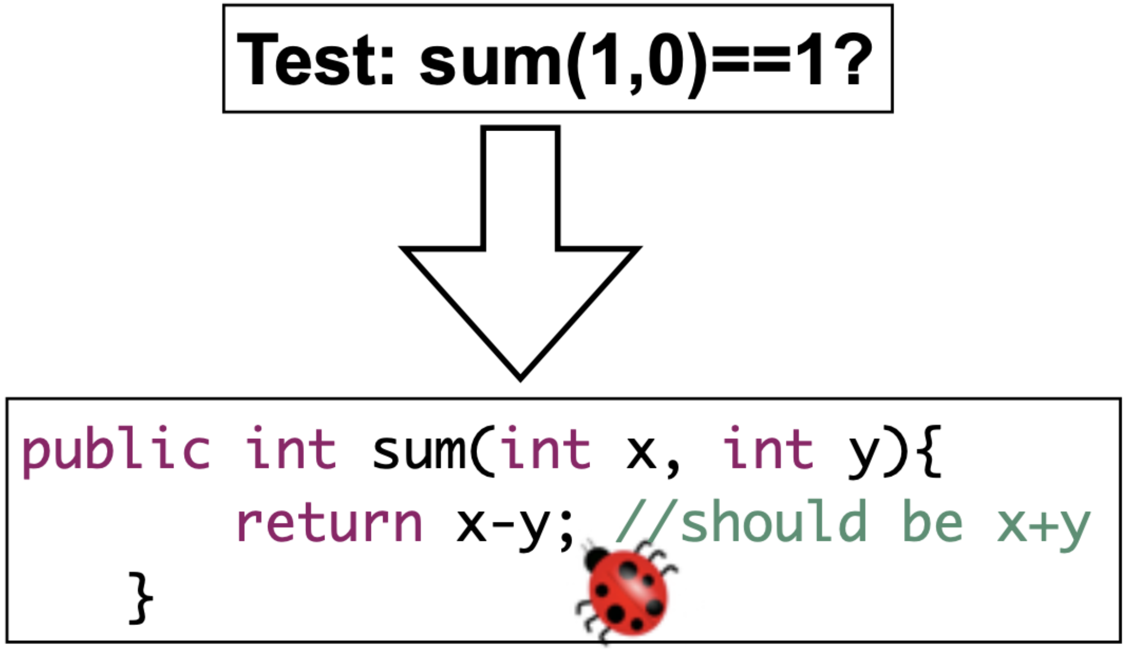
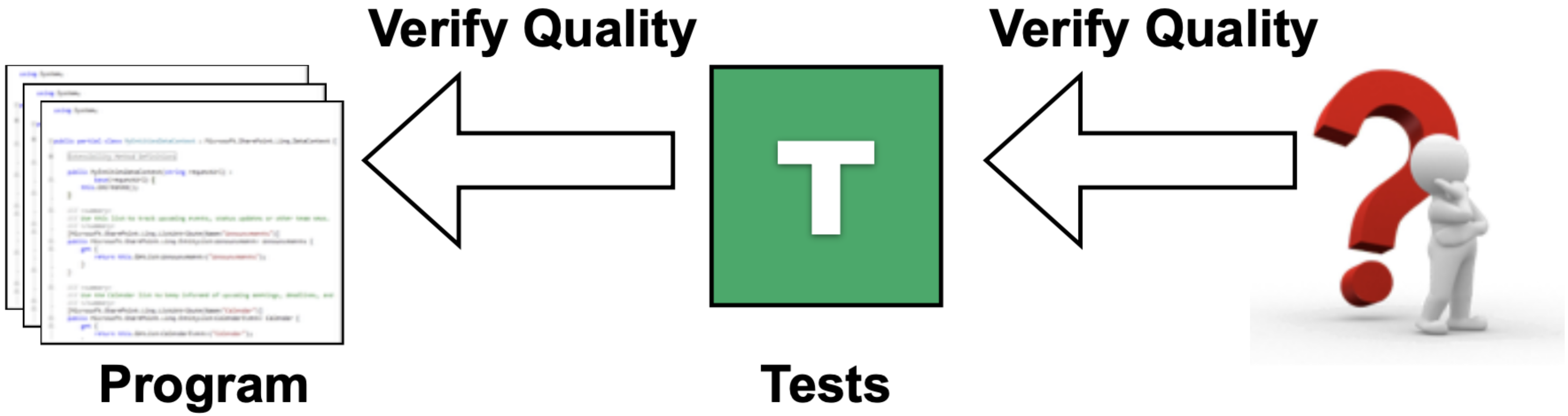
MUTATION TESTING

Khoa CNTT – Trường ĐHCN - ĐHQGHN

Motivating example



Who will test the tests?



