POLITECNICO
MILANO 1863

Software Engineering 2

Automated Testing

Concolic Execution

Fuzzing

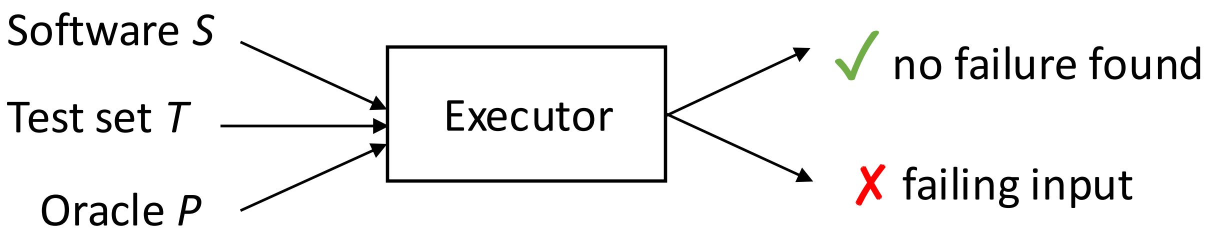
M Camilli, E Di Nitto, M Rossi

Verification & Validation

Dynamic Analysis, Testing

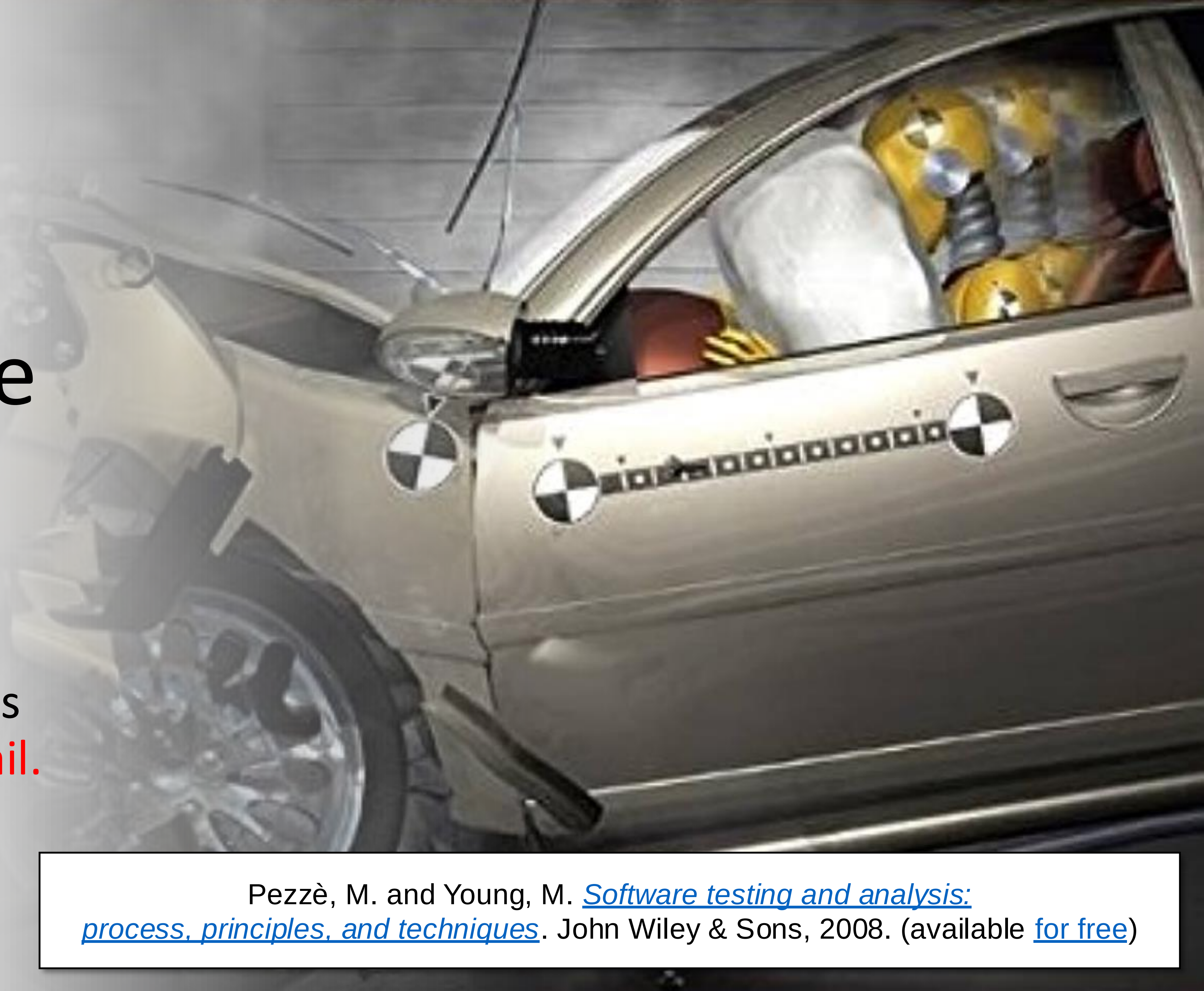
Dynamic analysis, aka testing

- The very idea
  - Analyzes program behavior
  - Properties are encoded as executable oracles, that represent
    - expected outputs, desired conditions (assertions)
  - It can run only finite sets of test cases → it's not exhaustive verification
