# Test Logiciel 1

EMN – GIPAD/GSI
2010-2011

Arnaud Gotlieb
INRIA / CELTIQUE



**Le Prof…** http://www.irisa.fr/lande/gotlieb/

**Le Cours…** Code-based Testing
OO Testing, Model-based Testing
Constraint-based Testing

**Les Outils…** CUnit, JUnit
LTG, Simulink
Euclide, Pex



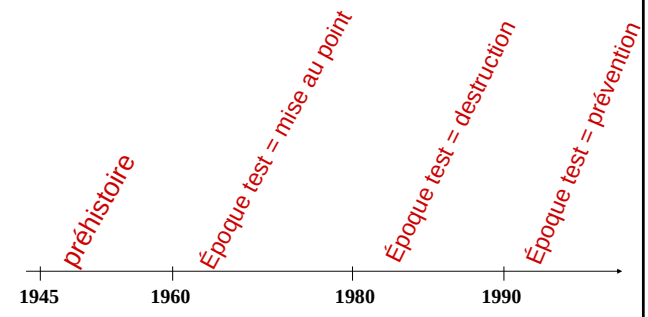
## Aujourd'hui…

- Introduction au Test Logiciel
- Test unitaire avec Cunit
- Critères de couverture
- Techniques de génération de cas de test



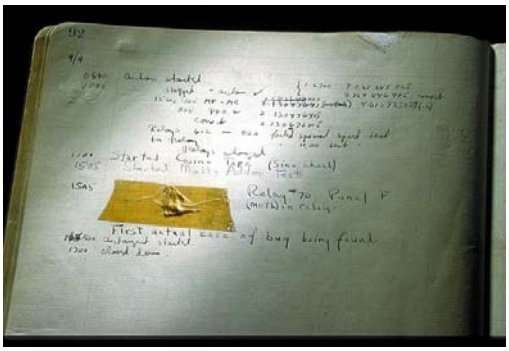
## INTRODUCTION AU TEST LOGICIEL

## historique et problèmatique du test






## 9/9/1945





## 1960-80 : Test = mise au point

Qu'avons-nous compris depuis ?

Chaîne causale : **erreur → faute → défaillance**

En fait, 3 activités distinctes :

* détection de défaillances (rôle du test)
* localisation de fautes (rôle de la mise au point)
* correction des erreurs (rôle de la mise au point)



## 1980-90 : Test = destruction

« *Testing is the process of executing a program with the intent of finding errors* »
[G. Myers *The Art of Software Testing* 1979]

Conséquence immédiate :
le testeur ne doit pas être le programmeur

Position dogmatique progressivement abandonnée !

## 1990-.. : Test = prévention

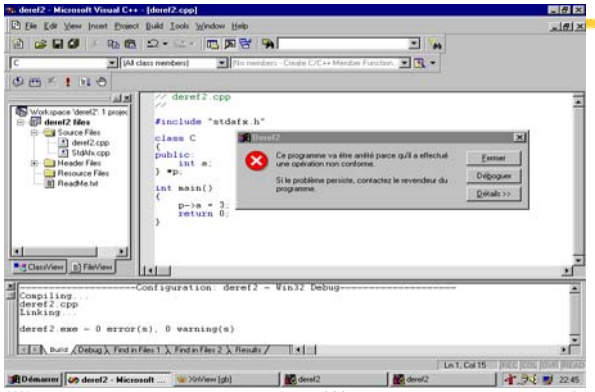
« *Se* ***convaincre****, par des techniques* ***d'analyse ou d'exécution,*** *qu'un programme répond à* ***ses spécifications*** »

- Analyse → Contrôle : vérifier des propriétés avant exécution

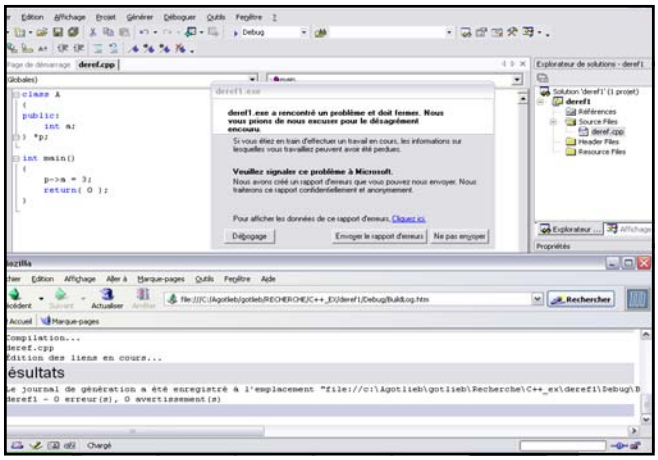
- Exécution → Test : évaluer un résultat après exécution



## Visual 1998






## Terminology

(IEEE Standard Glossary of SE, BCS's standard for Softw. Testing)

- ***Validation***: « The process of evaluating software at the end of software development to ensure compliance with intented usage »

- ***Verification***: « The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase »

- ***Testing:*** « Evaluating software by observing its execution »

## Test de programme : notre définition

- Tester = **exécuter** un programme P pour mettre en évidence **la présence de fautes,** par rapport à sa spécification F
- Recherche de contre-exemples :

$$\exists X \text{ tq } P(X) \neq F(X) \text{ ?}$$



## Correction de programme : limite fondamentale

Impossible de démontrer la correction d'un programme dans le cas général ← indécidabilité du **problème de l'arrêt d'une machine de Turing**

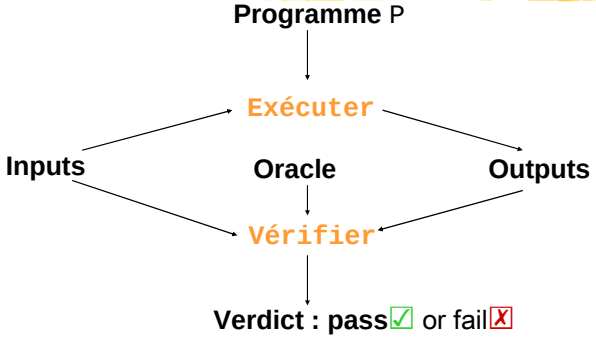
«*Program Testing can be used to prove the presence of bugs, but never their absence* » [Dijkstra 74]

---

PS : développeur expérimenté → 1 faute / 10 lignes de code
163 fautes / 1000 instructions
[B. Beizer *Software Testing Techniques* 1990]



## Processus de test






## Problème de l'oracle :
## Comment vérifier les sorties calculées ?

<u>En théorie :</u>

- par prédiction du résultat attendu
- à l'aide d'une formule issue de la spécification
- à l'aide d'un autre programme

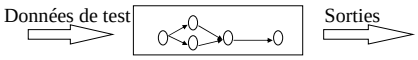
<u>En pratique :</u>

- prédictions approximatives (à cause des calculs flottants,…)
- formules inconnues (car programme = formule)
- oracle contenant des fautes

## Problème de la sélection des DT

A. Test fonctionnel : basé sur les spécifications


B. Test structurel : basé sur l'analyse du programme






## A. Test fonctionnel

A base d'un **modèle du programme issu des spécifications :**

- **Informelles** (test partitionnel, test aux limites, ...)
- **Semi-formelles** (cas d'utilisation, diagrammes de séquence, UML/OCL, graphes causes/effets…)
- **Formelles** (spéc. algébriques, machines B, systèmes de transitions IOLTS, …)



## B. Test structurel

A base d'un modèle **du code source du programme**

- modèle = représentation interne de la structure
- Utilisation importante de la **Théorie des Graphes**, notamment des techniques de couverture



## Le test structurel est indispensable (1)

Spécifications :
renvoie le produit de i par j

(i = 0, j = 0) --> 0
(i = 10, j = 100) -->1000
…

--> OK

```
prod(int i,int j )
 {
   int k ;
   if( i==2 )
      k := i << 1 ;
   else
     (faire i fois l 'addition de j)
   return k ;
 }
```

## Le test structurel est indispensable ! (2)

Spécifications :
renvoie le produit de i par j

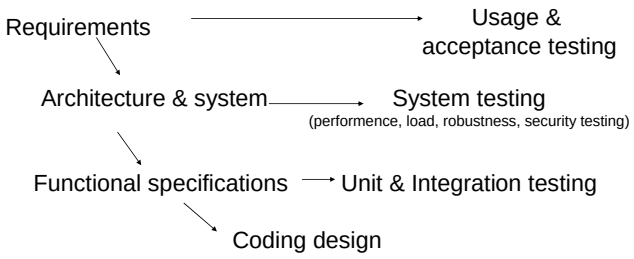
(i = 0, j = 0) --> 0
(i = 10, j = 100) -->1000
…

Faute non détectée par du test fonctionnel
patch -> `k := j << 1`

```
prod(int i,int j )
 {
   int k ;
   if( i==2 )
      k := i << 1 ;
   else
     (faire i fois l 'addition de j)
   return k ;
 }
```



## Software testing in the software developpement process

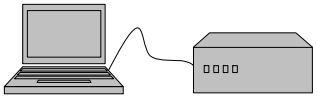





## Test natif / test croisé

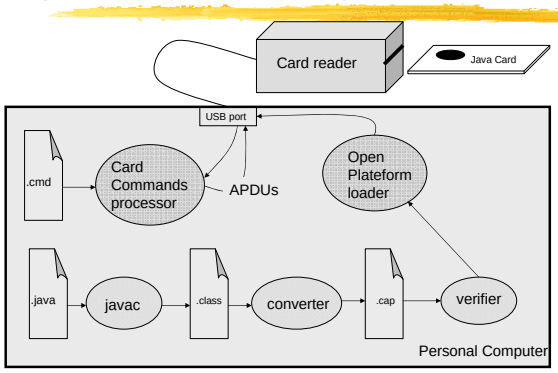
**Test natif :**
machine de test = machine de développement

**Test croisé :**
machine de test ≠ machine de développement

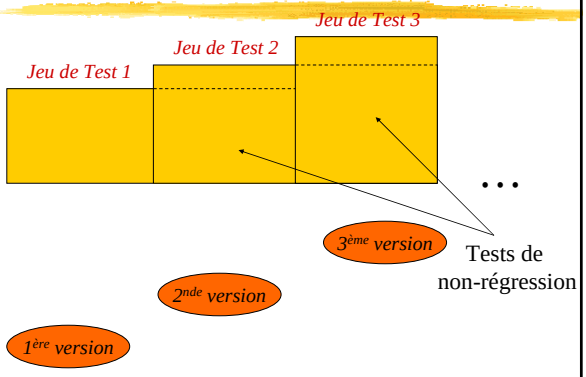




## Exemple de montage en test croisé : plateforme Java Card



## Tests de non-regréssion






## Bibliographie : quelques livres




## Bibliographie : quelques revues




## Et quelques sites intéressants...

Software Testing Online Resources `www.mtsu.edu/~storm/`

Software Testing Stuff `www.testdriven.com/`

Model-based Software Testing
`www.geocities.com/model_based_testing/`

Opensource ST `www.opensourcetesting.org/unit_java.php`

Tao Xie's homepage
`www.cs.washington.edu/homes/taoxie`

# TEST UNITAIRE AVEC CUNIT



## Units of language

- Functions in C
- Task in ADA
- Methods or classes in C++ and Java
- Predicates in Prolog
- ...



## Unit testing frameworks

- JUnit, Parasoft's Jtest for Java
- CUnit, CTC++, IBM Rational Test Real Time for C
- Parasoft's C++Test, Cantata++… for C++

…

http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks



## Unit testing: atomic operation

- Select test input variable values
- Define expected output values (a.k.a. oracle)
- Code a test script and register it in a database
- Run the test !

