# Cours de Test Logiciel

## Leçon 5 : test par mutations

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

sebastien.bardin@cea.fr
http://sebastien.bardin.free.fr

# Rappel : sélection de tests

## Méthodes Black Box

- combinatoire
- partitionnel
- aux limites
- couverture fonctionnelle

## Méthodes Probabilistes

- aléatoire
- statistique

## Méthodes White Box

- couverture structurelle (contrôle, données)
- mutation

# Plan

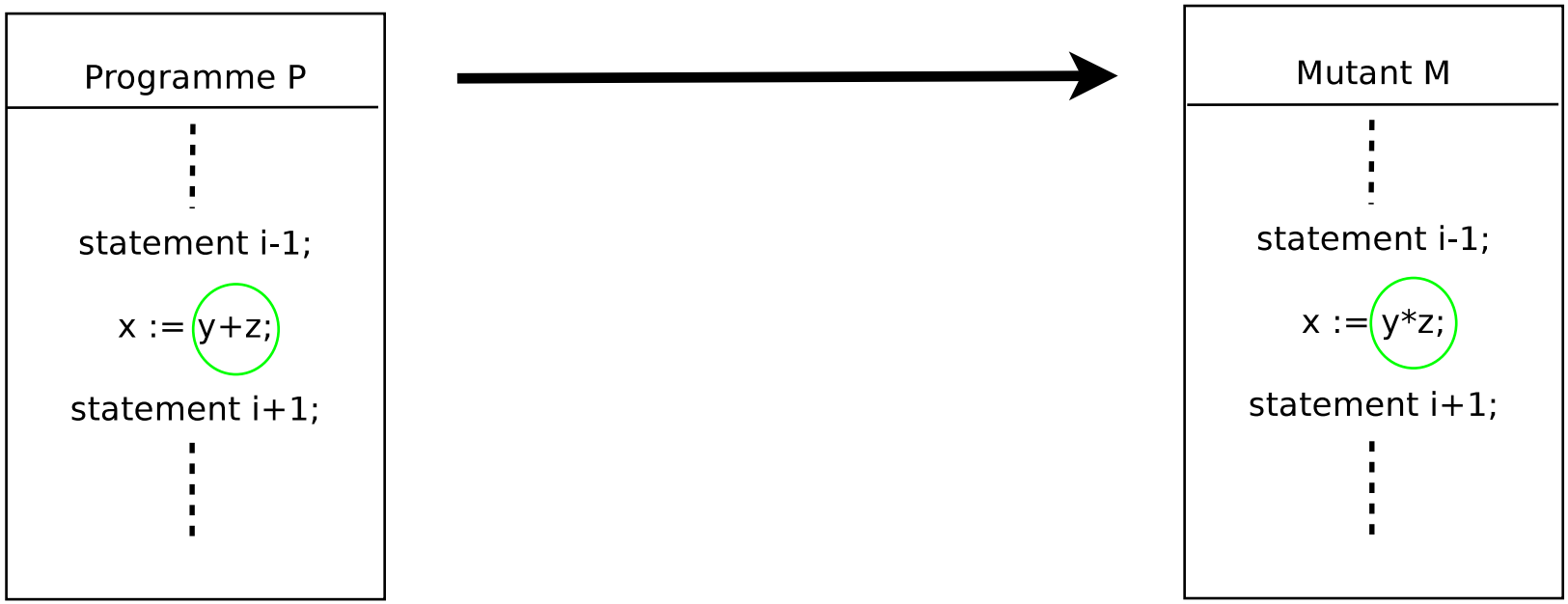
- Rappels
- Principe
- Expressivité
- En pratique : moteur de mutation, calcul de score
- Opérateurs de mutation
- Discussion