  - Failures come with concrete inputs that trigger them
  - Execution is automatic (definition of test cases and oracles may not)



What is the goal of testing?

The goal of testing is
making programs fail.



Pezzè, M. and Young, M. [Software testing and analysis: process, principles, and techniques](). John Wiley & Sons, 2008. (available [for free]())

What is the goal of testing?

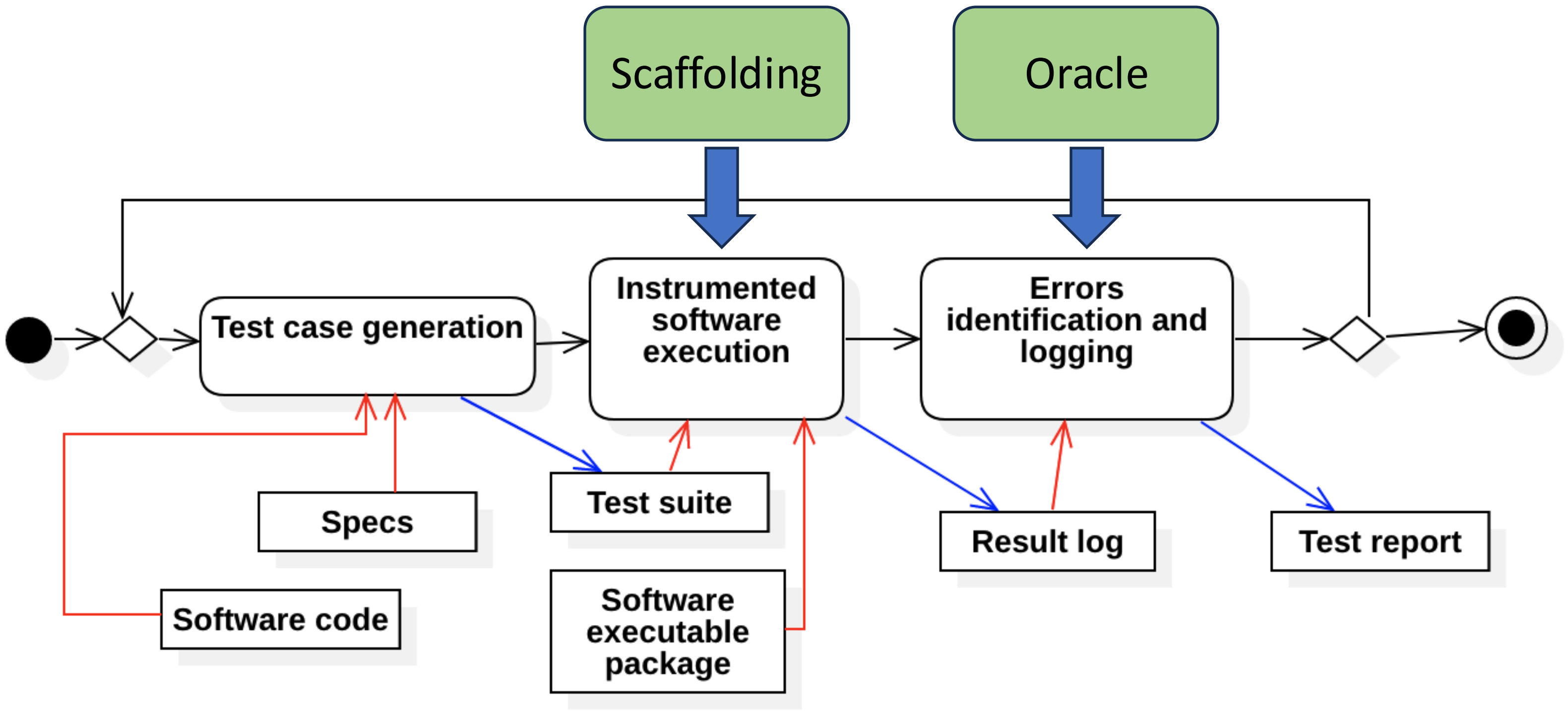
The main goal of testing is making programs fail

