

# Introduction aux tests du logiciel

F.X. Fornari

`xavier.fornari@esterel-technologies.com`

P. Manoury

`pascal.manoury@pps.univ-paris-diderot.fr`

2014

# Les Test Structurels

- ▶ Les tests structurels permettent l'analyse de la réalisation du logiciel

# Les Test Structurels

- ▶ Les tests structurels permettent l'analyse de la réalisation du logiciel
- ▶ Tests fonctionnels: est-ce que le logiciel fonctionne correctement ?
- ▶ Tests structurels: est-ce que le logiciel est bien fait ?

# Les Test Structurels

- ▶ Les tests structurels permettent l'analyse de la réalisation du logiciel
- ▶ Tests fonctionnels: est-ce que le logiciel fonctionne correctement ?
- ▶ Tests structurels: est-ce que le logiciel est bien fait ?
- ▶ On s'intéresse à la structure, c'est à dire aux instructions, et à leur couverture.

# Les Test Structurels

- ▶ Les tests structurels permettent l'analyse de la réalisation du logiciel
- ▶ Tests fonctionnels: est-ce que le logiciel fonctionne correctement ?
- ▶ Tests structurels: est-ce que le logiciel est bien fait ?
- ▶ On s'intéresse à la structure, c'est à dire aux instructions, et à leur couverture.
- ▶ Il y a différentes façons de couvrir le code.

## Couverture d'instructions

```
1  Found = False;
2  Indice = -1;
3  while ( ! Found && Indice < TabMax ) {
4      Indice++;
5      if ( Tab[Indice] == Value ) {
6          Found = True;
7      }
8  }
9  return Indice;
10 }
```

- ▶ Couverture de toutes les instructions

## Couverture d'instructions

```
1  Found = False;
2  Indice = -1;
3  while ( ! Found && Indice < TabMax ) {
4      Indice++;
5      if ( Tab[Indice] == Value ) {
6          Found = True;
7      }
8  }
9  return Indice;
10 }
```

- ▶ Couverture de toutes les instructions
- ▶ L'exemple  $Tab[0] = Value$  suffit

## Couverture d'instructions

```
1  Found = False;
2  Indice = -1;
3  while ( ! Found && Indice < TabMax ) {
4      Indice++;
5      if ( Tab[Indice] == Value ) {
6          Found = True;
7      }
8  }
9  return Indice;
10 }
```

- ▶ Couverture de toutes les instructions
- ▶ L'exemple  $Tab[0] = Value$  suffit
- ▶ Ce n'est pas une couverture suffisante, mais minimale



# Couverture de Décisions

```
1  Found = False;
2  Indice = -1;
3  while ( ! Found && Indice < TabMax ) {
4      Indice++;
5      if ( Tab[Indice] == Value ) {
6          Found = True;
7      }
8  }
9  return Indice;
10 }
```

- ▶ Couverture des branches, avec une décision vraie et fausse

# Couverture de Décisions

```
1 Found = False;  
2 Indice = -1;  
3 while ( ! Found && Indice < TabMax ) {  
4     Indice++;  
5     if ( Tab[Indice] == Value ) {  
6         Found = True;  
7     }  
8 }  
9 return Indice;  
10 }
```

- ▶ Couverture des branches, avec une décision vraie et fausse
- ▶ Il faut identifier toutes les décisions et comment les activer.

# Couverture de Décisions

```
1 Found = False;  
2 Indice = -1;  
3 while ( ! Found && Indice < TabMax ) {  
4     Indice++;  
5     if ( Tab[Indice] == Value ) {  
6         Found = True;  
7     }  
8 }  
9 return Indice;  
10 }
```

- ▶ Couverture des branches, avec une décision vraie et fausse
- ▶ Il faut identifier toutes les décisions et comment les activer.
- ▶ On ne tient pas compte de la combinaison entre les branches

## Couverture de Conditions

```
1  Found = False;
2  Indice = -1;
3  while (! Found && Indice < TabMax ) {
4      Indice++;
5      if ( Tab[Indice] == Value ) {
6          Found = True;
7      }
8  }
9  return Indice;
10 }
```

- ▶ Les conditions sont les composantes d'une décision

# Couverture de Conditions

```
1 Found = False;
2 Indice = -1;
3 while (! Found && Indice < TabMax ) {
4     Indice++;
5     if ( Tab[Indice] == Value ) {
6         Found = True;
7     }
8 }
9 return Indice;
10 }
```

- ▶ Les conditions sont les composantes d'une décision
- ▶ Il faut avoir les conditions à vraies et fausses.