## Cunit (http://cunit.sourceforge.net/) -- 1

- Lightweight system for **writing, managing and running unit test** for C programs
- Built as a **static library** which is linked with the user's test script
- provides a rich set of **assertions** for testing common data types.



## Cunit (http://cunit.sourceforge.net/) -- 2

- **Automated**
  Output to XML file, Non-interactive
- **Basic**
  Flexible programming interface, Non-interactive
- **Console**
  Console interface (ANSI C), Interactive
- **Curses**
  Graphical interface (Unix only), Interactive



## Example

```
typedef unsigned short ush;

ush gcd(ush u, ush v)
{
  while( u > 0 )
    {
      if ( v > u )
        {
          u = u + v;
          v = u - v;
          u = u * v;
        }
      u = u - v;
    }
  return v;
}
```

- Select a test input: (28, 24)
- Define expected output: 4
- Define another test input : (24,28)
- Expect output: 4
- Script and run the test !



## Typical test script: test_gcd.c

```
#include "Basic.h"

void test_gcd(void) {  CU_ASSERT(4 == gcd(24, 28));
                       CU_ASSERT(4 == gcd(28, 24)); }
main() {
  CU_pSuite pSuite = NULL;

  if (CUE_SUCCESS != CU_initialize_registry())
           return CU_get_error();

  pSuite = CU_add_suite("Suite_1", 0, 0);
  if (!pSuite) {  CU_cleanup_registry();
                  return CU_get_error(); }

  if ((!CU_add_test(pSuite, "test of fprintf()", test_gcd))
              {  CU_cleanup_registry();
                 return CU_get_error(); }

  CU_basic_run_tests();
  CU_cleanup_registry();

  return CU_get_error(); }
```

## Makefile

```
CUnitHeaders=/c/CUnit-2.1-0/include/CUnit
CUnitLibs=/c/CUnit-2.1-0/lib

test_gcd: gcd.o test_gcd.o
  $(CC) $^ -L$(CUnitLibs) -lcunit -o $@

gcd.o: gcd.c
  $(CC) -c -D static= gcd.c

test_gcd.o: test_gcd.c
  $(CC) -c -I$(CUnitHeaders) test_gcd.c

clean:
  $(RM) gcd.o test_gcd.o test_gcd.exe
```

## Let's play, now !

- Download and install Cunit (README)
  requires Jam, MinGW on windows
  http://cunit.sourceforge.net
- Code gcd.c, test_gcd.c and Makefile
- Run and explain...



## JUnit for Java

**http://www.irisa.fr/lande/gotlieb/resources/JUnit_Tutorial.mp4**



## CRITERES DE TEST

## Représentations internes

Abstractions de la structure d'un programme

- **Graphe de Flot de Contrôle** [GFC]

- **Graphe Def/Use** [GDU]

[- Program Dependence Graph, ...]



## Graphe de flot de contrôle (GFC)

Graphe orienté et connexe `(N,A,e,s)` où

N : ens. de sommets =
bloc d'instructions exécutés en séquence
E : relation de N x N =
débranchement possible du flot de contrôle

e : sommet "d'entrée" du programme

s : sommet de sortie du programme

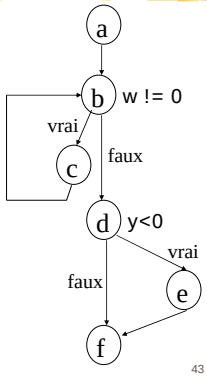


## Graphe de flot de contrôle (GFC) : exemple

```
double P(short x, short y) {
    short   w = abs(y) ;
    double z  = 1.0 ;

    while ( w != 0 )
       {
         z  = z * x ;
         w = w - 1 ;
       }

    if ( y<0 )
        z = 1.0 / z ;
  return(z) ;
  }
```



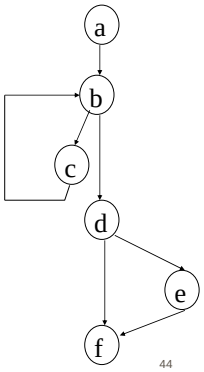



## Critère : *tous_les_sommets* | *toutes_les_instructions*

Motivation : couvrir toutes les instructions du programme au moins une fois

Déf: Un ensemble C de chemins du GFC `(N,A,e,s)` satisfait ***tous_les_sommets*** ssi $\forall n \in N, \exists C_i \in C$ tq n est un sommet de $C_i$

Exemple : Ici un seul chemin suffit
`a-b-c-b-d-e-f` [6/6 sommets]

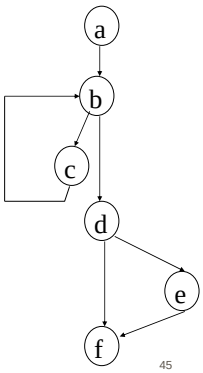


## Critère structurel : *tous_les_arcs* | *toutes_les_décisions*

Motivation : couvrir toutes les prises de décision au moins une fois

Déf : Un ensemble C de chemins du GFC (N,A,e,s) satisfait ***tous_les_arcs***
ssi $\forall a \in A, \exists C_i \in C$
tq a est un arc de $C_i$

Exemple : Ici 2 chemins sont nécessaires
`a-b-c-b-d-e-f` [6/7 arcs]
`a-b-d-f` [3/7 arcs]

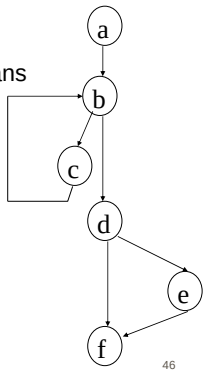





## Critère structurel : *tous_les_chemins_simples*

Motivation : couvrir toutes les chemins d'exécution, sans itérer plus d'une fois dans les boucles

Exemple : Ici 4 chemins complets sont nécessaires
`a-b-d-f`
`a-b-d-e-f`
`a-b-c-b-d-f`
`a-b-c-b-d-e-f`



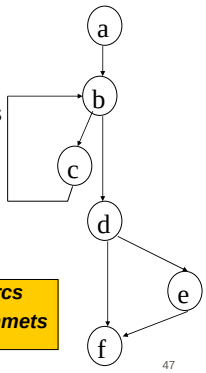



## Critère structurel : *tous_les_chemins*

Déf : Un ensemble C de chemins du GFC (N,A,e,s) satisfait ***tous_les_chemins***
ssi C contient tous les chemins de e à s

Ici, **impossible** car ∞ chemins et chemins non-exécutables !

***tous_les_chemins*** plus fort que ***tous_les_arcs***
***tous_les_arcs*** plus fort que ***toutes_les_sommets***



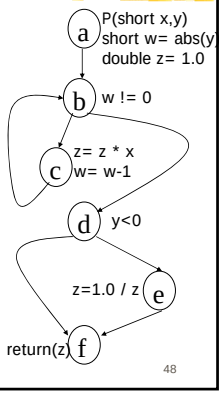



## Chemin sensibilisé : `exec(P,X)`

Principe:
X sensibilise un **unique** chemin du GFC

Déf: séquence de sommets du GFC, non nécessairement finie, empruntée lors de l'exécution de P avec X comme entrée

Exemples :
`exec(P,(0,0))` = `a-b-d-f`
`exec(P,(3,2))` = `a-b-(c-b)²-d-f`



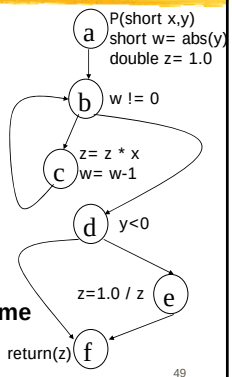
## Problème des chemins non exécutables

Soit c un chemin du GFC de P,
existence de X tq c=exec(P,X) ?

Ici, a-b-d-e-f est non-exécutable !

Weyuker 79
Déterminer si un sommet, un arc, ou un chemin du GFC est exécutable est indécidable dans le cas général

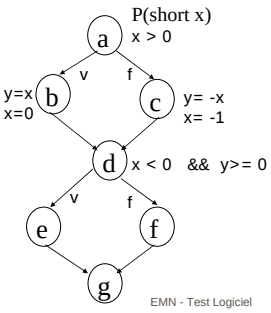
Idée de la preuve: réduction au **problème de l'arrêt d'une Machine de Turing**






## Exercice

Trouver tous les chemins non-exécutables de ce programme






## Measuring code coverage

- 3 distinct techniques
  - Instrumenting source code
    - + Easy to implement
    - + Powerful as everything regarding executions can be recorded
    - - Add untrusted code in trusted source code
  - Instrumenting binary code
    - + Do not modify source code
    - - Difficult to implement
  - Use a debugger
    - + Do not modify source code
    - - Specific to each compiler



## Condition / Décision dans un programme

**Condition** **(bool., expr. arith, ...)**

```
if( A && (B || C))
```

**Décision**
**(bool., expr. logique dans une structure de contrôle, ...)**

**Notation :** `Dec` **est la valeur de vérité de la décision**

## Critères de test liés aux décisions

```
if( A && (B || C))
```

1. Decision Criterion (DC) : `A=1,B=1,C=0 – Dec=1`
   `A=0,B=0,C=0 – Dec=0`

2. Condition Criterion (CC) : `A=1,B=1,C=0 – Dec=1`
   `A=0,B=0,C=1 – Dec=0`

3. Modified Condition/Decision Criterion (MC/DC)

4. Multiple Condition/Decision Criterion: $2^3$=8 cas de test



## Modified Condition/Decision Criterion (1)

Objectif : Démontrer l'action de chaque condition sur la valeur de vérité de la décision

```
if( A && (B || C))
```

Principe : pour chaque condition, trouvez 2 cas de test qui changent Dec lorsque toutes les autres conditions sont fixées

Ex : pour A `**A=0**, B=1,C=1 -- **Dec=0**`
`**A=1**, B=1,C=1 -- **Dec=1**`



## Modified Condition/Decision Criterion (2)

```
if( A && (B || C))
```

pour A
```
A=0, B=1,C=1  -- Dec=0
A=1, B=1,C=1  -- Dec=1
```

pour B
```
A=1, B=1,C=0  -- Dec=1
A=1, B=0,C=0  -- Dec=0
```

pour C
```
A=1, B=0,C=1  -- Dec=1
~~A=1, B=0,C=0  -- Dec=0~~
```

Ici, 5 cas de test suffisent pour MC/DC !