- Other common goals
  - Trigger different parts of a program to increase coverage
  - Make sure the interaction between components works (integration testing)
  - Support fault localization and error removal (debugging)
  - Ensure that bugs introduced in the past do not happen again (regression testing)
- Important note
  - *"Program testing can be used to show the presence of bugs, but never to show their absence!"* (Edsger W. Dijkstra)

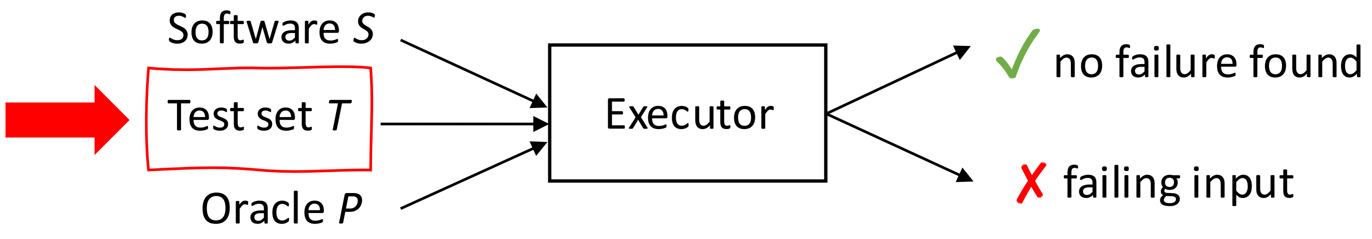
What is a test case?

- A test case is a set of inputs, execution conditions, and a pass/fail criterion
- Running a test case typically involves
  - Setup: bring the program to an **initial state** that fulfils the execution conditions
  - Execution: **run** the program on the actual inputs
  - Teardown: **record** the output, the final state, and any **failure** determined based on the pass/fail criterion
- A test set or test suite can include multiple test cases
- A test case specification is a requirement to be satisfied by one or more actual test cases
  - Example of test case specification: "*the input must be a sentence composed of at least two words*"
  - Example of test case input: "*this is a good test case input*"

