Concolic Testing: A modern software testing technique

Dr. Durga Prasad Mohapatra
Professor
Department of CSE, NIT Rourkela

Seminar Outline

1 Introduction

2 Fundamental Ideas

3 Survey of Related works

Introduction

Software Testing is an important phase in SDLC.

- Helps to achieve software dependability and improve quality.
- Time consuming: Approximately 40% of software development time is devoted to testing.

- Testing can be done in two ways – Manual vs. Automated.
- **An automation tool** for test case generation can effectively reduce the time required in testing.
- **Automation tool** can be designed to generate test cases for different test coverage criteria.

Reliability and Coverage Model -By Prof. Aditya Mathur

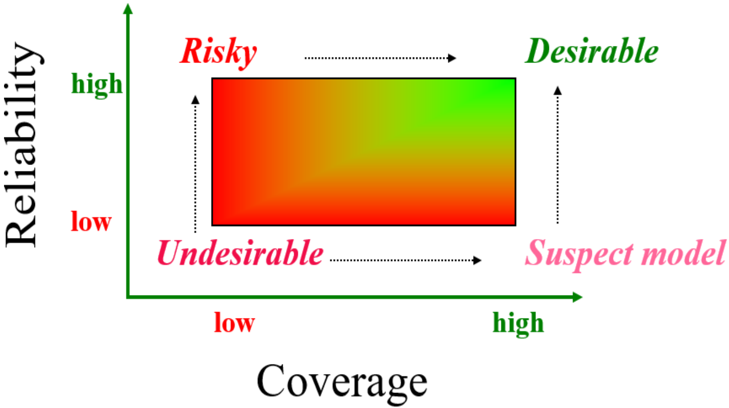


Figure 1: Reliability and Coverage

Software Testing

“No product of human intellect comes out right the first time. We rewrite sentences, rip out knitting stitches, replant gardens, remodel houses, and repair bridges. Why should software be any different?”

- By Wiener, Ruth: Digital Woes, Why We Should Not Depend on Software.

Software Testing

- The purpose of the verification process is to detect and report errors that have been introduced in the development process.
- The verification process must ensure that the produced software implements intended function completely and correctly, while avoiding unintended function.
- Verification is an integral process, which is coupled with every development step. Testing quality at the end of the life cycle is impractical.

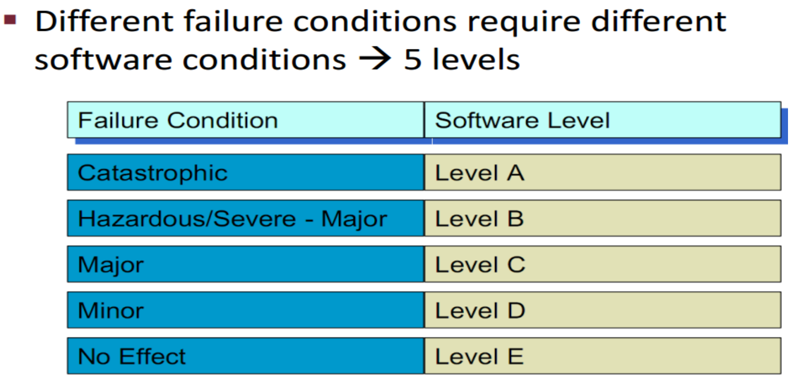
Software Testing: Coverage

- Coverage refers to the extent to which a given verification activity has satisfied its objectives: in essence, providing an exit criteria for when to stop. That is, what is “enough”is defined in terms of coverage.
- Coverage is a measure, not a method or a test. As a measure, coverage is usually expressed as the percentage of an activity that is accomplished.
- **Our goal, then, should be to provide enough testing to ensure that the probability of failure due to hibernating bugs is low enough to accept. “Enough”implies judgment.**

RTCA/DO178-B/C standards

- RTCA stands for Radio Technical Commission for Aeronautics.
- DO-178B (and DO-278) are used to assure safety of avionics software.
- DO-178C includes the coverage analysis of Object-Oriented Programs used in safety of avionics software.
- These documents provide guidance in the areas of SW development, configuration management, verification and the interface to approval authorities (e.g., FAA, EASA).

Levels of Software



- Different failure conditions require different software conditions → 5 levels

Failure Condition	Software Level
Catastrophic	Level A
Hazardous/Severe - Major	Level B
Major	Level C
Minor	Level D
No Effect	Level E

Figure 2: Levels of Software

Relation of Coverages

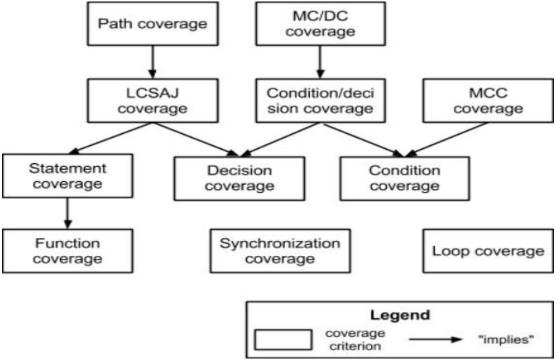


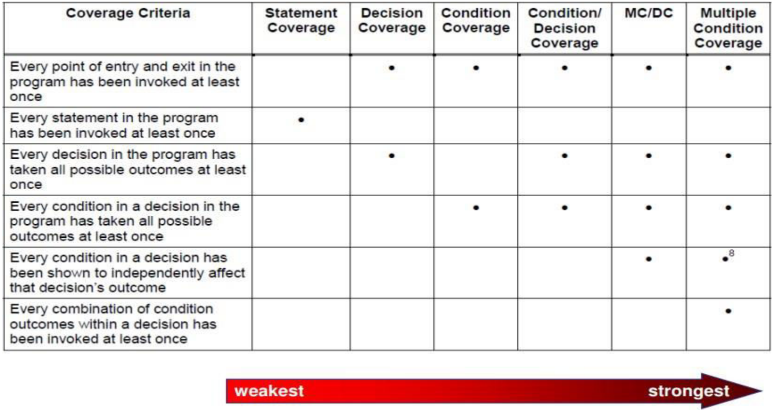
Figure 3: Relation of Coverages

[redacted]

LCSAJ Coverage

- **LCSAJ** stands for Linear Code Sequence and Jump.
- It is a white box **testing** technique to identify the **code coverage**.
- It begins at the start of the program or branch and ends at the end of the program or the branch.
- **LCSAJ** is ordinarily equivalent to **statement coverage**.

