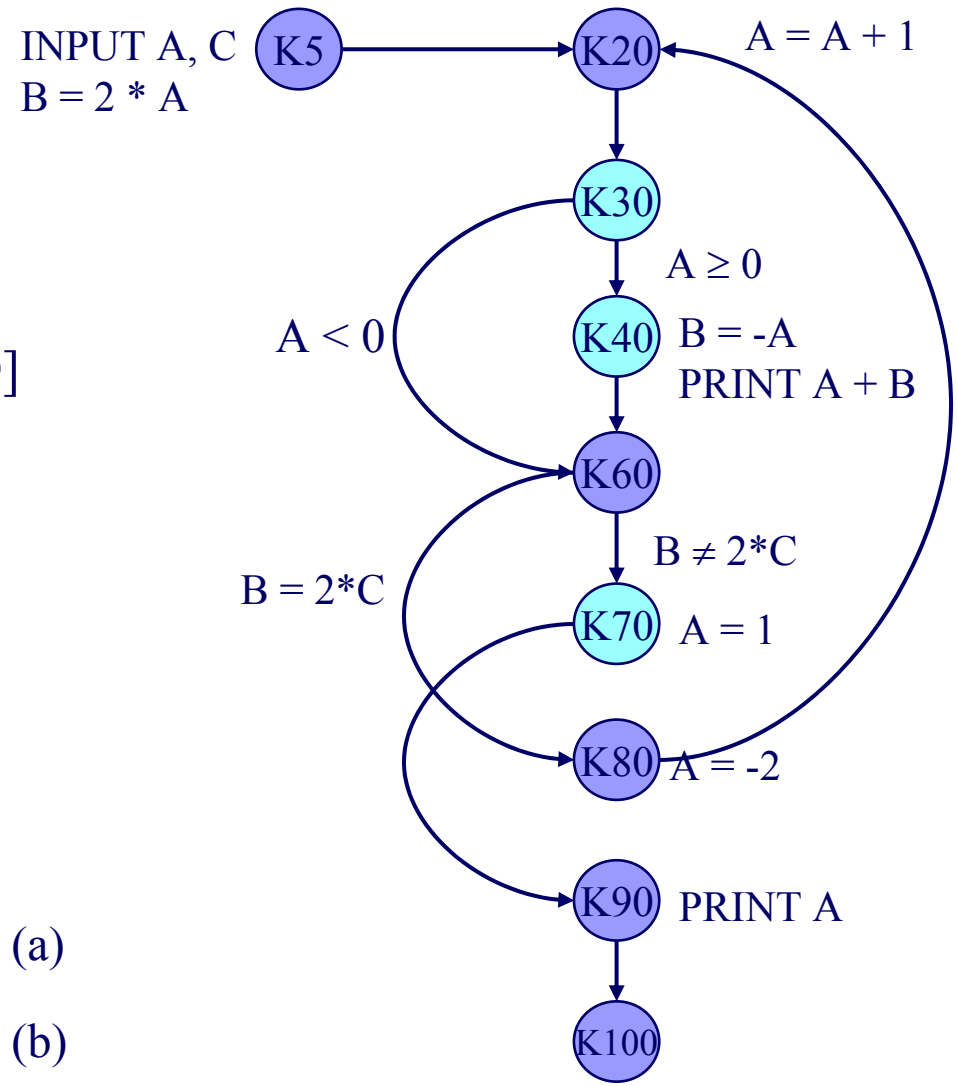
K5 Nœud type (a)

K70 Nœud type (b)

Couverture du critère PLCS – Exemple

Ce programme comprend 10 PLCS :

1) [K5, K20, K30, K60]
2) [K5, K20, K30, K40, K60, K80]
3) [K5, K20, K30, K40, K60, K70, K90]
4) [K20, K30, K60]
5) [K20, K30, K40, K60, K80]
6) [K20, K30, K40, K60, K70, K90]
7) [K60, K80]
8) [K60, K70, K90]
9) [K80, K20]
10) [K90, K100]



K5 Nœud type (a)

K70 Nœud type (b)

