

Test et Maintenance de Logiciel

Cours 3

Eléments de structure et test structurel

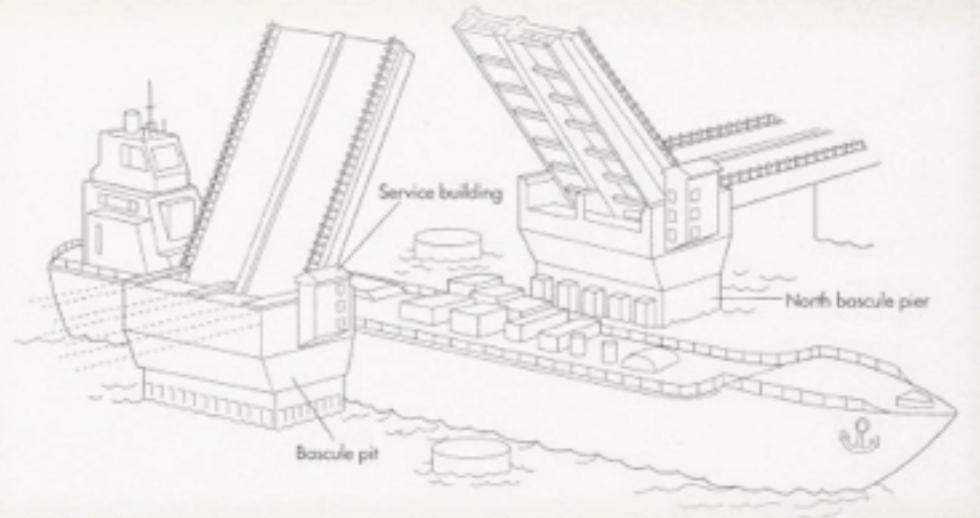
Ileana Ober

Maître de conférences

Université Paul Sabatier

IRIT

<http://www.irit.fr/~Ileana.Ober/>



Année Universitaire 2012-2013

©Ileana Ober

Plan du cours

- ❖ Graphe de contrôle
 - ❖ Bloc
 - ❖ Branchement
 - ❖ PLCS
 - ❖ Chemins d'exécution
- ❖ Chemins d'utilisations des variables

Graphe de contrôle

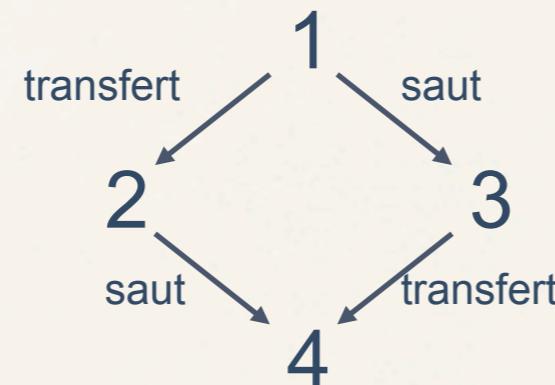
- ⊕ Appelé aussi Control-Flow Graph (CFG)
- ⊕ Graphe à une entrée et une sortie
- ⊕ Met en évidence le flot de contrôle
- ⊕ Permet d'étudier sa structure (code "mort", impasses ...)
- ⊕ Nœuds = portions de code : instructions élémentaires, instructions de décision (donc blocs de base)
- ⊕ Arcs orientés = branchements (sauts et transferts)

Graphe de contrôle - structures classiques

Séquence

1

Choix simple

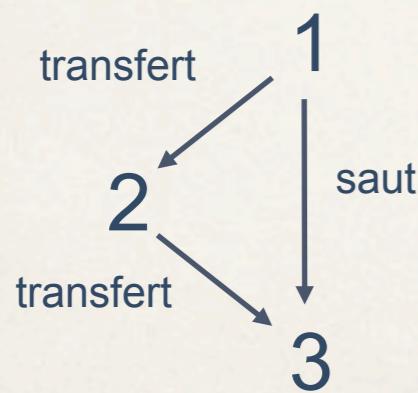


Répétition

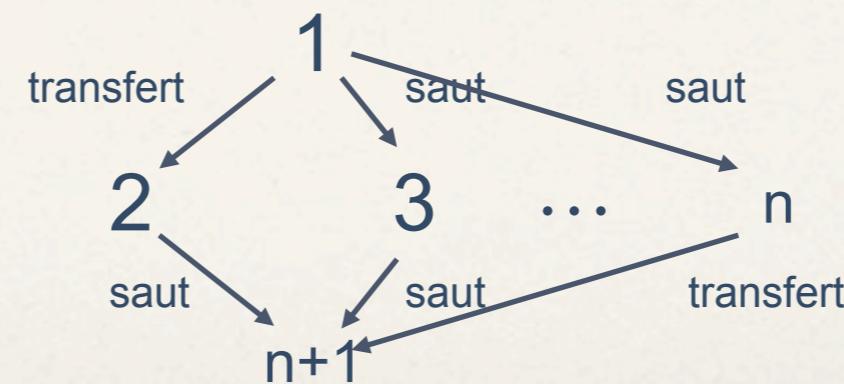


3

Conditionnelle



Choix multiple



Eléments de structure

- ❖ Eléments de structure
 - ❖ Bloc de base
 - ❖ Branchement
 - ❖ PLCS (Portion Linéaire de Code terminée par un Saut)
- ❖ Définitions
 - ❖ toutes basées sur la notion de "graphe de contrôle"
 - ❖ mais varient d'un auteur (et d'un outil) à l'autre !

Bloc de base

- * Plus grande séquence d'une ou plusieurs instructions exécutables consécutives, avec :
 - * un point d'entrée,
 - * un point de sortie,
 - * aucun branchement interne.
- * **Conséquence :**
 - * Toute instruction appartient à un et un seul bloc de base
 - * Si une instruction du bloc de base est exécutée, toutes le sont.
- * **Intérêt :**
 - * Analyse de la structure des programmes
 - * Optimisation

Bloc de base

- ✿ Obtenu à partir de la liste des branchements.
- ✿ Caractérisé par (début, fin), où :
 - ✿ Début
 - ✿ ligne destination d'un branchement,
 - ✿ ligne qui suit une ligne origine d'un branchement,
 - ✿ début du programme.
 - ✿ Fin
 - ✿ ligne origine d'un branchement,
 - ✿ ligne qui précède une ligne destination d'un branchement,
 - ✿ fin du programme.

Bloc de base exemple

```
1 if condition  
2 then  
3     instruction 1  
4     instruction 2  
5     instruction 3  
6 else  
7     instruction 4  
8 endif
```

Bloc de base exemple

```
1  if condition
2  then
3      instruction 1
4      instruction 2
5      instruction 3
6  else
7      instruction 4
8  endif
```

Bloc de base exemple

```
1   if condition
2     then
3       instruction 1
4       instruction 2
5       instruction 3
6     else
7       instruction 4
8   endif
```

Bloc de base exemple

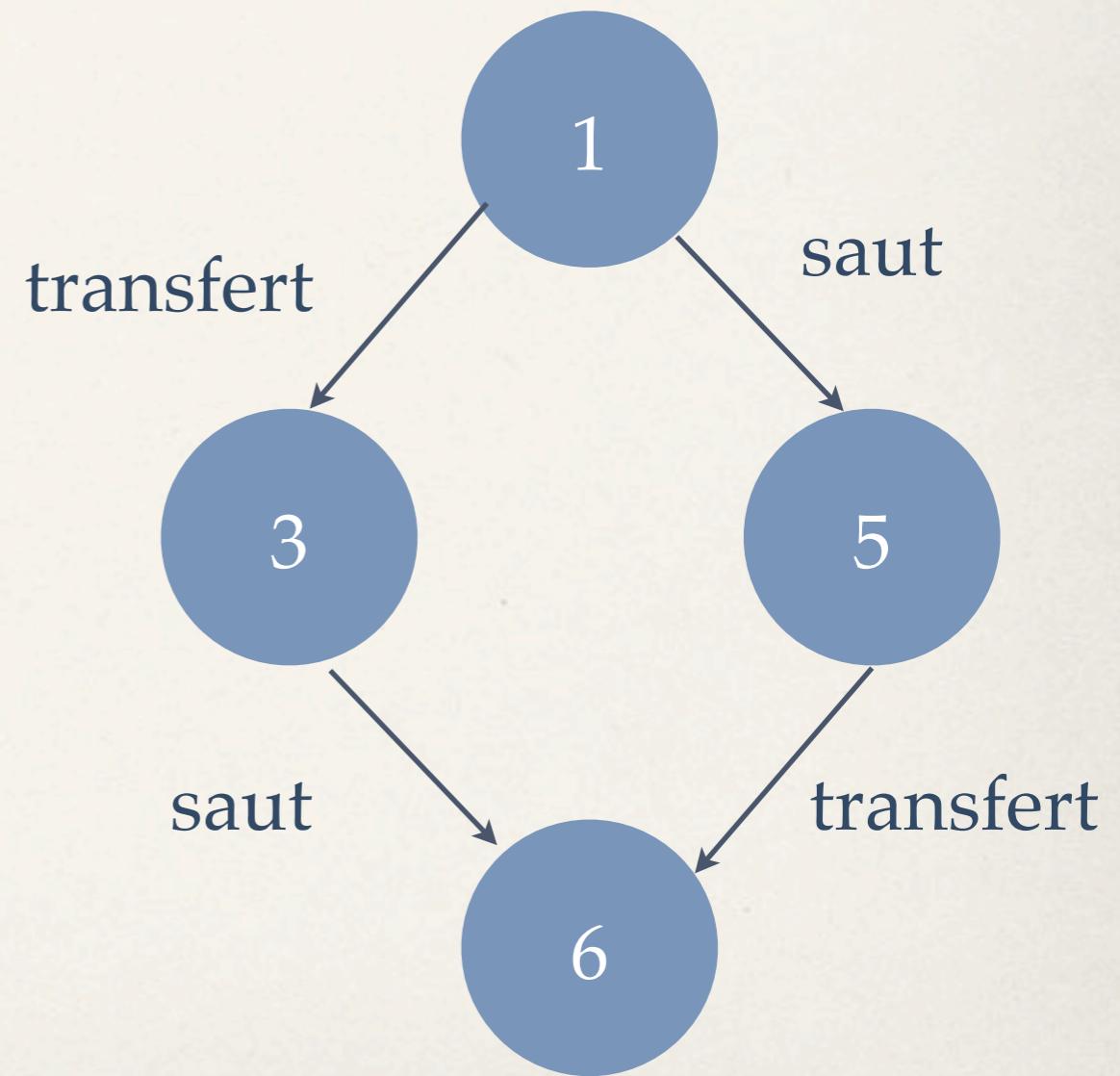
```
1   if condition
2   then
3       instruction 1
4       instruction 2
5       instruction 3
6   else
7       instruction 4
8   endif
```