Types of Structural Coverage



Coverage Criteria	Statement Coverage	Decision Coverage	Condition Coverage	Condition/ Decision Coverage	MC/DC	Multiple Condition Coverage
Every point of entry and exit in the program has been invoked at least once		•	•	•	•	•
Every statement in the program has been invoked at least once	•					
Every decision in the program has taken all possible outcomes at least once		•		•	•	•
Every condition in a decision in the program has taken all possible outcomes at least once			•	•	•	•
Every condition in a decision has been shown to independently affect that decision's outcome					•	•[8]
Every combination of condition outcomes within a decision has been invoked at least once						•

Figure 4: Types

Definitions

Predicate / Boolean Expression

A predicate is an expression that evaluates to a boolean value, and which is required for our approach.
A simple example is: $((a > b) \vee C) \wedge p(x)$. Predicates may contain boolean variables, non-boolean variables that are compared with relational operators, and calls to function that return a boolean value, all three of which may be joined with logical operators.

Definitions

Predicate Coverage

For each p $\in$ P, Test Requirement (TR) for predicate coverage contains two requirements: p evaluates to true, and p evaluates to false.

Definitions

Clause / Atomic Condition

A clause is a predicate that does not contain any of the logical operators.

The predicate $((a > b) \vee C) \wedge p(x)$ contains three clauses; a relational expression $(a > b)$, a boolean variable C and a boolean function call p(x).

Definitions

Clause Coverage

For each c $\in$ C, TR for clause coverage contains two requirements: c evaluates to true, and c evaluates to false.

Definitions

Statement Coverage

- The statement coverage based strategy aims to design the test cases so as to execute every statement in a program at least once.
- The principle idea governing the statement coverage strategy is that a unless a statement is executed, there is no way to determine whether as error exists in that statement.

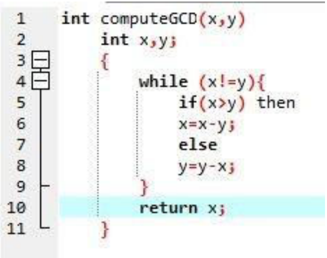
Definitions

Branch Coverage

Each decision should take all possible outcomes at least once either true or false.

For example if(m >n), the test cases are (1) m<=n, (2) m>n

Example



```
int computeGCD(x,y)
    int x,y;
    {
        while (x!=y){
            if(x>y) then
            x=x-y;
            else
            y=y-x;
        }
        return x;
    }
```

Figure 5: Euclid's GCD computation program

Test cases for Statement Coverage

- To design the test cases for the statement coverage, the conditional expression of the while statements needs to be made true and the conditional expression of the if statement needs to be made both true and false.
- By choosing the test set $\{(x=3,y=3), (x=4,y=3),(x=3,y=4)\}$, all the statements of the program would be executed at-least once.

Test cases for Statement Coverage

- For the GCD program, the test cases for branch coverage can be $\{(x=3,y=3),\ (x=3,y=2),\ (x=4,y=3),(x=3,y=4)\}$.

Observation

- It is easy to show that branch coverage based testing is a stronger testing than statement coverage-based testing.
- We can prove this by showing that branch coverage ensures statement coverage, but not vice-versa.

Definitions

Concolic Testing

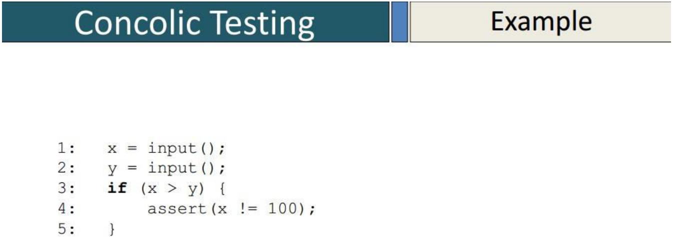
- The concept of CONCOLIC testing combines the CONCrete constraints execution and symBOLIC constraints execution to automatically generate test cases for full path coverage.
- This testing generates test suites by executing the program with random values.
- At execution time both concrete and symbolic values are saved for execution path.
- During execution, the variables are stored in some symbolic values such as x_0, and y_0, instead of x and y.
- The next iteration of the process forces the selection of different paths.

Definitions

Concolic Testing

- The tester selects a value from the path constraints and negates the values to create a new path value. Then the tester finds concrete constraints to satisfy the new path values.
- The selection of values is responsible by the Constraint Solver which is a part of Concolic tester.
- These constraints are inputs for all next executions. This concolic testing is performed iteratively until exceeds the threshold value or sufficient code coverage is obtained.