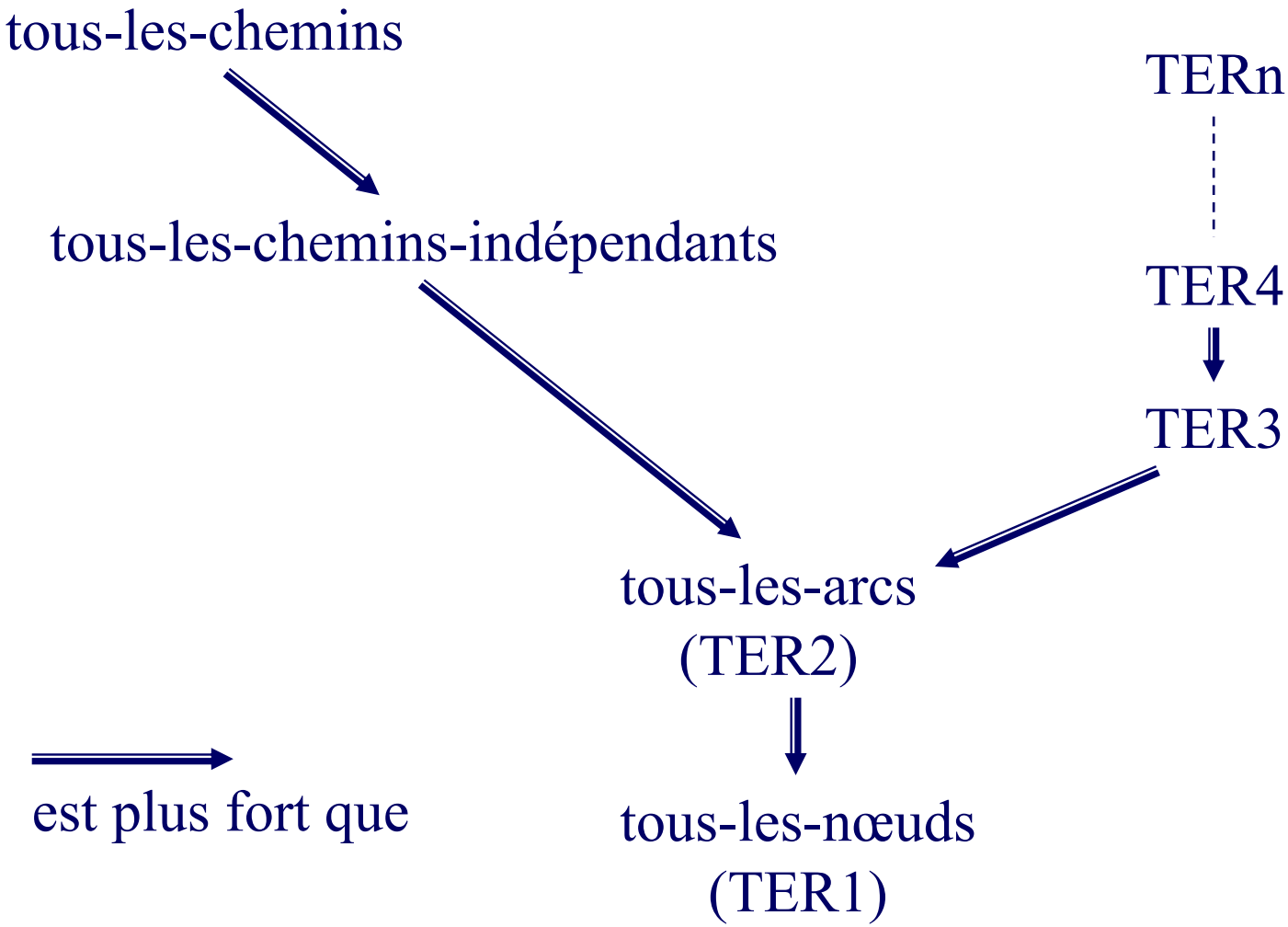
Critères de type PLCS

- Autres critères de type PLCS

$$TER4 = \frac{\text{Chemins composés de 2 PLCS}}{\text{Total des chemins composés de 2 PLCS}}$$

$$TER5 = \frac{\text{Chemins composés de 3 PLCS}}{\text{Total des chemins composés de 3 PLCS}}$$

Hiérarchie des critères basés sur le flot de contrôle



3-2 Critères de couverture basés sur le flot de données

- Les critères basés sur le flot de données sélectionnent les données de test en fonction des définitions et des utilisations des variables du programme
- Définitions sur les occurrences de variables :
  - une variable est *définie* lors d'une instruction si la valeur de la variable est modifiée (affectations),
  - Une variable est dite *référencée* si la valeur de la variable est utilisée.
- Si la variable référencée est utililsée dans le prédicat d'une instruction de décision (if, while, …), il s'agit d'une *p-utilisation*, dans les autres cas (par exemple dans un calcul), il s'agit d 'une *c-utilisation*.