# Test par Mutations

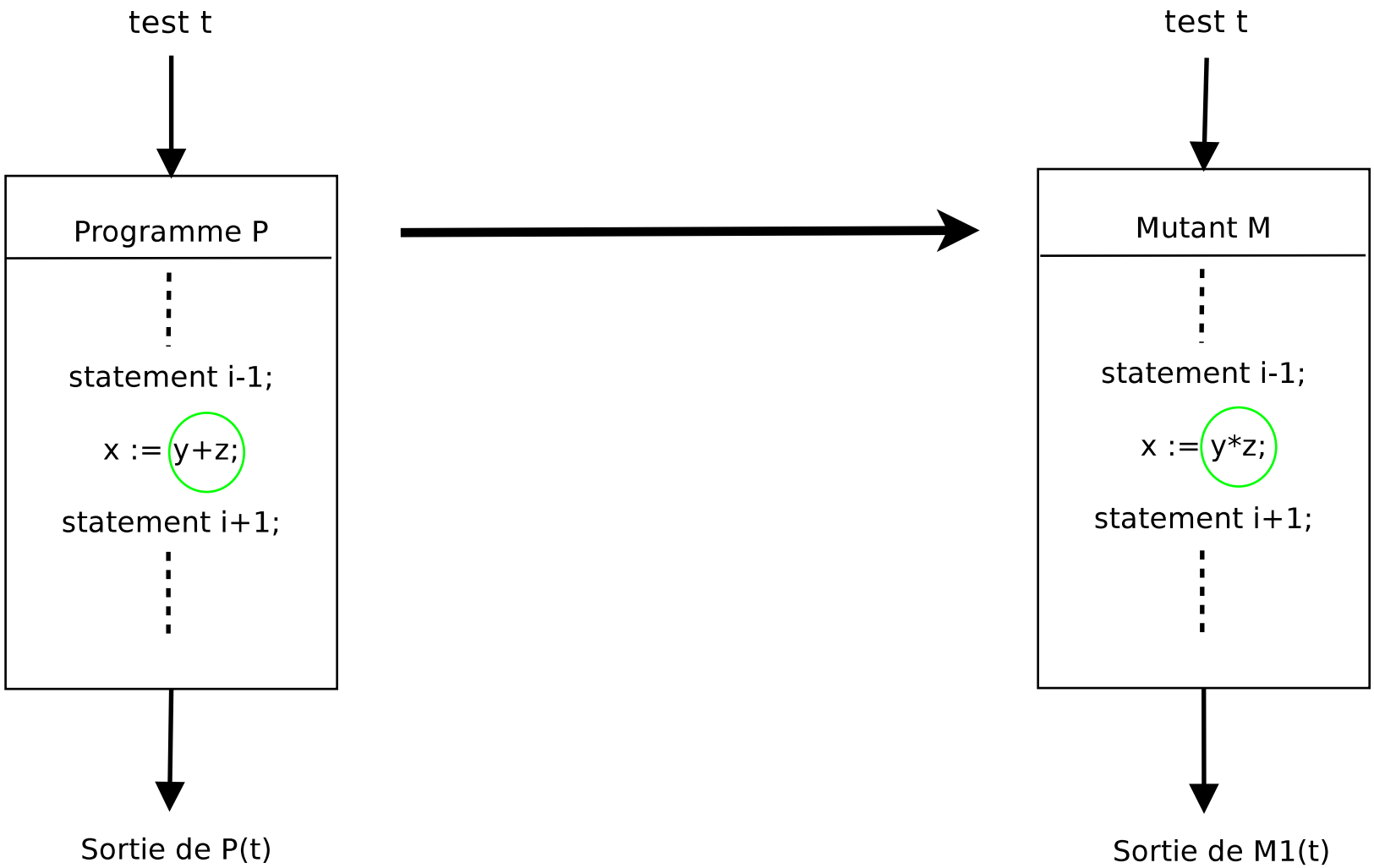
## Principe (mutations fortes)