Testing workflow



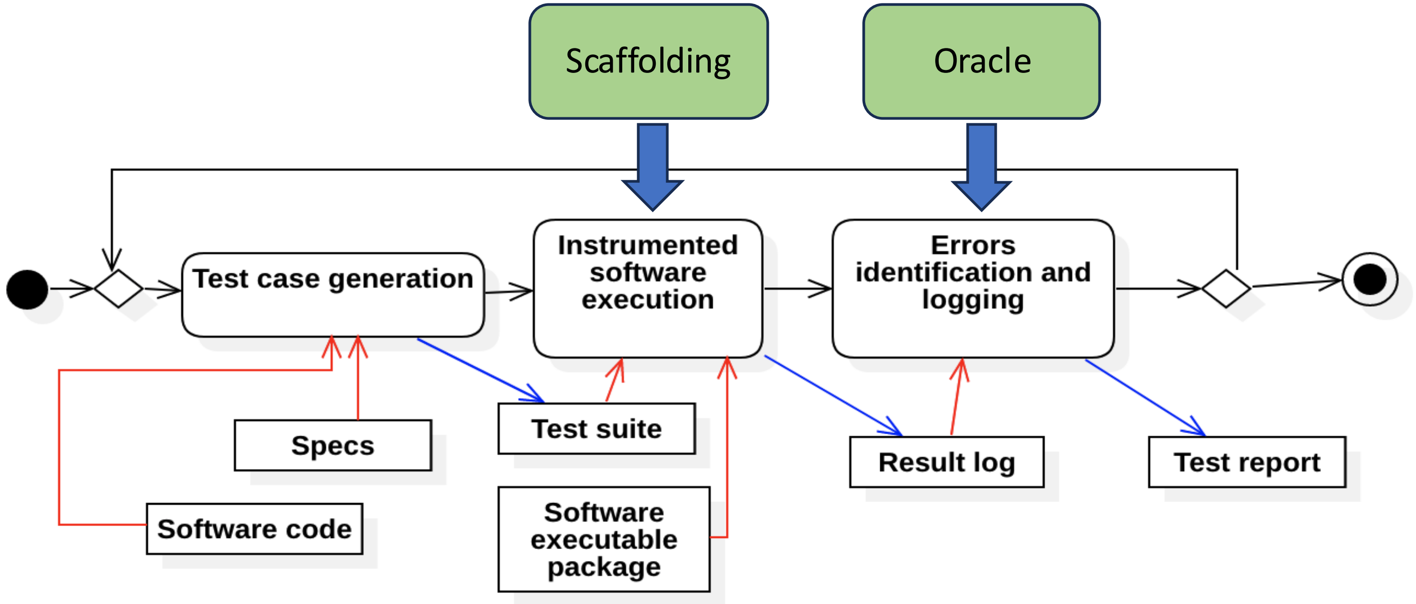
Test case generation



- Purpose: define *good quality* test sets
  - Showing a high probability of finding errors
  - Able to cover an acceptable amount of cases
  - Sustainable (we cannot run tests forewer…)

Test case generation

- Test cases can be generated in a black box or white box manner
  - **White box**: generation is based on code characteristics
  - **Black box**: generation is based on specs characteristics



Test case generation

- Test cases can be defined manually
- Test cases can be automatically generated (automated testing)
  - Combinatorial testing = enumerate all possible inputs following some policy (e.g., smaller to larger).. not in this course!
  - Symbolic execution (we have seen it)
  - Concolic execution = pseudo-random generation of inputs guided by symbolic path properties
  - Fuzz testing (fuzzing) = pseudo-random generation of inputs including invalid, unexpected inputs
  - Search-based testing = explores the space of valid inputs looking for those that improve some metrics (e.g., coverage, diversity, failure inducing capability)

Symbolic execution: test case generation

- **Symbolic execution** can be used to automatically generate test cases
- **Procedure**:
  - Give as input a set of target locations or paths
  - Run symbolic execution
  - If the path condition is SAT for a target location or path
  - Generate one or more input assignments (i.e., test cases) satisfying the path condition
  - Do it for all identified locations and paths to define a test suite
- However, we already discussed the **weaknesses** of symbolic execution…