Critères basés sur le flot de données

- Notion d'instruction utilisatrice :
  - Une instruction J2 est *utilisatrice* d'une variable x par rapport à une instruction J1, si la variable x qui a été définie en J1 est peut être directement référencée dans J2, c'est-à-dire sans redéfinition de x entre J1 et J2

Exemple :

```
(1)     x := 7;
(2)     a := x+2;
(3)     b := a*a;
(4)     a := a+1;
(5)     y := x + a;
```

Considérons l'instruction (5) :

(5) est c-utilisatrice de (1) pour la variable x

(5) est c-utilisatrice de (4) pour la variable a

Critères basés sur le flot de données

- Notion de chemin d'utilisation
  - Un chemin d'utilisation relie l'instruction de définition d'une variable à une instruction utilisatrice; un tel chemin est appelé chemin dr-strict

Exemple :

```
(1)     x := 7;
(2)     a := x+2;
(3)     b := a*a;
(4)     a := a+1;
(5)     y := x + a;
```

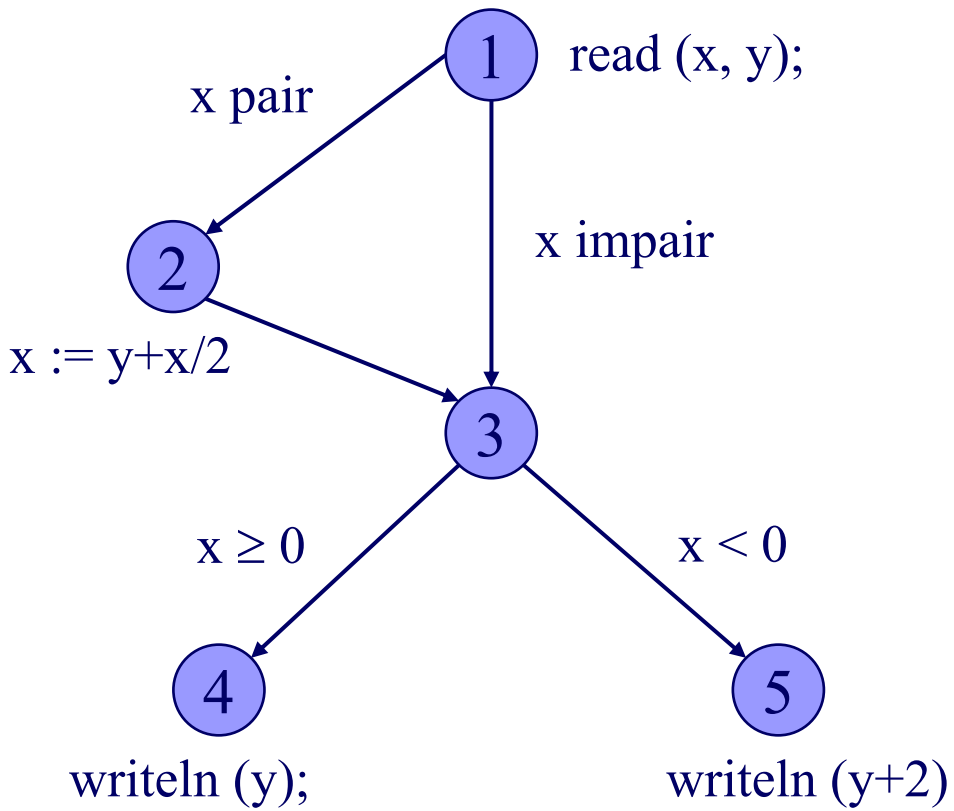
Le chemin [1,2,3,4,5] est un chemin dr-strict pour la variable x (mais pas pour la variable a)

