



POLITECNICO
MILANO 1863

Software Engineering 2

Static analysis in practice

Symbolic Execution

This slide deck includes an elaboration of some of Carlo A. Furia's slides available at <https://github.com/bugcounting/software-analysis/tree/master>
distributed under the Creative Commons license <https://creativecommons.org/licenses/by-nc-nd/4.0/>



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

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

The screenshot displays the SonarCloud web interface for the project 'AWS Java SDK :: Parent'. The left sidebar shows navigation options: Overview, Main Branch, Pull Requests (60), and Branches (2). The main content area is divided into a 'Filters' panel on the left and a list of issues on the right. The 'Filters' panel includes sections for 'Clean Code Attribute' (Consistency: 704, Intentionality: 4.9k, Adaptability: 443, Responsibility: 7), 'Software Quality' (Security: 7, Reliability: 86, Maintainability: 6k), and 'Severity' (High: 527, Medium: 1.3k). The issues list shows three items, each with a title, description, and metadata. The first issue is an 'Intentionality issue' titled 'Replace this if-then-else statement by a single return statement.' with a 'Maintainability' severity and '2min effort'. The second issue is an 'Intentionality issue' titled 'Add a default case to this switch.' with a 'Maintainability' severity and '5min effort'. The third issue is an 'Adaptability issue' titled 'This class has 7 parents which is greater than 5 authorized.' with a 'Maintainability' severity and '5h effort'.

sonarcloud

My Projects My Issues Explore

AWS Java SDK :: Parent

Amazon Web Services > AWS Java SDK :: Parent > master

Summary **Issues** Security Hotspots Measures Code Activity

The last analysis has warnings. [See details](#)

Select issues Navigate to issue 6,078 issues 82d effort

build-tools/.../awssdk/buildtools/checkstyle/NonJavaBaseModuleCheck.java

Intentionality issue

[Replace this if-then-else statement by a single return statement.](#)

No tags

Open Matthew Miller Maintainability Code Smell Minor 2min effort • 1 year ago

build-tools/.../awssdk/buildtools/checkstyle/SdkIllegalImportCheck.java

Intentionality issue

[Add a default case to this switch.](#)

No tags

Open Not assigned Maintainability Code Smell Critical 5min effort • 2 years ago

build-tools/.../awssdk/buildtools/checkstyle/SdkPublicMethodNameCheck.java

Adaptability issue

[This class has 7 parents which is greater than 5 authorized.](#)

