POLITECNICO
MILANO 1863

Software Engineering 2

Static analysis in practice

Symbolic Execution

Verification & Validation

Static analysis in practice

Static analysis tools

- Various tools available
- The analyses are language-specific but many tools support multiple programming languages
- The first static analysis tool was a Unix utility, Lint, developed in 1978 for C programs. From this, simple static analysis is also called linting
- Lists of currently available tools are available from various sources:
  - https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
  - https://github.com/analysis-tools-dev/static-analysis

Comparing some static analysis tools

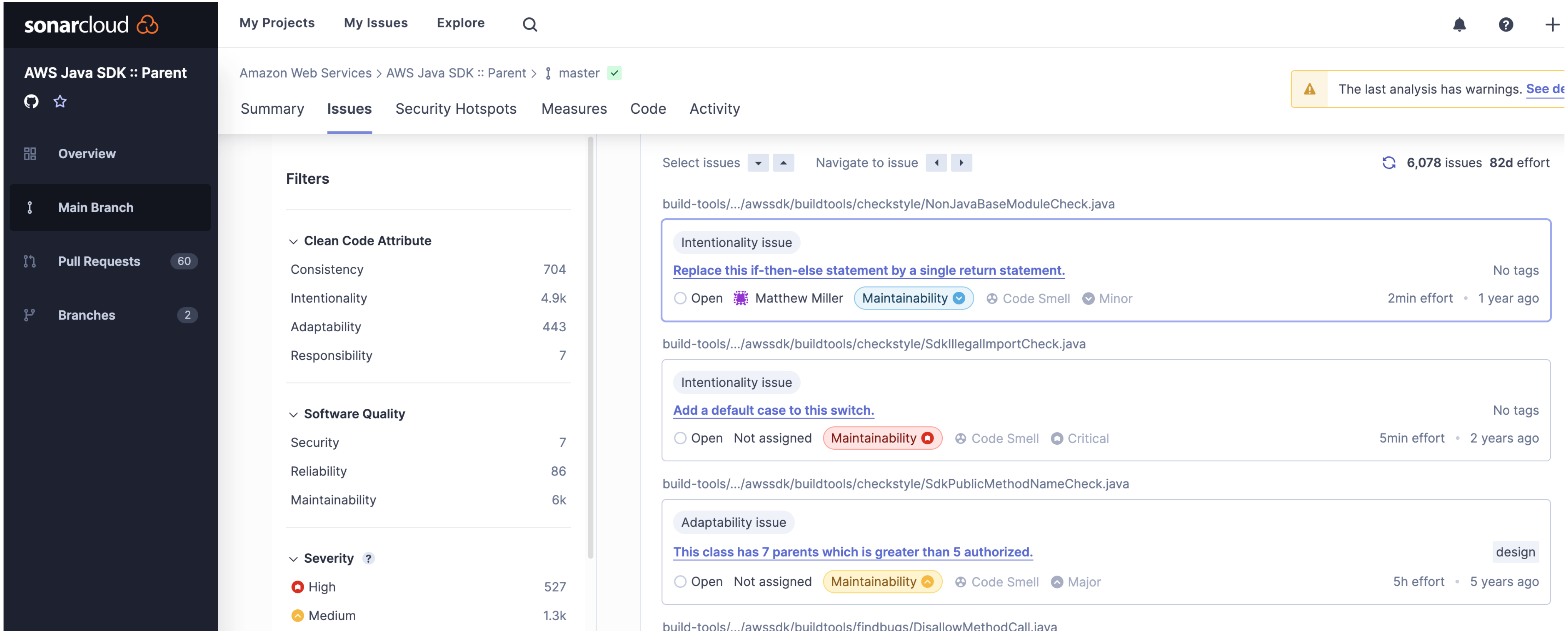
https://www.comparitech.com/net-admin/best-static-code-analysis-tools/