Branchement

- ❖ Tout *transfert* de contrôle qui résulte d'une *décision*
 - ❖ **Saut** :
 - ❖ Tout branchement qui **ne se fait pas** en séquence
 - ❖ **Transfert** :
 - ❖ Tout branchement qui **n'est pas un saut**
- ❖ Point de *branchement* : point à partir duquel ou vers lequel le contrôle peut être transféré

Branchement exemple

```
1 if condition  
2 then  
3   instruction 1  
4 else  
5   instruction 2  
6 endif
```



Blocs de base et branchements

Source

```
N : Natural ;
```

```
Natural_IO.get(N) ;
```

```
for l in 1..N loop
```

```
...
```

Blocs de base et branchements

Source

N : Natural ;

Natural_IO.get(N) ;

for l in 1..N loop

...

end loop ;

Text_IO.put("Fin") ;

Graphe de contrôle

saut

1

début du S/P

2

boucle

3

corp de boucle

saut

4

fin du S/P

saut

PLCS

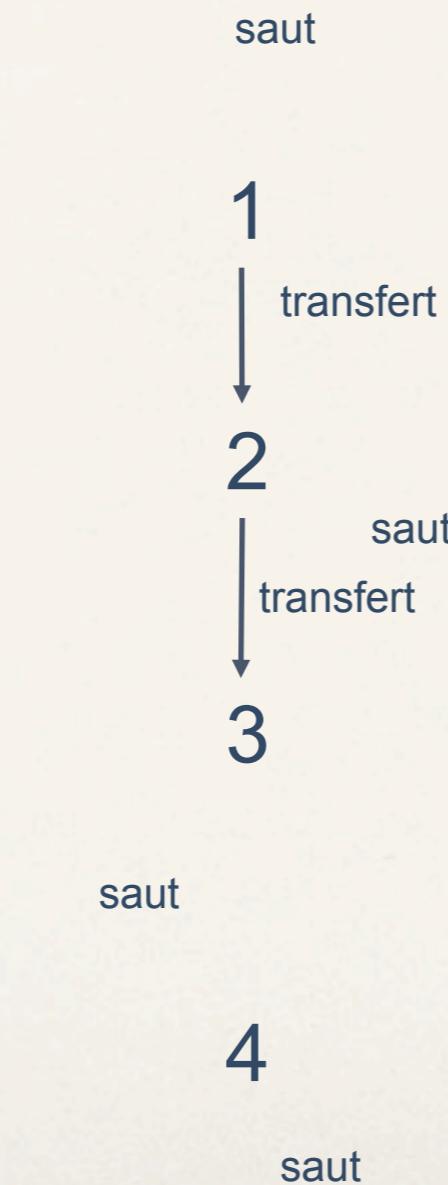
- ✳ Portion Linéaire de Code terminée par un Saut.
- ✳ Portion de saut à saut.
 - ✳ Début
 - ✳ ligne destination d'un branchement,
 - ✳ début du programme.
 - ✳ Fin
 - ✳ ligne origine d'un saut, séparée du début par une séquence linéaire de code
 - ✳ fin du programme.
- ✳ Plusieurs PLCS peuvent partager le même début, la même fin ou la même destination de saut.

PLCS

Source

```
N : Natural ;  
Natural_IO.get(N) ;  
  
for I in 1..N loop  
...  
end loop ;  
Text_IO.put("Fin") ;
```

Graphe de contrôle



PLCS

PLCS

Source

```
N : Natural ;  
Natural_IO.get(N) ;  
for I in 1..N loop  
...  
end loop ;  
Text_IO.put("Fin") ;
```

Graphe de contrôle

saut



PLCS

1&2, 4

1&2&3, 2

2,4

2&3,2

4, -1

Intérêt des PLCS pour le test

Source

```
N : Natural ;  
Natural_IO.get(N) ;  
for I in 1..N loop
```

...

```
end loop ;  
Text_IO.put("Fin") ;
```

Graphe de contrôle



PLCS

```
p1 : 1&2, 4  
p2: 1&2&3, 2  
p3: 2,4  
p4: 2&3,2  
p5: 4, -1
```

Intérêt des PLCS pour le test

Source

```
N : Natural ;  
Natural_IO.get(N) ;  
for I in 1..N loop
```

...

```
end loop ;  
Text_IO.put("Fin") ;
```

Graphe de contrôle



PLCS

p1 : 1&2, 4
p2: 1&2&3, 2
p3: 2,4
p4: 2&3,2
p5: 4, -1

Jeu de
test

Blocs exécutés

Branchements
exécutés

PLCS
exécutés

Remarques

Intérêt des PLCS pour le test

Source

```
N : Natural ;  
Natural_IO.get(N) ;  
for I in 1..N loop
```

...

```
end loop ;
```

```
Text_IO.put("Fin") ;
```

Graphe de contrôle



PLCS

p1 : 1&2, 4
p2: 1&2&3, 2
p3: 2,4
p4: 2&3,2
p5: 4, -1

Jeu de test

Blocs exécutés

Branchements exécutés

PLCS exécutés

Remarques

N=3

1, 2, 3, 2, 3, 2, 3, 4, -1

t1, t2, s1, t2, s1, t2, s1, s2, s3

p2, p4, p4, p3, p5

100% blocs, 100% branchements, 80% PLCS

Intérêt des PLCS pour le test

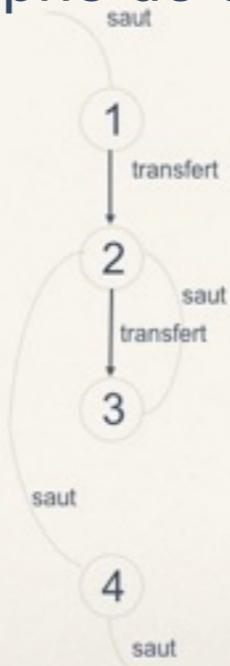
Source

```
N : Natural ;
Natural_IO.get(N) ;
for I in 1..N loop
```

...

```
end loop ;
Text_IO.put("Fin") ;
```

Graphe de contrôle



PLCS

```
p1 : 1&2, 4
p2: 1&2&3, 2
p3: 2,4
p4: 2&3,2
p5: 4, -1
```

Jeu de test	Blocs exécutés	Branchements exécutés	PLCS exécutés	Remarques
N=3	1, 2, 3, 2, 3, 2, 3, 4, -1	t1, t2, s1, t2, s1, t2, s1, s2, s3	p2, p4, p4, p3, p5	100% blocs, 100% branchements, 80% PLCS
N=0	1, 2, 4, -1	t1, s2, s3	p1, p5	nécessaire pour 100% PLCS

Intérêt des PLCS pour le test

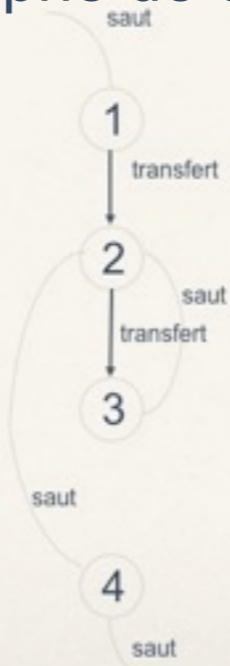
Source

```
N : Natural ;
Natural_IO.get(N) ;
for I in 1..N loop
```

...

```
end loop ;
Text_IO.put("Fin") ;
```

Graphe de contrôle



PLCS

p1 : 1&2, 4
p2: 1&2&3, 2
p3: 2,4
p4: 2&3,2
p5: 4, -1

Jeu de test	Blocs exécutés	Branchements exécutés	PLCS exécutés	Remarques
N=3	1, 2, 3, 2, 3, 2, 3, 4, -1	t1, t2, s1, t2, s1, t2, s1, s2, s3	p2, p4, p4, p3, p5	100% blocs, 100% branchements, 80% PLCS
N=0	1, 2, 4, -1	t1, s2, s3	p1, p5	nécessaire pour 100% PLCS
N=1	1, 2, 3, 2, 4, -1	t1, t2, s1, s2, s3	p2, p3, p5	redondant avec N=3

Chemins dans un graphe de contrôle

- ❖ **Chemin de contrôle** = séquence de blocs décrivant le flot de contrôle qui traverse un programme (ou sous-programme), d'un point d'entrée à un point de sortie.
 - ❖ PLCS = portions de chemins de contrôle.
- ❖ **Chemin d'exécution** = chemin de contrôle effectivement parcouru pendant l'exécution du programme (ou sous-programme).