design

Open Not assigned Maintainability Code Smell Major 5h effort • 5 years ago

build-tools/.../awssdk/buildtools/findbugs/DisallowMethodCall.java



POLITECNICO
MILANO 1863

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

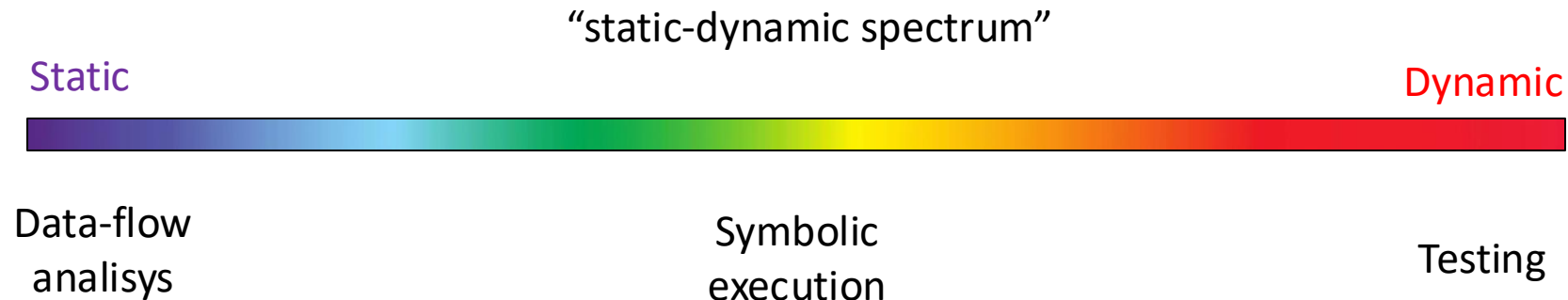
Static vs dynamic, again

- **Static** analysis

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

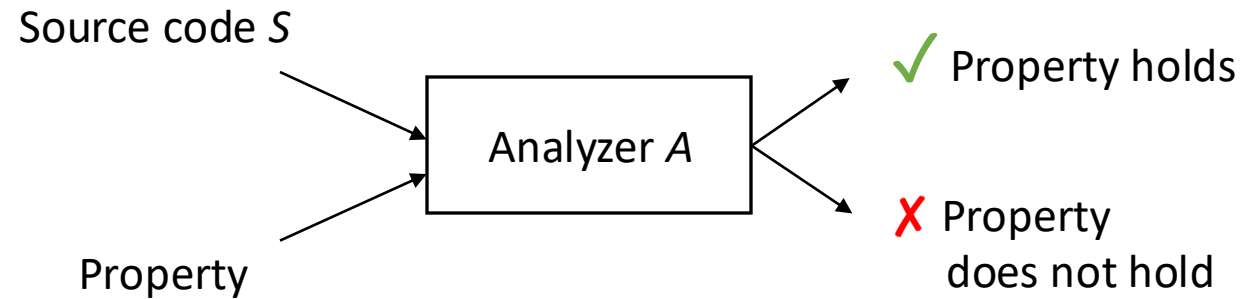
- **Dynamic** analysis

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



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

- **Reachability**: does some execution of the program reach the location **1** in S ?
 - Symbolic exec tries to **verify that 1 cannot be reached**, or alternatively **spots the condition under which 1 can be reached**

```
...                               ...
k:    try {                       l-1: if (x < 0) {
k+1:    ...                       1:    /* safe */
l-1: } catch (e) {
1:    /* error */
...    }
```

- **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**

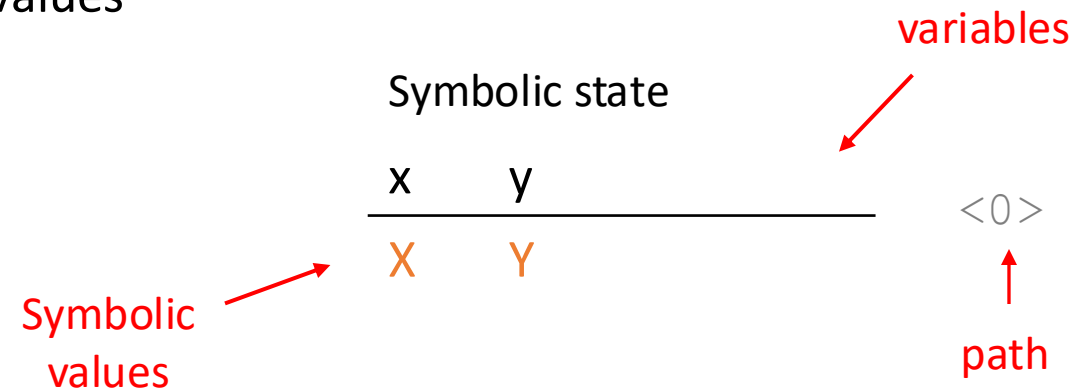
$$p = \langle 0, 1, \dots, k, \dots, n \rangle$$

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

```
0: void foo(int x, int y) {  
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 statement **updates** the symbolic **state**

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

Symbolic state

x	y	z	<0, 1>
X	Y	X	

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

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

```

Symbolic state

x	y	z	π	<0, 1, 2, 3>
X	Y	2X	X<Y	

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
					$X < Y \wedge 2X \geq Y$
Condition at point 4 false:		X	Y	2X	X<Y
					$X \geq Y \vee 2X < Y$