- modifier le programme P en P' en injectant une modification syntaxique correcte (P' compile toujours)
- idéalement, le comportement de P' est différent de celui de P
- sélectionner une DT qui met en évidence cette différence de comportement (= tuer le mutant P')

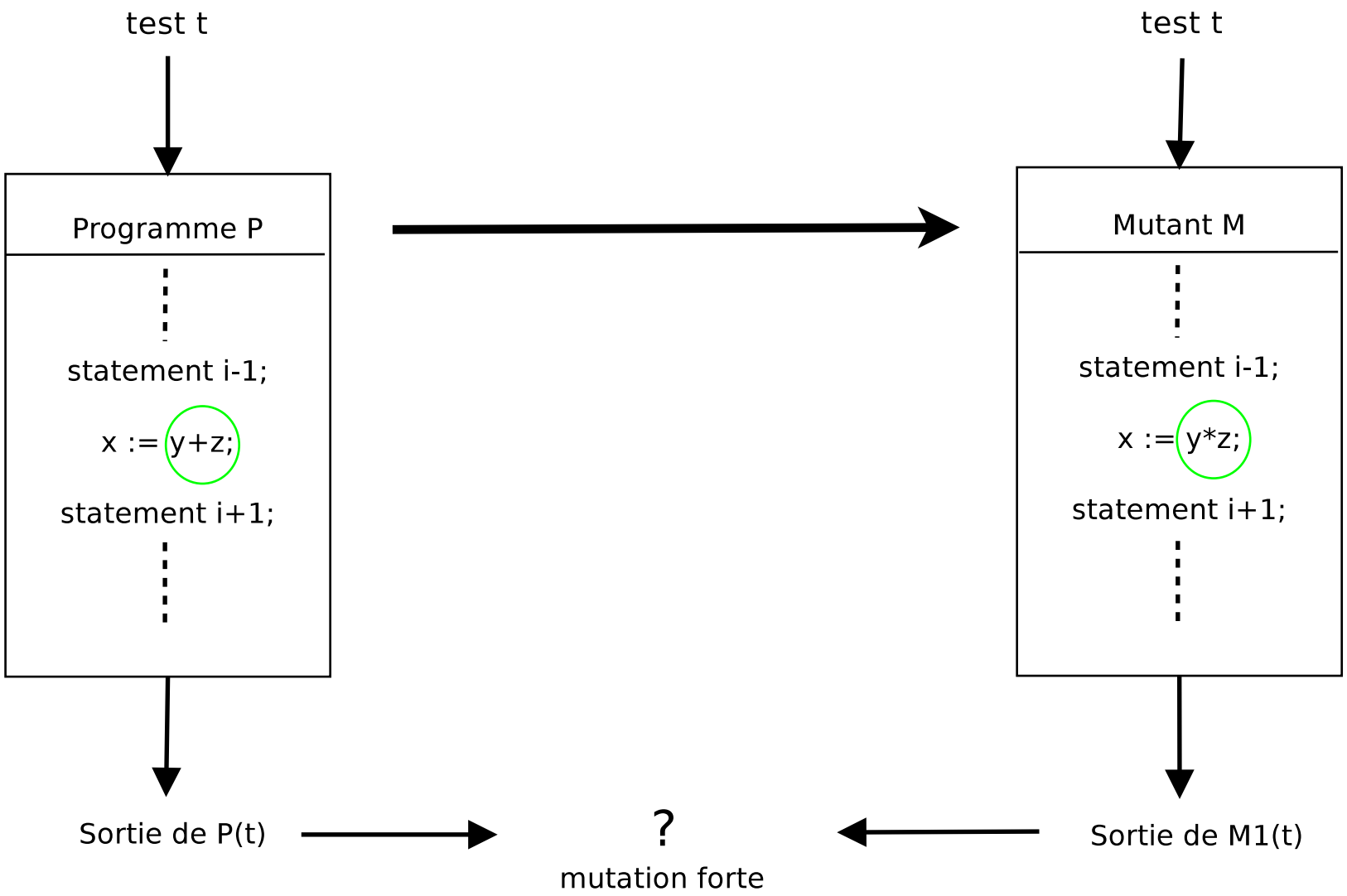
# Test par Mutations (fortes)