## Exercice : peut-on faire mieux ?

```
if( A && (B || C))
```

pour A
```
A= , B= ,C=   -- Dec=
A= , B= ,C=   -- Dec=
```

pour B
```
A= , B= ,C=   -- Dec=
A= , B= ,C=   -- Dec=
```

pour C
```
A= , B= ,C=  -- Dec=
A= , B= ,C=  -- Dec=
```

## Modified Condition/Decision Criterion (3)

Propriété : Si n = #conditions alors
couvrir MC/DC requiert au moins n+1 DT et au plus 2n DT

$$n+1 \leq \#\text{données de test} \leq 2*n$$

Conditions couplées : changer la valeur d'une condition modifie la valeur d'une autre

En l'absence de conditions couplées, le minimum (n+1) peut-être atteint



## Lien avec la couverture du code objet

Couvrir MC/DC $\Rightarrow$ couvrir les décisions du code objet

mais

Couvrir MC/DC $\not\Leftarrow$ couvrir les décisions du code objet

Couvrir les chemins du code objet $\Rightarrow$ couvrir MC/DC

mais

Couvrir les chemins du code objet $\not\Leftarrow$ couvrir MC/DC



## From the Galileo development standard

| *Structural coverage* | DAL A | DAL B | DAL C | DAL D | DAL E |
|---|---|---|---|---|---|
| Statement coverage (source code) | N/A | **100%** | **100%** | **90%** | N/A |
| Statement coverage (object code) | **100%** | N/A | N/A | N/A | N/A |
| Decision coverage (source code) | N/A | **100%** | N/A | N/A | N/A |
| Modified Condition & Decision Coverage (Source code) | **100%** | N/A | N/A | N/A | N/A |



## Représentations internes

Abstractions de la structure d'un programme

- **Graphe de flot de contrôle** [GFC]

- **Graphe Def/Use** [GDU]
  GFC + décoration des sommets/arcs avec infos définitions/utilisations sur les variables

[- Program Dependence Graph, …]

## Graphe Def/Use (GDU) – [Rapps & Weyuker 85]

A chaque **sommet** j est associé

- ***def(j)*** : {v ∈ Var(P) | v est définie en j }
- ***c-use(j)*** : {v ∈ Var(P) | v est utilisée en j dans un calcul et ∃k ≠ j tel que v ∈ def(k)}

A chaque **arc** j-k issu d'une décision est associé

- ***p-use(j,k)*** : {v ∈ Var(P) | v est utilisée dans le prédicat j et conditionne l'exécution de k}

Hypothèses : Toute variable utilisée possède une définition et à chaque définition correspond au moins une utilisation

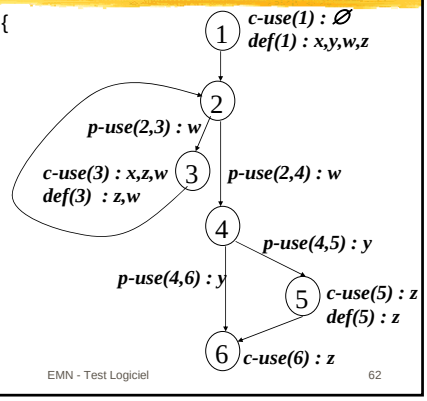


## Graphe Def/Use : exemple

```
double P(int x, int y) {
    w = abs(y) ;
    z  = 1.0 ;

    while ( w != 0 )
      {
        z  = z*x ;
        w = w-1 ;
       }

    if ( y<0 )
        z = 1.0 / z ;
   return(z) ;
   }
```






## Chemin sans définition (def-clear path)

`i-`$n_1$`-…-`$n_m$`-j` est un **chemin sans déf.** de x de `i` à `j`
ssi ∀k ∈ {1,..,m}, x ∉ *def*($n_k$)

Soient `i` un sommet d'un GDU (N,A,e,s) et x ∈ *def(i)*

`dcu(x,i)`: {**`j ∈ N`** | x ∈ *c-use(j)* et ∃un chemin sans déf. de x de `i` à `j`}

`dpu(x,i)`: {**`j-k ∈ A`** | x ∈ *p-use(j, k)* et ∃un chemin sans déf. de x de `i` à `j`}

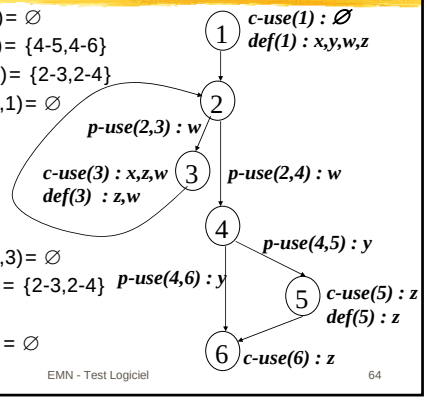


## dcu(x,j) et dpu(x,j) : exemples

dcu(x,1)= {3}, dpu(x,1)= ∅
dcu(y,1)= ∅, dpu(y,1)= {4-5,4-6}
dcu(w,1)= {3}, dpu(w,1)= {2-3,2-4}
dcu(z,1)= {3,5,6}, dpu(z,1)= ∅

dcu(z,3)= {3,5,6}, dpu(z,3)= ∅
dcu(w,3)= {3}, dpu(w,3)= {2-3,2-4}

dcu(z,5) = {6}, dpu(z,5) = ∅



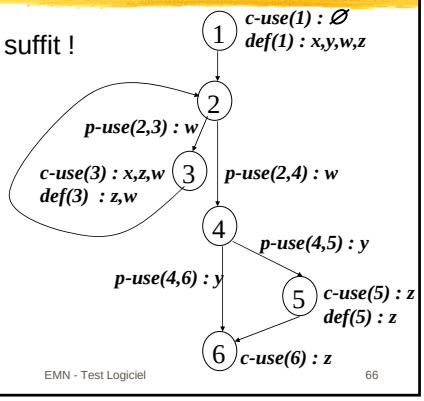
## Critère structurel : *toutes_les_définitions*

Déf : Un ensemble C de chemins du GDU (N, A, e, s) satisfait ***toutes_les_définitions*** ssi
$\forall n \in N$ et $\forall x \in def(n)$,
$\exists C_i \in C$ tel que
$C_i$ inclut un chemin sans déf. de x
de n à un élément de dcu(x,n) ou dpu(x,n)



## Couverture de *toutes_les_définitions*

Ici, un seul chemin suffit !
1-2-3-2-4-5-6






## Critère structurel : *toutes_les_utilisations*

Déf : C satisfait ***toutes_les_utilisations***
ssi $\forall n \in N$ et $\forall x \in def(n)$, $\exists C_i \in C$
tq $C_i$ inclut un chemin sans déf. de x de n à
**tous les éléments** de dcu(x,n) et dpu(x,n)

Exemple : Ici, trois chemins sont nécessaires !
par exemple : 1-2-4-6, 1-2-3-2-4-6 et 1-2-(3-2)$^2$-4-5-6

***toutes_les_utilisations*** est plus fort que ***toutes_les_définitions***



## Chemin définition/utilisation p.r.à. x (du-path w.r.t. x)

Déf: p = i-..-j-k est
un **chemin définition/utilisation par rapport à** x ssi :

1. $x \in def(i)$

2a. soit $x \in c\text{-}use(k)$ alors p est un chemin sans déf. de x de i à k et p se trouve sur un chemin simple

2b. soit $x \in p\text{-}use(j,k)$ alors p est un chemin sans déf. de x de i à j et p se trouve sur un chemin simple

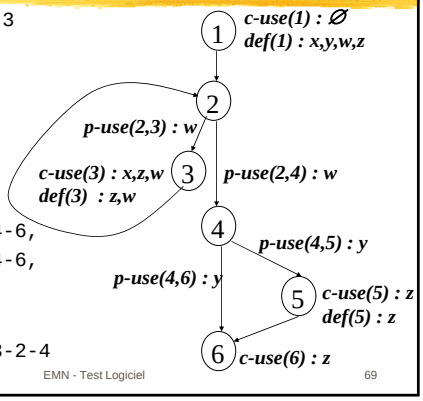
## Chemin définition/utilisation : exemples

du_path w.r.t. x : 1-2-3

du_paths w.r.t. y :
1-2-4-5, 1-2-4-6,
1-2-3-2-4-5,
1-2-3-2-4-6

du_paths w.r.t. z :
1-2-3,1-2-4-5,1-2-4-6,
3-2-3,3-2-4-5,3-2-4-6,
5-6

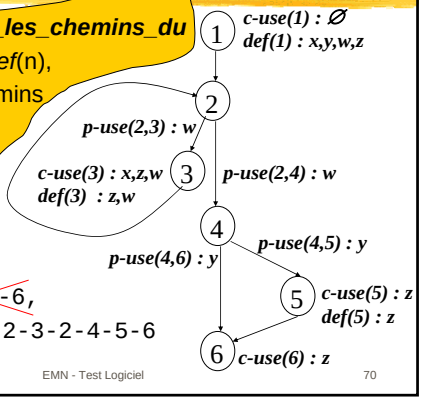
du_paths w.r.t. w :
1-2-3,1-2-4,3-2-3,3-2-4






## Critère structurel : *tous_les_chemins_du*

Déf: C satisfait ***tous_les_chemins_du*** ssi ∀n∈ N et ∀x∈ *def*(n), C inclut tous les chemins définition/utilisation par rapport à x




Ici, 5 chemins !
1-2-4-6, 1-2-4-5-6,
1-2-(3-2)$^{2}$-4-6,1-2-3-2-4-5-6
1-2-3-2-4-6



## Relation entre les critères (“*est plus fort que*”)

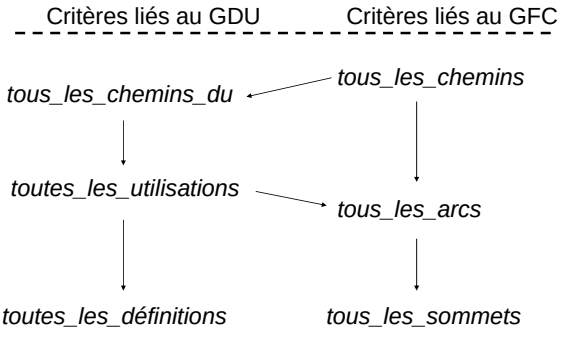
$C_1$ subsume $C_2$ (noté $C_1 \Rightarrow C_2$) ssi
∀GDU, ∀ensemble de chemins P qui satisfait $C_1$,
P satisfait aussi $C_2$

Propriétés :
- relation transitive ( $C_1 \Rightarrow C_2$ et $C_2 \Rightarrow C_3$ alors $C_1 \Rightarrow C_3$ )
- définit un ordre partiel entre les critères

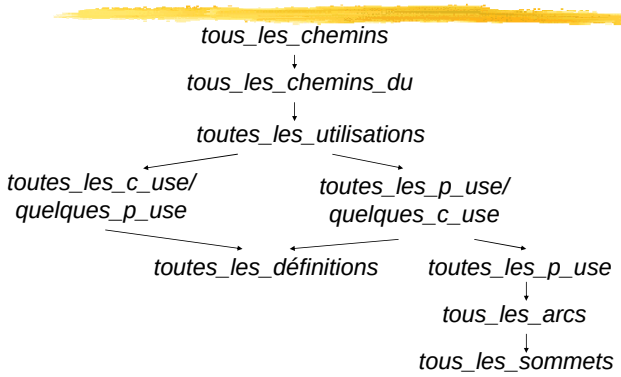
*Ex: tous_les_chemins_du* subsume *toutes_les_utilisations*



## Les liens entre critères liés au GDU et au GFC



## Et en plus… [Rapps & Weyuker 1985]






## Quelques outils disponibles...

Freeware:

- Gcov / gcc `www.netadmintools.com/html/1gcov.man.html`
- Hansel (Oregon University) `hansel.sourceforge.net`
- Quilt / Junit `quilt.sourceforge.net`

Outils commerciaux :

- IPL Cantata++ / ADATest `www.iplbath.com/products`
- IBM Rational Test Realtime `www-306.ibm.com/software/awdtools/test/realtime`
- LDRA Testbed `www.ldra.co.uk/pages/testbed.htm`



## Synthesis

⌘ Nowadays, Software Testing
1. relies on solid theoretical foundations
2. is always tool-supported

⌘ Code-based testing / Model-based testing are complementary techniques

⌘ Measuring code coverage is desirable to assess the test quality

# Test Logiciel 2

EMN – GIPAD/GSI
2010-2011

Arnaud Gotlieb
Celtique, INRIA Rennes



## Plan du cours d'aujourd'hui

1. Techniques de test élémentaires
2. Test de logiciel orientés-objet
3. Model-Based Testing



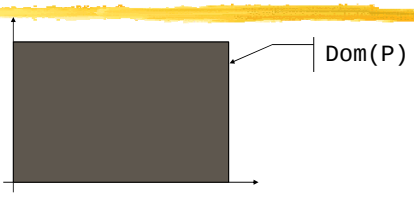
## Génération automatique de données de test (1)

⌘ Techniques élémentaires

- Test exhaustif
- Test par échantillions
- Test aléatoire



## Test exhaustif



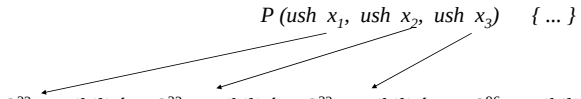

- Parcours exhaustif du domaine d'entrée des programmes

- Séléction de toutes les données de test possible

- Equivalent à une preuve de correction du programme

## Test exhaustif : limitation et intérêt

- Mais, généralement impraticable !