<0, 1, 2, 3, 4>

<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	$X < Y$	$\langle 0, 1, 2, 3, 4, 5 \rangle$
			$X < Y \wedge 2X \geq Y$	

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 $\langle 0, 1, 2, 4, 5 \rangle$ 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	x	y	z	π	
point 4 true:	X	Y	X	$X \geq Y$	$\langle 0, 1, 2, 4 \rangle$

$X < Y \wedge X \geq Y$

UNSAT exit

There is **no satisfying** assignment

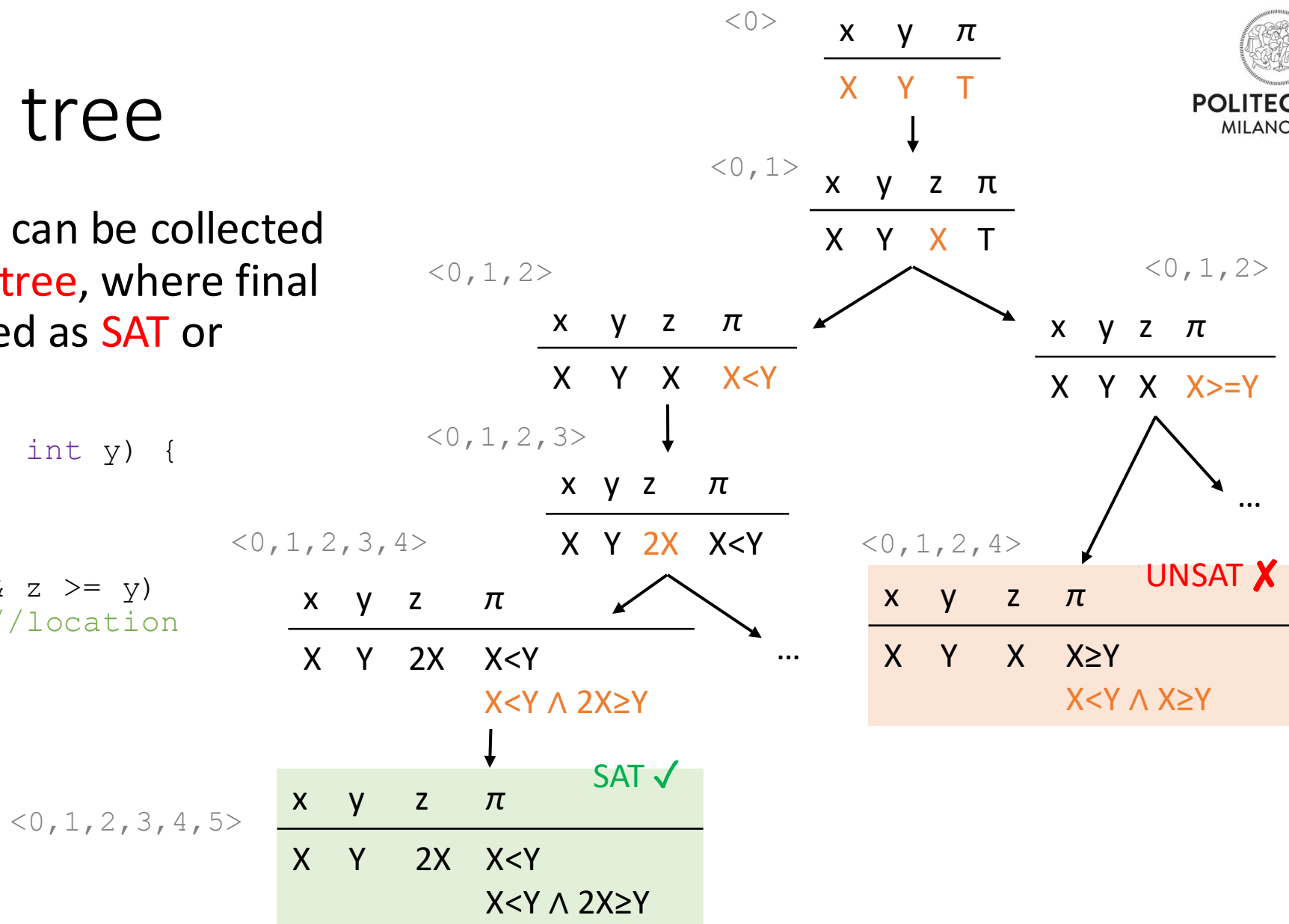
Execution tree

- 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?