# Couverture de Conditions

```
1 Found = False;
2 Indice = -1;
3 while (! Found && Indice < TabMax ) {
4     Indice++;
5     if ( Tab[Indice] == Value ) {
6         Found = True;
7     }
8 }
9 return Indice;
10 }
```

- ▶ Les conditions sont les composantes d'une décision
- ▶ Il faut avoir les conditions à vraies et fausses.
- ▶ Dans l'exemple il faut *Found* et *Indice* à vrai et faux. 2 tests suffiraient.

# Analyse de conditions/décisions

- ▶ Combiner conditions/décisions: augmentation de la couverture mais:

# Analyse de conditions/décisions

- ▶ Combiner conditions/décisions: augmentation de la couverture mais:
  - ▶ "TT" et "FF" ne distinguent pas "A or B" et "A and B"
  - ▶ pourtant, conditions et décisions sont bien activées



# Analyse de conditions/décisions

- ▶ Combiner conditions/décisions: augmentation de la couverture mais:
  - ▶ "TT" et "FF" ne distinguent pas "A or B" et "A and B"
  - ▶ pourtant, conditions et décisions sont bien activées
- ▶ il faut couvrir tous les chemins

# Analyse de conditions/décisions

- ▶ Combiner conditions/décisions: augmentation de la couverture mais:
  - ▶ "TT" et "FF" ne distinguent pas "A or B" et "A and B"
  - ▶ pourtant, conditions et décisions sont bien activées
- ▶ il faut couvrir tous les chemins
- ▶ 2 méthodes: LCSAJ et MC/DC

## Couverture de code: LCSAJ

“Linear Code Subpath And Jump” : il s’agit d’une portion linéaire de code suivie d’un saut.

L’idée est de “couper” le code en tronçons linéaires, et de couvrir tous les chemins.

Pour un LCSAJ, nous avons:

- ▶ le début de la séquence;
- ▶ sa fin;
- ▶ la cible du saut.

Ces éléments sont exprimés en numéros de ligne.

LCSAJs ont un ratio d’efficacité de test de niveau 3.

# Test Effectiveness Ratio

C'est une mesure de l'efficacité de couverture des tests.

$$TER1 = \frac{\# \text{ d'instructions executees}}{\# \text{ total d'instructions executables}}$$

$$TER2 = \frac{\# \text{ de branches executees}}{\# \text{ total de branches}}$$

$$TER3 = \frac{\# \text{ LCSAJs executes}}{\# \text{ total de LCSAJs}}$$

A noter:

$$TER3 = 100\% \Rightarrow TER2 = 100\% \text{ et } TER1 = 100\%$$

# Un exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <math.h>
4
5  #define MAXCOLUMNS 26
6  #define MAXROW 20
7  #define MAXCOUNT 90
8  #define ITERATIONS 750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

- ▶ Combinaison de boucle et de branchement;
- ▶ Nous avons donc des "bouts" de code séquentiel, ainsi que des sauts (branchements)
- ▶ Les sauts correspondent aux tests, et aux branches explicites ou implicites.

# Un exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <math.h>
4
5  #define MAXCOLUMNS 26
6  #define MAXROW 20
7  #define MAXCOUNT 90
8  #define ITERATIONS 750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

| LCSAJ | Start | End | Jmp |
|-------|-------|-----|-----|
| 1     | 10    | 17  | 28  |
| 2     | 10    | 21  | 25  |
| 3     | 10    | 26  | 17  |
| 4     | 17    | 17  | 28  |
| 5     | 17    | 21  | 25  |
| 6     | 17    | 26  | 17  |
| 7     | 25    | 26  | 17  |
| 8     | 28    | 28  | -1  |

# Un exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <math.h>
4
5  #define MAXCOLUMNS 26
6  #define MAXROW 20
7  #define MAXCOUNT 90
8  #define ITERATIONS 750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

| LCSAJ | Start | End | Jmp |
|-------|-------|-----|-----|
| 1     | 10    | 17  | 28  |
| 2     | 10    | 21  | 25  |
| 3     | 10    | 26  | 17  |
| 4     | 17    | 17  | 28  |
| 5     | 17    | 21  | 25  |
| 6     | 17    | 26  | 17  |
| 7     | 25    | 26  | 17  |
| 8     | 28    | 28  | -1  |