Critères toutes-les-définitions et *tous-les-utilisateurs*

- *toutes-les-définitions* : pour chaque définition, il y a au moins un chemin dr-strict dans un test
- *tous-les-utilisateurs* : pour chaque définition et pour chaque référence accessible à partir de cette définition, couverture de tous les utilisateurs (nœuds c-utilisateurs ou arcs p-utilisateurs)

tous-les-utilisateurs ➔ *toutes-les-définitions*

Critères toutes-les-définitions et *tous-les-utilisateurs* - exemple



Couverture du critère toutes-les-définitions :

[1,3,5]
[1,2,3,5]

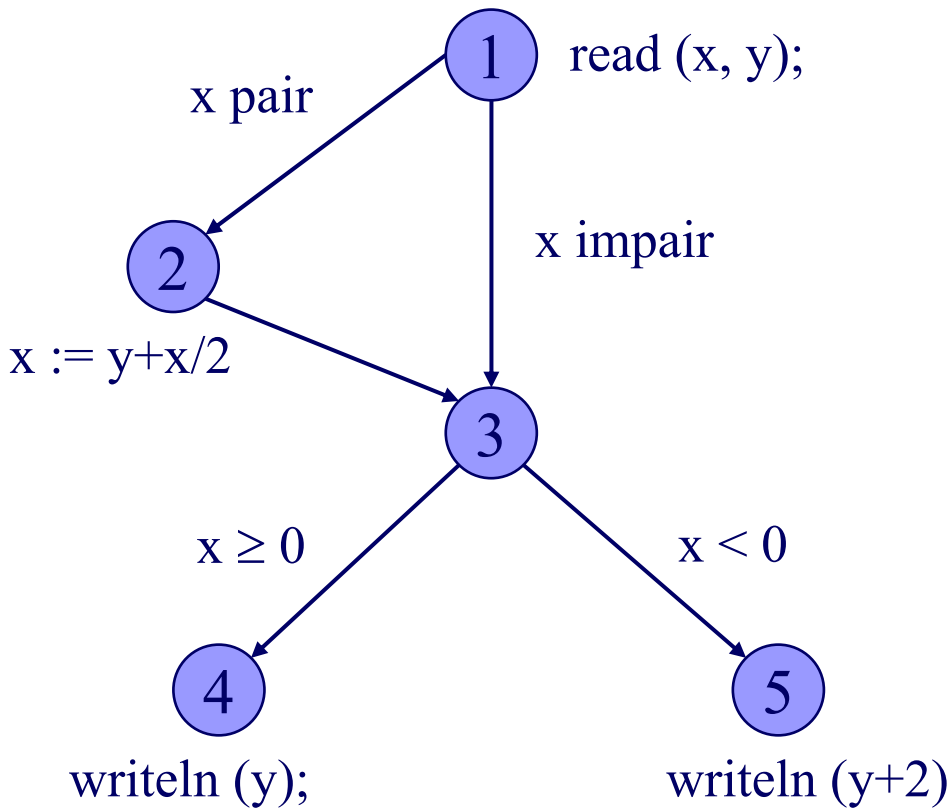
Couverture du critère tous-les-utilisateurs :

[1,3,4]
[1,2,3,4]
[1,3,5]
[1,2,3,5]

Autrès critères basés sur le flot de données

- Le critère *tous-les-utilisateurs* nécessite souvent la sensibilisation d'un grand nombre de chemin, deux autres critères, intermédiaires entre *tous-les-utilisateurs* et *toutes-les-définitions* sont proposés :
  - *tous-les-p-utilisateurs/quelques-c-utilisateurs* : pour chaque définition, et pour chaque p-utilisation accessible à partir de cette définition et pour chaque branche issue de la condition associée, il y a un chemin dr-strict prolongé par le premier segment de cette branche ; s'il n'y a aucune p-utilisation, il suffit d'avoir un chemin dr-strict entre la définition et l'une des c-utilisation,
  - *tous-les-c-utilisateurs/quelques-p-utilisateurs* : pour chaque définition, et pour chaque c-utilisation accessible à partir de cette définition, il y a un chemin dr-strict ; s'il n'y a aucune c-utilisation, il suffit d'avoir un chemin dr-strict entre la définition et l'une des p-utilisation.