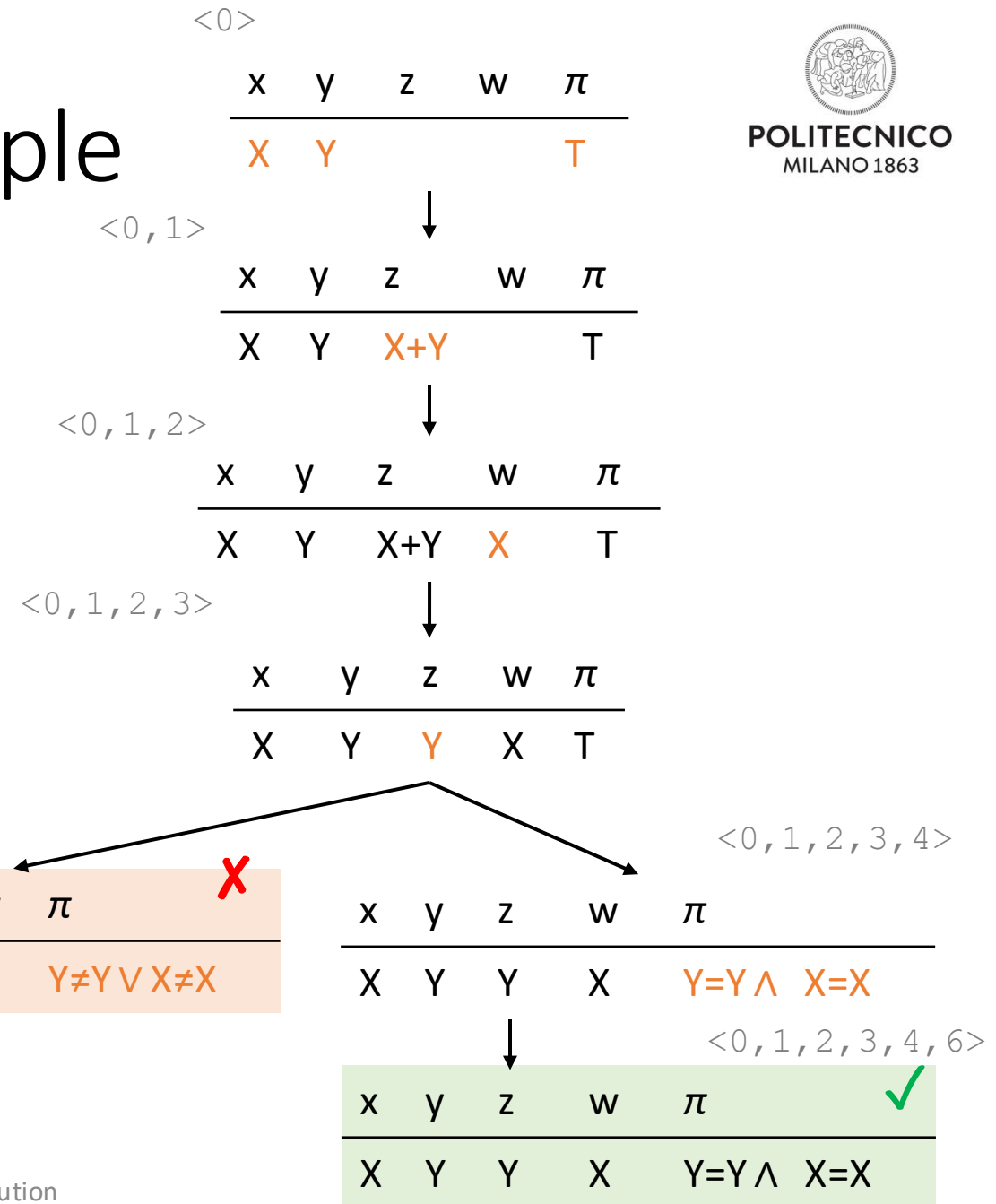
```

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: }