$\{\text{chemins d'exécution}\} \subseteq \{\text{chemins de contrôle}\}$

Chemins de contrôle

Bloc A

avec $n(A)$ branches
indépendantes

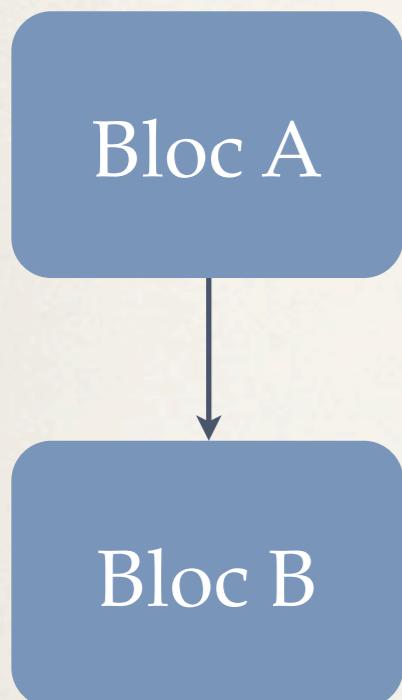
Bloc B

avec $n(B)$ branches
indépendantes

Nombre de chemins dans un graphe de contrôle

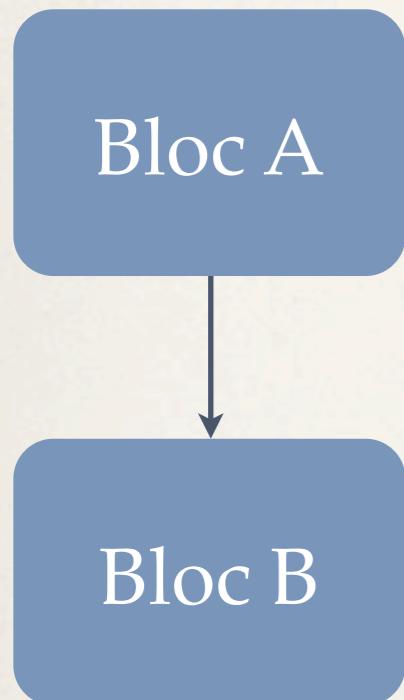
Nombre de chemins dans un graphhe de contrôle

Cas
séquentiel

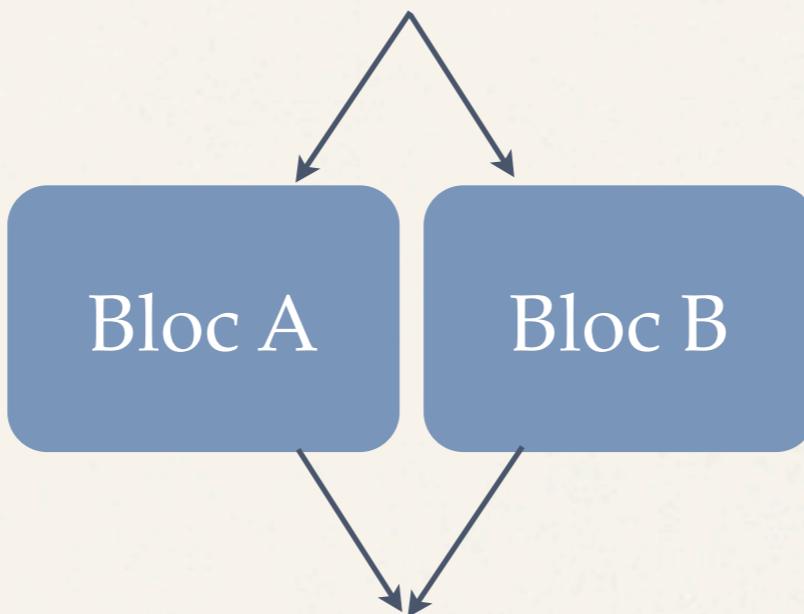


Nombre de chemins dans un graphhe de contrôle

Cas
séquentiel

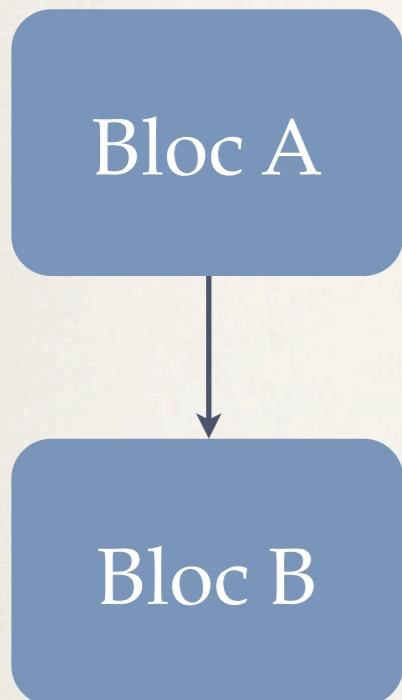


Cas
alternatif

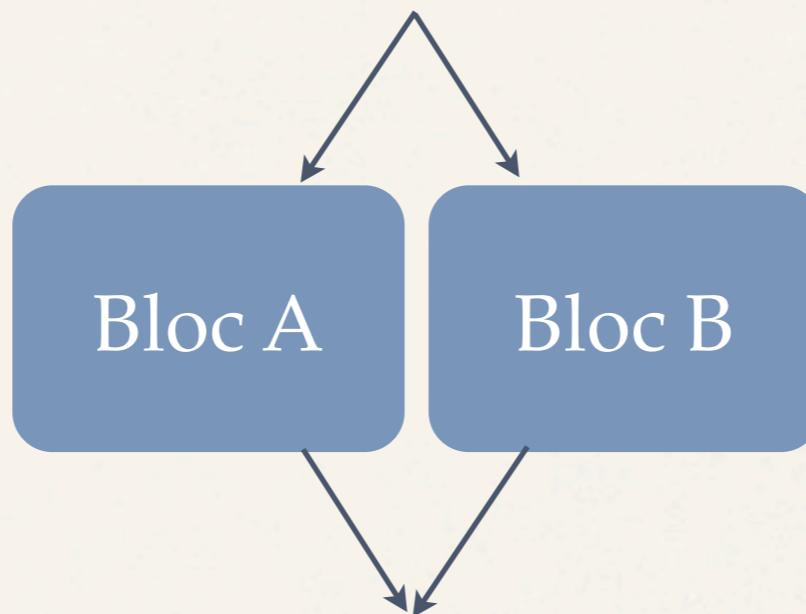


Nombre de chemins dans un graphe de contrôle

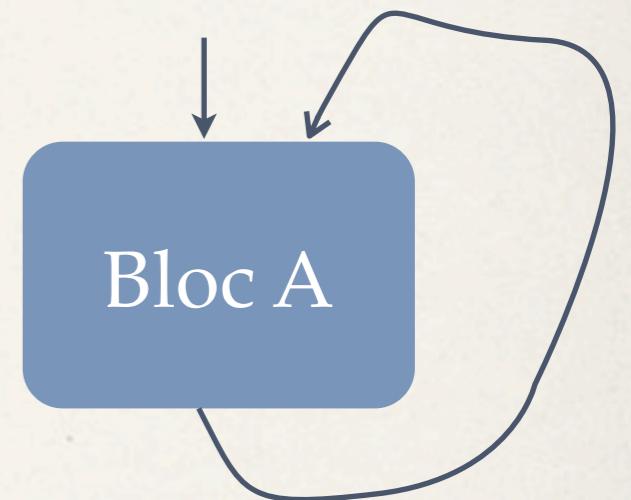
Cas
séquentiel



Cas
alternatif

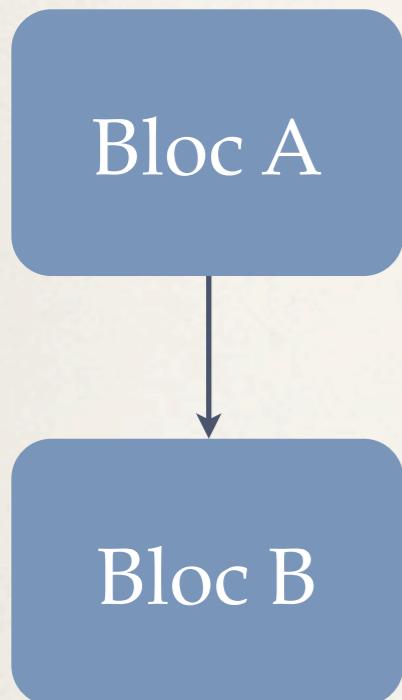


Cas itératif
(max. m itérations)



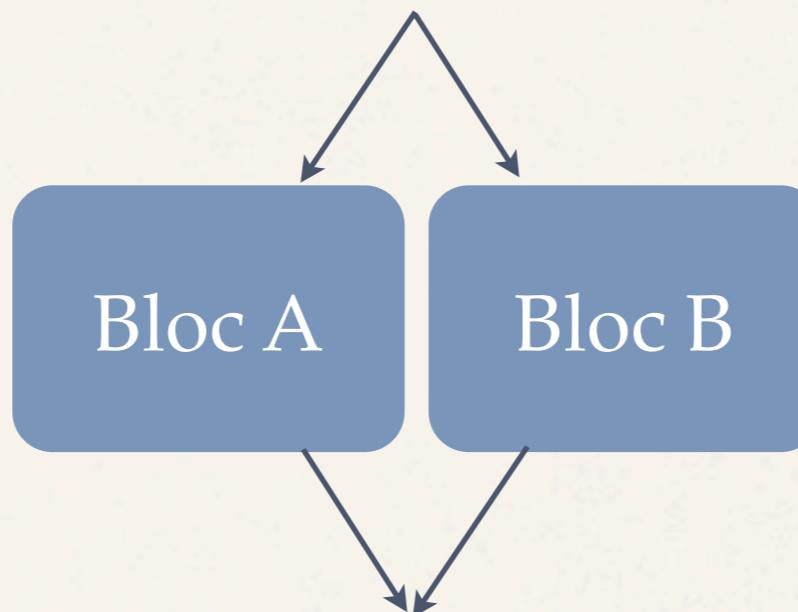
Nombre de chemins dans un graphe de contrôle