Tool/Features	SonarQube	Checkmarx	Synopsys Coverity	Micro Focus Fortify SCA	Veracode Static Analysis	Snyk Code
Language Support	Multiple	Multiple	Multiple	Multiple	Multiple	Multiple
Integrations	Various IDEs, CI/CD	Various IDEs, CI/CD	Various IDEs, CI/CD	Various IDEs, CI/CD	Various IDEs, CI/CD	Various IDEs, CI/CD
Free Trial	Yes	Yes	No	Yes	Yes	Yes
On-Premises/Cloud	Both	Both	Both	Both	Both	Cloud
Automated Scans	Yes	Yes	Yes	Yes	Yes	Yes
Compliance Reporting	Yes	Yes	Yes	Yes	Yes	Yes
Vulnerability Database	Yes	Yes	Yes	Yes	Yes	Yes
Real-Time Feedback	Yes	Yes	No	No	No	Yes

Example of issues report from SonarCloud/SonarQube

POLITECNICO MILANO 1863

https://sonarcloud.io/project/issues?resolved=false&id=aws_aws-sdk-java-v2



Verification & Validation

Symbolic Execution

Static vs dynamic, again

- Static analysis
  - without executing the software
  - on generic/abstract inputs (symbolic constraints)
  - based on over-approximation

- Dynamic analysis
  - while executing the software
  - on specific/concrete inputs
  - based on concrete sequences of states

Symbolic Execution

Symbolic execution executes programs on symbolic values, so that the output is expressed as a function of symbolic input

Static vs dynamic, again

- Static analysis
  - without executing the software
  - on generic/abstract inputs (symbolic constraints)
  - based on over-approximation
- Dynamic analysis
  - while executing the software
  - on specific/concrete inputs
  - based on concrete sequences of states

"static-dynamic spectrum"

Static

Dynamic

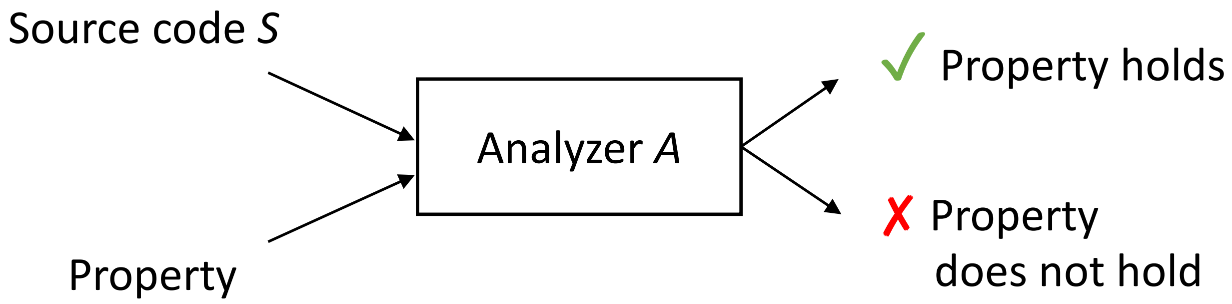
Data-flow analisys

Symbolic execution

Testing

Symbolic execution

- The very idea
  - Analyzes **real** source code
  - Analyzes **reachability** and **path feasibility** properties
  - Can be used to **generate test cases**
  - Is **automatic**
  - **May fail** to analyze all possible paths



Types of checked properties

POLITECNICO MILANO 1863

- Reachability: does some execution of the program reach the location *l* in *S*?
  - Symbolic exec tries to **verify that *l* cannot be reached**, or alternatively **spots the condition under which *l* can be reached**

```
…
k:    try {
k+1:    …
l-1:  } catch (e) {
l:      /* error */
…     }
```

```
…
l-1: if (x < 0) {
l:     /* safe */
```

- Path feasibility: Is the given path *p* feasible?
  - Symbolic exec tries to **verify that *p* cannot be executed**, or alternatively **spots the condition under which *p* can be executed**

p = <0,1,…,k,…,n>

Symbolic state and path condition

- Symbolic execution executes programs on symbolic values
  - Symbolic states keep track of the (symbolic) value of variables