```

Result

- Path $\langle 0, 1, 2, 3, 4, 5, 6 \rangle$ unfeasible
- Path $\langle 0, 1, 2, 3, 4, 6 \rangle$ feasible



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

$\langle 0 \rangle$ <table border="0"> <tr> <td>x</td> <td>y</td> <td>π</td> </tr> <tr> <td colspan="2"></td> <td>T</td> </tr> </table>	x	y	π			T	$\langle 0, 1 \rangle$ <table border="0"> <tr> <td>x</td> <td>y</td> <td>π</td> </tr> <tr> <td>X</td> <td></td> <td>T</td> </tr> </table>	x	y	π	X		T	$\langle 0, 1, 2 \rangle$ <table border="0"> <tr> <td>x</td> <td>y</td> <td>π</td> </tr> <tr> <td>X</td> <td>X>0</td> <td></td> </tr> </table>	x	y	π	X	X>0		$\langle 0, 1, 2, 3 \rangle$ <table border="0"> <tr> <td>x</td> <td>y</td> <td>π</td> </tr> <tr> <td>X</td> <td>2X</td> <td>X>0</td> </tr> </table>	x	y	π	X	2X	X>0
x	y	π																									
		T																									
x	y	π																									
X		T																									
x	y	π																									
X	X>0																										
x	y	π																									
X	2X	X>0																									
$\langle 0, 1, 2, 3, 4 \rangle$ <table border="0"> <tr> <td>x</td> <td>y</td> <td>π</td> </tr> <tr> <td>X</td> <td>2X</td> <td>X>0</td> </tr> <tr> <td colspan="2"></td> <td>X>10</td> </tr> </table>	x	y	π	X	2X	X>0			X>10	$\langle 0, 1, 2, 3, 4, 5 \rangle$ <table border="0"> <tr> <td>x</td> <td>y</td> <td>π</td> </tr> <tr> <td>X</td> <td>X-1</td> <td>X>10</td> </tr> </table>	x	y	π	X	X-1	X>10	$\langle 0, 1, 2, 3, 4, 5, 8 \rangle$ <table border="0"> <tr> <td>x</td> <td>y</td> <td>π</td> </tr> <tr> <td>X-1</td> <td>X-1</td> <td>X>10</td> </tr> </table>	x	y	π	X-1	X-1	X>10				
x	y	π																									
X	2X	X>0																									
		X>10																									
x	y	π																									
X	X-1	X>10																									
x	y	π																									
X-1	X-1	X>10																									
$\langle 0, 1, 2, 3, 4, 5, 8, 2 \rangle$ <table border="0"> <tr> <td>x</td> <td>y</td> <td>π</td> </tr> <tr> <td>X-1</td> <td>X-1</td> <td>X>10</td> </tr> <tr> <td colspan="2"></td> <td>X-1>0</td> </tr> </table>	x	y	π	X-1	X-1	X>10			X-1>0	$\langle 0, 1, 2, 3, 4, 5, 8, 2, 3 \rangle$ <table border="0"> <tr> <td>x</td> <td>y</td> <td>π</td> </tr> <tr> <td>X-1</td> <td>2(X-1)</td> <td>X>10</td> </tr> </table>	x	y	π	X-1	2(X-1)	X>10	$\langle 0, 1, 2, 3, 4, 5, 8, 2, 3, 4 \rangle$ <table border="0"> <tr> <td>x</td> <td>y</td> <td>π</td> </tr> <tr> <td>X-1</td> <td>2(X-1)</td> <td>X>10</td> </tr> <tr> <td colspan="2"></td> <td>X-1≤10</td> </tr> </table>	x	y	π	X-1	2(X-1)	X>10			X-1≤10	
x	y	π																									
X-1	X-1	X>10																									
		X-1>0																									
x	y	π																									
X-1	2(X-1)	X>10																									
x	y	π																									
X-1	2(X-1)	X>10																									
		X-1≤10																									