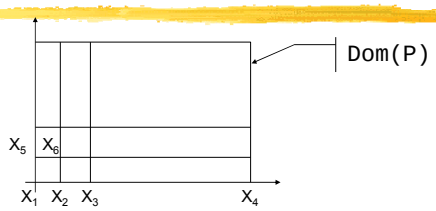

$2^{32}$ possibilités × $2^{32}$ possibilités × $2^{32}$ possibilités = $2^{96}$ possibilités

- Estimation de la taille de l'espace de recherche face à un objectif de test

ex d'objectif : atteindre une instruction dans le programme



## Test par échantillion




Forme faible du test exhaustif : **test par échantillion**

Exemples :

{0, 1, 2, $2^{32}$-1} pour un ush

{NaN, -INF,-3.40282347e+38, -1.17549435e-38, -1.0, -0.0,.. } pour un float (IEEE 754)



## Test aléatoire

Loi de probabilité **uniforme** sur le domaine d'entrée

(i.e. chaque DT est équiprobable)

- Utilisation de générateurs de valeurs **pseudo-aléatoires**

- Nécessité de disposer d'un **oracle automatique**

- Condition d'arrêt à déterminer (nombre de DT à générer, critère de test couvert)

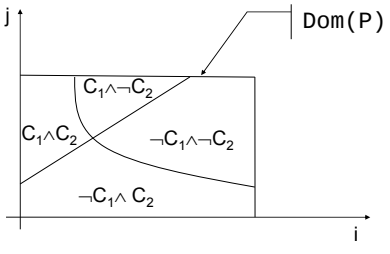


## Critère de sélection C

- Procédé de sélection de données de test
- Induit parfois une « partition » sur le domaine d'entrée (ex : chemins de P)

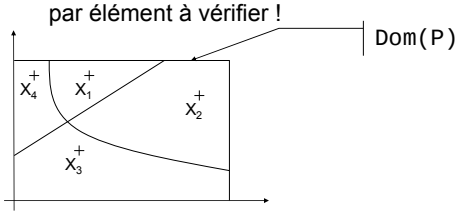
```
P(int i,int j)
{
if( C1 )
else ...

if( C2)
else ...
}
```



## Couverture **déterministe** d'un critère C

Génération d'une seule donnée de test
par élément à vérifier !

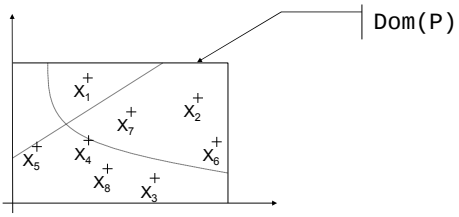



Repose sur une hypothèse d'uniformité :
une seule donnée de test suffit pour tout le sous-domaine



## Couverture **probabiliste** du critère C

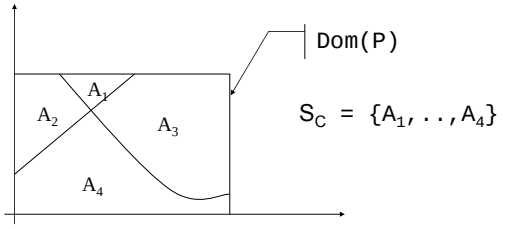
Génération aléatoire de données de test
selon une distribution de probabilités
(ou profil d'entrée)






## Efficacité du test aléatoire pour couvrir un critère ?

$p\{x \in A\}$ : probabilité qu'une DT aléatoire x couvre l'élément A




$S_C = \{A_1, .., A_4\}$

Ici $p\{x \in A_1\} < p\{x \in A_2\} < p\{x \in A_3\} < p\{x \in A_4\}$ donc le test aléatoire couvre « mieux » $A_4$ que $A_1$, ce qui n'est pas satisfaisant ! → Test statistique structurel [Thévenod 90]

## Génération automatique de données de test (2)

- Techniques de test fonctionnel
  - Analyse partitionnelle
  - Test aux limites
  - Test à partir de Graphes Cause/Effet

## Un test pour soi-même

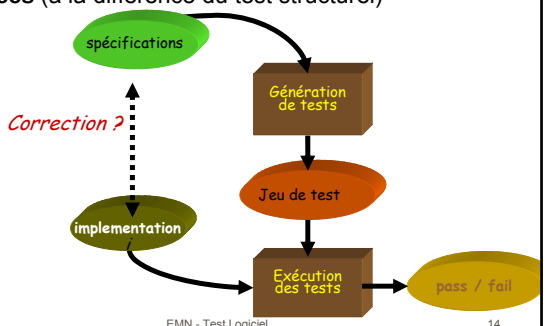
« The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isocele, or equilateral » [Myers 79]

Ecrire un jeu de test que vous sentez être adéquate pour tester ce programme



## Quelques remarques générales sur le test fonctionnel

- Génération de données de test + **oracle pour ces données** (à la différence du test structurel)






## Analyse partitionnelle (1)

- Technique de test majoritairement employé en présence de spécifications informelles

- Idée : Découper le domaine d'entrée en classes d'équivalence : ensembles d'entrées aboutissant au même comportement fonctionnel

- Soit D le domaine d'entrée, $\{C_1,..,C_n\}$ est une partition de $D$ ssi :

$$\bigcup_{i=1}^{n} C_i = D \quad \text{et} \quad \forall i, j \in 1..n, C_i \cap C_j = \varnothing$$



## Analyse partitionnelle (2)

- Permet au testeur de passer d'un nombre infini de données à un nombre fini et limité de données de test

- Division du domaine d'entrée en classes valides et invalides

- Choix d'un représentant dans chacune des classes d'équivalence

- Existence de règles générales de découpage (non uniques)
  - Entier : a ≤ 0, a> 0 ou a<0, a=0,a>0
  - Intervalle : ∅, [a,b] où a < b < ∞, [-∞,+∞]
  - Ensemble : ∅, {a}, {a,b}, {a,b,c}, ...
  - Arbre bin. : nil, a , ...

## Partitionnement unidimensionnel

- Considérer une variable à la fois. Ainsi, chaque variable d'entrée du programme conduit à une partition du domaine d'entrée. Cette méthode est appelée *partitionnement unidimensionnel*
- Ce type de partitionnement est le plus utilisé (et le plus simple à mettre en oeuvre). Néanmoins, il n'exerce pas certaines combinaisons pouvant conduire à détecter des fautes



## Partitionnement multidimensionnel

- Considérer le domaine d'entrée comme le produit Cartésien des domaines de chaque variable d'entrée. Cette méthode conduit à la création d'une unique partition. Cette méthode est appelée *partitionnement multidimensionnel.*
- Grand nombre de cas de test, difficiles à gérer manuellement. Cependant, bien que certaines classes créées puissent être invalides, cette méthode offre une grande variété de cas de test.



## Partitionnement unidimensionnel : example

PGCD(ENTIER a, ENTIER b)

Classes valides
C1 : a = 0,
C2 : a > 0,
D1 : b = 0,
D2 : b > 0

*Problème :*
*a = b = 0 appartient à C1 et à D1*

Classes invalides
C3 : a < 0,
D3 : b < 0



## Partitionnement multidimensionnel : example

PGCD(ENTIER a, ENTIER b)

Classes valides
C1 : a = b = 0
C2 : a = b, a > 0, b > 0
C3 : a >= 0, b >= 0, a ≠ b

Classes invalides
C4 : a < 0, b < 0
C5 : a < 0, b > 0
C6 : a > 0, b < 0

## Erreurs aux limites

L'expérience montre que les programmeurs font souvent des erreurs aux bornes des classes d'équivalence

Par exemple, si le programme P est sensé calculer une fonction F1 lorsque x<0 est vraie et calculer F2 sinon et qu'il calcule F1 uniquement lorsque x< -1 alors P contient une faute qui peut ne pas être révelée par une analyse partitionnelle (x ≤ 0, x > 0 ou x<0, x=0, x>0).



## Test aux limites

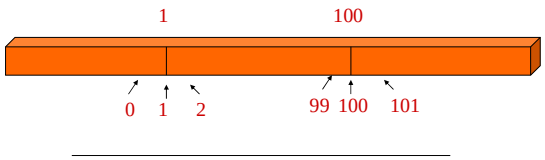
- Idée : Identification des données de test « limites » définies par la spécification suite à l'analyse partitionnelle

- Sélection de données de test aux bornes valides et invalides

- Formalisation de la notion de limite difficile dans le cas général ➔ heuristiques de choix des valeurs



## Test aux limites de l'intervalle [1,100]




Mais combinatoire importante en présence de vecteurs d'entrées ➔ définition de voisinages discrets

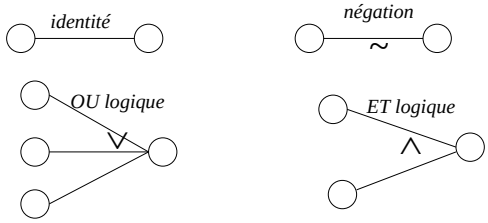
Ecrire des définitions possibles de voisinages discrets d'1 point en dimension 2, puis généralisez en dimension n



## Graphes cause-effet

- Idée : Réseau logique qui relie les effets d'un programme (sorties) à leurs causes (entrées)

4 types de symboles :



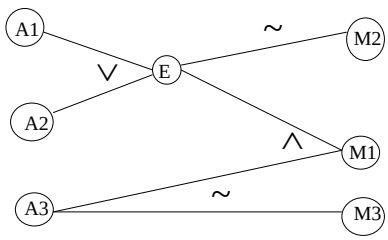
## Graphes cause-effet : example (1)

**« Le caractère dans la colonne 1 doit être un A ou un B. Dans la colonne 2, il doit y avoir un chiffre. Si le premier caractère est erroné, le message d'erreur X12 doit être imprimé. Si dans la colonne 2 nous n'avons pas de chiffre, alors le message X13 est imprimé »**

Analyse des causes :
A1 : le caractère de la première colonne est A
A2 : le caractère de la première colonne est B
A3 : le caractère de la deuxième colonne est un chiffre
Analyse des effets :
M1 : programme ok
M2 : message X12
M3 : message X13



## Graphes cause-effet : example (2)




Analyse des causes :
A1 : le car de col 1 est A
A2 : le car de col 1 est B
A3 : le car de col 2 est un chiffre
Analyse des effets :
M1 : programme ok
M2 : message X12
M3 : message X13