Test effectiveness evaluation

- More bugs detected, more effective test suite
- However,
  - We don't know how many bugs the program has and where they are
  - It's hard to evaluate test effectiveness in detecting future bugs
- How about creating artificial bugs to simulate real bugs for evaluating test effectiveness?

Mutation testing

- Mutation testing injects changes to program statements to generate artificial bugs
- More mutation (artificial) bugs detected → capability to detect more real bugs → More effective test suite

Mutation testing

- Mutation testing: is an approach for evaluating test effectiveness
- Mutant: a modified program
    - A mutant is killed if it causes failure for at least one test (there is at least one failed test)

Example

Test: sum(1,0)==1?

```
public int sum(int x, int y){
        return x-y; //should be x+y
    }
```

```
public int sum(int x, int y){
        return x*y;
    }
```

```
public int sum(int x, int y){
        return x-0;
    }
```

mutants

...

Mutation can be stronger than control/data-flow coverage

Mutation testing

- Step 1: Applies artifical changes based on mutation operators to generate mutants
- Step 2: Execute the test suite against each mutant
- Step 3: Compute the mutation score: The higher, the better

Mutation testing

- Step 1: Applies artifical changes based on mutation operators to generate mutants

```
int foo(int num [ ], n){
    int r = 0;
    for  i = 1 to n do:
        if(num[i] > num[r]):
            r = i;
    return r;
}
```

```
int foo(int num [ ], n){
    int r = 0;
    for  i = 1 to n do:
        if(num[i] > num[r]):
        if(i > num[r]):
            r = i;
    return r;
}
```

Java mutation operators (Mujava tool)

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Language Feature	Operator	Description
Encapsulation	AMC	**A**ccess **m**odifier **c**hange
Inheritance	IHD	**H**iding variable **d**eletion
	IHI	**H**iding variable **i**nsertion
	IOD	**O**verriding method **d**eletion
	IOP	**O**verriding method calling **p**osition change
	IOR	**O**verriding method **r**ename
	ISI	**super** keyword **i**nsertion
	ISD	**super** keyword **d**eletion
	IPC	Explicit call to a **p**arent's **c**onstructor deletion
Polymorphism	PNC	**new** method call with **c**hild class type
	PMD	**M**ember variable **d**eclaration with parent class type
	PPD	**P**arameter variable **d**eclaration with child class type
	PCI	Type **c**ast operator **i**nsertion
	PCC	**C**ast type **c**hange
	PCD	Type **c**ast operator **d**eletion
	PRV	**R**eference assignment with other comparable **v**ariable
	OMR	**O**verloading **m**ethod contents **r**eplace
	OMD	**O**verloading **m**ethod **d**eletion
	OAC	**A**rguments of **o**verloading method call **c**hange
Java-Specific Features	JTI	**this** keyword **i**nsertion
	JTD	**this** keyword **d**eletion
	JSI	**static** modifier **i**nsertion
	JSD	**static** modifier **d**eletion
	JID	Member variable **i**nitialization **d**eletion
	JDC	Java-supported **d**efault **c**onstructor deletion
	EOA	**R**eference **a**ssignment and content assignment replacement
	EOC	**R**eference **c**omparison and content comparison replacement
	EAM	**A**cessor **m**ethod change
	EMM	**M**odifier **m**ethod change

ROR – Relational operator replacement

```
int foo(int num [ ], n){
    int r = 0;
    for  i = 1 to n do:
        if(num[i] > num[r]):
            r = i;
    return r;
}
```

>=, <, <=, !=, ==

ROR – Relational operator replacement

```
int foo(int num [ ], n){
    int r = 0;
    for  i = 1 to n do:
        if(num[i] > num[r]):
            r = i;
    return r;
}
```

```
int foo(int num [ ], n){
    int r = 0;
    for  i = 1 to n do:
        if(num[i] > num[r]):
        if(num[i] != num[r]):
            r = i;
    return r;
}
```

AOR – Arithmetic operator replacement

sum = x + y

sum = x – y
sum = x * y
sum = x / y
sum = x % y

COR – Conditional operator replacement

if (a && b) →

if (a || b)
if (a)
if (b)
if (true)
if (false)