Cas
séquentiel

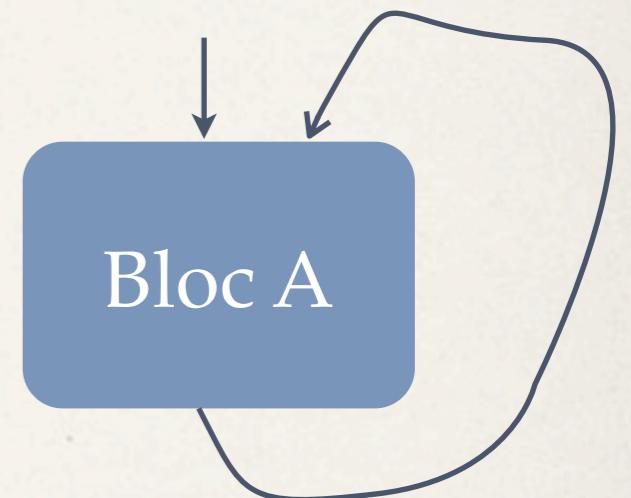


$$n(A) \times n(B)$$

Cas
alternatif

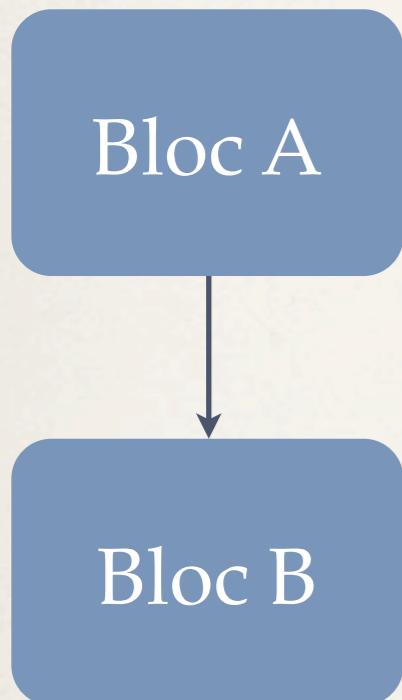


Cas itératif
(max. m itérations)



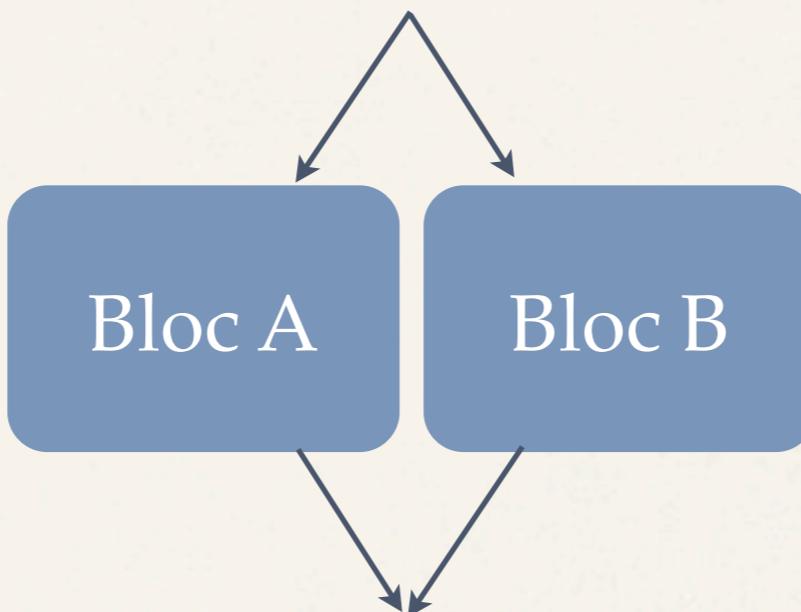
Nombre de chemins dans un graphe de contrôle

Cas
séquentiel



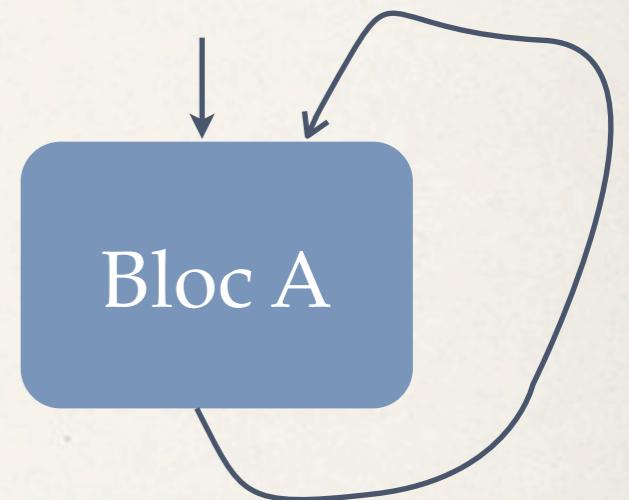
$$n(A) \times n(B)$$

Cas
alternatif



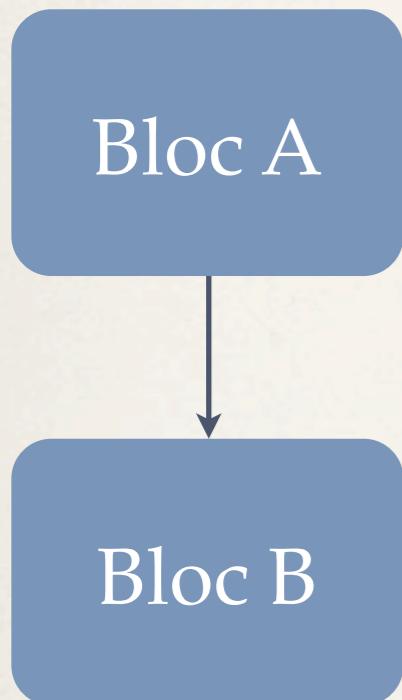
$$n(A) + n(B)$$

Cas itératif
(max. m itérations)



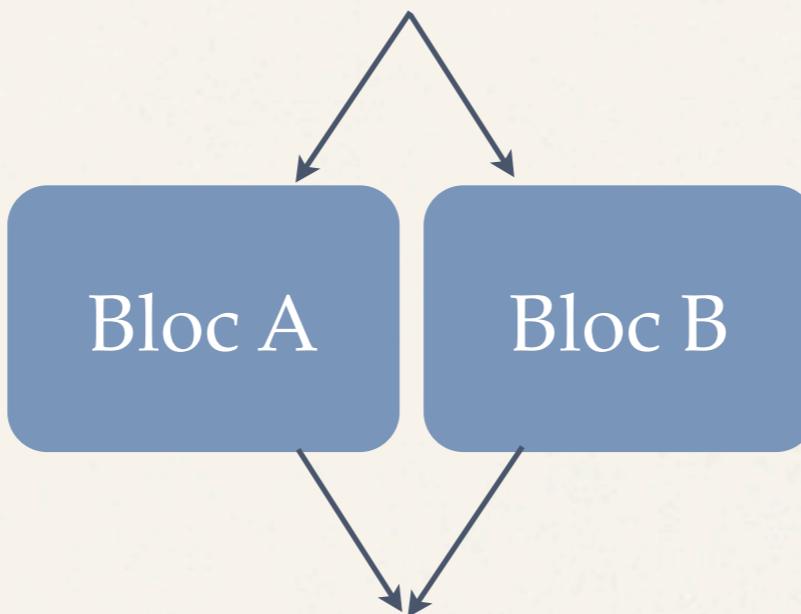
Nombre de chemins dans un graphe de contrôle

Cas
séquentiel



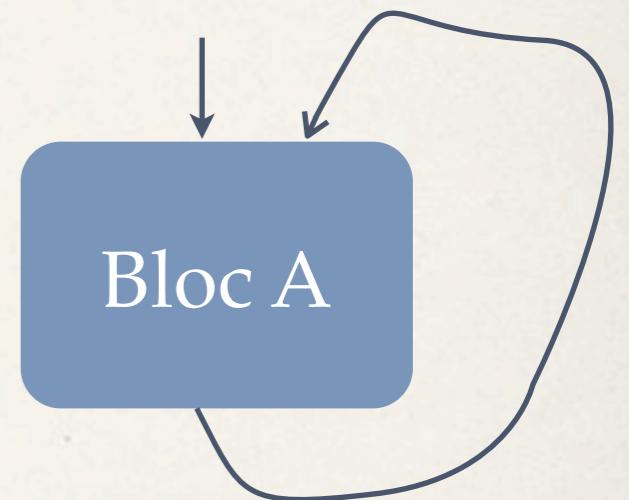
$$n(A) \times n(B)$$

Cas
alternatif



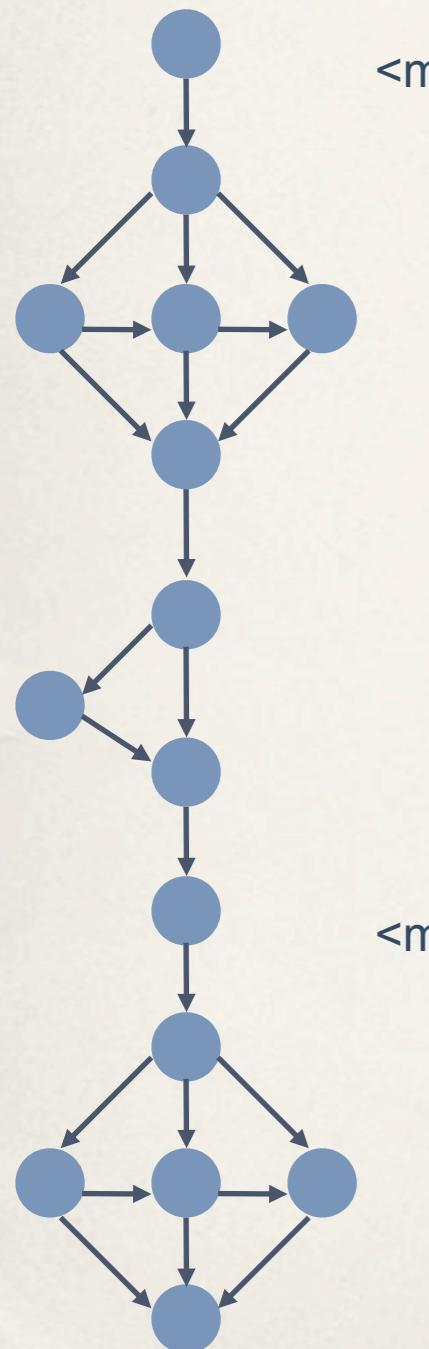
$$n(A) + n(B)$$

Cas itératif
(max. m itérations)



$$1 + n(A) + n(A)^2 + \dots + n(A)^m$$

Chemins de contrôle



$$m = 12 \Rightarrow (6 + 6^2 + \dots + 6^{12}) \times 2 \times (6 + 6^2 + \dots + 6^{12})$$

soit environ 10^{19} chemins de contrôle !