## Graphes cause-effet : génération de cas de test

- Causes et effets sont vus comme des variables booléennes

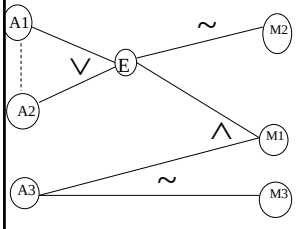
- Génération automatique d'une matrice de décision (matrice booléenne, Vrai : 1 Faux : 0) à partir du graphe cause-effet

- Elimination des combinaisons invalides : «comportements inobservables» (représentées par une contrainte sur le graphe cause effet)

  Par ex: A1 et A2 vrais où A1 : le car de col 1 est A
  A2 : le car de col 1 est B



## Matrice de décision




| Cause -effet | A1 | A2 | A3 | M1 | M2 | M3 |
|---|---|---|---|---|---|---|
| CT 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| CT 2 | 0 | 0 | 1 | 0 | 1 | 0 |
| CT 3 | 0 | 1 | 0 | 0 | 0 | 1 |
| CT 4 | 0 | 1 | 1 | 1 | 0 | 0 |
| CT 5 | 1 | 0 | 0 | 0 | 0 | 1 |
| CT 6 | 1 | 0 | 1 | 1 | 0 | 0 |

## Génération de cas de test concrets

| | A1 | A2 | A3 | M1 | M2 | M3 | |
|---|---|---|---|---|---|---|---|
| CT 1 | 0 | 0 | 0 | 0 | 1 | 1 | *« Ca », X12, X13 émis* |
| CT 2 | 0 | 0 | 1 | 0 | 1 | 0 | *« C1 », X12 émis* |
| CT 3 | 0 | 1 | 0 | 0 | 0 | 1 | *« Bz », X13 émis* |
| CT 4 | 0 | 1 | 1 | 1 | 0 | 0 | *« B3 », ok* |
| CT 5 | 1 | 0 | 0 | 0 | 0 | 1 | *« Ak », X13 émis* |
| CT 6 | 1 | 0 | 1 | 1 | 0 | 0 | *« A4 », ok* |

Analyse des causes :
A1 : le car de col 1 est A
A2 : le car de col 1 est B
A3 : le car de col 2 est un chiffre

Analyse des effets :
M1 : programme ok
M2 : message X12
M3 : message X13



## Réduction de la combinatoire

- Si n = nombre de causes, m = nombre d'effets
  alors matrice de décision de taille $2^n$ x (n+m)
  dans le pire cas → ne peut-être gérée automatiquement si n est grand (> 30)

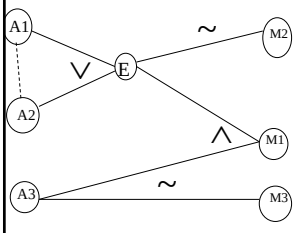
- Comment réduire la combinatoire ?

  Idée : Eliminer les CT redondants en identifiant ceux ayant des comportements similaires (mêmes effets) bien qu'ayant des causes différentes

  Ex: CT3 et CT5



## Technique de réduction sur le graphe cause-effet




*Par construction,*
*(A1 = 1, A2 = 1)*
*(A1 = 1, A2 = 0)*
*(A1 = 0, A2 = 1)*
*conduisent à E = 1 et aux mêmes effets*

*Seul un des trois est choisi*

*Règles similaires pour les autres connecteurs*



## Exercice :

**The Sendfile command.**
In a given network, the sendfile command is used to send a file to a user on a different file server. The sendfile command takes three arguments: the first argument should be an existing file in the sender's home directory, the second argument should be the name of the receiver's file server, and the third argument should be the receiver's userid. If all the arguments are correct, then the file is successfully sent ; otherwise the sender obtains an error message.

1. Modéliser cette commande avec un graphe cause-effet
2. Générer une matrice de décision

## Plan du cours d'aujourd'hui

1. Techniques de test élémentaires
2. Test de logiciel orientés-objet
3. Model-Based Testing



## Langages orientés objet / langages impératifs

- **Langages impératifs** : fonctions qui transforment des entrées en sortie
- **Langages OO** : objets qui modélisent un problème et collaborent entre eux pour trouver une solution

-> les **techniques de test** utilisées pour les programmes impératifs doivent être **adaptées** pour prendre en compte les spécificités des langages OO



## Langages OO et test : un aperçu des difficultés

- Encapsulation des données
  - Comment connaître les états internes des objets dont les attributs sont privés ?
- Dépendances fortes entre certaines classes
  - Dans quel ordre tester les classes ?
  - Que tester dans une classe qui hérite d'une autre ?
- Polymorphisme
  - Plusieurs parties de code peuvent "se cacher" derrière un appel de méthode polymorphe



## References

[1] *Testing Object-oriented Systems, Models, Patterns, and Tools*, Robert V. Binder, 2000, Addison-Wesley

[2] *A Practical Guide to Testing Object-Oriented Software*, John D. McGregor et David A. Sykes, 2001, Addison-Wesley

Partie 1

# Test de classe



## Principe du test de classe

- Unité = la classe
- Augmente la confiance que l'on a dans l'implémentation des unités
  - Vérifie que l'implémentation d'une classe répond à sa spécification
  - -> si l'implémentation de la classe est correcte, chaque instance de la classe devrait fonctionner correctement
- Une fois les classes testées séparément : les erreurs lors de l'intégration de la classe sont plus probalement liées à un interfaçage incorrect des unités qu'à une implémentation incorrecte de cette classe



## Quelles classes tester ?

- Pour chaque classe, décider si on la teste :
  - comme une unité
  - comme un composant d'une partie plus large du système
- Facteurs de décision :
  - Rôle de la classe dans le système, degré de risque associé
  - Complexité de la classe
    - Nombres d'états, d'opérations, associations avec les autres classes
  - Effort à fournir pour tester la classe
- Illustration : si la classe fait partie d'une librairie, la tester intensivement même si tester coûte cher / prend du temps



## Quand tester ?

- Avant l'intégration de la classe
- Test de régression après tout changement de code
- Modification de code liée à un bug : des cas de test doivent être ajoutés ou modifiés pour détecter ce bug lors des tests futurs

## Comment tester ?

- Construction d'un script de test qui exécute les cas de test pour une classe et collecte les résultats des exécutions
- Un script de test :
  - crée des instances de classe et un environnement ;
  - envoie un message à une instance ;
  - vérifie le résultat de ce message : valeur retournée, état de l'instance, modification des paramètres d'entrée, valeur des attributs statiques de la classe ;
  - Si langage avec gestion mémoire, détruit les instances créées.

## Comment choisir des cas de test pertinents ?

- Le développeur est susceptible de faire des erreurs d'interprétation de la spécification
- N'utiliser que le code pour identifier les cas de test risque de propager ces erreurs
- Méthode préconisée :
  - créer des cas de test à partir de la spécification ;
  - augmenter la suite de tests selon l'implémentation.

Remarque : pour le reste de cette partie 1, nous nous restreignons au test de classes dites primitives, c'est à dire indépendantes des autres classes.



## Utilisation des pre- et postconditions

- Objectif : définir des cas de test pour une méthode à partir de ses pre- et postconditions
- Idée générale :
  - déterminer toutes les combinaisons possibles de situations pour lesquelles une précondition peut être vérifiée et une postcondition peut être atteinte ;
  - créer des cas de test pour couvrir ces situations avec des valeurs spécifiques, incluant des valeurs typiques et des valeurs aux limites ;
  - dans le cas d'une programmation de type défensive, prévoir aussi des cas où les préconditions ne sont pas respectées



## Utilisation des preconditions

| Precondition en OCL | Contribution |
|---|---|
| true | (true,Post) |
| *pre1* | (*pre1*,Post)<br>(not *pre1*,Exception)• |
| not *pre1* | (not *pre1*,Post)<br>(*pre1*,Exception) • |
| *pre1* and *pre2* | (*pre1* and *pre2*,Post)<br>(not *pre1* and *pre2*, Exception) •<br>(*pre1* and not *pre2*, Exception) •<br>(not *pre1* and not *pre2*, Exception) • |
| *pre1* or *pre2* | (*pre1*,Post)<br>(*pre2*,Post)<br>(*pre1* and *pre2*, Post)<br>(not *pre1* and not *pre2*, Exception) • |

• Dans le cas d'une programmation défensive

## Utilisation des preconditions (suite)

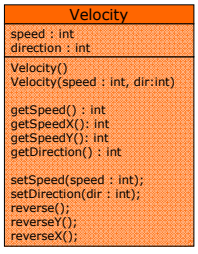
| Precondition en OCL | Contribution |
|---|---|
| *pre1* xor *pre2* | (*pre1* and not *pre2*, Post)<br>(not *pre1* and *pre2*, Post)<br>(*pre1* and *pre2*, Exception) •<br>(not *pre1* and not *pre2*, Exception) • |
| *pre1* implies *pre2* | (not *pre1*,Post)<br>(*pre2*,Post)<br>(not *pre1* and *pre2*, Post)<br>(*pre1* and not *pre2*, Exception) • |
| If *pre1* then *pre2*<br>else *pre3* endif | (*pre1* and *pre2*,Post)<br>(not *pre1* and *pre3*,Post)<br>(*pre1* and not *pre2*, Exception) •<br>(not *pre1* and not *pre3*, Exception) • |

• Dans le cas d'une programmation défensive

## Utilisation des postconditions

| Postconditions en OCL | Contribution |
|---|---|
| *post1* | (Pre,*post1*) |
| *post1* and *post2* | (Pre, *post1* and *post2*) |
| *post1* or *post2* | (Pre, *post1*)<br>(Pre, *post2*)<br>(Pre, *post1* and *post2*) |
| *post1* xor *post2* | (Pre, *post1* and not *post2*)<br>(Pre, not *post1* and *post2*) |
| *post1* implies *post2* | (Pre, not *post1* or *post2*) |
| If *post1* then *post2*<br>else *post3* endif | (Pre and ◘, *post2*)<br>(Pre and not ◘, *post3*)<br>avec ◘ une condition qui rend *post1* vraie au moment où s'applique la postcondition |

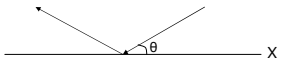
## Exercice: bâtir des cas de test pour la classe Velocity, dans le cadre d'une Programmation Défensive




*Exemple extrait de [2]*

Velocity::setDirection(dir:int)
pre : 0<=dir and dir<360
post : direction=dir and speed=speed@pre

Velocity::reverseX()
pre : true
post : speed = speed@pre and direction =
if direction@pre<=180
then (180 – direction@pre)
else (540-direction@pre)




## Critères de test basés sur le flot de contrôle

⌘ Critères pour le test de programmes impératifs sont applicables

- Couverture de toutes les instructions
- Couverture de toutes les branches
- Etc.