Inputs are initialized with symbolic (generic) values

```
void foo(int x, int y) {
    …
```

Symbolic state

variables

x	y
X	Y

<0>

Symbolic values

path

Symbolic state and path condition

- Symbolic execution executes programs on symbolic values
  - Symbolic states keep track of the (symbolic) value of variables

Executing a statement updates the symbolic state

```
0: void foo(int x, int y) {
1:   int z := x
```

Symbolic state

x	y	z
X	Y	X

<0,1>

Symbolic state and path condition

- Symbolic execution executes programs on symbolic values
  - Symbolic states keep track of the (symbolic) value of variables

Executing a branch splits the symbolic state
A path condition π represents the constraint of a path

```
0: void foo(int x, int y) {
1:    int z := x
2:    if (z < y)
```

Symbolic state

Condition at point 2 true:

x	y	z	π
X	Y	X	$X<Y$

<0,1,2>

Condition at point 2 false:

x	y	z	π
X	Y	X	$X \geq Y$

<0,1,2>

Symbolic state and path condition

POLITECNICO MILANO 1863

- Symbolic execution executes programs on symbolic values
  - Symbolic states keep track of the (symbolic) value of variables

The execution continues along feasible paths (path condition π is satisfiable)

```
0: void foo(int x, int y) {
1:   int z := x
2:   if (z < y)
3:     z := z*2
```

Symbolic state

x	y	z	π
X	Y	2X	X<Y

<0,1,2,3>

Symbolic state and path condition

- Symbolic execution executes programs on symbolic values
  - Symbolic states keep track of the (symbolic) value of variables

The execution continues along feasible paths (path condition π is satisfiable)

```
0: void foo(int x, int y) {
1:   int z := x
2:   if (z < y)
3:     z := z*2
4:   if (x < y && z >= y)
```

Symbolic state

	x	y	z	π	
Condition at point 4 true:	X	Y	2X	$X<Y$ $X<Y \land 2X \geq Y$	<0,1,2,3,4>

	x	y	z	π	
Condition at point 4 false:	X	Y	2X	$X<Y$ $X \geq Y \lor 2X<Y$	<0,1,2,3,4>

Final states

- Possible outcomes of symbolic execution:
  - SAT exit (π is satisfiable): any satisfying assignment to variables in π is an **input** that satisfies the given property in a **concrete execution**
  - UNSAT exit (π is not satisfiable): the given property cannot be satisfied by **any concrete execution**
- **Example**: is location 5 reachable?

```
0: int foo(int x, int y) {
1:   int z := x
2:   if (z < y)
3:     z := z*2
4:   if (x < y && z >= y)
5:     print(z) //location
6: }
```

x	y	z	π
X	Y	2X	$X<Y$
			$X<Y \land 2X \geq Y$

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

SAT exit

Example of satisfying assignment: $X = 2$, $Y = 3$

Final states

- Possible outcomes of symbolic execution:
  - SAT exit (π is satisfiable): any satisfying assignment to variables in π is an **input** that satisfies the given property in a **concrete execution**
  - UNSAT exit (π is not satisfiable): the given property cannot be satisfied by **any concrete execution**
- **Example**: is path `<0,1,2,4,5>` feasible?

```
0: int foo(int x, int y) {
1:   int z := x
2:   if (z < y)
3:     z := z*2
4:   if (x < y && z >= y)
5:     print(z) //location
6: }
```

Condition at point 4 true:

x	y	z	π	
X	Y	X	$X \geq Y$	`<0,1,2,4>`
			$X<Y \land X \geq Y$	

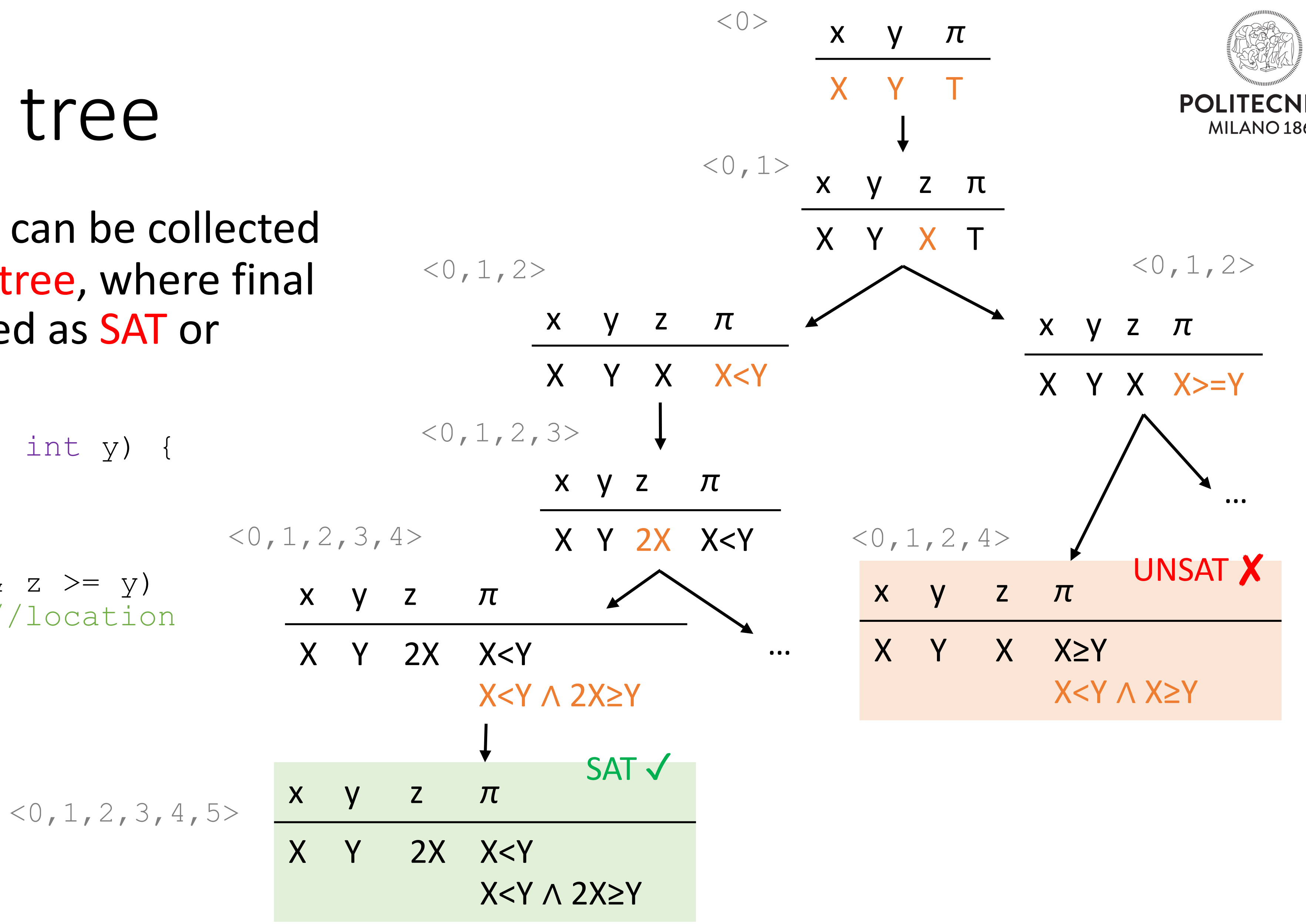
UNSAT exit

There is no satisfying assignment

Execution tree

- Execution paths can be collected in an execution tree, where final states are marked as SAT or UNSAT

```
0: int foo(int x, int y) {
1:   int z := x
2:   if (z < y)
3:     z := z*2
4:   if (x < y && z >= y)
5:     print(z) //location
6: }
```



Symbolic execution: example

- What are the feasible paths of the program `swap`?

```
0: void swap(int x, int y) {
1:   int z := x + y
2:   int w := z - y
3:   z := z - w
4:   if (z != y || w != x)
5:     print("error")
6: }
```