Une stratégie de test basée sur les seuls chemins de contrôle est donc absurde, car le test exhaustif est impossible !

Chemins exécutables et non exécutables

- ❖ Heureusement, les branches ne sont pas aussi indépendantes qu'on l'a supposé
 - ❖ la plupart des chemins de contrôle ne sont pas des chemins d'exécution
 - ❖ parfois plus de 90% !
 - ❖ leur nombre croît de manière exponentielle avec la taille du programme.

Chemins exécutables et non exécutables

- code **non exécutable** = code utilisé par le compilateur, mais à caractère passif (ex. déclarations)
- code **non atteignable** = code qui ne peut être atteint par aucun chemin (de contrôle)

```
return;  
x := 10; -- code non atteignable
```

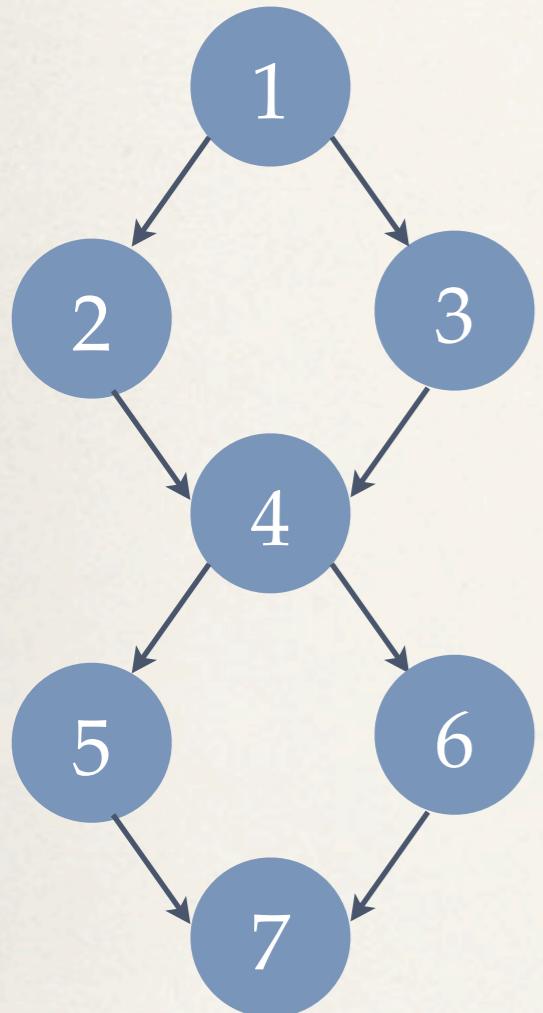
- code **non faisable** = code atteignable par au moins un chemin (de contrôle), mais tel que l'ensemble des prédictats qui doivent être satisfaits n'a pas de solution (pas de chemin d'exécution).

```
if A = B then  
...      -- pas de modification de A et B  
if A !=B then  
...      -- non faisable
```

Exemple 1

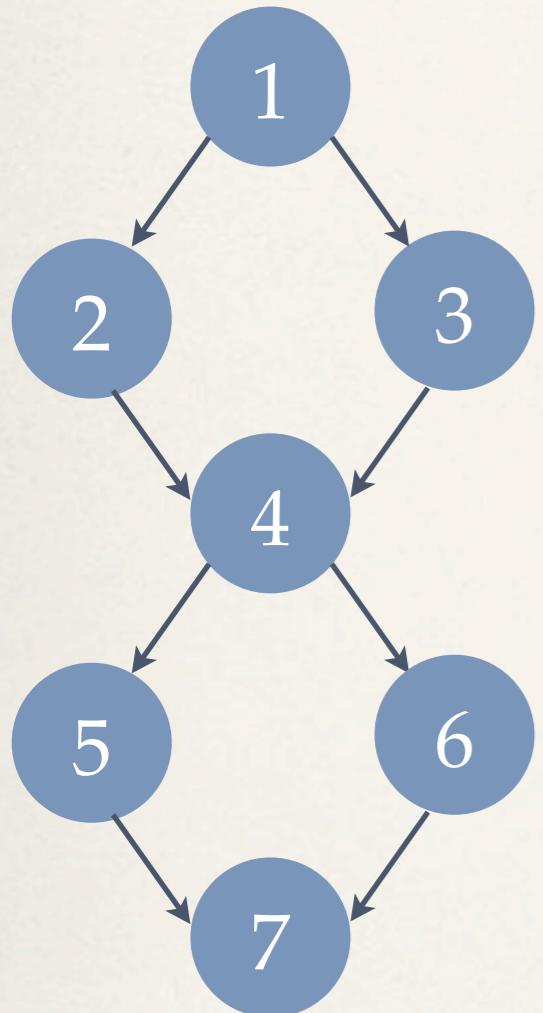
```
if A = B then
    instructions-1 ;
else
    instructions-2 ;
end if ;
... -- A,B non modifiés
if A = B then
    instructions-3 ;
else
    instructions-4 ;
end if ;
```

Exemple 1



```
if A = B then  
    instructions-1 ;  
else  
    instructions-2 ;  
end if ;  
... -- A,B non modifiés  
if A = B then  
    instructions-3 ;  
else  
    instructions-4 ;  
end if ;
```

Exemple 1



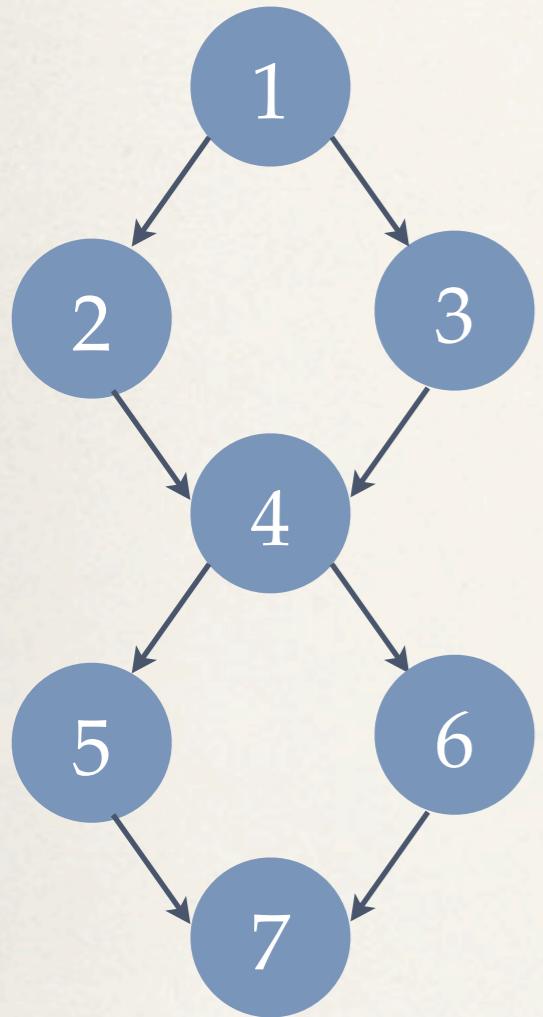
```
if A = B then  
    instructions-1 ;  
else  
    instructions-2 ;  
end if ;  
... -- A,B non modifiés  
if A = B then  
    instructions-3 ;  
else  
    instructions-4 ;  
end if ;
```

- * A,B non altérés par 1er if, ni par ce qui suit
- * 4 chemins de contrôle
- * 2 chemins d'exécution
- * Pas de code non faisable, mais doublets impossibles :
 - * (instructions-1, instructions-4),
 - * (instructions-2, instructions-3)

Exemple 2

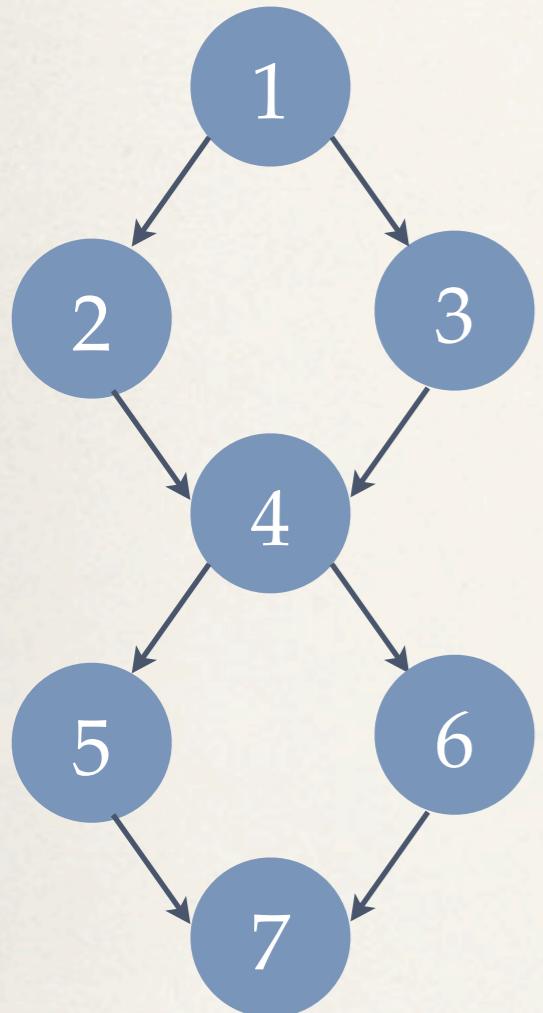
```
if x <= 0 then
    x := -x ;
else
    x := 1 - x ;
end if ;
...
if x = -1 then
    x := 1 ;
else
    x := x + 1 ;
end if ;
```