⌘ Mais, des critères spécifiques et mieux adaptés peuvent être définis

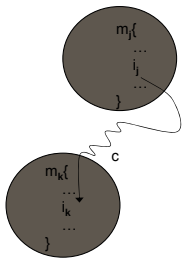
## Exemple d'un critère de test basé sur le flot des données

- Soient $m_j$ et $m_k$ des méthodes appelées par la méthode m
- Soient :

  $v$ : variable d'instance

  $i_j$ : instruction de la méthode $m_j$ qui définit $v$

  $i_k$ : instruction de la méthode $m_k$ qui utilise $v$




- Le critère est couvert si :

  $\forall (v, i_j, i_k)$ faisable, $\exists$ un cas de test pour m qui sensibilise un chemin $c$ entre $i_j$ et $i_k$ sans redéfinition de v

## Partie 2

# Test d'intégration



## Notion de bouchon de test

- Bout de code sans fonctionnalité qui vise à remplacer un code fonctionnel
- Utile lorsque le code fonctionnel n'est pas disponible, ou en test d'intégration pour tester unitairement les méthodes

  Ex: int m(int a, int b) {return 0;} à la place de

  int m(int a, int b) {if(…) return(f(a)) else return(f(b));…}

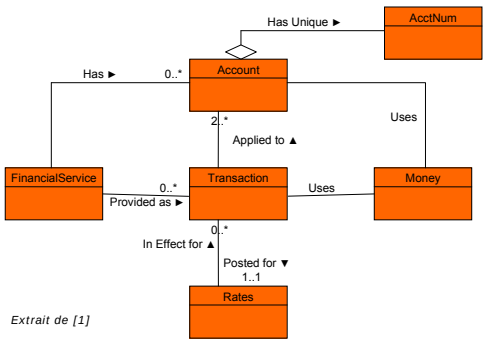
- L'utilisation de bouchons est une pratique courante, en test croisé de logiciels embarqués (cibles non dispo.) et en test de logiciels OO (composants non dispo.)



## Analyse de dépendances

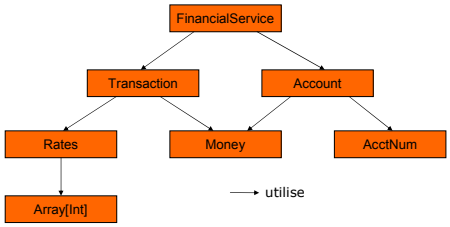
- Exemples de dépendances entre classes :
  - Invocation de méthode de B dans A
  - Objets B utilisés en tant que paramètres de méthodes de A
  - Héritage (cf partie 3)
- ➔ Créer un graphe de dépendances

## Analyse de dépendances: diagramme de classes




*Extrait de [1]*

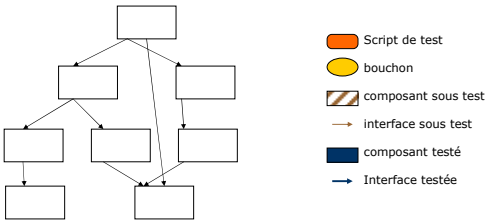
## Analyse de dépendances: graphe de dépendances





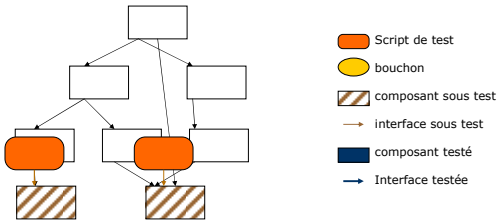

## Intégration Bottom-Up

- Méthode d'intégration recommandée pour les programmes objets
- L'intégration débute par les composants qui sont les plus indépendants

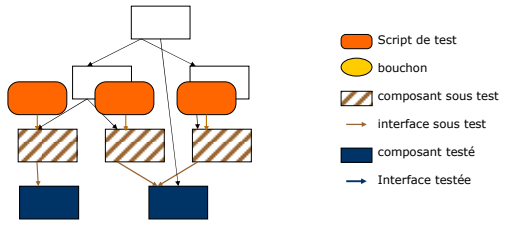





## Intégration Bottom-Up

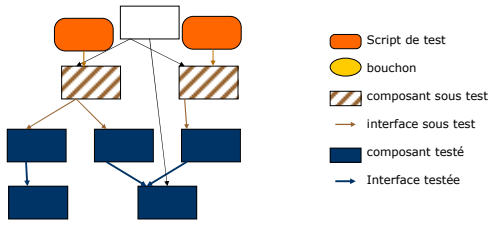


## Intégration Bottom-Up



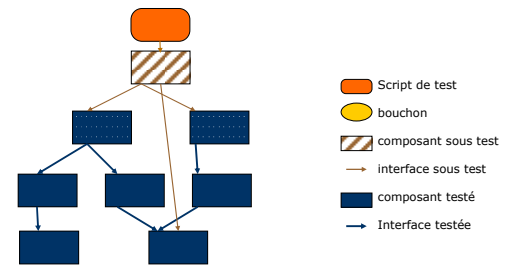



## Intégration Bottom-Up



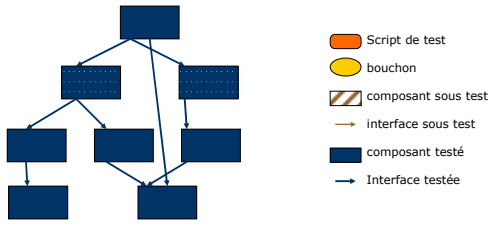



## Intégration Bottom-Up






## Intégration Bottom-Up



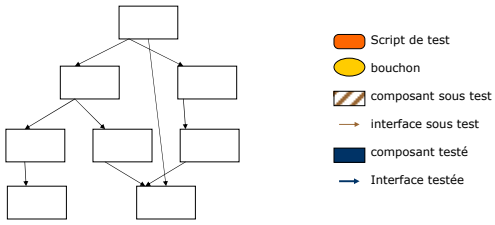
## Intégration Bottom-Up

- Avantages :
  - Possibilité de débuter l'intégration dès qu'un composant de niveau feuille dans le graphe de dépendance est prêt
  - **Pas d'écriture de bouchons**
- Inconvénients :
  - Beaucoup de scripts de test à écrire
  - Si un composant proche des feuilles dans le graphe est modifié, certains tests de composants qui en dépendent doivent être ré-exécutés
  - Le composant principal du système est souvent à la racine du graphe. Il est testé en dernier et risque de ne pas être assez testé si le temps manque.



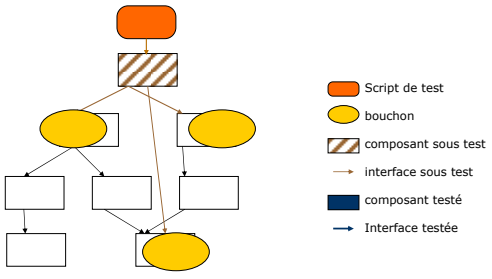
## Intégration Top-down

- Méthode d'intégration privilégiant le test des composants des niveaux les plus élevés de l'arbre de dépendance



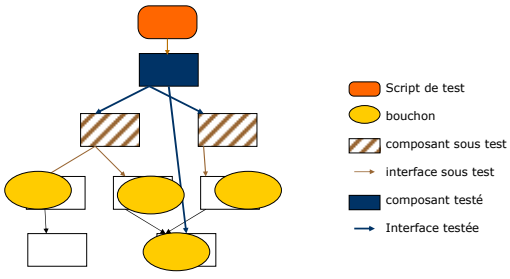



## Intégration Top-down

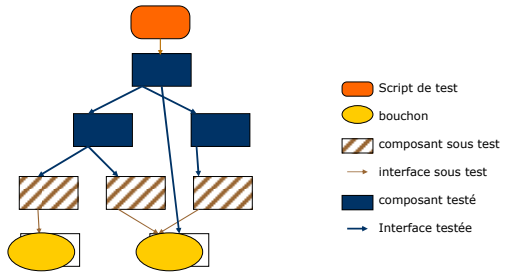





## Intégration Top-down

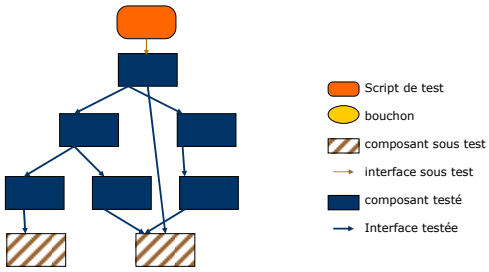


## Intégration Top-down



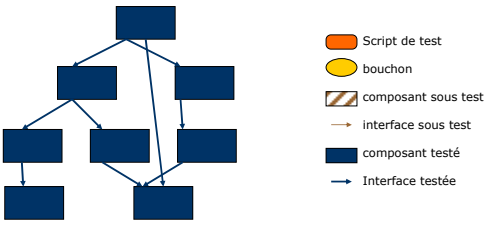



## Intégration Top-down






## Intégration Top-down






## Intégration Top-down

- Avantages :
  - Possibilité de tester tôt les composants les plus dépendants (les plus hauts dans l'arbre de dépendance)
    - démonstration des fonctionnalités des composants de haut niveau disponible rapidement
  - Peu de scripts de test à écrire
  - Test de non régression systématique à chaque ajout de composant

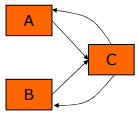
## Intégration Top-down

- Inconvénients :
  - **Beaucoup de bouchons à écrire**
  - L'ajout d'une contrainte (ex : précondition) à un composant proche des feuilles peut rendre nécessaires des changements dans le code de beaucoup de composants qui en dépendent
  - Difficulté d'atteindre une forte couverture des critères de test pour les composants les plus bas dans l'arbre. Ils ne sont testés que dans le contexte de l'application sous test.



## Cas de dépendances cycliques

- Le graphe de dépendance peut comporter des composantes fortement connexes

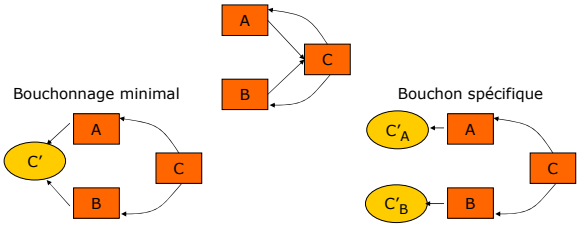



- **Possibilité 1 : intégration Big-Bang**
  - Principe : tester toutes les classes ensemble. Les dépendances ne sont pas exploitées pour définir un ordre de test.
  - Avantage : nombre d'exécutions de cas de test limité
  - Inconvénient : en cas d'erreur, toutes les classes sont aussi susceptibles les unes que les autres de l'avoir provoquée.



## Cas de dépendances cycliques

- **Possibilité 2** : utilisation de bouchons




  - Avantage : intégration progressive des composants
  - Inconvénient : écriture de bouchons



# Partie 3

## Test en présence d'héritage

## Utilisation de l'héritage pour le test

- Principe: si une classe a déjà été testée alors l'effort de test pour les classes qui en héritent peut s'en trouver réduit
- *Hypothèse* : le comportement associé à une sous-classe est un raffinement du comportement de la classe de base.
  - Soit D héritant de C :
    - préconditions de D : les mêmes ou moins fortes que celles de C
      ( précond(D) ➔ précond(C) )
    - postconditions de D : les mêmes ou plus fortes que celles de C ;
      ( postcond(C) ➔ postcond(D) )
    - Invariant de D : le même ou plus fort que celui de C
      ( inv(C) ➔ inv(D) )



## Héritage de cas de test

- Si D hérite de C :
  - Sont aussi hérités de C par D :
    - les cas de test basés sur la spécification ;
    - la plupart des cas de test basés sur l'implémentation ;
    - la plupart des cas de test issus des tests d'intégration
  - Des cas de test doivent parfois être ajoutés aux cas de test hérités

-> processus de test incrémental et hiérarchique. Les cas de test hérités et additionnels dépendent des modifications apportées dans D par rapport à C



## Quels cas de test ajouter pour la sous-classe ?

- Dans la suite de cette partie, C désigne une classe et D une de ses sous-classes.