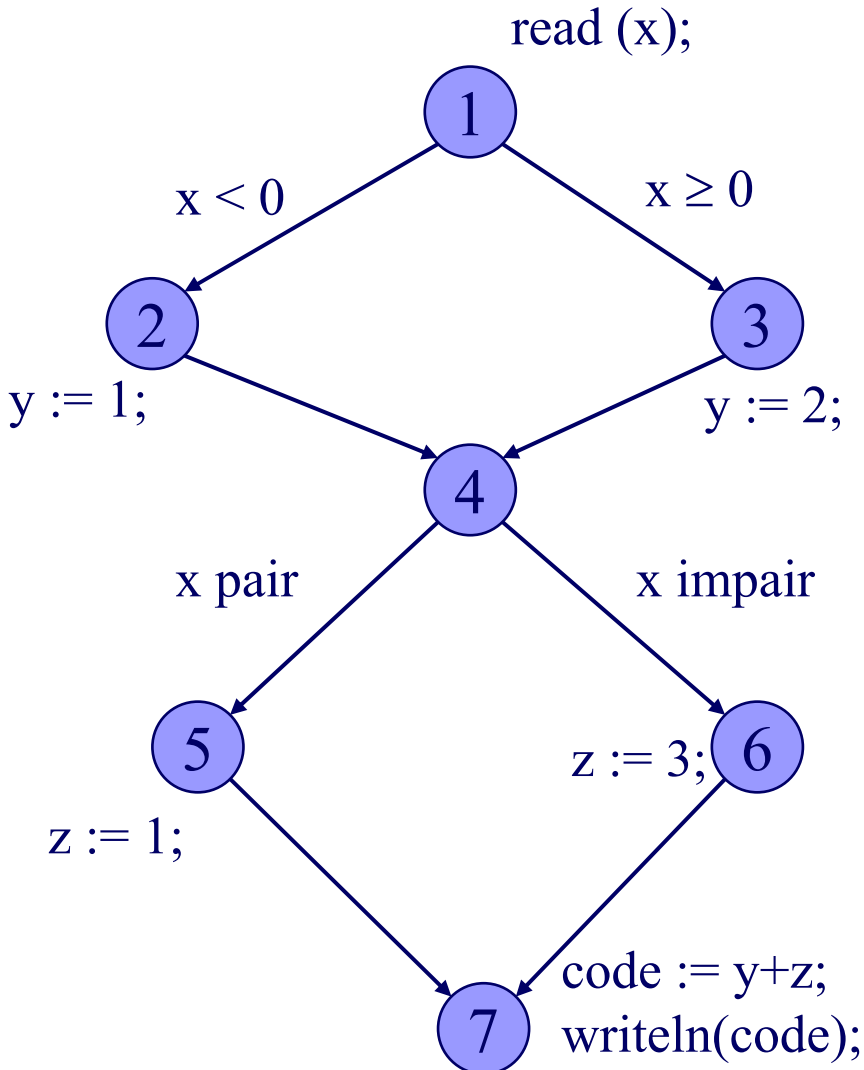
Critère *tous-les-p-utilisateurs/quelques-c-utilisateurs* - exemple



Couverture du critère tous-les-p-utilisateurs/ quelques-c-utilisateurs :

[1,3,4]
[1,2,3,5]

Limite du critère *tous-les-utilisateurs*



Couverture du critère tous-les-utilisateurs :

[1,2,4,5,7]
[1,3,4,6,7]

Ces deux tests ne couvrent pas tous les chemins d'utilisation :
(7) peut être utilisatrice de (2) pour la variable y