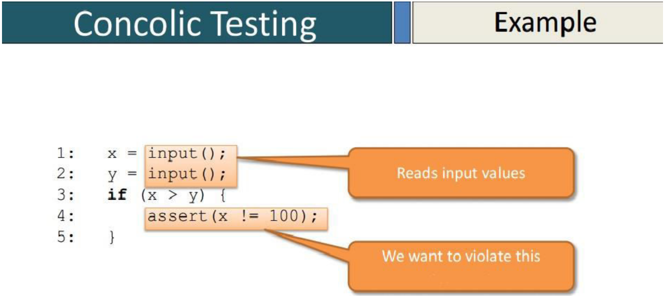
Concolic Testing Example



Concolic Testing | Example

```
x = input();
y = input();
if (x > y) {
    assert(x != 100);
}
```

Concolic Testing Example



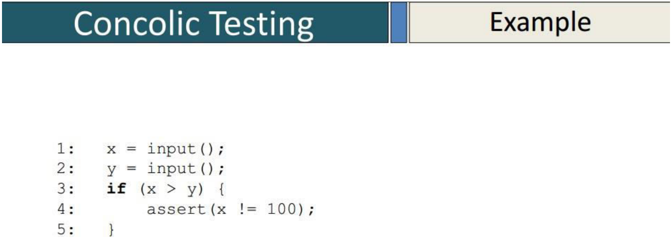
Concolic Testing | Example

```
x = input();
y = input();
if (x > y) {
    assert(x != 100);
}
```

Reads input values

We want to violate this

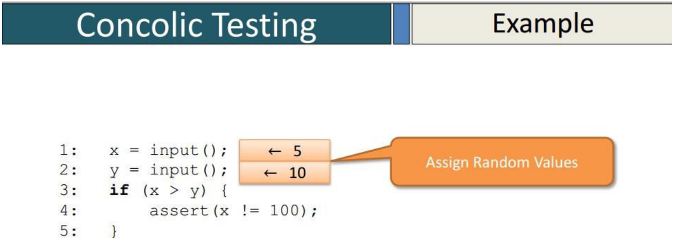
Concolic Testing Example



Concolic Testing | Example

```
x = input();
y = input();
if (x > y) {
    assert(x != 100);
}
```

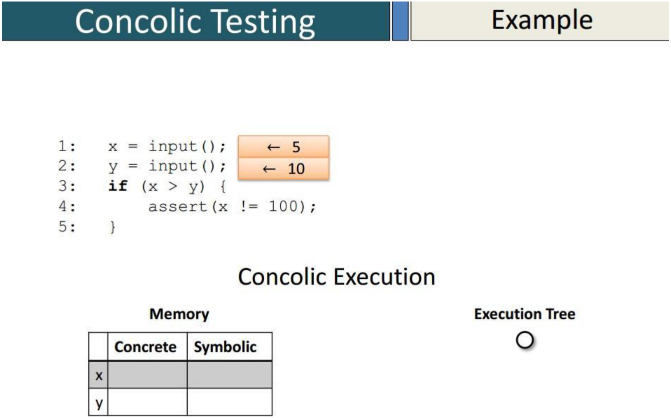
Concolic Testing Example



Concolic Testing | Example

```
x = input();
y = input();
if (x > y) {
    assert(x != 100);
}
```

Concolic Testing Example



Concolic Testing | Example

```
x = input();   ← 5
y = input();   ← 10
if (x > y) {
    assert(x != 100);
}
```

Concolic Execution

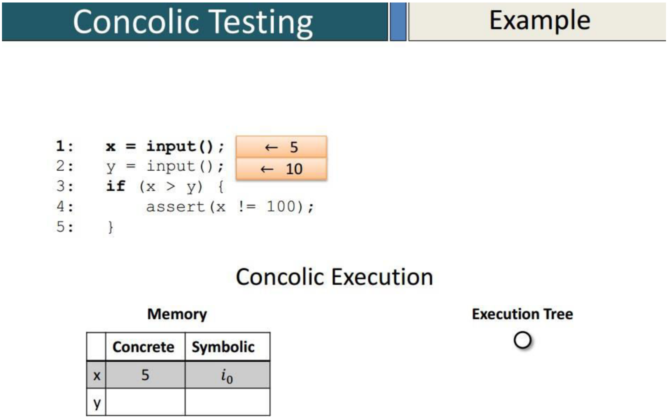
Memory

	Concrete	Symbolic
x		
y		

Execution Tree

○

Concolic Testing Example



Concolic Testing | Example

```
x = input();          ← 5
y = input();          ← 10
if (x > y) {
    assert(x != 100);
}
```

Concolic Execution

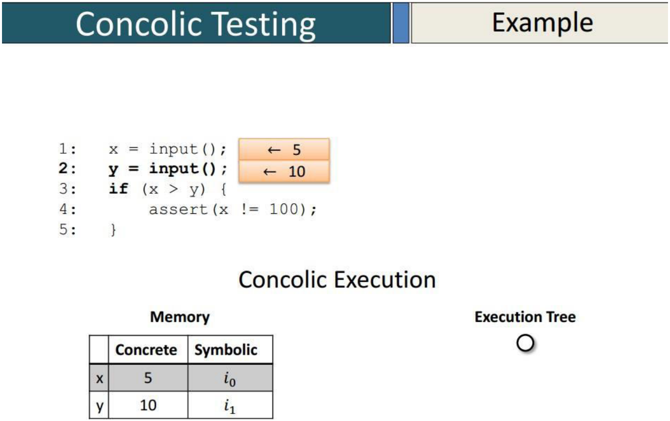
Memory

	Concrete	Symbolic
x	5	i_0
y		

Execution Tree

○

Concolic Testing Example



Concolic Testing | Example

```
x = input();       ← 5
y = input();       ← 10
if (x > y) {
    assert(x != 100);
}
```

Concolic Execution

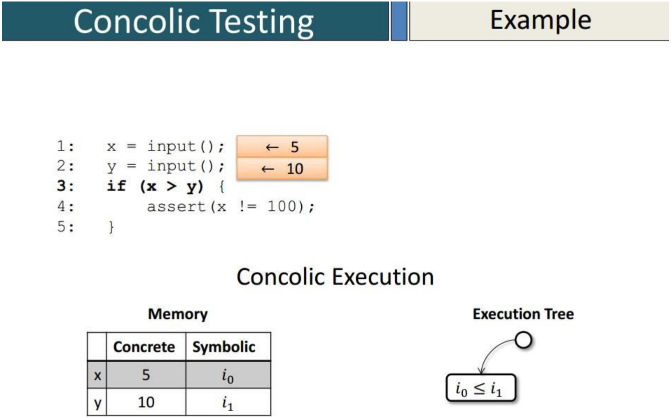
Memory

	Concrete	Symbolic
x	5	i_0
y	10	i_1

Execution Tree

○

Concolic Testing Example



Concolic Testing | Example

```
x = input();     ← 5
y = input();     ← 10
if (x > y) {
    assert(x != 100);
}
```