- Ajout d'une opération dans l'interface de D et potentiellement dans son implémentation :
  - Ajout de cas de test basés sur sa spécification
  - Si l'opération est implémentée : ajout de cas de test basés sur l'implémentation et pour le test d'intégration



## Quels cas de test ajoutés pour la sous-classe ?

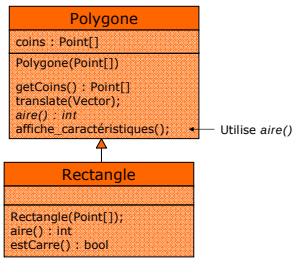
- Modification dans D de la spécification d'une opération m déclarée dans C :
  - Ajout de cas de test basés sur la spécification de m dans D
  - Ré-exécution des cas de tests de C pour m dans D
- Surcharge dans D d'une opération m héritée de C :
  - Réutilisation de tous les cas de test basés sur la spécification
  - Revoir les cas de test basés sur l'implémentation et ceux pour le test d'intégration : réutilisation d'une partie de ceux de C et ajout de nouveaux cas de test

## Quels cas de test ajouter pour la sous-classe ?

- Méthodes héritées telles quelles par D :
  - Les cas de test de C pour ces méthodes sont encore valides
    - ils n'ont pas besoin d'être réexécutés en général…
    - …sauf pour les méthodes qui utilisent des méthodes qui ont été modifiées dans D. En ce cas, des cas de test basés sur l'implémentation doivent aussi être ajoutés
- Remarque sur la ré-exécution de cas de test : en pratique, tous les cas de test hérités sont ré-exécutés. En effet, l'effort nécessaire à la sélection des cas de test à ne pas ré-exécuter est supérieur à l'effort nécessaire pour les exécuter de nouveau



## Exercice: quels cas de test ajouter pour tester Rectangle, connaissant les tests de Polygone ?

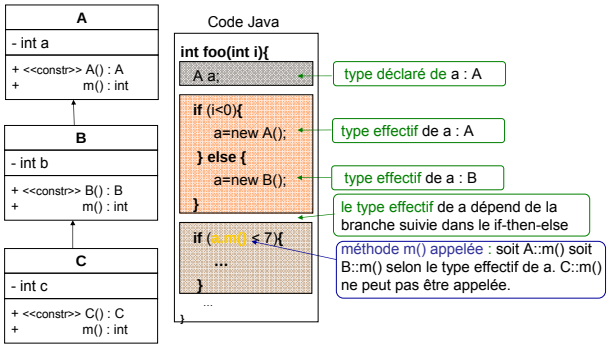





## Comment tester en présence de polymorphisme par liaison dynamique ?



## Rappel sur la liaison dynamique



## Critère de test structurel et polymorphisme

⌘ Soient $m_j$ et $m_k$ des m

⌘ Soient :

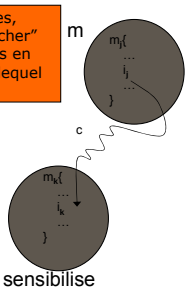
$v$ : variable d'instance

$i_j$ : instruction de la méthode $m_j$ qui définit $v$

$i_k$ : instruction de la méthode $m_k$ qui utilise $v$

Si $m_j$ et/ou $m_k$ sont polymorphes, plusieurs codes peuvent "se cacher" derrière les appels de méthodes en fonction du type de l'objet sur lequel elles s'appliquent

m




⌘ Le critère est couvert si :

$\forall (v, i_j, i_k)$ faisable, $\exists$ un cas de test pour m qui sensibilise un chemin $c$ entre $i_j$ et $i_k$ sans redéfinition de v

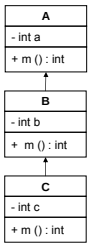
Si $m_j$ et/ou $m_k$ sont polymorphes, il faut déterminer ces triplets pour chaque corps de méthode potentiellement exécuté.



## Critère de test structurel et polymorphisme

⌘ Soit m polymorphe, appelée par o de type déclaré A:

- A::m, B::m ou C::m peuvent être appelées
- Cependant, le type effectif de o ne peut être que A ou B, ainsi l'appel de C::m() ne peut être couvert !




⌘ Déterminer les types effectifs possible nécessite une analyse globale de l'application (coûteux et complexe)

-> la mesure fiable du critère de couverture est compromise (problème à rapprocher de la détermination des chemins executables)



## Comment tester les classes abstraites ?



## Problématique des classes abstraites

⌘ **Classe abstraite =** classe sans instance, à l'implémentation incomplète – Utile pour la généricité et la lisibilité

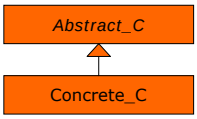
⌘ **Problème pour le test : impossible à tester en tant que telle puisque pas instanciable.**

⌘ Pratique de vérification usuelle : revue du code uniquement !

⌘ **Nécessité de tester les opérations de la classe abstraite au travers des sous-classes**

## Comment tester une classe abstraite ?

⌘ **Approche 1 :** définition d'une classe concrète uniquement pour pouvoir tester la classe abstraite




⌘ Dans la classe concrète, les méthodes abstraites de la classe abstraite sont implémentées -> bouchons



## Comment tester une classe abstraite ?

⌘ **Problèmes posés par l'approche 1 :**

- Les bouchons sont parfois difficiles à écrire :
  - Soit f une méthode implémentée dans la classe abstraite, qui appelle la méthode abstraite g de la même classe.

```
abstract class Abstract_C{
  abstract int g(int a);
  int f(int a){
      int i = g(a);
      …
  }
}
```

  - L'implémentation de g doit permettre de respecter la postcondition de la fonction g.

-> le coût est parfois très élevé si g est une fonction complexe

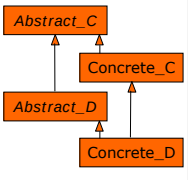


## Comment tester une classe abstraite ?

⌘ **Problèmes posés par l'approche 1 :**

- Soient, Abstract_C et Abstract_D deux classes telles que Abstract_D hérite de Abstract_C ;
- Concrete_C et Concrete_D les classes qui leur sont respectivement associées à des fins de test.

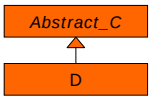
***L'héritage multiple*** est nécessaire pour que Concrete_D hérite des bouchons créés dans Concrete_C
-> OK en C++, mais en Java l'héritage multiple de classe n'est pas autorisé






## Comment tester une classe abstraite ?

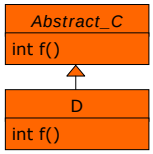
⌘ **Approche 2 :** Tester la classe abstraite avec son premier descendant




⌘ *Hypothèse sous-jacente*: si D passe sans problème les cas de test alors Abstract_C les passe également sans problème

## Comment tester une classe abstraite ?

- **Problème posé par l'approche 2 :**
  - L'hypothèse n'est pas toujours valide !
    - Si D surcharge une méthode *f* de Abstract C alors la méthode *f* de Abstract_C n'est pas testée. Il faudra la tester dans une autre sous-classe de Abstract_C qui ne surcharge pas cette méthode *f*.






## Comment tester une classe abstraite ?

- **Approche 3 :** Utiliser une compilation conditionnelle pour avoir dans un même fichier :
  - la classe abstraite sous test avec ses méthodes abstraites
  - Une version de la classe où chaque méthode abstraite est remplacée par une implémentation (bouchon)

- Ex : #ifdefined(TEST)
  <bouchon>
  #endif

- **Problème posé par l'approche 3 :**
  - le code est peu lisible



## Test de logiciel orienté-objet: conclusions

- Spécificités à prendre en compte pour le test (abstraction, encapsulation, modularité) pour ne pas rejouer trop de test inutiles

- Difficultés particulières :

  - Test en présence de polymorphisme (liaison dynamique)

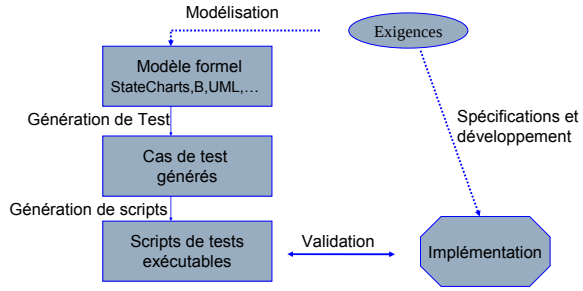
  - Test de classes abstraites



## Plan du cours d'aujourd'hui

1. Techniques de test élémentaires
2. Test de logiciel orientés-objet
3. Model-Based Testing

## Processus de test à partir de modèles

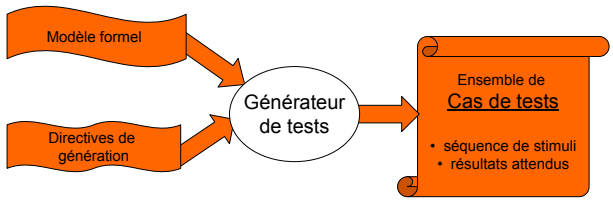



Originalité du MBT : construire le modèle dans le but d'automatiser le test





## Génération de tests à partir de modèles

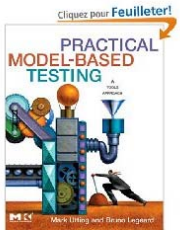



- Directives de génération (définition d'un scénario de tests) :
  - Critères de couverture du modèle
  - Critères de sélection sur le modèle



## Reference

« Practical Model-based Testing »
M. Utting, B. Legeard, 2008






- Modéliser pour tester
- Stratégies de génération de test
- Sélection des tests – Critères de couverture
- Exemple d'outils : SIMULINK DESIGN VERIFIER

## Modéliser pour tester (1)

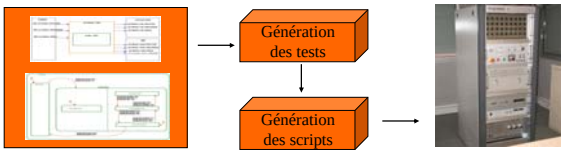
- Modèle abstrait
  - modèle fonctionnel (et non modèle d'architecture et/ou d'implémentation)
- Modèle détaillé et précis
  - Le modèle doit permettre de calculer les cas de test et les résultats attendus
  - ➔ établir automatiquement le verdict de test
- Modèle validé et vérifié
  - Si le verdict de test « Fail » : l'erreur est-elle dans l'implantation ou dans le modèle ?