<0>

x	y	z	w	π
X	Y			T

<0,1>

x	y	z	w	π
X	Y	X+Y		T

<0,1,2>

x	y	z	w	π
X	Y	X+Y	X	T

<0,1,2,3>

x	y	z	w	π
X	Y	Y	X	T

<0,1,2,3,4> ✗

x	y	z	w	π
X	Y	Y	X	Y≠Y ∨ X≠X

<0,1,2,3,4>

x	y	z	w	π
X	Y	Y	X	Y=Y ∧ X=X

<0,1,2,3,4,6> ✓

x	y	z	w	π
X	Y	Y	X	Y=Y ∧ X=X

- Result
  - Path <0,1,2,3,4,5,6> unfeasible
  - Path <0,1,2,3,4,6> feasible

Symbolic execution: exercise

- Consider the following program `bar`. Is the path `<0,1,2,3,4,5,8,2,3,4,7,8,2,10,11>` feasible?

```
0:  int bar( ) {
1:    int x := input()
2:    while (x > 0) {
3:      int y := 2*x
4:      if (x > 10)
5:        y := x - 1
6:      else
7:        x := x + 2
8:      x := x - 1
9:    }
10:   x := x - 1
11:   return x
12: }
```

<0>

x	y	π
		T

<0,1>

x	y	π
X		T

<0,1,2>

x	y	π
X		X>0

<0,1,2,3>

x	y	π
X	2X	X>0

<0,1,2,3,4>

x	y	π
X	2X	X>0
		X>10

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

x	y	π
X	X-1	X>10

<0,1,2,3,4,5,8>

x	y	π
X-1	X-1	X>10

<0,1,2,3,4,5,8,2>

x	y	π
X-1	X-1	X>10
		X-1>0

<0,1,2,3,4,5,8,2,3>

x	y	π
X-1	2(X-1)	X>10

<0,1,2,3,4,5,8,2,3,4>

x	y	π
X-1	2(X-1)	X>10
		X-1≤10

Symbolic execution: exercise

- Consider the following program `bar`. Is the path `<0 1 2 3 4 5 8 2 3 4 7 8 2 10 11>` feasible?

```
0:  int bar( ) {
1:    int x := input()
2:    while (x > 0) {
3:      int y := 2*x
4:      if (x > 10)
5:        y := x - 1
6:      else
7:        x := x + 2
8:      x := x - 1
9:    }
10:   x := x - 1
11:   return x
11: }
```