# Test par Mutations (fortes)



# Test par Mutations (fortes)



# Vocabulaire

- mutation de P : modification syntaxique de P
- mutant de P : P' obtenu par mutation de P
- TS tue P : $\exists t \in TS$ tq P(t) $\neq$ P'(t)
- score de mutation de TS

  (# mutants tués)/(# mutants)

# Détecter un mutant

Les 3 conditions suivantes doivent être remplies

Accessibilité : P(t) atteint la mutation

Infection : P(t) $\neq$ P(t') juste après la mutation

Propagation : P(t) $\neq$ P(t') à la fin du programme

# Exemples de mutations

Pour l'instruction suivante :

```
if a > 8 then x := y+1
```

on peut considérer les mutants suivants :

- `if a < 8 then x := y+1`
- `if a ≥ 8 then x := y+1`
- `if a > 8 then x := y-1`
- `if a > 8 then x := y`

Pour un programme donné, on considèrera un très grande nombre de mutants

# Quelques exemples d'opérateurs de mutation

Opérateurs classiques de mutations [Offutt-Ammann]

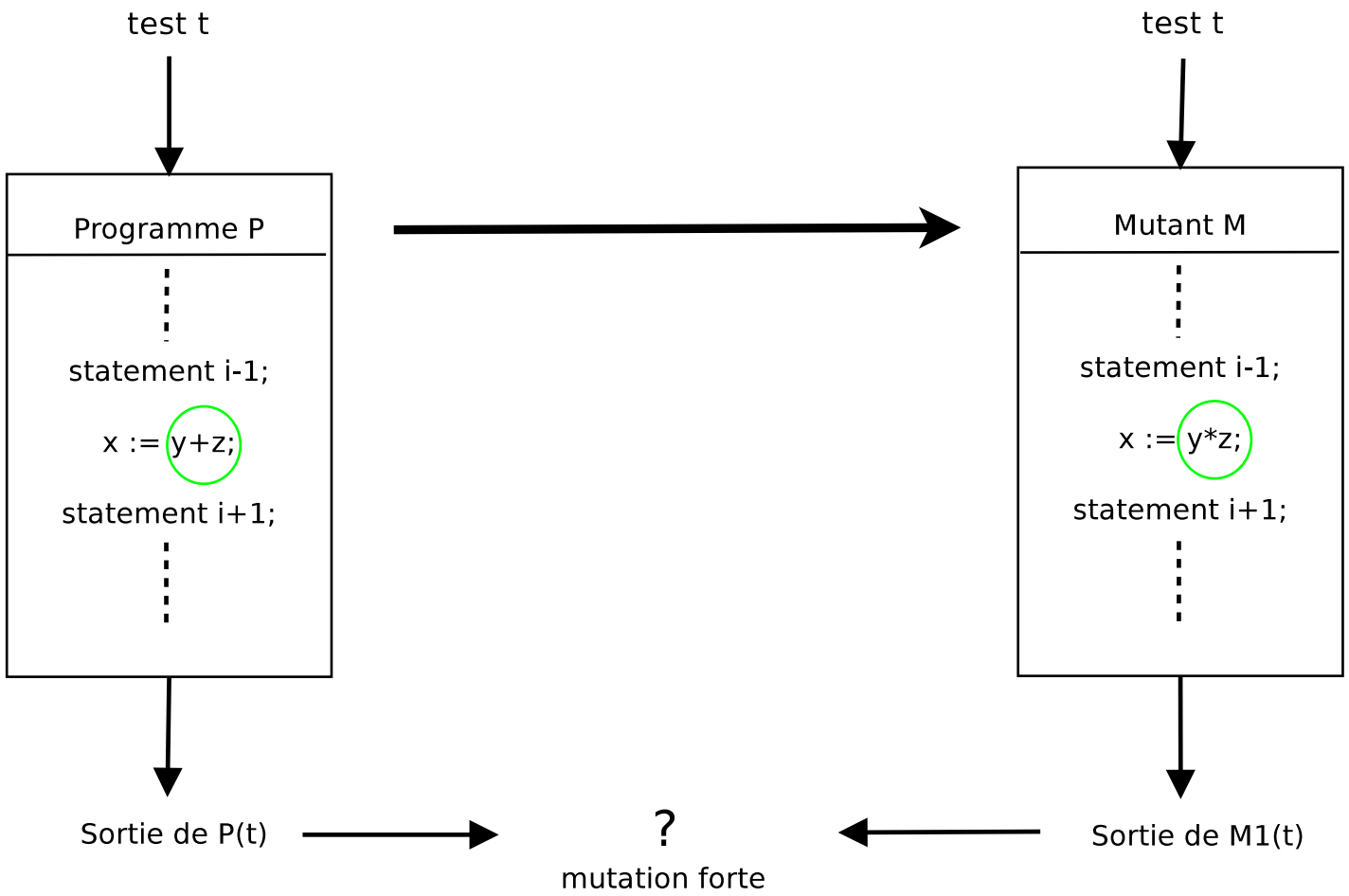
- bomb : ajouter `assert(false)` après une instruction
- modifier une expression arithmétique $e$ en $|e|$ (ABS)
- modifier un opérateur relationnel arith par un autre (ROR)
- modifier un opérateur arith par un autre (AOR)
- modifier un opérateur booléen par un autre (COR)
- modifer une expression bool/arith en ajoutant $-$ ou $\neg$ (UOI)
- modifier un nom de variable par un autre
- modifier un nom de variable par une constante du bon type
- modifier une constante par une autre constante du bon type
- ...