Exemple 2



```
if x <= 0 then  
    x := -x ;  
else  
    x := 1 - x ;  
end if ;  
...  
if x = -1 then  
    x := 1 ;  
else  
    x := x + 1 ;  
end if ;
```

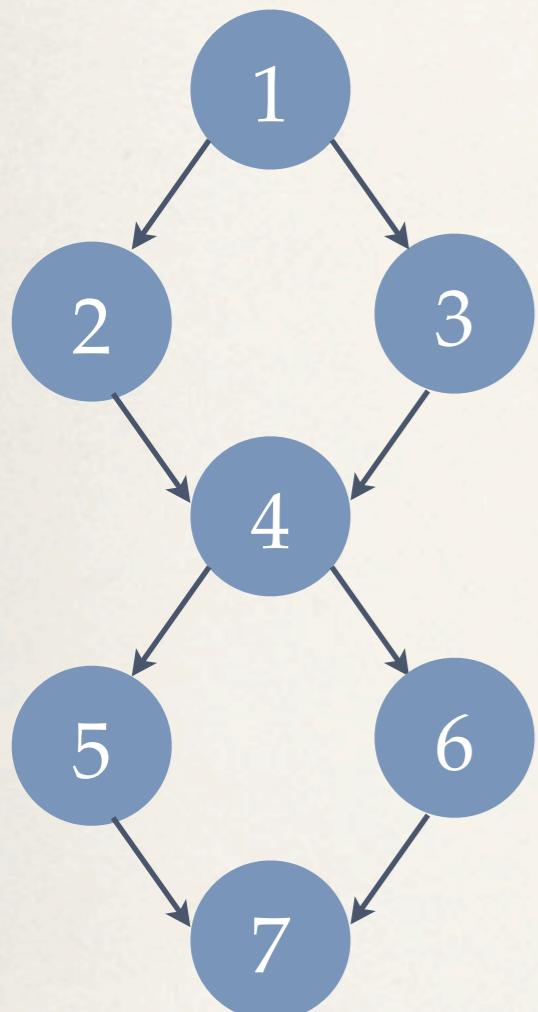
Exemple 2



```
if x <= 0 then  
    x := -x ;  
else  
    x := 1 - x ;  
end if ;  
...  
if x = -1 then  
    x := 1 ;  
else  
    x := x + 1 ;  
end if ;
```

- ✳ X est altéré par le 1er if
- ✳ 4 chemins de contrôle
- ✳ 3 chemins d'exécution
- ✳ Pas de code non faisable, mais **doublet impossible** : (2,5).
- ✳ (1, 2, 4, 5, 7) est un chemin de contrôle, mais pas un chemin d'exécution
- ✳ généralisation possible aux doublets, triplets

Exemple 2



```
if x <= 0 then  
    x := -x ;  
else  
    x := 1 - x ;  
end if ;  
...  
if x = -1 then  
    x := 1 ;  
else  
    x := x + 1 ;  
end if ;
```

- ✿ X est altéré par le 1er if
- ✿ 4 chemins de contrôle
- ✿ 3 chemins d'exécution
- ✿ Pas de code non faisable, mais **doublet impossible** : (2,5).
- ✿ (1, 2, 4, 5, 7) est un chemin de contrôle, mais pas un chemin d'exécution
- ✿ généralisation possible aux doublets, triplets

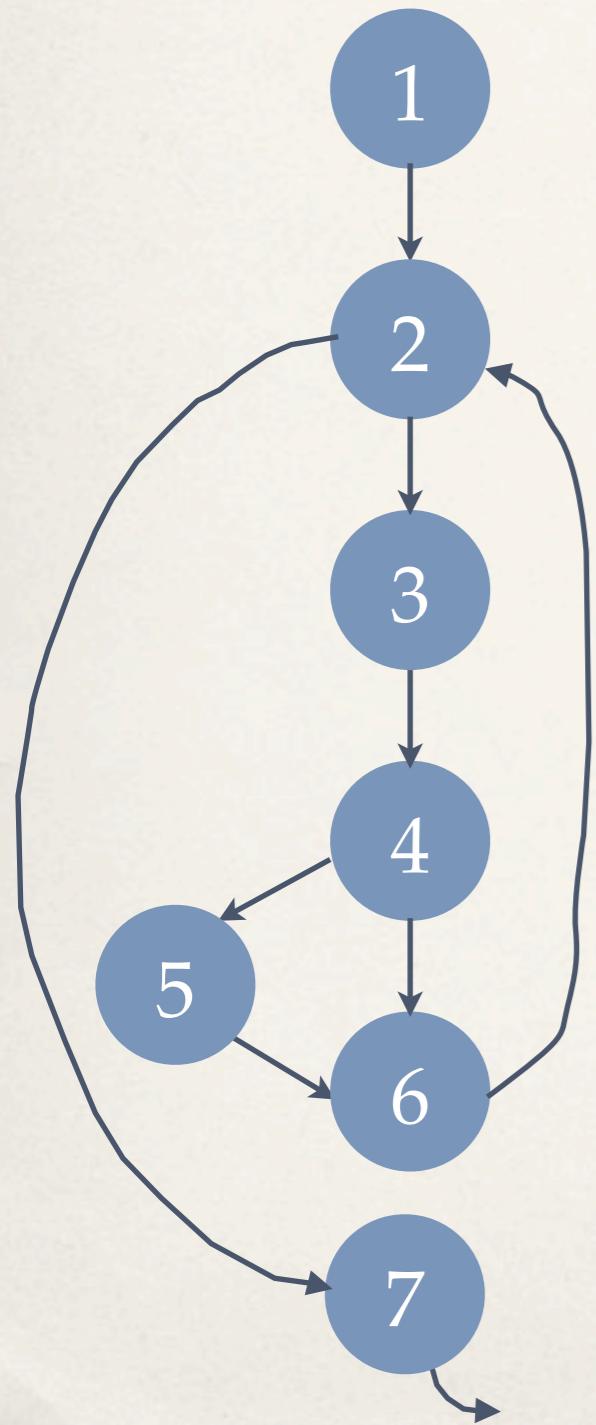
PLCS non faisable = mauvaise programmation !

Exemple 3 - tri à bulles

```
FINI : Boolean := False ;
while not FINI loop
    -- prendre les éléments consécutifs deux à deux
    -- et permuter ceux qui sont mal ordonnés
    ...
    if ... then -- pas d'échange
        FINI := True ;
    end if ;
end loop ;
```

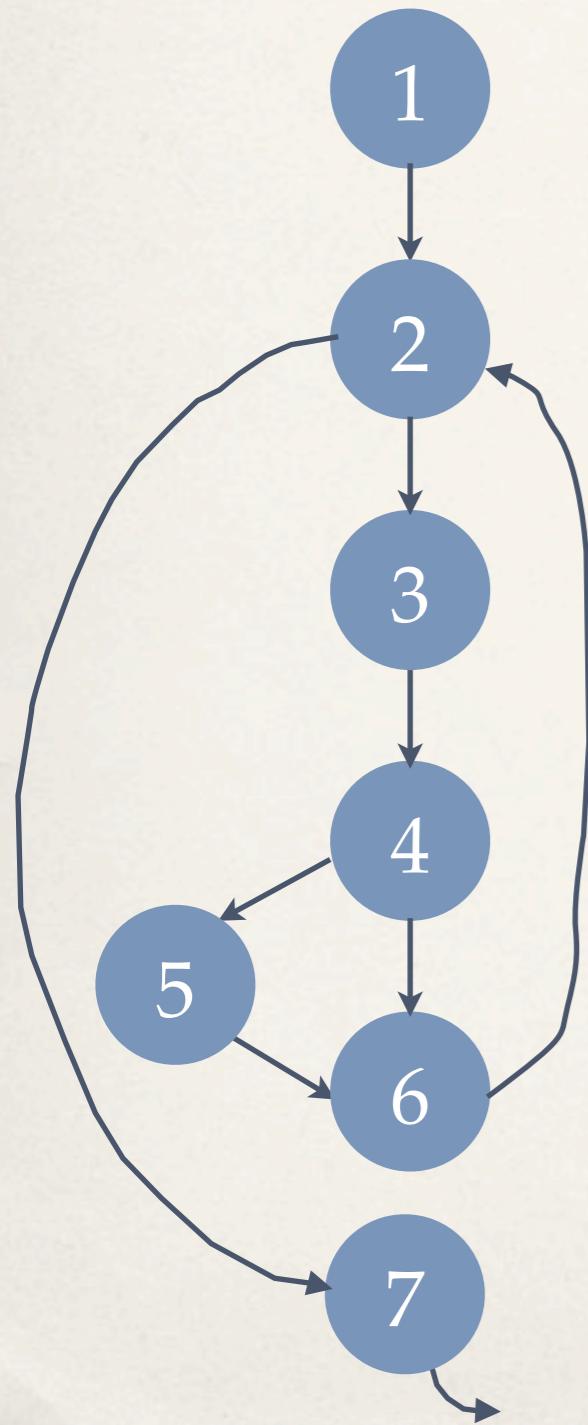


Exemple 3 - tri à bulles



```
FINI : Boolean := False ;
while not FINI loop
    -- prendre les éléments consécutifs deux à deux
    -- et permuter ceux qui sont mal ordonnés
    ...
    if ... then -- pas d'échange
        FINI := True ;
    end if ;
end loop ;
```

Exemple 3 - tri à bulles



```
FINI : Boolean := False ;  
while not FINI loop  
    -- prendre les éléments consécutifs deux à deux  
    -- et permuter ceux qui sont mal ordonnés  
    ...  
    if ... then -- pas d'échange  
        FINI := True ;  
    end if ;  
end loop ;
```

- ✿ La PLCS 1&2&7, -1 n'est pas faisable.
- ✿ Elle correspond au test inutile au 1er passage
⇒ restructurer éventuellement en repeat.