UOI– Unary Operator Insertion

result = x % y

result = x % (- y)
result = x % (-- y)
result = x % (++y)
result = x % (y--)
result = x % (y++)

Mutation testing

- Step 2: Execute the test suite against each mutant

```
int foo(int num [ ], n){
    int r = 0;
    for  i = 1 to n do:
        ~~if(num[i] > num[r]):~~
        if(i > num[r]):
            r = i;
    return r;
}
```

- Test 1:
  - Input: [1, 2, 3], Expected output: 2
  - Actual Output: 2
- Test 2:
  - Input: [1, 2, 1], Expected output: 1
  - Actual Output: 2 → failed
- Test 3:
  - Input: [3, 2, 1], Expected output: 0
  - Actual Output: 0

→ Mutant is killed

Mutation testing

- Step 2: Execute the test suite against each mutant

```
int foo(int num [ ], n){
    int r = 0;
    for  i = 1 to n do:
        if(num[i] > num[r]):
        if(num[i] >= num[r]):
            r = i;
    return r;
}
```

- Test 1:
  - Input: [1, 2, 3], Expected output: 2
  - Actual Output: 2
- Test 2:
  - Input: [1, 2, 1], Expected output: 1
  - Actual Output: 1
- Test 3:
  - Input: [3, 2, 1], Expected output: 0
  - Actual Output: 0

→ Mutant is still alive

Mutation testing

- Step 3: Compute the mutation score
  - The higher, the better

- In the previous example,
  - There are 2 mutants
  - One is killed, and one is still alive
  - Mutation score is 0.5

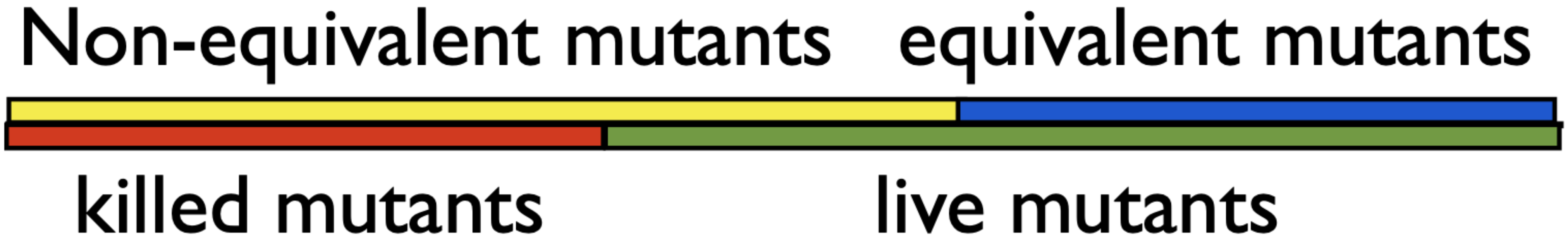
Mutation testing

- Step 1: Applies artifical changes based on mutation operators to generate mutants
- Step 2: Execute the test suite against each mutant
- Step 3: Compute the mutation score: The higher, the better

Type of mutants

- Killed mutants
- Live mutants
- Equivalent mutants
- Non-equivalent mutants

Types of mutants



all mutants

Example – equivalent mutant

```
int foo(int num [ ], n){
    int r = 0;
    ~~for  i = 1 to n do:~~
    for  i = 0 to n do:
        if(num[i] > num[r]):
            r = i;
    return r;
}
```

- Test 1:
  - Input: [1, 2, 3], Expected output: 2
  - Actual Output: 2
- Test 2:
  - Input: [1, 2, 1], Expected output: 1
  - Actual Output: 1
- Test 3:
  - Input: [3, 2, 1], Expected output: 0
  - Actual Output: 0

Mutation testing criteria

- Test suite T kills a mutant m iff there exist a test t in T kills m
- The effectiveness of test suites can be measured using mutation score:

$$\mathrm{MS}(T) = \frac{\#\mathrm{KilledMutants}}{\#\mathrm{AllMutants} - \#\mathrm{EquivalentMutants}}$$

Mutation testing criteria

- Mutation testing is extremely costly, since we need to run the test suite against each mutant

E.g. Using 108 mutation operators, **Proteum** generates 4,937 mutants for **tcas** (only 137 lines)

Optimization techniques

- Do fewer
  - Selective mutation
- Do smarter
  - Weak mutation
- Do faster
  - Schema-based analysis, specific compilation, parallel execution

Do fewer

- Selective mutation testing: Select a subset of mutants to do mutation testing
- How to select?
  - Operator-based mutant selection
  - Random mutant selection