Concolic Execution

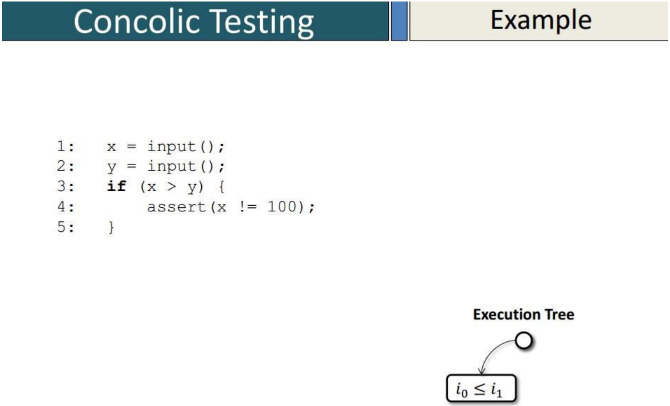
Memory

	Concrete	Symbolic
x	5	i_0
y	10	i_1

Execution Tree

$i_0 \leq i_1$

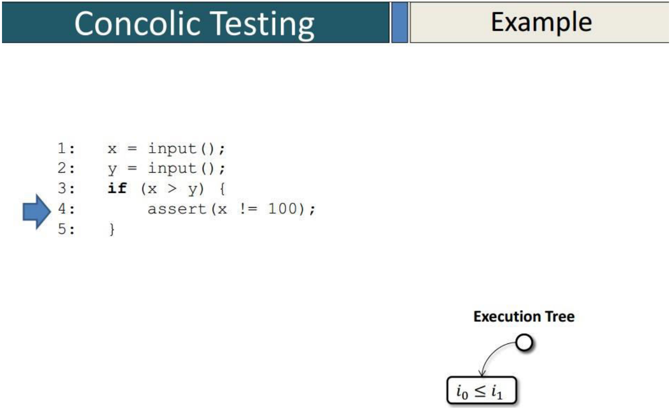
Concolic Testing Example



Concolic Testing | Example

```
x = input();
y = input();
if (x > y) {
    assert(x != 100);
}
```

Concolic Testing Example



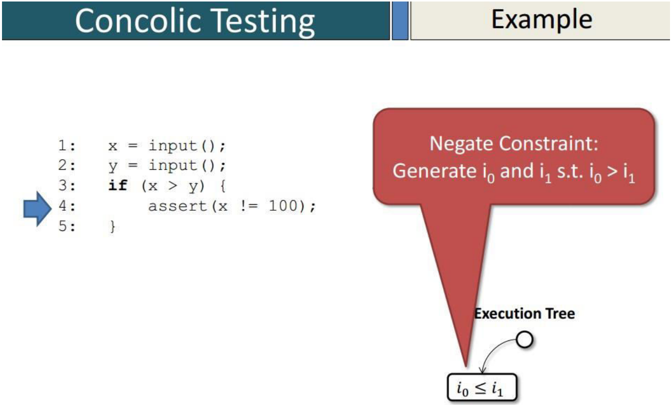
Concolic Testing

Example

```
x = input();
y = input();
if (x > y) {
    assert(x != 100);
}
```

Execution Tree

Concolic Testing Example

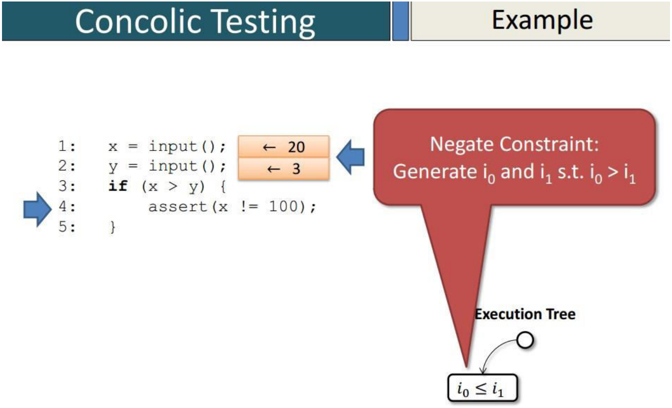


Concolic Testing | Example

```
x = input();
y = input();
if (x > y) {
    assert(x != 100);
}
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$

Concolic Testing Example



Concolic Testing | Example

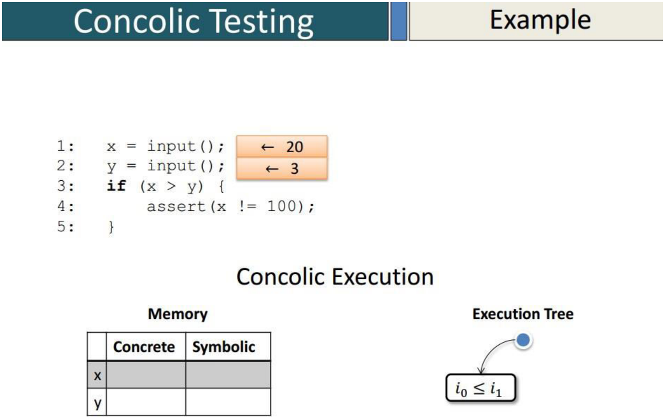
```
x = input();
y = input();
if (x > y) {
    assert(x != 100);
}
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$

Execution Tree

$i_0 \le i_1$

Concolic Testing Example



Concolic Testing | Example

```
x = input();   ← 20
y = input();   ← 3
if (x > y) {
    assert(x != 100);
}
```