# Quelques règles pour les mutants

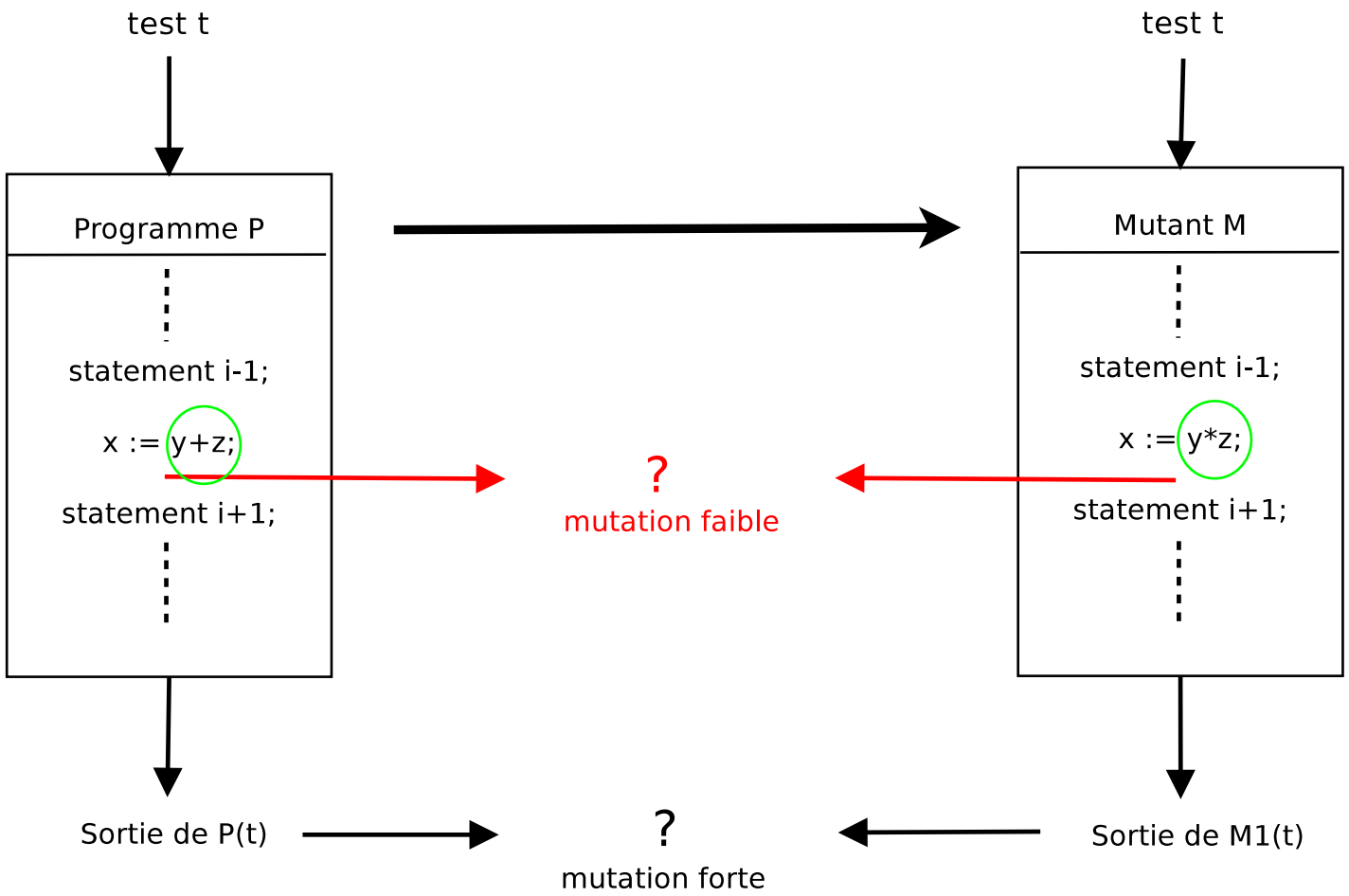
Les opérateurs de mutations doivent préserver la correction syntaxique et le typage, sinon le mutant est trivialement distingué (ne compile même pas).

On ne considère des mutants ne différant que par une instruction du programme original (mutation atomique, ou mutants d'ordre 1)
[justification empirique]

# Mutations faibles



# Mutations faibles



# Intérêts des mutations faibles

Point de vue théorique : plus facile pour comparaison aux critères structurels usuels

Point de vue pratique : plus simple pour trouver les DT : juste respecter la phase d'accessibilité et infection, pas besoin de suivre la phase d'infection.

- vrai pour génération manuelle et automatique

Comparaison de wM et sM

- théorie : sM $\succ$ wM
- pratique : sM $\approx$ wM

# Plan

- Rappels
- Principe
- Expressivité
- En pratique : moteur de mutation, calcul de score
- Opérateurs de mutation
- Discussion

# Opérateur de mutations

Opérateur de mutation $O$ : transforme un programme P syntaxiquement correct en un ensemble de programmes P' syntaxiquement corrects.

On note M($O$) la couverture des mutants créés par l'opérateur de mutations $O$.

On note sM($O$) et wM($O$) pour distinguer mutants forts / faibles

# Expressivité

Théorème 1 : pour tout critère de couverture $\mathcal{C} \in \{I, D, C, DC, MC\}$, il existe un opérateur de mutation $O(\mathcal{C})$ tq $sM(O(\mathcal{C})) \succeq \mathcal{C}$

Théorème 2 : pour tout critère de couverture $\mathcal{C} \in \{I, D, C, DC, MC\}$, il existe un opérateur de mutation $O(\mathcal{C})$ tq $wM(O(\mathcal{C})) = \mathcal{C}$