Verification & Validation

Automated Testing: Concolic (Concrete-Symbolic) Execution

Concrete-symbolic (concolic) execution

- The very idea
  - Perform symbolic execution **alongside** a concrete one (concrete inputs)
  - The **state** of concolic execution combines a **symbolic** part and a **concrete** part, used as needed to make progress in the exploration
- Steps
  - Concrete to symbolic: derive conditions to explore new paths
  - Symbolic to concrete: simplify conditions to generate concrete inputs

Concolic execution: example
(concrete to symbolic)

- **Example**: Let's explore all paths of procedure `m`
- We start from a (random) concrete input, at the same time, we build the symbolic condition of the explored path

```
0: void m(int x, int y) {
1:   int z := 2 * y
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

- {x=22, y=7} → path `<0,1,2,6,7>`, path condition: $2Y \neq X$
- To explore another path, negate the path condition: $\neg(2Y \neq X)$

`<0>`

x	y	z	π
X	Y		T
22	7		

`<0,1>`

x	y	z	π
X	Y	2Y	T
22	7	14	

`<0,1,2>`

x	y	z	π
X	Y	2Y	$2Y \neq X$
22	7	14	

`<0,1,2,6,7>`

x	y	z	π
X	Y	2Y	$2Y \neq X$
22	7	14	

Concolic execution: example
(symbolic to concrete)

x	y	z	π
X	Y		T
2	1		

<0>

- If we can solve the constraint, we start again with another concrete input that satisfies the new constraint $\neg(2Y \neq X)$

```
void m(int x, int y) {
  int z := 2 * y
  if (z == x) {
    z := y + 10
    if (x <= z)
      print("Log message.")
  }
}
```

Solve: $\neg(2Y \neq X)$

Concolic execution: example
(concrete to symbolic)

- We explore the new path and apply again the concrete-to-symbolic step

```
0: void m(int x, int y) {
1:   int z := 2 * y
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

- {x=2, y=1} → path <0,1,2,3,4,5,6,7> with path condition 2Y=X ∧ X ≤ Y + 10

<0>

x	y	z	π
X	Y		T
2	1		

<0,1>

x	y	z	π
X	Y	2Y	T
2	1	2	

<0,1,2>

x	y	z	π
X	Y	2Y	2Y=X
2	1	2	

<0,1,2,3>

x	y	z	π
X	Y	Y+10	2Y=X
2	1	11	

<0,1,2,3,4,5,6,7>

x	y	z	π
X	Y	Y+10	2Y=X
2	1	11	X≤Y+10

Partial negation: last condition only

2Y=X ∧ ¬(X ≤ Y + 10)

Concolic execution: example
(symbolic to concrete)

x	y	z	π
X	Y		T
30	15		

- New input values are identified

```
void m(int x, int y) {
  int z := 2 * y
  if (z == x) {
    z := y + 10
    if (x <= z)
      print("Log message.")
  }
}
```

Solve: $2Y=X \wedge \neg(X \leq Y + 10) \equiv$
$2Y=X \wedge X > Y + 10$

Concolic execution: example
(concrete to symbolic)

- We explore the new path and apply again the concrete-to-symbolic step

```
0: void m(int x, int y) {
1:   int z := 2 * y
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

<0>

x	y	z	π
X	Y		T
30	15		

<0,1>

x	y	z	π
X	Y	2Y	T
30	15	30	

<0,1,2>

x	y	z	π
X	Y	2Y	2Y=X
30	15	30	

<0,1,2,3>

x	y	z	π
X	Y	Y+10	2Y=X
30	15	25	

<0,1,2,3,4,6,7>

x	y	z	π
X	Y	Y+10	2Y=X
30	15	25	X>Y+10

- Conclusion: we have been able to cover all paths with the following test cases:
  - <0,1,2,6,7>: {x=22, y=7}
  - <0,1,2,3,4,5,6,7>: {x=2, y=1}
  - <0,1,2,3,4,6,7>: {x=30, y=15}

Concolic execution: example2

- **Example**: Let's explore again all paths of the procedure `m2`