Concolic Execution

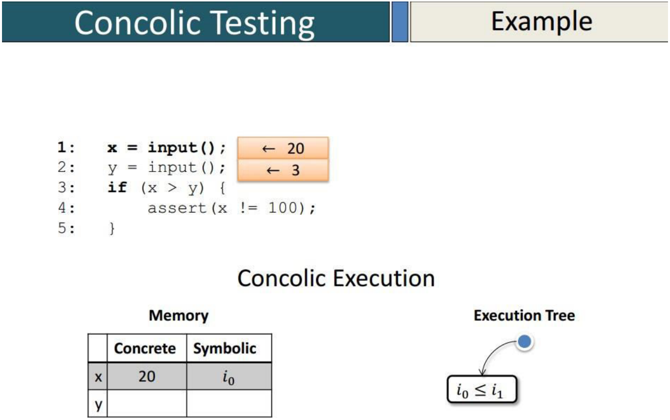
Memory

	Concrete	Symbolic
x		
y		

Execution Tree

$i_0 \leq i_1$

Concolic Testing Example



Concolic Testing | Example

```
x = input();      ← 20
y = input();      ← 3
if (x > y) {
    assert(x != 100);
}
```

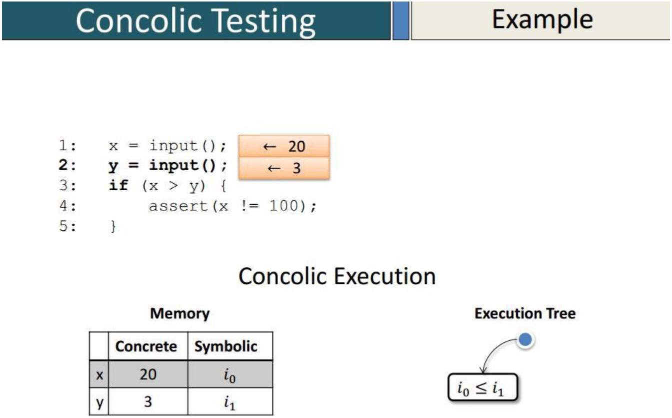
Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y		

Execution Tree

Concolic Testing Example



Concolic Testing | Example

```
x = input();        ← 20
y = input();        ← 3
if (x > y) {
    assert(x != 100);
}
```

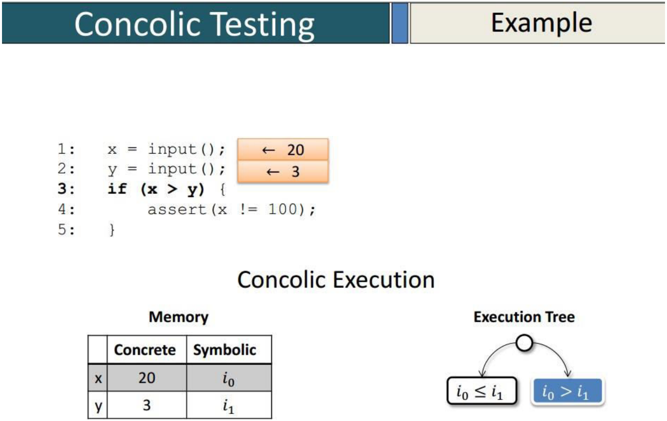
Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y	3	i_1

Execution Tree

Concolic Testing Example



Concolic Testing | Example

```
x = input();      ← 20
y = input();      ← 3
if (x > y) {
      assert(x != 100);
}
```

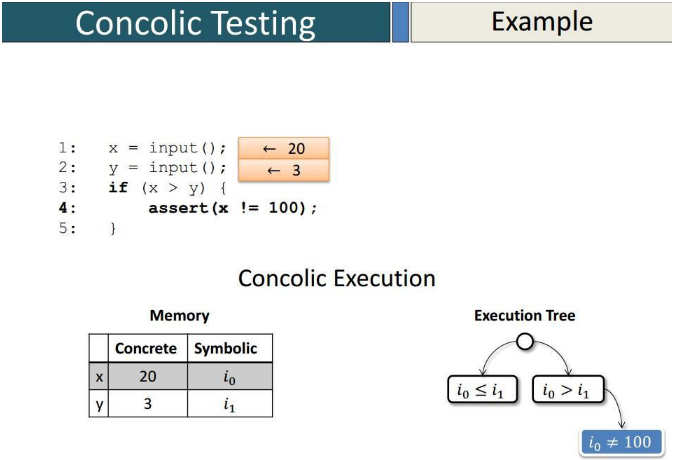
Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y	3	i_1

Execution Tree

Concolic Testing Example



Concolic Testing | Example

```
x = input();     ← 20
y = input();     ← 3
if (x > y) {
    assert(x != 100);
}
```

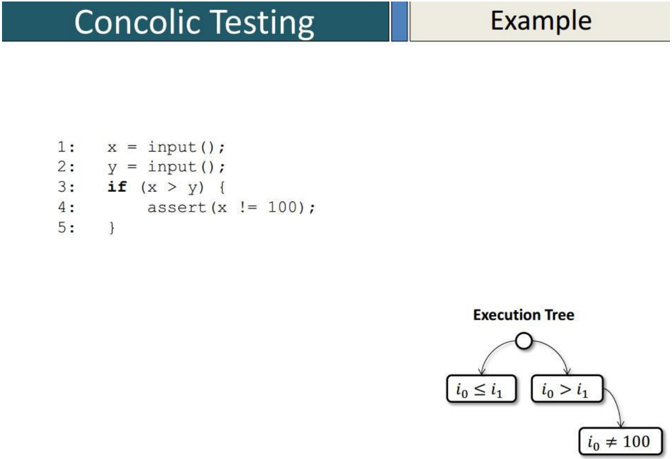
Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y	3	i_1