*Note : les mutations faibles permettent d'émuler les critères structurels usuels*

# Démonstration

exercice

# Expressivité (2)

On a aussi les propriétés suivantes :

- wM $\succ$ MC
- wM $\neq$ all-paths
- wM $\succ$ MCDC
- wM $\succ$ all-def
- wM $\succ$ all-use
- wM $\neq$ all-du-all-paths
- wM $\neq$ all-du-one-paths, mais $\succ$ souvent vérifié en pratique

# Plan

- Rappels
- Principe
- Expressivité
- En pratique : moteur de mutation, calcul de score
- Opérateurs de mutation
- Discussion

# En pratique

Créer les mutants : très très grand nombre, souvent $|M| \approx |P|$

Compiler les mutants : $|M|$ compilations d'un prog. de taille $|P|$

- une compilation par mutant
- (rappel) : couv. D : 1 compilation

Calculer le score de mutation : $(|M| + 1) \times |T|$ exécutions

- exec. chaque test contre chaque mutant (+ prog. initial)
- (rappel) : couv. D : $|T|$ exécutions

# Améliorations : compilation

## Muter directement la représentation bas niveau

- 1 seule compilation !
- limite : seulement sur bytecode (Java, .NET, etc.)

## Agréger les mutants en 1 seul "méta-mutant"

- cf suite
- 1 seule compilation, mais méta-mutant très gros
- toujours applicable (bytecode ou code natif)

# Méta-mutant, exemple

```
void foo(type1 arg1, type2 arg2) {
  ...
  x := x + y;
  a := b + 1;
  ...
}
```

```
void foo(type1 arg1, type2 arg2, int m_id) {
  ...
  switch m_id {
    case 1 :  x := x × y; break;
    case 2 :  x := x − y; break;
    default :  x := x + y;     // no mutation here
  };
  switch m_id {
    case 3 :  a := b − 1; break;
    case 4 :  a := b; break;
    default :  a := b+1;     // no mutation here
  };
  ...
}
```

# Améliorations : calcul du score

## Calcul de score incrémental

- idée : chaque mutant tué n'est pas rejoué contre les tests restants
- en pratique beaucoup moins de rejeu
  - car les 1er tests tuent souvent bcp de mutants "faciles"
- mais on perd de l'information
  - on calcule juste le score
  - pas les mutants tués par chaque test
  - par ex., minimisation du jeu de tests devient impossible

# Trouver les DT

Souvent 90% des mutants simples à couvrir ($\approx$ D)

- ex : `x := y+1` $\mapsto$ `x := y` forcément tué si mutation atteinte

Ceux restants sont très difficiles à couvrir

- mais ce sont eux qui font la force de l'approche

# Plan

- Rappels
- Principe
- Expressivité
- En pratique : moteur de mutation, calcul de score
- Opérateurs de mutation
- Discussion

# Quelques exemples d'opérateurs de mutation (rappel)

Opérateurs classiques de mutations [Offutt-Ammann]

- bomb : ajouter `assert(false)` après une instruction
- modifier une expression arithmétique $e$ en $|e|$ (ABS)
- modifier un opérateur relationnel arith par un autre (ROR)
- modifier un opérateur arith par un autre (AOR)
- modifier un opérateur booléen par un autre (COR)
- modifer une expression bool/arith en ajoutant $-$ ou $\neg$ (UOI)
- modifier un nom de variable par un autre
- modifier un nom de variable par une constante du bon type
- modifier une constante par une autre constante du bon type
- ...

Attention : les mutations doivent préserver la correction syntaxique et le typage, sinon le mutant est trivialement distingué (ne compile même pas).

Opérateurs spéciaux pour les langages avancés

- Java : opérateurs de classe de $\mu$Java

# Choix des opérateurs de mutation

Les opérateurs de mutations se basent en général sur :

- imposer des critères de couverture structurelle (bomb)
- couvrir les domaines des variables (intermédiaires) du programme (abs)
- modèles de fautes simples usuelles
- autres ?