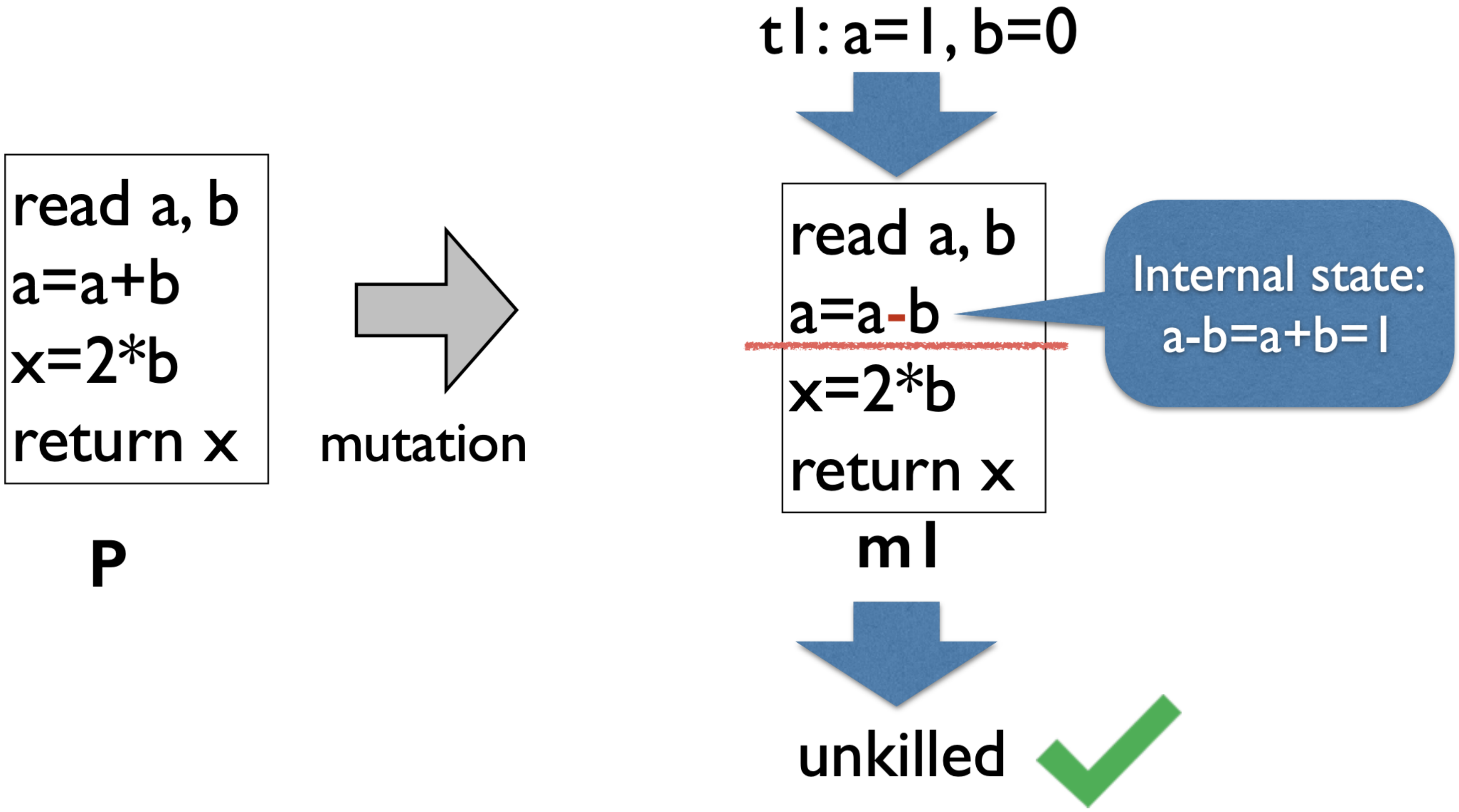
Do smarter

- Weak mutation testing
  - Only check the reachability and necessity conditions
  - A mutant is deemed as **weakly killed** as long as it cause internal state change
- Pros: More efficient, do not need to each mutant completely
- Cons: Imprecise, weakly killed mutants may not be actually killed