Execution Tree

Concolic Testing Example

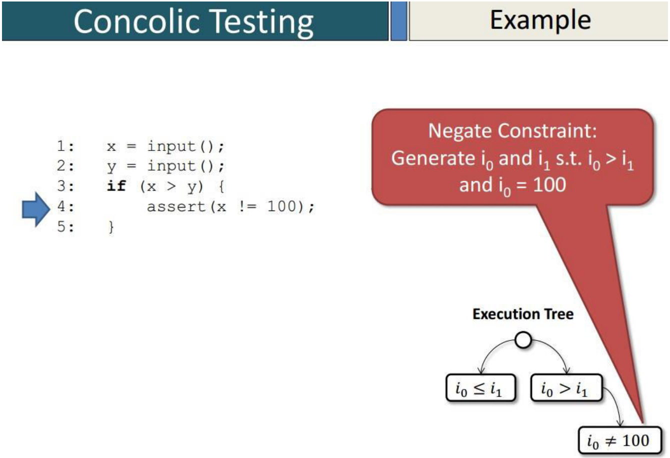


Concolic Testing | Example

```
x = input();
y = input();
if (x > y) {
    assert(x != 100);
}
```

Execution Tree

Concolic Testing Example

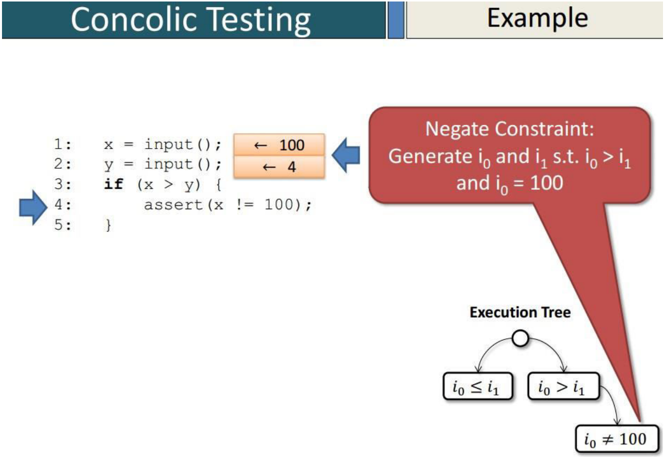


Concolic Testing | Example

```
x = input();
y = input();
if (x > y) {
    assert(x != 100);
}
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$
and $i_0 = 100$

Concolic Testing Example

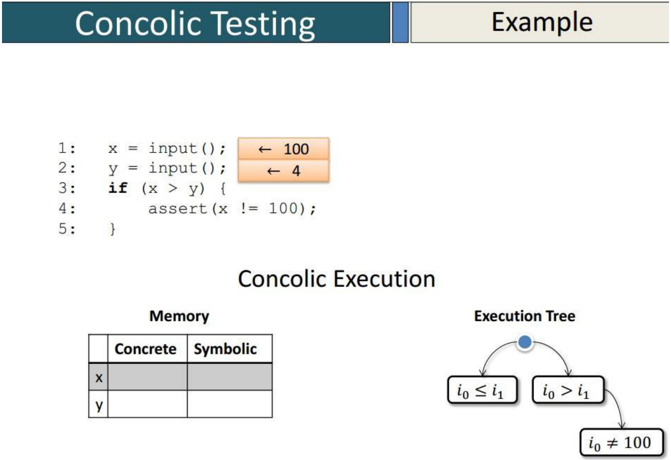


Concolic Testing | Example

```
x = input();     ← 100
y = input();     ← 4
if (x > y) {
    assert(x != 100);
}
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$ and $i_0 = 100$

Concolic Testing Example



Concolic Testing | Example

```
x = input();       ← 100
y = input();       ← 4
if (x > y) {
    assert(x != 100);
}
```

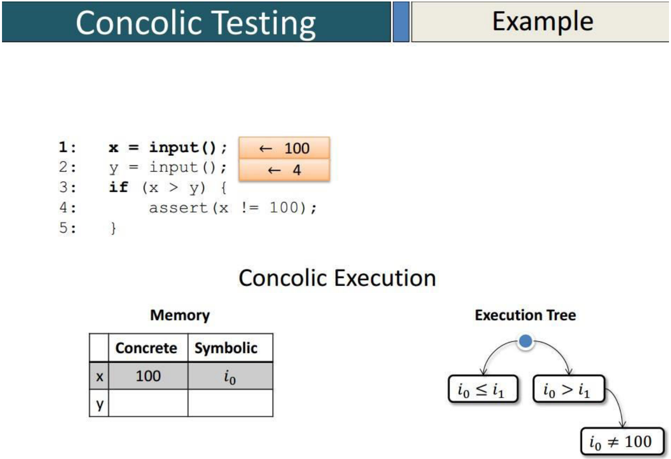
Concolic Execution

Memory

	Concrete	Symbolic
x		
y		

Execution Tree

Concolic Testing Example



Concolic Testing | Example

```
x = input();    ← 100
y = input();    ← 4
if (x > y) {
    assert(x != 100);
}
```

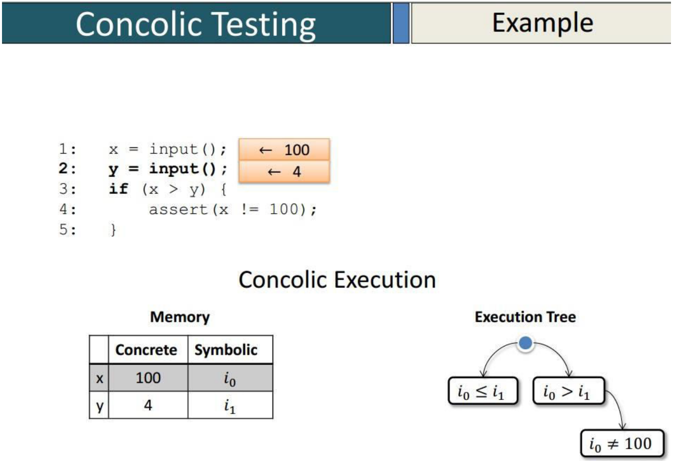
Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y		