# Un exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <math.h>
4
5  #define MAXCOLUMNS 26
6  #define MAXROW 20
7  #define MAXCOUNT 90
8  #define ITERATIONS 750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

| LCSAJ | Start | End | Jmp |
|-------|-------|-----|-----|
| 1     | 10    | 17  | 28  |
| 2     | 10    | 21  | 25  |
| 3     | 10    | 26  | 17  |
| 4     | 17    | 17  | 28  |
| 5     | 17    | 21  | 25  |
| 6     | 17    | 26  | 17  |
| 7     | 25    | 26  | 17  |
| 8     | 28    | 28  | -1  |



# Un exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <math.h>
4
5  #define MAXCOLUMNS 26
6  #define MAXROW 20
7  #define MAXCOUNT 90
8  #define ITERATIONS 750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

| LCSAJ | Start | End | Jmp |
|-------|-------|-----|-----|
| 1     | 10    | 17  | 28  |
| 2     | 10    | 21  | 25  |
| 3     | 10    | 26  | 17  |
| 4     | 17    | 17  | 28  |
| 5     | 17    | 21  | 25  |
| 6     | 17    | 26  | 17  |
| 7     | 25    | 26  | 17  |
| 8     | 28    | 28  | -1  |

# Un exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <math.h>
4
5  #define MAXCOLUMNS 26
6  #define MAXROW 20
7  #define MAXCOUNT 90
8  #define ITERATIONS 750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

| LCSAJ | Start | End | Jmp |
|-------|-------|-----|-----|
| 1     | 10    | 17  | 28  |
| 2     | 10    | 21  | 25  |
| 3     | 10    | 26  | 17  |
| 4     | 17    | 17  | 28  |
| 5     | 17    | 21  | 25  |
| 6     | 17    | 26  | 17  |
| 7     | 25    | 26  | 17  |
| 8     | 28    | 28  | -1  |

# Un exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <math.h>
4
5  #define MAXCOLUMNS 26
6  #define MAXROW 20
7  #define MAXCOUNT 90
8  #define ITERATIONS 750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

| LCSAJ | Start | End | Jmp |
|-------|-------|-----|-----|
| 1     | 10    | 17  | 28  |
| 2     | 10    | 21  | 25  |
| 3     | 10    | 26  | 17  |
| 4     | 17    | 17  | 28  |
| 5     | 17    | 21  | 25  |
| 6     | 17    | 26  | 17  |
| 7     | 25    | 26  | 17  |
| 8     | 28    | 28  | -1  |

# Un exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <math.h>
4
5  #define MAXCOLUMNS 26
6  #define MAXROW 20
7  #define MAXCOUNT 90
8  #define ITERATIONS 750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

| LCSAJ | Start | End | Jmp |
|-------|-------|-----|-----|
| 1     | 10    | 17  | 28  |
| 2     | 10    | 21  | 25  |
| 3     | 10    | 26  | 17  |
| 4     | 17    | 17  | 28  |
| 5     | 17    | 21  | 25  |
| 6     | 17    | 26  | 17  |
| 7     | 25    | 26  | 17  |
| 8     | 28    | 28  | -1  |