Guide :

- seulement des mutants “d’ordre 1” (une seule mutation)
- privilégier les opérateurs les plus efficaces (cf après)

# Efficacité relative des opérateurs de mutation

Certains opérateurs de mutations sont plus puissants que d'autres, aux sens où (en théorie ou en pratique) :

- M(OP) $\succeq$ M(OP')

Expérimentalement :

- ABS et ROR : 97% de l'efficacité
- ABS, ROR, AOR, COR, UOI : 99% de l'efficacité

# Plan

- Rappels
- Principe
- Expressivité
- En pratique : moteur de mutation, calcul de score
- Opérateurs de mutation
- Discussion

# Mutants équivalents

Parfois la mutation ne peut être mise en évidence, c-à-d

$$\nexists\, dt,\ P(dt) \neq P'(dt)$$

- mutation non accessible (ex : code mort)
- infection impossible (ex : `x + 0` $\mapsto$ `x - 0` )
- propagation impossible (ex : valeur non utilisée)

On parle de mutant équivalent

Les mutants équivalents gênent la méthode

- score de mutation artificiellement bas
- effort pour couvrir un mutant non couvrable

Détecter les mutants équivalents : indécidable

# Mutants équivalents (2)

Détecter les mutants équivalents est indécidable

Cependant on peut utiliser des techniques de vérification pour repérer les mutants équivalents les plus évidents :

- analyse de code mort
- analyse des dépendances du résultat du programme
- analyse de valeur / signe pour les mutants de type $\mathtt{x} \mapsto |\mathtt{x}|$
- theorem proving (local) pour tout mutant de type $\mathtt{x} \mapsto \mathtt{f}(\mathtt{x})$
  demander si $x \neq f(x)$ est satisfiable

# À propos des mutants

Les mutations montrent les avantages suivants

- point de vue théorique : est plus puissant que nombre de critères de couverture structurelle
- point de vue pratique : critère très bien corrélé à la découverte de bugs

Nécessite un très grand nombre de mutants (souvent $\approx |P|$)

- une grande partie des mutants sont faciles à tuer (souvent $\geq 90\%$)
- l'efficacité de la technique réside dans la couverture des derniers $5 - 15\%$

Difficile et coûteux, mais grande qualité des tests générés

- même le calcul de score est coûteux
- plutôt à utiliser comme critère de qualité

# À propos des mutants (2)

Réflexions sur l'efficacité du test par mutations
(avis personnel)

Mauvaise raison (avis personnel) : la couverture de mutants trouve des bugs car les mutations représentent des bugs possibles

- vous arrive-t-il souvent d'écrire `x:= a*b` au lieu de `x:=a+b` ?

Meilleure raison (avis personnel) : couvrir les mutations oblige à créer un jeu de tests exerçant beaucoup de comportements différents du programme, il devrait donc aussi trouver beaucoup de bugs.

- souvent la couverture des mutants permet aussi de couvrir : DC, all-use, all-def-one-use
- on peut ajouter des mutations pour simuler d'autres critères (D,C,MC)

# Mutations en pratique

Rarement utilisé pour le moment (intérêt plus théorique : comparaison des critères de qualité)

Critère très fin mais très coûteux

Peut être utile pour évaluer la qualité du jeu de tests

Outils de calcul de score : Proteum (C), mujava/muclipse (Java)

# Mutations en pratique (2)

## Améliorer l'efficacité

- se restreindre à des mutants atomiques
- introduction incrémentale des opérateurs
- ne chercher les mutants équivalents que quand le score de mutation obtenu est déjà élevé (bcp - de candidats potentiels)

## Ne pas commencer directement par la couverture des mutants

- déjà couvrir I, D et/ou les domaines d'entrée
- le jeu de tests créé permettra d'éliminer bcp de mutants