Execution Tree

Concolic Testing Example



Concolic Testing | Example

```
x = input();      ← 100
y = input();      ← 4
if (x > y) {
    assert(x != 100);
}
```

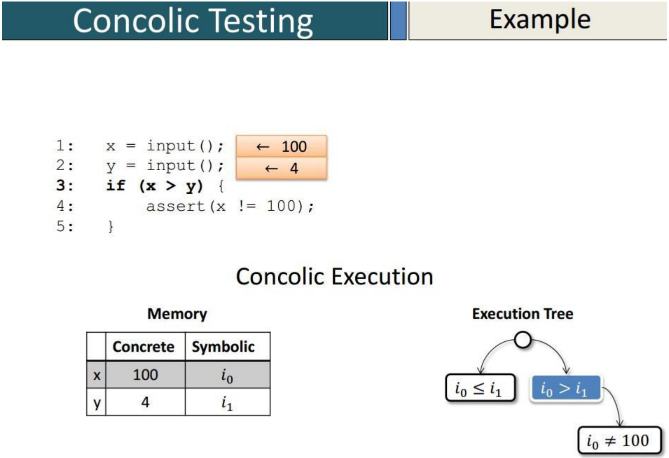
Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y	4	i_1

Execution Tree

Concolic Testing Example



Concolic Testing | Example

```
x = input();   ← 100
y = input();   ← 4
if (x > y) {
    assert(x != 100);
}
```

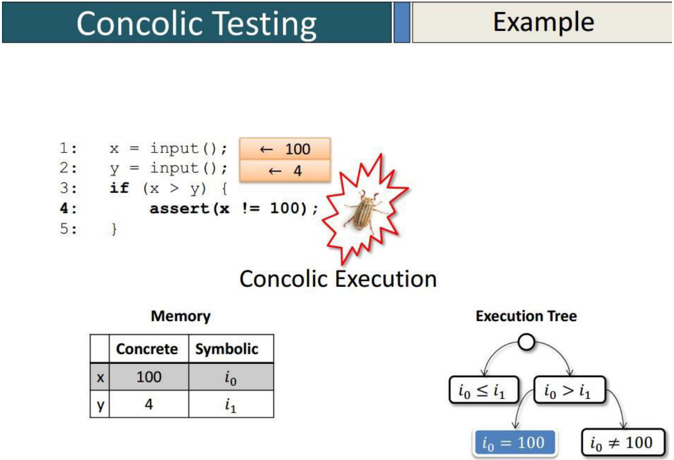
Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y	4	i_1

Execution Tree

Concolic Testing Example



Concolic Testing | Example

```
x = input();          ← 100
y = input();          ← 4
if (x > y) {
      assert(x != 100);
}
```

Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y	4	i_1

Execution Tree

Definitions

Concolic Testing: Final report

- Concolic testing technique explored total 3 paths for the example program.
- It has generated total three test cases for the variables x and y. These are shown in Table 1.

Table 1: Test cases

Test cases	TC1	TC2	TC3
x	5	20	100
y	10	3	4

Definitions

Distributed Concolic Testing

- *Distributed Concolic Testing (DCT)* is an extension of the original concolic testing approach that uses several computing nodes in a distributed manner.
- It significantly reduces the time to generate test cases with better efficiency.
- Concolic testing is the combination of concrete and symbolic testing. In addition, distributed concolic testing is scalable and achieves a linear speedup by using a large number of computing nodes for test case generation.

Concolic Testers

In Table 1, we have compared concolic testing tools with respect to the following parameters/features: 1) variables types; 2) pointers; 3) native calls; 4) non-linear arithmetic operations; 5) bitwise operations; 6) array offsets and 7) function pointers.

The abbreviation used in Table 1 are the following: “Y

- ”means the tool supports the feature. “N”means
- the tool does not support the feature.
- “P”means the tool can partially support the feature.
- “NA”means unknown.

Concolic Testers

Table 2: Summary of concolic tester with their properties.

Tool Name	Supporting Language	Supporting Platform	Support Constraints Solver	Support for float/double	Support for pointer	Support for native call	Support for non-linear arithmetic op.	Support for bitwise op.	Support for offset	Support for function pointer
DART	C	NA	LP_SOLVER	N	N	N	NA	NA	N	N
SMART	C	LINUX	LP_SOLVER	N	N	N	NA	NA	N	N
CUTE	C	LINUX	LP_SOLVER	N	Y	N	NA	NA	N	N
jCUTE	**JAVA**	**LINUX/WINDOWS**	**NA**	**N**	**-**	**N**	**NA**	**NA**	**N**	**N**
CREST	**C**	**LINUX**	**YICES**	**N**	**N**	**N**	**P**	**P**	**N**	**N**
EXE	C	LINUX	STP	N	Y	N	Y	Y	Y	N
KLEE	C	LINUX	STP	N	Y	P	Y	Y	Y	NA
RWSET	C	LINUX	STP	N	Y	N	Y	Y	Y	NA
jFUZZ	JAVA	LINUX	BUILT ON JPF	N	NA	N	N	N	NA	NA
PATHCRAWLER	C	NA	NA	NA	NA	N	NA	NA	NA	NA
PEX	.NET	WINDOWS	Z3	N	NA	N	NA	NA	NA	NA
SAGE	MACHINE CODE	WINDOWS	DISOLVER	NA	N	Y	NA	NA	NA	NA
APOLLO	PHP	WINDOWS	CHOCO	NA	NA	N	NA	N	NA	NA
SCORE	**C**	**LINIX**	**Z3 SMT Solver**	**Y**	**N**	**N**	**Y**	**N**	**NA**	**NA**

Comparison of related works

Table 3: Summary of different work on concolic testing.