# Un exemple

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <math.h>
4
5  #define MAXCOLUMNS 26
6  #define MAXROW 20
7  #define MAXCOUNT 90
8  #define ITERATIONS 750
9
10 int main (void)
11 {
12     int count = 0, totals[MAXCOLUMNS], val = 0;
13
14     memset (totals, 0, MAXCOLUMNS * sizeof(int));
15
16     count = 0;
17     while ( count < ITERATIONS )
18     {
19         val = abs(rand()) % MAXCOLUMNS;
20         totals[val] += 1;
21         if ( totals[val] > MAXCOUNT )
22         {
23             totals[val] = MAXCOUNT;
24         }
25         count++;
26     }
27
28     return (0);
29
30 }
```

| LCSAJ | Start | End | Jmp |
|-------|-------|-----|-----|
| 1     | 10    | 17  | 28  |
| 2     | 10    | 21  | 25  |
| 3     | 10    | 26  | 17  |
| 4     | 17    | 17  | 28  |
| 5     | 17    | 21  | 25  |
| 6     | 17    | 26  | 17  |
| 7     | 25    | 26  | 17  |
| 8     | 28    | 28  | -1  |

Calculée automatiquement d'ordinaire, inutile de s'inquiéter !

**Avantage** La couverture est sensiblement meilleure que la couverture des branches sans être trop explosive (bon ratio coût/niveau de confiance, entre couverture des branches et couverture des chemins).

**Inconvénient** Elle ne couvre pas la totalité de la spécification. Elle dépend du code du programme plus que de la structure réelle du graphe de contrôle.

# Couverture MC/DC

- ▶ c'est une méthodologie pour assurer une combinaison des décisions et des conditions en essayant d'être efficace

# Couverture MC/DC

- ▶ c'est une méthodologie pour assurer une combinaison des décisions et des conditions en essayant d'être efficace
- ▶ théorisée par un article de la NASA, et standard obligatoire dans l'aéro, mais se diffuse



# Couverture MC/DC

- ▶ c'est une méthodologie pour assurer une combinaison des décisions et des conditions en essayant d'être efficace
- ▶ théorisée par un article de la NASA, et standard obligatoire dans l'aéro, mais se diffuse
- ▶ But: trouver des ensembles minimaux de tests qui:

# Couverture MC/DC

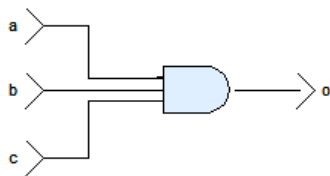
- ▶ c'est une méthodologie pour assurer une combinaison des décisions et des conditions en essayant d'être efficace
- ▶ théorisée par un article de la NASA, et standard obligatoire dans l'aéro, mais se diffuse
- ▶ But: trouver des ensembles minimaux de tests qui:
  - ▶ assurent que les décisions soient vraies et fausses

# Couverture MC/DC

- ▶ c'est une méthodologie pour assurer une combinaison des décisions et des conditions en essayant d'être efficace
- ▶ théorisée par un article de la NASA, et standard obligatoire dans l'aéro, mais se diffuse
- ▶ But: trouver des ensembles minimaux de tests qui:
  - ▶ assurent que les décisions soient vraies et fausses
  - ▶ démontrent l'indépendance des conditions entre elles sur les décisions

# Couverture MC/DC

Un exemple avec l'opérateur "and" :

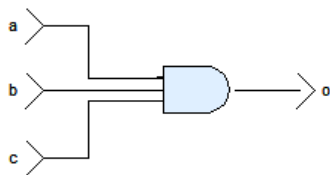


| Test | 1 | 2 | 3 | 4 |
|------|---|---|---|---|
| a    | T | F | T | T |
| b    | T | T | F | T |
| c    | T | T | T | F |
| o    | T | F | F | F |

- ▶ La sortie "o" est modifiée par une seule entrée
- ▶ 4 tests au lieu de  $2^3 = 8$

# Couverture MC/DC

Un exemple avec l'opérateur "and":



- ▶ La sortie "o" est modifiée par une seule entrée
- ▶ 4 tests au lieu de  $2^3 = 8$
- ▶ Le problème du masquage: la cause "unique" n'est pas toujours possible.
- ▶ AND: FF inutile, OR: TT inutile, etc...

| Test | 1 | 2 | 3 | 4 |
|------|---|---|---|---|
| a    | T | F | T | T |
| b    | T | T | F | T |
| c    | T | T | T | F |
| o    | T | F | F | F |

| Test | 1 | 2 | 3 | 4 |
|------|---|---|---|---|
| a    | T | F | F | T |
| b    | T | T | F | T |
| c    | T | T | T | F |
| o    | T | F | F | F |

# MC/DC: méthodologie

1. Avoir une représentation schématique du code (portes logiques)
2. Identifier les tests en fonction des spécifications. Propager les tests sur la représentation
3. Eliminer les cas de masquages
4. Valider que les tests restants forment la couverture MC/DC
5. Examiner la sortie des tests pour confirmer la correction du logiciel