Chemins exécutables et non exécutables

- ❖ Les chemins impossibles :
 - ❖ rendent la procédure de test confuse ;
 - ❖ entravent la maintenance (déroutent le lecteur).
- ❖ Le nombre de chemins exécutables est encore très élevé ; par contre, le nombre de PLCS ne croît que de manière **linéaire** avec la taille du programme.
 - ❖ on pourra donc bâtir une stratégie de test viable sur les PLCS, à condition de pouvoir identifier les PLCS faisables et non faisables.

Graphe de dépendance

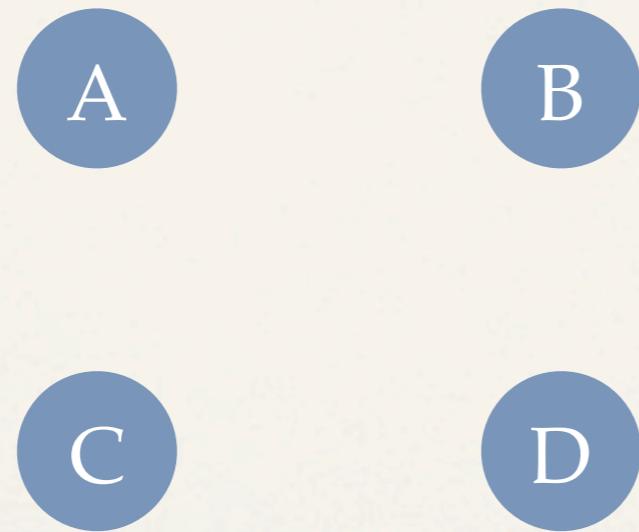
- ✿ Met en évidence les dépendances entre variables
- ✿ Permet d'étudier les dépendances entre segments de programme, les répercussions de modifications ...
- ✿ Facilite l'élaboration de jeux de test
- ✿ Nœuds = variables
- ✿ Arcs orientés = relation "utilisé pour définir"

Exemple - graphe de dépendance

```
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;
```

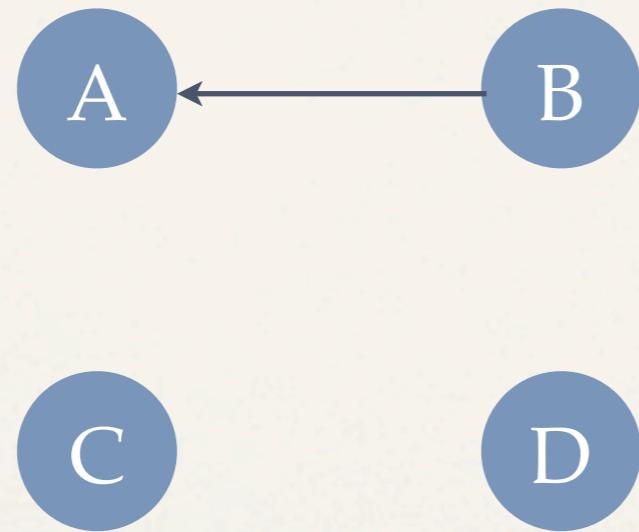
Exemple - graphe de dépendance

```
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;
```



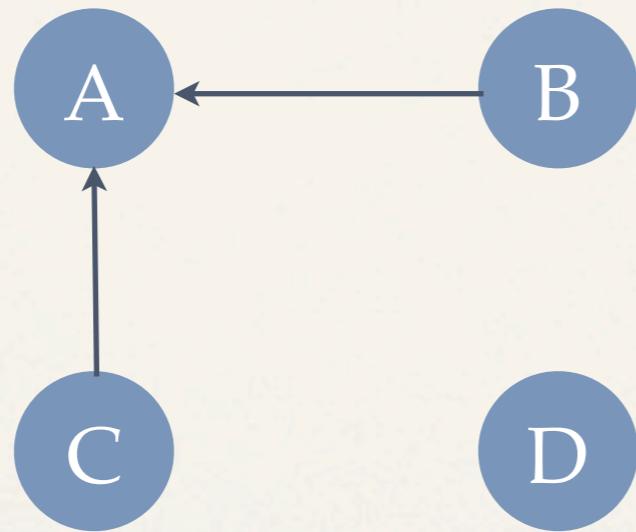
Exemple - graphe de dépendance

```
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;
```



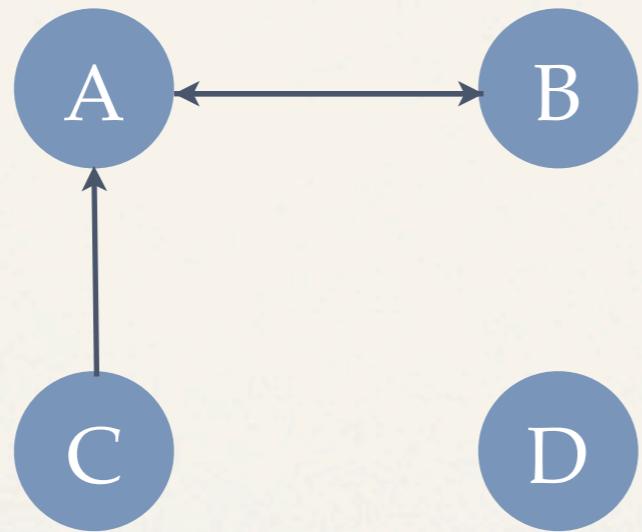
Exemple - graphe de dépendance

```
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;
```



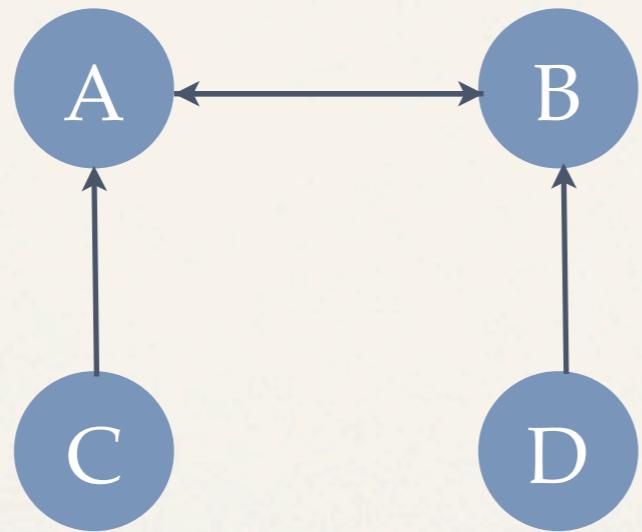
Exemple - graphe de dépendance

```
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;
```



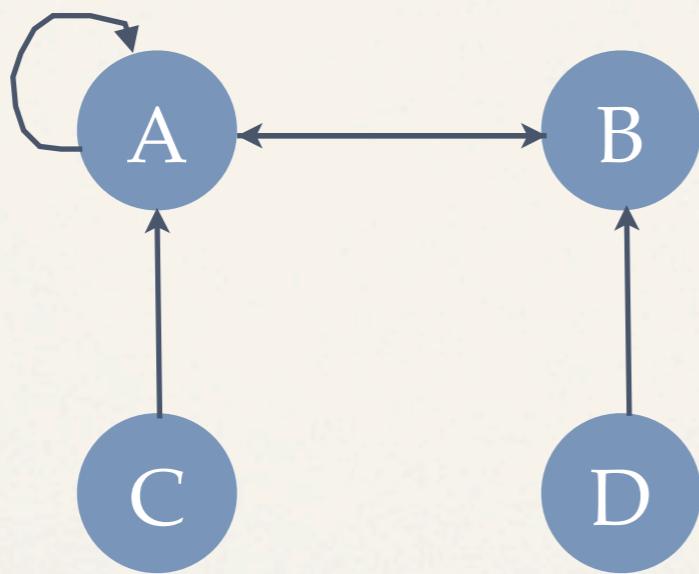
Exemple - graphe de dépendance

```
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;
```



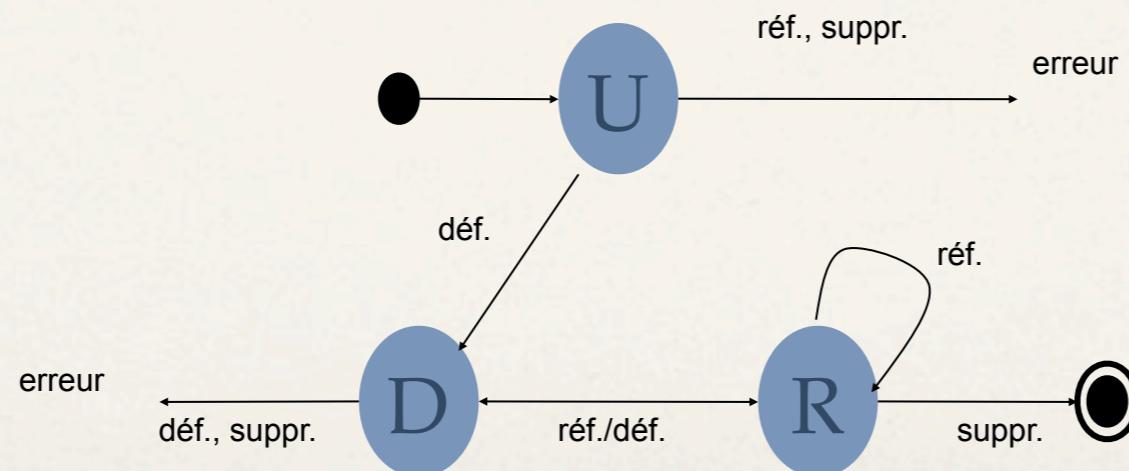
Exemple - graphe de dépendance

```
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;
```



Graphe de transitions des variables

- * Met en évidence les transitions entre états d'une variable
- * Permet de contrôler le bon usage des variables
- * Nœuds : états d'une variable
- * Arcs orientés : transitions entre états, liées aux instructions



Chemins d'utilisation des états des variables

- * définition: chemins dans le graphe de transitions des états des variables
- * but: détection des anomalies du flot de données

Chemins d'utilisation des états des variables

- * définition: chemins dans le graphe de transitions des états des variables
- * but: détection des anomalies du flot de données