`<0,1,2,3,4,5,8,2,3,4>`

x	y	π
X-1	2(X-1)	X>10 X≤11

`<0,1,2,3,4,5,8,2,3,4,7>`

x	y	π
X+1	2(X-1)	X>10 X≤11

`<0,1,2,3,4,5,8,2,3,4,7,8>`

x	y	π
X	2(X-1)	X>10 X≤11

`<0,1,2,3,4,5,8,2,3,4,7,8,2>`

x	y	π
X	2(X-1)	X>10 X≤11 X≤0

- Conclusion: path `<0,1,2,3,4,5,8,2,3,4,7,8,2,10,11>` is unfeasible

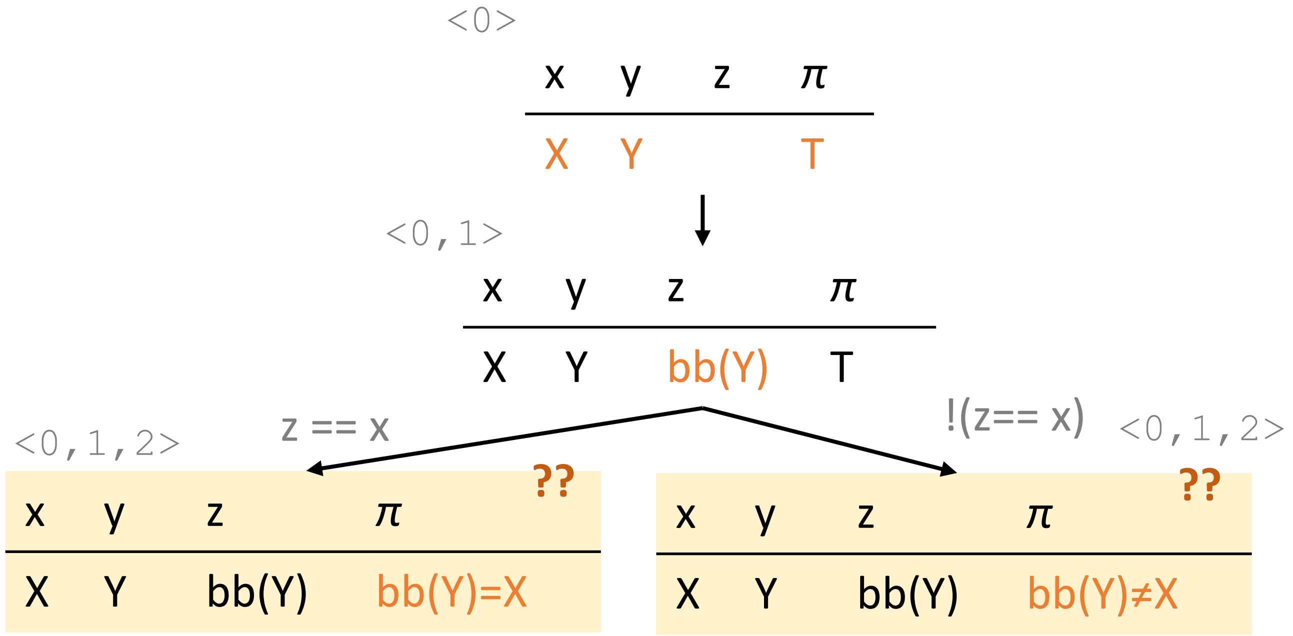
Symbolic execution: weaknesses

- It seems symbolic execution can be used to verify the correctness of any program, however…
- Limitations
  - Path conditions may be too complex for constraint solvers
    - Solvers are very good at checking linear constraints
    - It is harder for them to reason on non-linear arithmetic, bit-wise operations, string manipulation
  - Impossible/hard to use when number of paths to be explored is infinite/huge
    - unbounded loops give rise to infinite sets of paths
    - Even if set of paths is finite, checking all loops is expensive/unfeasible in practice
      - rule of thumb: approximate the analysis by considering 0, 1, and 2 iterations
  - There may be external code
    - Sources not available (e.g., pre-compiled library) → unknown behavior for the solver

Limitations: example

```
void bar(int x, int y) {
  int z := bb(y) //black-box function
  if (z == x) {
    z := y + 10
    if (x <= z)
      print("error")
  }
  print("end") //location
}
```

Solver cannot find satisfying assignments (*bb* behavior is unknown) =>
Paths remain unexplored



Symbolic execution: evolution

POLITECNICO
MILANO 1863

- Symbolic execution was first introduced in 1976
  - James C. King. (IBM Thomas J. Watson Research Center) 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252
- Became practical about 30 years later
  - Progress in constraint solving (SMT solvers)
  - Concolic techniques (presented as part of testing):
    - Combination of concrete (or dynamic) and symbolic execution
    - Can alleviate several weaknesses of classical symbolic execution
    - Can be used to generate test cases covering alternative execution paths

References

- Patrick Thomson, Static Analysis: An Introduction. ACMQueue 2021 https://queue.acm.org/detail.cfm?id=3487021
- Nichols, W. R., Jr. 2020. The cost and benefits of static analysis during development.arXiv:2003.03001; https://ui.adsabs.harvard.edu/abs/2020arXiv200303001N
- James C. King. (IBM Thomas J. Watson Research Center) 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252
- Static analysis services https://sonarcloud.io/ https://www.sonarqube.org/
- Carlo A. Furia. Material for the Software Analysis course. https://github.com/bugcounting/software-analysis/tree/master