# Résumé des couvertures

**Instruction** couverture des instructions uniquement, mais les conditions ne sont pas couvertes complètement

**Décision** Couverture des branches, il faut un test à *vrai* et un test à *faux*

**Condition** Chaque condition doit être à *vrai* ou *faux*: Ex:  
“A or B”  $\Rightarrow$  TF et FT

**Condition/Decision** Combinaison des 2 précédentes, mais ne permet pas de distinguer TT et FF pour “A and B” et “A or B”

**MC/DC** *Modified Condition/Decision Coverage*. Demande l'indépendance de chaque condition sur la sortie.  
“A or B”  $\Rightarrow$  TF, FT et FF

**Multiple Condition** Faire les  $2^n$  cas. N'est pas praticable en général.

# Impact des couvertures

| Critère de couverture  | Couverture d'instruction | Couverture de Décision | Couverture de Condition | Couverture Condition Décision | MC/DC | Couverture Multiple de Condition |
|--|--------------------------|------------------------|-------------------------|-------------------------------|-------|----------------------------------|
| Chaque point d'entrée/de sortie du programme exécuté au moins une fois               |                          | •                      | •                       | •                             | •     | •                                |
| Chaque instruction du programme exécutée une fois                                    | •                        |                        |                         |                               |       |                                  |
| Chaque décision a pris toutes les valeurs possible au moins une fois                 |                          | •                      |                         | •                             | •     | •                                |
| Chaque condition d'une décision a pris toutes les valeurs possible au moins une fois |                          |                        | •                       | •                             | •     | •                                |
| Chaque condition d'une décision a affecté la décision de manière indépendante        |                          |                        |                         |                               | •     | •                                |
| Chaque combinaison de conditions d'une décision a été faite au moins une fois        |                          |                        |                         |                               |       | •                                |

Critères satisfaits en fonction des couvertures.



## Couverture des itérations

| <b>procedure p is</b> | test 1 | test 2 | test 3    |
|-----------------------|--------|--------|-----------|
| <b>begin</b>          |        |        |           |
| <b>while c1</b>       | F      | V,F    | V,...,V,F |
| <b>loop</b>           |        |        |           |
| s1;                   |        | s1     | s1;...;s1 |
| <b>end loop;</b>      |        |        |           |
| s2;                   | s2     | s2     | s2        |
| <b>end p;</b>         |        |        |           |

## Couverture des itérations

|                              |        |        |           |
|------------------------------|--------|--------|-----------|
| <b>procedure</b> p <b>is</b> | test 1 | test 2 | test 3    |
| <b>begin</b>                 |        |        |           |
| <b>while</b> c1              | F      | V,F    | V,...,V,F |
| <b>loop</b>                  |        |        |           |
| s1;                          |        | s1     | s1;...;s1 |
| <b>end loop;</b>             |        |        |           |
| s2;                          | s2     | s2     | s2        |
| <b>end</b> p;                |        |        |           |

### Pour chaque boucle

- ▶ 0 itération
- ▶ 1 itération
- ▶ max - 1 itérations
- ▶ max itérations
- ▶ max + 1 itérations

# Couverture des données

- ▶ Dans les couvertures précédentes : on ne s'est intéressé ni au format des données manipulées ni aux valeurs numériques permettant de rendre une condition vraie ou fausse.