S.No	Authors	Testing Type	FrameWork Type	Input Type	Output Type
1	Das et al. [16]	Concolic Testing, MC/DC	BCT,CREST, CA	C-Program	MC/DC%
2	Bokil et al. [24]	SC, DC, BC,MC/DC	AutoGen	C-Program	Test data, Time
3	Kim et al. [20]	HCT	SMT Solver, CREST	Flash storage Platform Software	Reduction Ratio
4	Majumdar et al. [17]	HCT, BC	CUTE	Editor in C-Language	Test Cases
5	Burnim et al. [21]	Heuristics Concolic Testing, BC	CREST	Software Application in C	Branch Covered
6	Kim et al. [23]	Concolic Testing	CREST	Embedded C Application	Branch Covered
7	Kim et al. [22]	Concolic Testing	CONBOL	Embedded Software	BC%, Time
8	Kim et al. [19, 25]	Distributed Concolic Testing	SCORE	Embedded C Program	BC%, Effectiveness
9	Sen et al. [26]	Concolic Testing, BC	CUTE, JCUTE	C and Java Programs	Test Cases, BC%, Time

Characteristics of different approaches

Table 4: Characteristics of different approaches on concolic testing.

Sl No	Authors	Generated Test Cases	Measuring Coverage%	Determined Time Constraints	Computed Speed
1	Das et al. [16]	C	C	X	X
2	Bokil et al. [24]	C	X	C	X
3	Kim et al. [20]	C	X	X	X
4	Majumdar et al. [17]	C	X	X	X
5	Burnim et al. [21]	C	X	X	X
6	Kim et al. [23]	C	X	X	X
7	Kim et al. [22]	C	C	C	X
8	Kim et al. [19,25]	C	C	X	C
9	Sen et al. [26]	C	C	C	X

References ≫ I

1. Palacios, M., Garc´ıa-Fanjul, J., Tuya, J. and Spanoudakis, G., 2015. Coverage-based testing for service level agreements. IEEE Transactions on Services Computing, 8(2), pages 299-313.
2. Fraser, G. and Arcuri, A., 2014. A large-scale evaluation of automated unit test generation using EvoSuite. ACM Transactions on Software Engineering and Methodology (TOSEM), 24(2), page 8.
3. Jones, JA. and Harrold, MJ. 2013. Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Transactions on Software Engineering. 29(3), pages 195-209.
4. Baluda, M., Braione, P., Denaro, G. and Pezz`e, M., 2011. Enhancing structural software coverage by incrementally computing branch executability. Software Quality Journal, 19(4), pages 725-751.

References ≫ II

5 Jiang B, Tse TH, Grieskamp W, Kicillof N, Cao Y, Li X, Chan WK. 2011. *Assuring the model evolution of protocol software specifications by regression testing process improvement*. Software: Practice and Experience. 41(10). pages 1073-1103.

6 McMinn, P. 2004. Search-based software test data generation: a survey: Research articles. *Software Testing, Verification and Reliability, 14(2)*, pages 105–156.

7 Harman M., Hu L., Hierons R., Wegener J., Sthamer H., Baresel A., and Roper M., 2004. Testability Transformation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 30(1), pages 1-14.

8 Wegener, J., Baresel, A. and Sthamer, H., 2001. Evolutionary test environment for automatic structural testing. Information and Software Technology, 43(14), pages 841-854.

9 Duran, J.W. and Ntafos, S.C., 1984. An evaluation of random testing. IEEE transactions on software engineering, (4), pages 438-444.

References ≫ III

[10] Miller, W., and Spooner, DL. 1976. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223-226,

[11] Kuhn, DR. 1999. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 8(4), pages 411-424.

[12] Ferguson, R. and Korel, B., 1996. The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology (TOSEM), 5(1), pages. 63-86.

DeMillo, R., and Offutt, J., 1993. Experimental results from an Automatic Test Case Generation. *ACM Transaction on Software Engineering Methodology*, 2(2), pages 109-175.

Ntafos, S.C., 1988. A comparison of some structural testing strategies. IEEE Transactions on software engineering, 14(6), pages 868-874.

Chilenski, J., and Miller. S. 1994 Application of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, pages 193-200.

References ≫ IV

16 Das A., and Mall R., 2013. Automatic Generation of MC/DC Test Data. *International Journal of Software Engineering, Acta Press* 2(1).

17 Majumder R., and Sen K., 2007. Hybrid Concolic Testing. *Proceedings of the 29th International Conference on Software Engineering,IEEE Computer Society,Washington, DC, USA* pages 416-426.

18 Hayhurst, KJ., Veerhusen DS., Chilenski, JJ., Rierson, LK. 2001. A Practical Tutorial on Modified Condition/Decision Coverage, NASA/TM-2001-210876.

19 Kim Y., and Kim M., 2011. SCORE: a scalable concolic testing tool for reliable embedded software. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* pages 420-423.

Kim, M., Kim, Y. and Choi, Y., 2012. Concolic testing of the multi-sector read operation for flash storage platform software. Formal Aspects of Computing, 24(3), pages 355-374.

References ≫ V

[21] Burnim J., and Sen K., 2008. Heuristics for scalable dynamic test generation. *In Proc. ASE*, pages 443-446, Washington, D.C., USA.

[22] Kim Y., Kim Y., Kim T., Lee G., Jang Y., and Kim M., 2013. Automated unit testing of large industrial embedded software using concolic testing. *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)* pages 519-528.

[23] Kim M., and Kim Y.,and Jang Y., 2012. Industrial application of concolic testing on embedded software: Case studies. *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 390-399.

Bokil, P., Darke, P., Shrotri, U., and Venkatesh, R., 2009. Automatic Test Data Generation for C Programs. *In proceedings 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement.*

References ≫ VI

[25] Kim M., Kim Y., and Rothermel G., 2012. A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation. *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 340-349.

[26] Sen K., and Agha G., 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools (Tools Paper). *DTIC Document*.

Thank You!