Exemple 1:

```
...
x :=1;
x :=2; -- x définie 2 fois
...
```

Chemins d'utilisation des états des variables

- * définition: chemins dans le graphe de transitions des états des variables
- * but: détection des anomalies du flot de données

Exemple 1:

```
...
x :=1;
x :=2; -- x définie 2 fois
...
```

Exemple 2:

```
declare
    x : int
begin
    ...
    x := 2; -- x définie, mais non utilisée
end
```

Chemins d'utilisation des états des variables

- * définition: chemins dans le graphe de transitions des états des variables
- * but: détection des anomalies du flot de données

Exemple 1:

```
...
X :=1;
X :=2; -- X définie 2 fois
...
```

Exemple 2:

```
declare
    X : int
begin
    ...
    X := 2; -- X définie, mais non utilisée
end
```

Exemple 3:

```
declare
    X, Y : int
begin
    Y:=X; -- X référencée sans être initialisée
...
...
```

Chemins d'utilisation des états des variables

- * définition: chemins dans le graphe de transitions des états des variables
- * but: détection des anomalies du flot de données

Exemple 1:

```
...
X :=1;
X :=2; -- X définie 2 fois
...
```

Exemple 2:

```
declare
    X : int
begin
    ...
    X := 2; -- X définie, mais non utilisée
end
```

Exemple 3:

```
declare
    X, Y : int
begin
    Y:=X; -- X référencée sans être initialisée
...
...
```

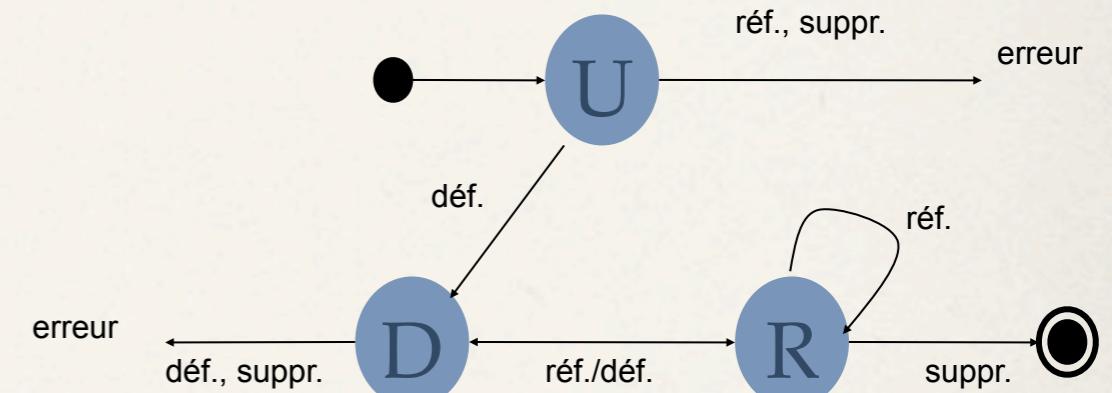
Exemple 2:

```
declare
    X : int
begin
    ...
    ... -- X déclarée, mais non utilisée
end
```

Chemins d'utilisation des variables

- Expressions des chemins dans le graphe de transition des états declare

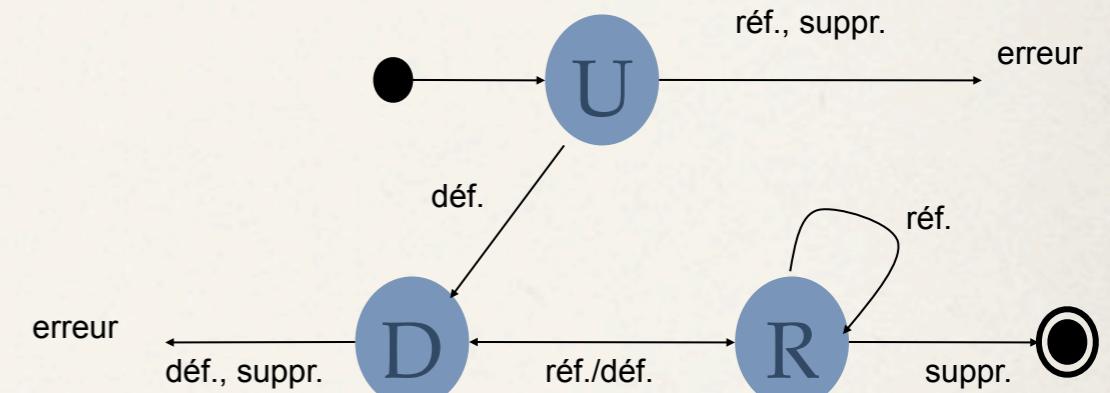
```
A, X : float ;  
begin  
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;  
end ;  
-- on suppose B, C, D définies
```



Chemins d'utilisation des variables

- Expressions des chemins dans le graphe de transition des états declare

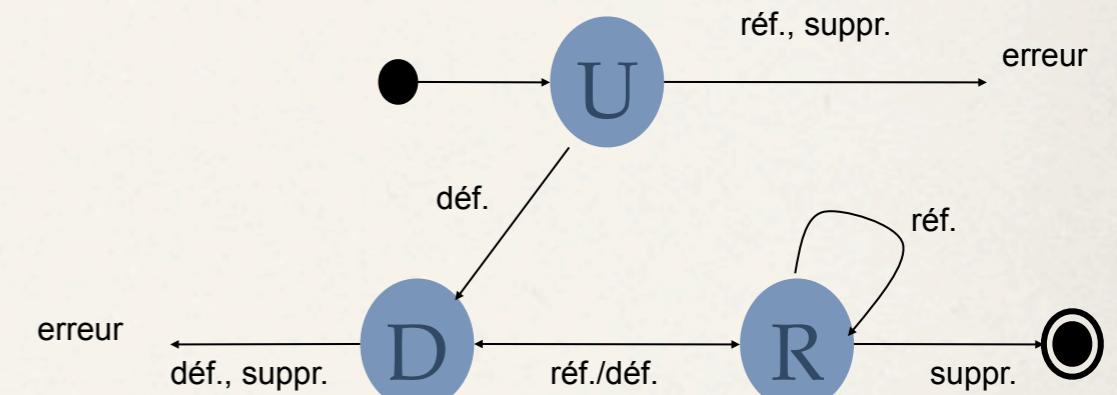
```
A, X : float ;  
begin  
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;  
end ;  
-- on suppose B, C, D définies
```



Chemins d'utilisation des variables

- Expressions des chemins dans le graphe de transition des états declare

```
A, X : float ;  
begin  
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;  
end ;  
-- on suppose B, C, D définies
```

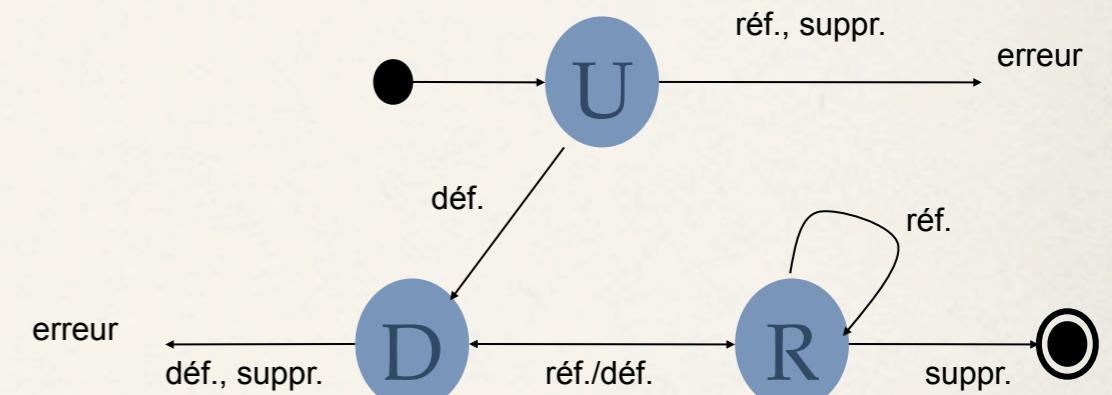


Chemins d'utilisation des variables

- Expressions des chemins dans le graphe de transition des états declare

```
A, X : float ;  
begin  
A := B + C ;  
B := A + D ;  
A := A + 1.0 ;  
B := A + 2.0 ;  
end ;  
-- on suppose B, C, D définies
```

- Expressions des chemins
- pour A : UDRRDR
- Trouver pour B, C, D ...



Anomalies

- ✳ URs
- ✳ sDDs
- ✳ sD

où s est une suite arbitraire de R et D

Chemins d'utilisation des variables - le cas des boucles

- ❖ L'expression des chemins correspondant à deux passages suffit
 - ❖ Le cas 0 passage est en général inclus
 - ❖ 1 passage ne suffit pas : toute combinaison anormale détectée pour 1 passage se retrouve dans l'expression de chemin correspondant à 2 passages, mais pas l'inverse.
 - ❖ Les suivants n'apportent plus de nouvelle combinaison

Chemins d'utilisation des variables - le cas des boucles

- ❖ L'expression des chemins correspondant à deux passages suffit
 - ❖ Le cas 0 passage est en général inclus
 - ❖ 1 passage ne suffit pas : toute combinaison anormale détectée pour 1 passage se retrouve dans l'expression de chemin correspondant à 2 passages, mais pas l'inverse.
 - ❖ Les suivants n'apportent plus de nouvelle combinaison

```
...          -- while, ou for
loop        -- pour X, en 0 passage : rien
  X := 1 ;  -- en 1 passage : ... DRD ...
  ...
  X := X + 1 ;
end loop ;           -- en 2 passages : ... DRDDRD ...
```

Bilan test structurel

- ❖ Complémentaire aux autres techniques de test
- ❖ S'appuie sur l'analyse du code source
- ❖ Automatisable, mais oracles pas facile à trouver
- ❖ Très outillé
- ❖ Difficultés issues du fait qu'on ne peut pas distinguer entre chemins exécutables et non-exécutables => métriques faussées