```
0: void m2(int x, int y) {
1:   int z := bb(y) //black-box function
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

- We try to follow the same approach, but, in some cases, we cannot solve the symbolic condition…
- Behavior of `bb` is unknown. We execute it with the identified input cases
  - Example: run bb(7) returns 14
- Now the condition can be solved

x	y	z	π
X	Y		T
22	7		

<0>

x	y	z	π
X	Y	bb(Y)	T
22	7	14	

<0,1>

<0,1,2,6,7>

x	y	z	π
X	Y	bb(Y)	bb(Y)≠X
22	7	14	

Negate: ¬(bb(Y)≠X)

Solve: ¬(bb(Y)≠X) → cannot do it symbolically!

Symbolic-to-concrete step

¬(14≠X) ≡ 14=X

Concolic execution: example2

x	y	z	π
X	Y		T
14	7		

<0>

- Now the constraint can be solved and we can start a new exploration

```
void m2(int x, int y) {
  int z := bb(y) //black-box function
  if (z == x) {
    z := y + 10
    if (x <= z)
      print("Log message.")
  }
}
```

Solve: $\neg(14 \neq X) \equiv 14 = X$

Concolic execution: example2

- New explorations follow the same approach

```
0: void m2(int x, int y) {
1:   int z := bb(y) //black-box function
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

<0>

x	y	z	π
X	Y		T
14	7		

<0,1>

x	y	z	π
X	Y	bb(Y)	T
14	7	14	

<0,1,2>

x	y	z	π
X	Y	bb(Y)	bb(Y)=X
14	7	14	

<0,1,2,3>

x	y	z	π
X	Y	Y+10	bb(Y)=X
14	7	17	

<0,1,2,3,4,5,6,7>

x	y	z	π
X	Y	Y+10	bb(Y)=X
14	7	17	X≤Y+10

Negate last condition:

bb(Y)=X ∧ ¬(X≤Y+10) ≡ bb(Y)=X ∧ X>Y+10

Solve: bb(Y)=X ∧ X>Y+10 → cannot!

Concolic execution: example2

x	y	z	π
X	Y		T
34	17		

<0>

POLITECNICO MILANO 1863

- New explorations follows the same approach

```
void m2(int x, int y) {
  int z := bb(y) //black-box function
  if (z == x) {
    z := y + 10
    if (x <= z)
      print("Log message.")
  }
}
```

Concretize: $\boxed{\text{bb(Y)}}$=X ∧ X>Y+10
We select Y randomly, execute bb(Y) and check if the formula holds
Example: Y=17, bb(Y)=34
Solve: Y=17 ∧ bb(Y)=34 ∧ bb(Y) =X ∧ X>Y+10

Concolic execution: example2

- New explorations follows the same approach

```
0: void m2(int x, int y) {
1:   int z := bb(y) //black-box function
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

<0>

x	y	z	π
X	Y		T
34	17		

<0,1>

x	y	z	π
X	Y	bb(Y)	T
34	17	34	

<0,1,2>

x	y	z	π
X	Y	bb(Y)	bb(Y)=X
34	17	34	

<0,1,2,3>

x	y	z	π
X	Y	Y+10	bb(Y)=X
34	17	27	

<0,1,2,3,4,6,7>

x	y	z	π
X	Y	Y+10	bb(Y)=X
34	17	27	X>Y+10

Concolic execution: pros and cons

- Advantages
  - Can deal with black-box functions in path conditions (not possible with symbolic exec)
  - As symbolic execution, can generate concrete test cases automatically
- Limitations
  - Finds just **one input example** per path, however…
    - Failures typically occur with certain inputs only
    - If failures are rare events, it's unlikely to spot them with concolic exec
  - #paths explode due to complex nested conditions → **large search space**
  - **Does not guide the exploration**, it just explores possible paths one by one as long as we have budget (e.g., time, #runs)

Positioning concolic execution in the testing workflow