## Modéliser pour tester (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 Banc de test





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

- Applications type contrôle-commande (automobile, énergie, aéronautique, …)
  - ➔ Statecharts, Matlab/Simulink, Lustre, Signal
- 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)





## Notations de modélisation

- Systèmes de transitions (Etats / Transitions / Evénements)
  - Flow Charts / CFGs
  - Data Flow Diagrams
  - Diagrammes d'état
- Diagrammes objet & association (hiérarchie, héritage, …)
  - Diagrammes de classe (UML)
  - Modèles Entité-Relation
- Représentation Pré/Post conditions
  - OCL (Object Constraint Language) pour UML
  - JML (Java Modeling Language) pour Java
  - Machine Abstraite B

- Représentation : Graphique (plus « intuitive » ) ou textuelle (plus précise)

## Choix de notations

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





## Quelques outils…

- 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, …)
- Systèmes hybrides (discret/continus)
  - Simulink (Mathworks)
- Langages synchrones
  - Lustre (GATEL – CEA)



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

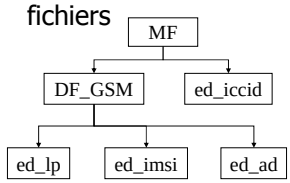
- La norme GSM 11-11 défini l'interface entre le Mobile (téléphone GSM) et la puce SIM – Subscriber Identifier Module
- La puce SIM est une carte à puces qui contient toutes les données utilisateurs et gére la protection de l'accès
- Le standard GSM 11-11 propose 21 API incluant la gestion du PIN – Personal 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_unblock1, 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

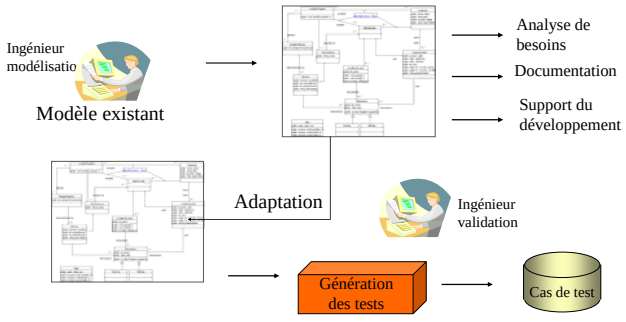
```
sw ← CHANGE_Pin(old_code, new_code) =
    PRE
    old_code ∈ Nat ∧ new_code ∈ Nat
    THEN
        IF counter_pin = 0
            THEN
            sw := 9840
            ELSE
              IF code_pin = old_code
                  THEN
                  code_pin := new_code          ||
                  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 ;
```



- Modéliser pour tester
- Stratégies de génération de test
- Sélection des tests – Critères de couverture
- Exemple d'outils : SIMULINK DESIGN VERIFIER

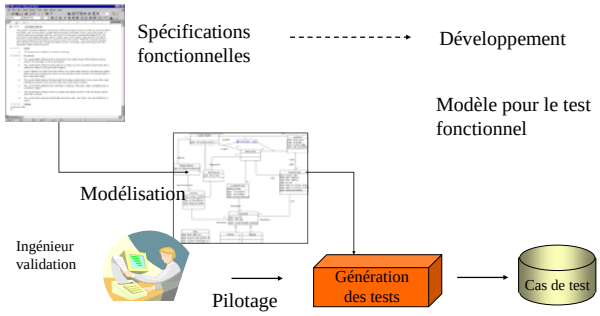


## Processus Model-Based Testing – Schéma 1






## Processus Model-Based Testing – Schéma 2



## Génération automatique de tests fonctionnels

Automatiser les stratégies classiques :

- **Analyse partitionnelle** des données d'entrée
- **Test aux limites**
- Couverture de critères de test défini sur le modèle

Ou bien sélection de comportements à tester en priorité

Objectif : Maîtriser la combinatoire des cas de test



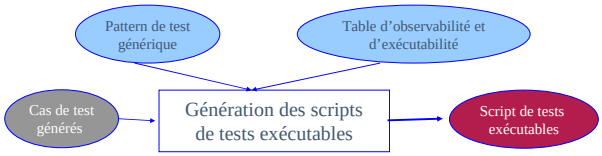
## Composition d'un cas de test

- 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




## 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.**

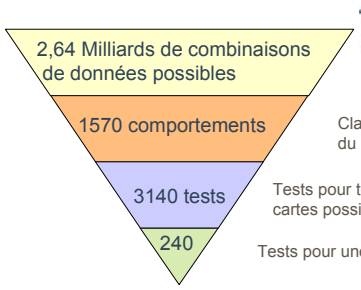





## Problématique de la traduction des tests générés en scripts exécutables

- 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 (TTCN, JUnit, Cunit, Pluto, …)
  - → 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

## Maîtriser la combinatoire : exemple de la validation terminal Carte Bancaire




MagIC 500

*Schlumberger device*

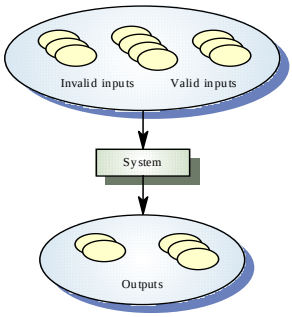
Classes de comportements du système

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

Tests pour une configuration particulière



## 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.



- Modéliser pour tester
- Stratégies de génération de test
- Sélection des tests – Critères de couverture
- Exemple d'outils : SIMULINK DESIGN VERIFIER



## 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

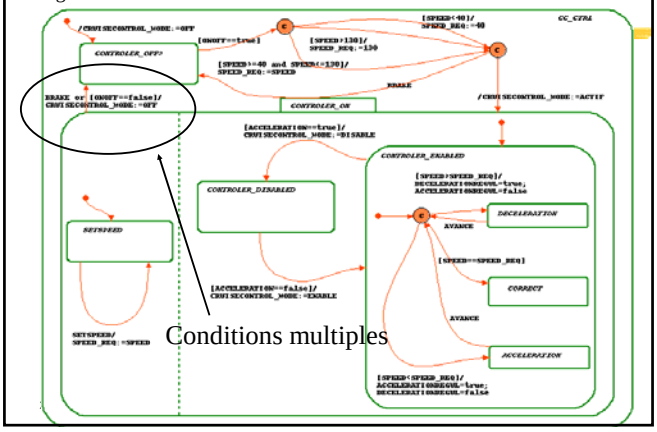
## Critères de couverture

- Critères de conditions multiples dans les décisions
  - Toutes les décisions
  - Toutes les conditions
  - Toutes les décisions / conditions modifiées (MC/DC)
  - Toutes les conditions/decisions multiples
  - …

  SUR LE MODELE…



Diagramme d'états du contrôleur de vitesse

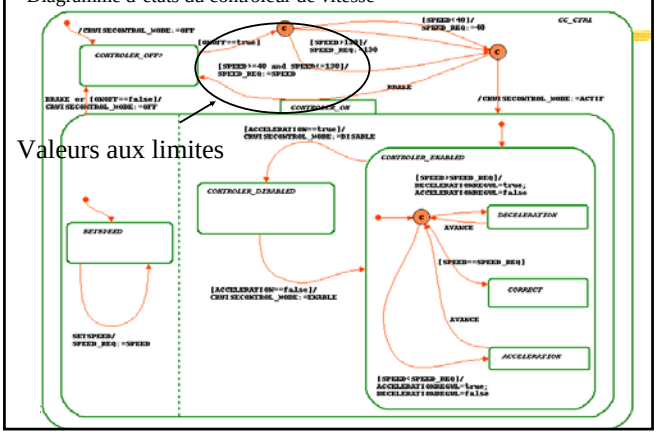



## Critères de couverture

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



Diagramme d'états du contrôleur de vitesse

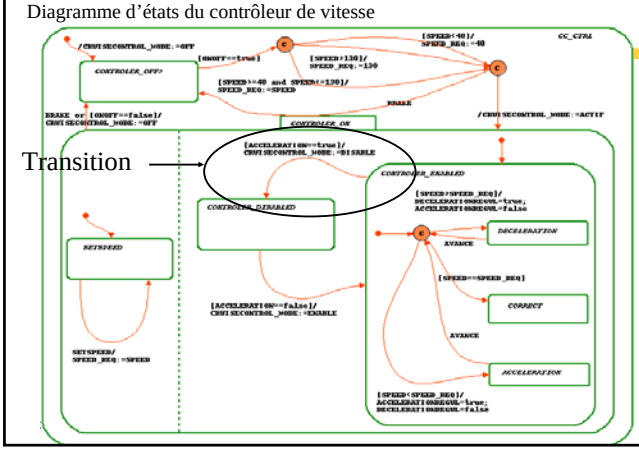


## Critères de couverture

- Critères de conditions multiples dans les décisions
  - Toutes les décisions
  - Toutes les conditions/décisions
  - Toutes les décisions / conditions modifiées (MC/DC)
  - Toutes les conditions/decisons multiples
- 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



- Modéliser pour tester
- Stratégies de génération de test
- Sélection des tests – Critères de couverture
- Exemple d'outils : SIMULINK DESIGN VERIFIER

## Synthèse du cours

- Validation des logiciels dans le Monde Industriel essentiellement basée sur le test logiciel
- Spécificité du test des logiciels orientés-objet
- Approches complémentaires:

  Code-based Testing vs Model-based Testing

- Nombreux outils pour tous les langages et notations de modèles

Soit le programme suivant codé dans le langage C (on rappelle que le type `unsigned char` code une valeur non signée sur 8 bits) :

```
unsigned char F(unsigned char x, unsigned char y)
{
1.   while( x*y == 6 )
2.        while( x+y == 5 )
3.             {  x++ ; y-- ; }
4.   return(x-y) ;
}
```

Question 1 : Déssiner le graphe de flot de contrôle associé à F

Question 2 : Donner une base de chemins, de taille minimum, qui couvre le critère tous_les_arcs.

Question 3 : Quelle est la valeur du programme F si on lui soumet la donnée de test (0,1) ?

Question 4 : Le programme F est-il toujours terminant ? (justifiez brièvement votre réponse)

Question 5 : Quelles sont les valeurs possibles renvoyées par F si le flot suit le début de chemin 1-2-3-… ?

Question 6 : Quels sont les chemins exécutables de F qui démarrent par la séquence 1-2-3-… ?

On considère le programme P suivant, écrit en langage C :

```
signed short  P(signed short x) {
signed short t1, t2, t3 ;

1.  t1 = x + 1 ;
2.  t2 = t1 * t1 ;
3.  t3 = t2 – t1 ;
4.  if (((x²-16 =< 0) && (t3 != 0)) || ((x < -1) && (x²-25 > 0))
5.        t1 = x – t1 ;
  else
6.        t1 = t1 – x ;
7. return t1 ;
   }
```

**1.** En remplaçant chaque condition par une variable booléenne libre dans la décision 4, donner un ensemble de valeurs pour ces variables, de taille minimum, qui couvre le critère de test « Modified Condition/Decision » *(Indication : dans cette question, on ignore les liens entre les différentes conditions de la décision).*

**2.** Donner, si possible, un jeu de test pour le programme P qui couvre le critère des « Conditions » pour la décision 4.

**3.** Donner, si possible, un jeu de test qui couvre le critère « Modified Condition/Decision » *(justifiez brièvement)*

**4.** Donner, si possible, un jeu de test qui couvre le critère « Multiple Condition/Decision » *(justifiez brièvement)*

## "TEST STRUCTUREL DE PROGRAMMES IMPERATIFS

On considère le programme P suivant, écrit en langage C :

```
short  P(short x1, short x2, short x3) {
     short y;
 1.  y = x3;
 2.  if (x2 < x3)
 3.     if (x1 < x2)
 4.         y = x2 ;
        else
 5.         if (x1 < x3)
 6.             y = x1 ;
     else
 7.     if (x1 > x2)
 8.         y = x2 ;
        else
 9.        if (x1 > x3)
 10.             y = x1 ;
 11. return y;
}
```

*Question 1.* Donner une spécification précise de P et déssiner le graphe Def/Use de P

*Question 2.* Donner un ensemble de chemins de P, de taille minimum, qui satisfait

*1)* le critère *toutes_les_définitions*

*2)* le critère *tous_les_arcs*

*Justifier(brièvement) les réponses*

*Question 3.* Donner un jeu de test qui sensibilise les chemins de P qui satisfont les critères *toutes_les_définitions* et *tous_les_arcs* (Nota : on indiquera aussi les sorties attendues).

*Question 4.* Comment se comparent les critères *toutes_les_définitions* et *tous_les_arcs* pour le programme P ? Montrer que ces deux critères sont incomparables dans le cas général (indication : on recherchera des contres-exemples)

*Question 5.* On remplace la ligne 9 dans le programme P par :

```
9.     if (x2 > x3)
```

et on appelle P' le programme ainsi obtenu.

1) les jeux de test donnés à la troisième question de cette partie sont-ils réussis pour P' ?

2) Montrer que P' est incorrect par rapport à la spécification de la première question de cette partie