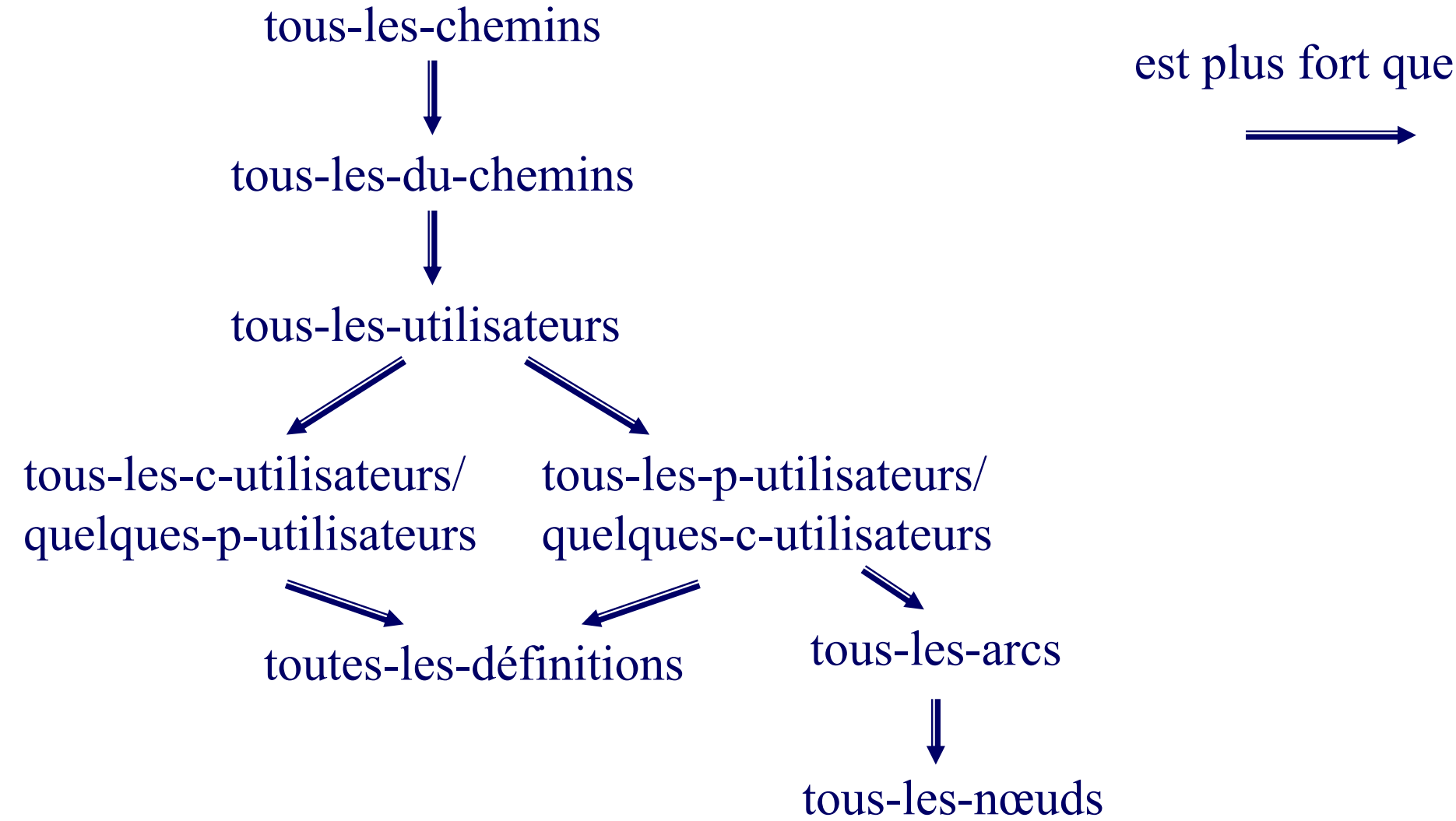
=> critère *tous-les-du-chemins*

Critère *tous-les-du-chemins*

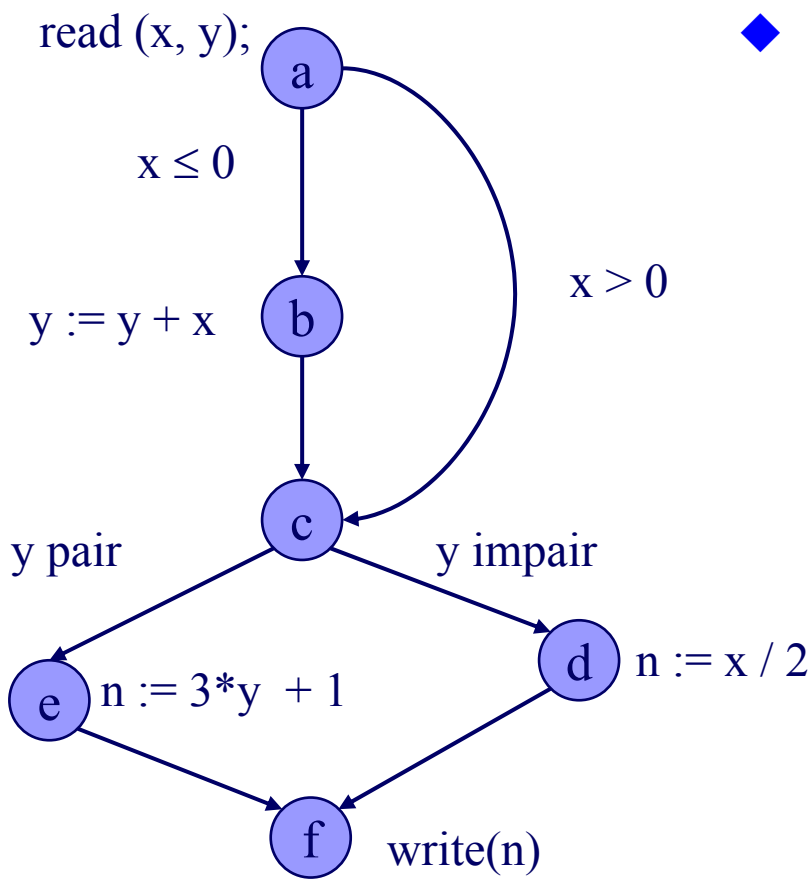
- Ce critère rajoute au critère *tous-les-utilisateurs* le fait qu'on doit couvrir tous les chemins possibles entre la définition et la référence, en se limitant aux chemins sans cycle.
- Sur l'exemple précédent, ce critère sensibilise :

[1,2,4,5,7]
[1,3,4,6,7]
[1,2,4,6,7]
[1,3,4,5,7]

Hiérarchie des critères basés sur le flot de données

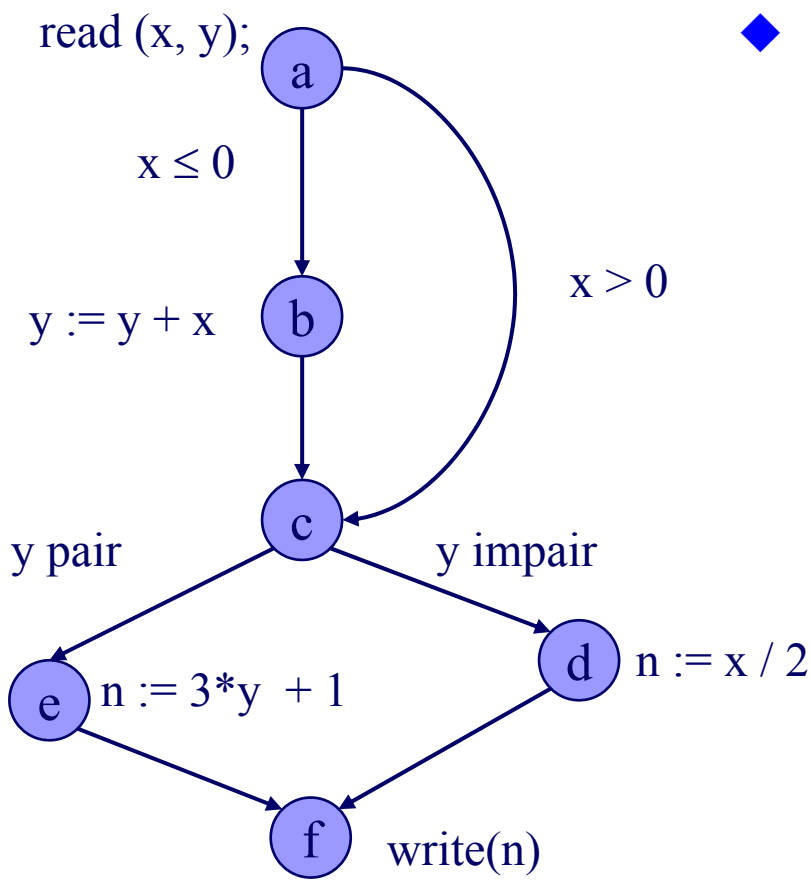


Méthodes de test structurel basés sur la couverture du flot de données - Exercice



- Critères de test :
  - *toutes-les-définitions*
  - *tous-les-utilisateurs*

Méthodes de test structurel basés sur la couverture du flot de données - Solution



- Critères de test :
  - *toutes-les-définitions*
    - » [a,c,d,f] : x=1 & y=3
    - » [a,b,c,e,f] : x=-1 & y=3
  - *tous-les-utilisateurs*
    - » [a,c,d,f] : x=1 & y=3
    - » [a,c,e,f] : x=-1 & y=3
    - » [a,b,c,e,f] : x=0 & y=2
    - » [a,b,c,d,f] : x=0 & y=3