# Couverture des données

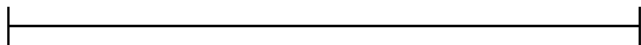
- ▶ Dans les couvertures précédentes : on ne s'est intéressé ni au format des données manipulées ni aux valeurs numériques permettant de rendre une condition vraie ou fausse.
- ▶ La couverture des données consiste à choisir :
  - ▶ les “bonnes valeurs numériques” pour rendre une condition vraie ou fausse (valeurs remarquables)
  - ▶ les valeurs limites des données d'entrée
  - ▶ les valeurs numériques permettant d'obtenir les valeurs ou les erreurs prévisibles (overflow, division par zéro, ...)

## Couverture des données: domaine

Il faut distinguer les domaines de définitions des données, et les domaines d'utilisation:

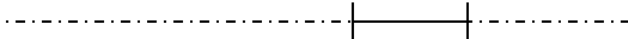
-32768

32767



0

360



## Couverture d'une donnée

Un exemple de valeurs compte tenu de l'intervall fonctionnel, du test, et du type de donnée:

|                       |        |        |        |        |        |        |
|-----------------------|--------|--------|--------|--------|--------|--------|
| <b>procedure p is</b> | test 1 | test 2 | test 3 | test 4 | test 5 | test 6 |
| <b>begin</b>          |        |        |        |        |        |        |
| <b>if e &gt; 30</b>   | 360    | 31     | 30     | 0      | -32768 | 32767  |
| <b>then</b>           |        |        |        |        |        |        |
| s1;                   | s1     | s1     |        |        |        | s1     |
| <b>endif;</b>         |        |        |        |        |        |        |
| <b>end p;</b>         |        |        |        |        |        |        |

# Couverture d'une donnée

- ▶ Une procédure évalue les données pour des valeurs remarquables.
- ▶ Par exemple la conditionnelle :  $e > 30$  montre que la valeur 30 est une valeur remarquable pour la donnée  $e$ .
- ▶ Il est possible de découper le domaine de chaque donnée en classes d'équivalence. Une classe définit un comportement propre à la donnée.
- ▶ Sur l'exemple précédent, la donnée peut être découpée en 2 classes d'équivalence  $[\text{val\_min}; 30]$ ,  $[31; \text{val\_max}]$ .

# Couverture d'une donnée

Du point de vue du testeur, cela nécessite deux types de test :

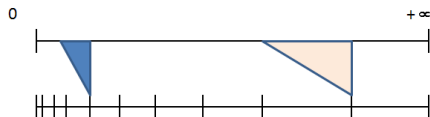
- ▶ les tests **aux limites** pour vérifier le comportement du logiciel pour chaque borne de chaque classe d'équivalence (les limites fonctionnelles des données et les valeurs remarquables)
- ▶ les tests **hors limites** pour vérifier le comportement du logiciel avec des valeurs en dehors de son domaine fonctionnel.



## Cas des réels

Le cas des réels pose le problème de la représentation de flottants:

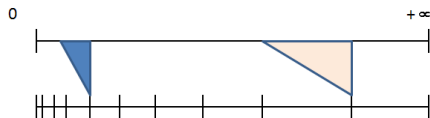
- ▶ Les valeurs sont discrétisées, et les intervalles ne sont pas les mêmes (nombre de bits fixe).



## Cas des réels

Le cas des réels pose le problème de la représentation de flottants:

- ▶ Les valeurs sont discrétisées, et les intervalles ne sont pas les mêmes (nombre de bits fixe).



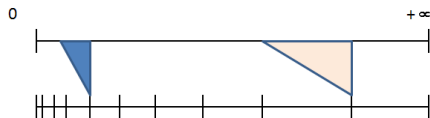
- ▶ Selon le calcul, le résultat peut changer d'intervalle.



## Cas des réels

Le cas des réels pose le problème de la représentation de flottants:

- ▶ Les valeurs sont discrétisées, et les intervalles ne sont pas les mêmes (nombre de bits fixe).



- ▶ Selon le calcul, le résultat peut changer d'intervalle.

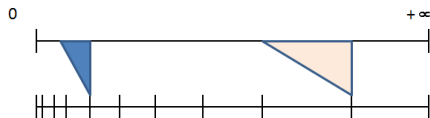


- ▶ Un programme devrait faire:  $\text{if}(val1 - val2) < \epsilon) \dots$  et non  $\text{if}(val1 = val2) \dots$

## Cas des réels

Le cas des réels pose le problème de la représentation de flottants:

- ▶ Les valeurs sont discrétisées, et les intervalles ne sont pas les mêmes (nombre de bits fixe).



- ▶ Selon le calcul, le résultat peut changer d'intervalle.



- ▶ Un programme devrait faire:  $\text{if}(val1 - val2) < \epsilon) \dots$  et non  $\text{if}(val1 = val2) \dots$
- ▶ Idem pour tester les résultats: il faut définir la précision attendue par rapport à la spécification.

# Couverture de données: enumeration

Considérons le programme C suivant:

```
switch (size-bound) {  
  case 3:  
    value += mBvGetbit(p_bv, byte_idx)<<2;  
    byte_idx--;  
  case 2:  
    value += mBvGetbit(p_bv, byte_idx)<<1;  
    byte_idx--;  
  case 1:  
    value += mBvGetbit(p_bv, byte_idx);  
}
```

# Couverture de données: enumeration

Considérons le programme C suivant:

```
switch (size-bound) {  
  case 3:  
    value += mBvGetbit(p_bv, byte_idx)<<2;  
    byte_idx--;  
  case 2:  
    value += mBvGetbit(p_bv, byte_idx)<<1;  
    byte_idx--;  
  case 1:  
    value += mBvGetbit(p_bv, byte_idx);  
}
```

- ▶ Si  $(size - bound) = 3$ , tout le code est couvert.
- ▶ Certains outils l'acceptent, d'autres demandent la couverture de tous les cas
- ▶ Vérifier que **toutes** les valeurs possibles de l'énumération sont effectivement prises
- ▶ Faire le test avec des valeurs hors énumération (facile en C, plus dur en Ada)

# Developpement des tests structurels

- ▶ Pour les tests structurels, il faut écrire des programmes de tests pour exécuter les tests
- ▶ Ces programmes contiennent les tests, et sont *liés* avec le composant à tester.
- ▶ Ils sont souvent exécutés automatiquement par des logiciels d'exécution de tests (Ex: RTRT d'IBM, LDRA TestBed, ...)

# Developpement des tests structurels

- ▶ Attention: le code n'est pas toujours disponible
- ▶ C'est la spécification détaillée qui est utilisée
  - ▶ elle doit décrire le rôle de chaque fonction, et le détail des algos
  - ▶ le test doit déterminer si le code est correct: ex:  $x < N$  ou  $x \leq N$ ?
- ▶ Méthodes agiles: la spécification est donnée par des *stories*, c'est-à-dire des cas d'utilisation



## Execution d'un test



# Programmes de tests

Les programmes de test sont architecturés comme suit :

- ▶ Initialisation de la valeur de retour du test à **KO** !

# Programmes de tests

Les programmes de test sont architecturés comme suit :

- ▶ Initialisation de la valeur de retour du test à **KO** !
- ▶ Initialisation des variables **globales** utilisées par le composant à tester à partir des valeurs d'entrée des jeux de tests

# Programmes de tests

Les programmes de test sont architecturés comme suit :

- ▶ Initialisation de la valeur de retour du test à **KO** !
- ▶ Initialisation des variables **globales** utilisées par le composant à tester à partir des valeurs d'entrée des jeux de tests
- ▶ Appel du composant à tester avec les paramètres initialisées à partir des valeurs d'entrée des jeux de tests
  - ▶ En fonction des spécifications

# Programmes de tests

Les programmes de test sont architecturés comme suit :

- ▶ Initialisation de la valeur de retour du test à **KO** !
  - ▶ Initialisation des variables **globales** utilisées par le composant à tester à partir des valeurs d'entrée des jeux de tests
  - ▶ Appel du composant à tester avec les paramètres initialisés à partir des valeurs d'entrée des jeux de tests
    - ▶ En fonction des spécifications
    - ▶ Les données sont lues ou mise dans le source du test.
- Attention:** Peut s'avérer complexe.

# Programmes de tests

Les programmes de test sont architecturés comme suit :

- ▶ Initialisation de la valeur de retour du test à **KO** !
- ▶ Initialisation des variables **globales** utilisées par le composant à tester à partir des valeurs d'entrée des jeux de tests
- ▶ Appel du composant à tester avec les paramètres initialisés à partir des valeurs d'entrée des jeux de tests
  - ▶ En fonction des spécifications
  - ▶ Les données sont lues ou mise dans le source du test.  
**Attention:** Peut s'avérer complexe.
- ▶ Récupération des sorties produites par le composant (paramètres en sortie **et** variables globales)

# Programmes de tests

Les programmes de test sont architecturés comme suit :

- ▶ Initialisation de la valeur de retour du test à **KO** !
- ▶ Initialisation des variables **globales** utilisées par le composant à tester à partir des valeurs d'entrée des jeux de tests
- ▶ Appel du composant à tester avec les paramètres initialisés à partir des valeurs d'entrée des jeux de tests
  - ▶ En fonction des spécifications
  - ▶ Les données sont lues ou mise dans le source du test.  
**Attention:** Peut s'avérer complexe.
- ▶ Récupération des sorties produites par le composant (paramètres en sortie **et** variables globales)
- ▶ Comparaison des valeurs des sorties obtenues avec les valeurs des sorties décrites dans les jeux de tests

# Programmes de tests

Les programmes de test sont architecturés comme suit :

- ▶ Initialisation de la valeur de retour du test à **KO** !
- ▶ Initialisation des variables **globales** utilisées par le composant à tester à partir des valeurs d'entrée des jeux de tests
- ▶ Appel du composant à tester avec les paramètres initialisés à partir des valeurs d'entrée des jeux de tests
  - ▶ En fonction des spécifications
  - ▶ Les données sont lues ou mise dans le source du test.  
**Attention:** Peut s'avérer complexe.
- ▶ Récupération des sorties produites par le composant (paramètres en sortie **et** variables globales)
- ▶ Comparaison des valeurs des sorties obtenues avec les valeurs des sorties décrites dans les jeux de tests
  - ▶ La comparaison peut-être plus ou moins simple (chaîne de caractères, réel vs flottant, structures, bitstream....)



# Programmes de tests

Les programmes de test sont architecturés comme suit :

- ▶ Initialisation de la valeur de retour du test à **KO** !
- ▶ Initialisation des variables **globales** utilisées par le composant à tester à partir des valeurs d'entrée des jeux de tests
- ▶ Appel du composant à tester avec les paramètres initialisés à partir des valeurs d'entrée des jeux de tests
  - ▶ En fonction des spécifications
  - ▶ Les données sont lues ou mise dans le source du test.  
**Attention:** Peut s'avérer complexe.
- ▶ Récupération des sorties produites par le composant (paramètres en sortie **et** variables globales)
- ▶ Comparaison des valeurs des sorties obtenues avec les valeurs des sorties décrites dans les jeux de tests
  - ▶ La comparaison peut-être plus ou moins simple (chaîne de caractères, réel vs flottant, structures, bitstream....)
- ▶ Enregistrement et affichage du résultat de test.
  - ▶ Certains tests sont manuels: il faut aussi enregistrer leurs résultats !

# Bouchonnage

Le composant fait appel à une fonction externe  $f$ .  
Que l'on dispose ou non de cette fonction, on souhaite dans un premier temps tester le comportement du composant indépendamment de celui de  $f$ . La technique du bouchonnage consiste donc à :

- ▶ ajouter une entrée *artificielle* dans les jeux de test. Cette entrée correspondra à la sortie de la fonction  $f$
- ▶ écrire un *bouchon* pour remplacer le véritable code de la fonction  $f$ : ce code ne fait que retourner la valeur positionnée dans le jeu de test.

Ainsi le testeur peut maîtriser complètement les tests réalisés.

# Code défensif

- ▶ Certaines parties ne sont pas accessibles durant l'exécution *normale* du code:

```
typedef
    enum {RED, ORANGE, GREEN}
    Light_t;
switch (light) {
case RED: ...
    break;
case ORANGE: ...
    break;
case GREEN: ...
    break;
default:
    /* handler */
}
```

```
void GetName(Person* pP) {
    assert(pP != (Person*)0);
    ...
}
```

- ▶ Normalement, ce type de situation ne doit pas se produire, seulement sur des versions en tests.
- ▶ Cela peut se compliquer avec l'utilisation d'exceptions.

# En Résumé

- ▶ Il existe différents tests structurels, qui offrent plus ou moins de garanties
- ▶ Le test structurel est dépendant du code
  - ▶ Toute modification du code peut entraîner des modifications des tests
  - ▶ Le code est plus ou moins proche de la spécification (degré de liberté du développeur)
  - ▶ Il faut une bonne traçabilité, et une gestion de configuration
- ▶ Quelques idées générales ont été présentées, à adapter
  - ▶ En fonction des applications, des données, des algorithmes
  - ▶ En fonction des langages: C, C++/Java (héritage, dispatch, ...), O'Caml, assembleur.

# En Résumé

- ▶ Il existe différents tests structurels, qui offrent plus ou moins de garanties
- ▶ Le test structurel est dépendant du code
  - ▶ Toute modification du code peut entraîner des modifications des tests
  - ▶ Le code est plus ou moins proche de la spécification (degré de liberté du développeur)
  - ▶ Il faut une bonne traçabilité, et une gestion de configuration
- ▶ Quelques idées générales ont été présentées, à adapter
  - ▶ En fonction des applications, des données, des algorithmes
  - ▶ En fonction des langages: C, C++/Java (héritage, dispatch, ...), O'Caml, assembleur.

# En Résumé

- ▶ Il existe différents tests structurels, qui offrent plus ou moins de garanties
- ▶ Le test structurel est dépendant du code
  - ▶ Toute modification du code peut entraîner des modifications des tests
  - ▶ Le code est plus ou moins proche de la spécification (degré de liberté du développeur)
  - ▶ Il faut une bonne traçabilité, et une gestion de configuration
- ▶ Quelques idées générales ont été présentées, à adapter
  - ▶ En fonction des applications, des données, des algorithmes
  - ▶ En fonction des langages: C, C++/Java (héritage, dispatch, ...), O'Caml, assembleur.