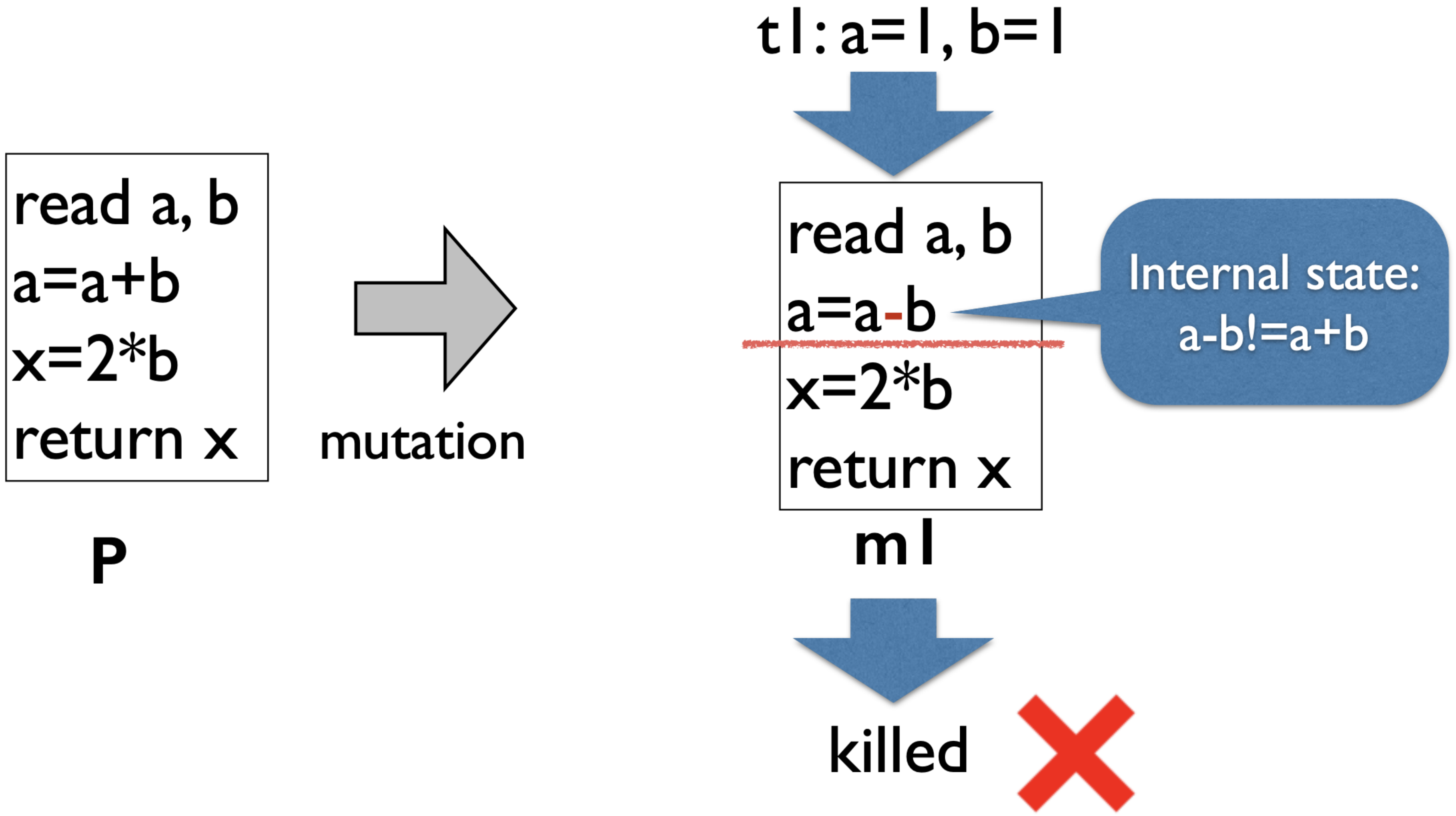
Do smarter

- Weak mutation testing
  - Only check the reachability and necessity conditions
  - A mutant is deemed as **weakly killed** as long as it cause internal state change
- Pros: More efficient, do not need to each mutant completely
- Cons: Imprecise, weakly killed mutants may not be actually killed

Weak mutation



Weak mutation



Do faster

- Use parallel execution to speedup mutation testing

Mutation testing tools

- Java
  - PIT: https://pitest.org
  - MuJava: https://cs.gmu.edu/~offutt/mujava/
  - MAJOR: http://mutation-testing.org
  - Javalanche: https://github.com/david-schuler/javalanche/
- Python
  - MutPy: https://pypi.org/project/MutPy/0.4.0/

Application of mutation

- Mutation-based test generation
- Mutation-based fault localization
- Mutation-based program repair