3-3 Critères de couverture basés sur les fautes – le test mutationnel

- L'idée est de considéré des variantes du programme – les mutants – ne diffèrent que par une instruction
- Par exemple, pour l'instruction suivante dans un programme p quelconque :
  - if a > 8 then x := y

  on peut considérer les mutants suivants :
  - if a < 8 then x := y
  - if a >= 8 then x := y
  - if a >10then x := y
  - if a > 8 then x := y+1
  - if a > 8 then x := x
  - …….

La création des mutants est basée sur des modèles de faute

Le test mutationnel

- Cette technique vise à évaluer les Données de Tests vis-à-vis de la liste des fautes les plus probables qui ont été envisagées (modèles de fautes).
- Les mutations correspondent à des transformations syntaxiques élémentaires de programme
- Pour un programme P, soit M(P) l'ensemble de tous les mutants obtenus par le modèle de faute. Certains mutants peuvent avoir le même comportement que P, noter E(P). On fait exécuter la suite de tests T à chacun des mutants de M(P) et on note DM(P) l'ensemble de ceux qui ne fournissent pas le même résultat que P.
- Le score de mutation MS(P,T) correspond à :

$$MS(P,T) = \frac{DM(P)}{M(P) - E(P)}$$

Le test mutationnel - Synthèse

- Le test mutationnel est plus une approche pour évaluer la qualité d'une suite de tests : combien de mutants fonctionnellement distinguables sont « tués »,
- Cette méthode est couteuse de mise en œuvre, en particulier pour établir les mutants fonctionnement distinguables des autres

  => test mutationnel faible : que sur des composants élémentaires

Règles de mutations d'opérateurs (instructions)

operator set

1. expression deletion
2. boolean expression negation
3. term associativity shift
4. arithmetic operator by arithmetic operator
5. relational operator by relational operator
6. logical operator by logical operator
7. logical negation
8. variable by variable replacement
9. variable by constant replacement
10. constant by required constant replacement

Opérateurs de mutations (Statecharts)

Statecharts operator set

1. wrong-start-state
2. arc-missing
3. event-missing
4. event-extra
5. event-exchanged
6. destination-exchanged
7. output-missing
8. output-exchanged

4- Les outils de l'automatisation du test

- Le test de logiciels est une activité très coûteuse qui gagne à être supportée par des outils.

⇨ Plus de 278 outils référencés sur le site spécialisé www.stickminds.com

⇨ Les principales catégories d'outils concernent :

- L'exécution des tests
- La gestion des campagnes
- Le test de performance
- La génération de tests fonctionnels
- La génération de tests structurels

Outils d'aide à l'exécution des tests

- Principales fonctions :
  - Capture et re-exécution des scripts réalisés sur une IHM
  - Sauvegarde des tests et des résultats de tests
  - Génération de scripts de test en fonction des langages et des plate-formes

- Exemples de produits :
  - Mercury Winrunner
  - IBM Rational Robot
  - Segue Silktest

Outils de gestion des campagnes de test

- Principales fonctions :
  - Définition d'une campagne de test
  - Historisation des résultats
  - Gestion des tests de non-regression
- Exemples de produits :
  - IBM Rational Test Manager (inclut dans la Suite TestStudio)
  - Segue Silkplan Pro

Outils de test de performance

- Principales fonctions :
  - Test de montée en charge
  - Simulation d'environnement
  - Evolution aggressive de l'accès aux ressources

- Exemples de produits :
  - Empirix e-Load
  - Mercury LoadRunner
  - Segue Silkperformer

Outils de génération de tests fonctionnels

- Principales fonctions :
  - Génération des tests à partir d'un modèle des spécifications
  - Animation des cas de tests
  - Traçabilité des tests

- Exemple de produits :
  - Leirios Test Generator
  - T-Vec
  - Reactis
  - Conformiq

Outils de génération de tests structurels

- Principales fonctions :
  - Critères de couvertures
  - Evaluation du ratio de couverture
  - Simulation d'environnement – Bouchonnage (Stubbing)

- Exemples de produits :
  - IPL Cantata ++
  - Parasoft CodeWizard