Symbolic execution: exercise (continue)

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

<pre> 0: int bar() { 1: int y, x := input() 2: while (x > 0) { 3: 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: }</pre>	<0,1,2,3,4,5,8,2,3,4>			<0,1,2,3,4,5,8,2,3,4,7>		
	x	y	π	x	y	π
	X-1	2(X-1)	X>10	X+1	2(X-1)	X>10
			X≤11			X≤11
	<0,1,2,3,4,5,8,2,3,4,7,8>			<0,1,2,3,4,5,8,2,3,4,7,8,2>		
	x	y	π	x	y	π
	X	2(X-1)	X>10	X	2(X-1)	X>10
			X≤11			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

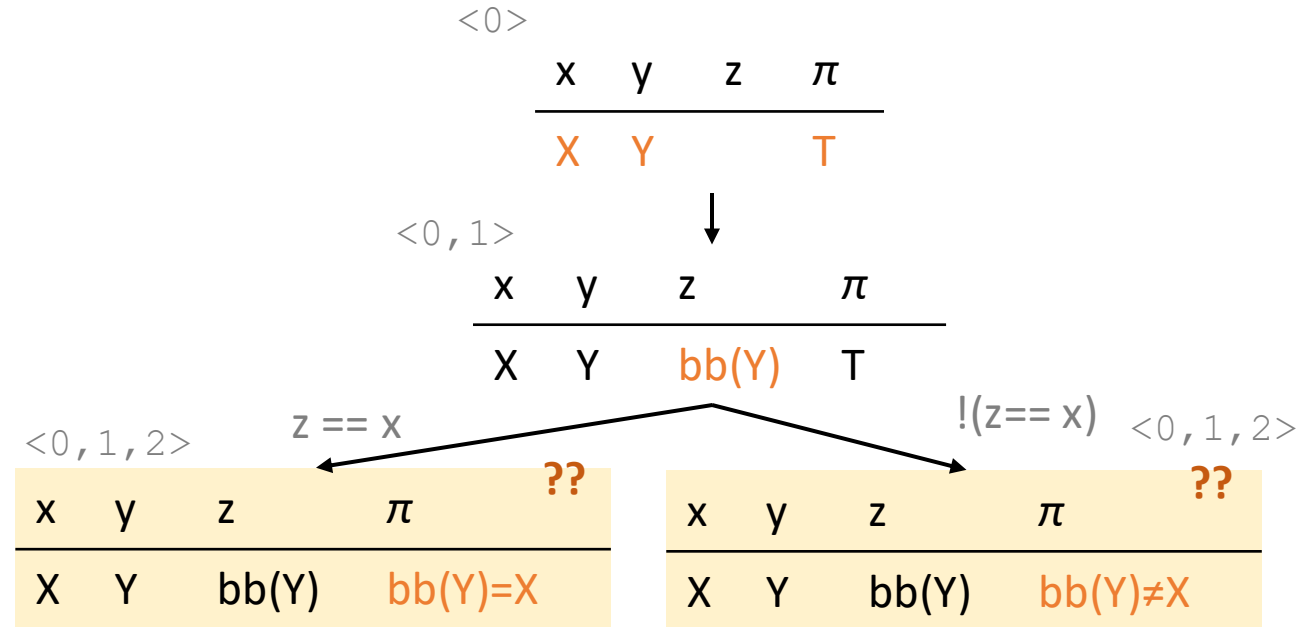
```

0: void bar(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("error")
6:   }
7:   print("end") //location
8: }

```

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

Paths remain **unexplored**



Symbolic execution: evolution

- 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>