Software Quality Engineering

04. Control Flow & Symbolic Execution

Achiya Elyasaf

אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

SISE
הנדסת מערכות תוכנה ומידע
Software and Information Systems Engineering

Software Testing and QA Theory and Practice, Naik andTripathy

Today's Agenda

- Control Flow — analyzing possible paths
  - Basics
  - Criteria
- Symbolic Execution — finding inputs for desired paths
  - Definitions
  - Examples

Control Flow

\ Control flow testing is a structural testing strategy

\ Uses a program control flow as a model

\ White box testing

  \ the structure, design, and code of the software should be known

Control flow - basics

\ Order of statements execution

\ Assignment statements

\ Conditional statements

No conditional statements => executed line by line

```
public boolean foo(String[] strings) {
    System.out.println("Hello");
    System.out.println("World");

    return true;
}
```

Control flow - basics

\ Conditions

Small number of conditional statements

can lead to

a *complex* control flow

```
public boolean foo(String[] strings) {
    System.out.println("Hello");
    System.out.println("World");

    if(strings.length > 2) {
        System.out.println("More than 2");

        return false;
    }

    return true;
}
```

Control flow - basics

\ Function call

  \ A call to ***foo(…)*** will pass the control to ***foo***

\ Return statement

  \ Exit the function (redirect all return statements to Exit point)

\ A well-defined entry point and a well-defined exit point

\ A path (characterized by input and output)

  \ Structurally - a sequence of statements

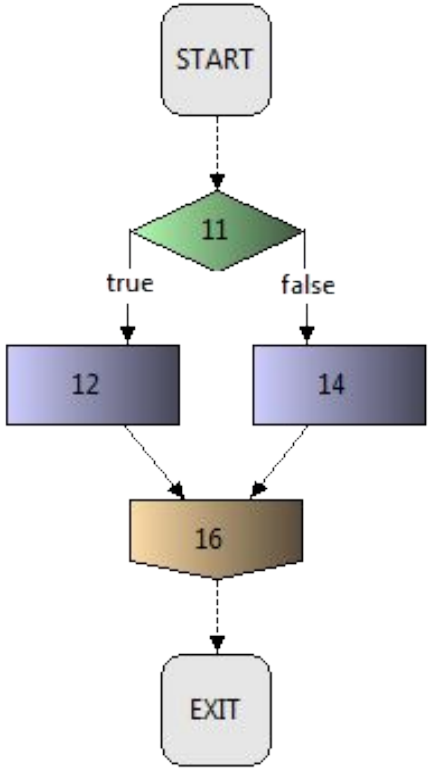
  \ Semantically - an execution instance of the unit (e.g., the path of an actual thread)

Control flow - basics

Two paths **depend on x**

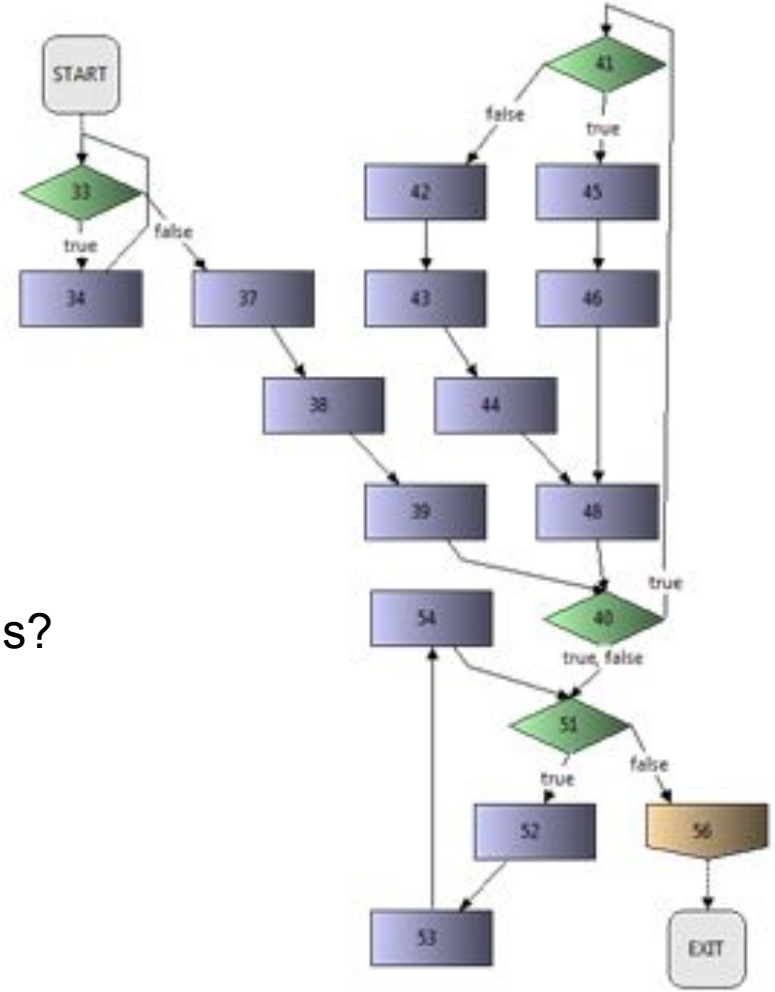
Executing as many paths as possible has **high cost** and may **not be effective** in finding defects

```
9   public static int foo(int x) {
10      int res;
11      if (x > 0) {
12          res = 1;
13      } else {
14          res = 0;
15      }
16      return res;
17  }
```



Graphs can be complicated…

How do we identify the **best** paths?

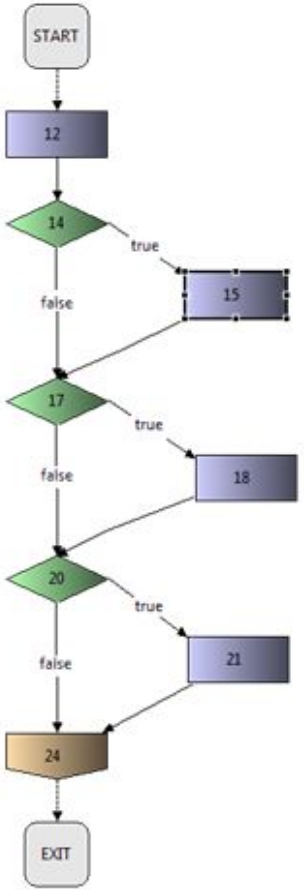


Control Flow Graph

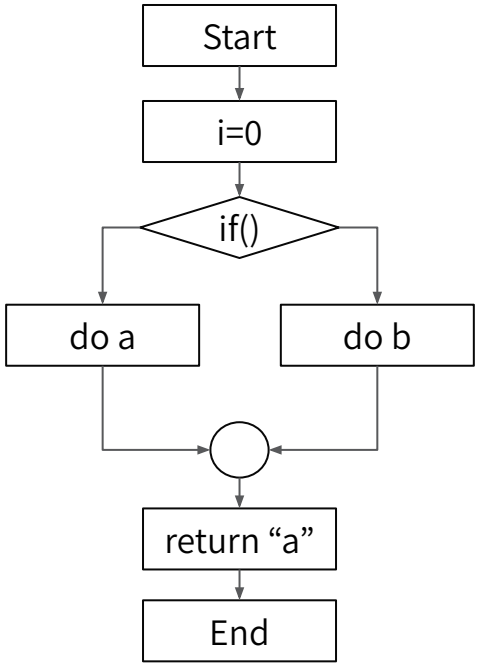
<u>First step</u>

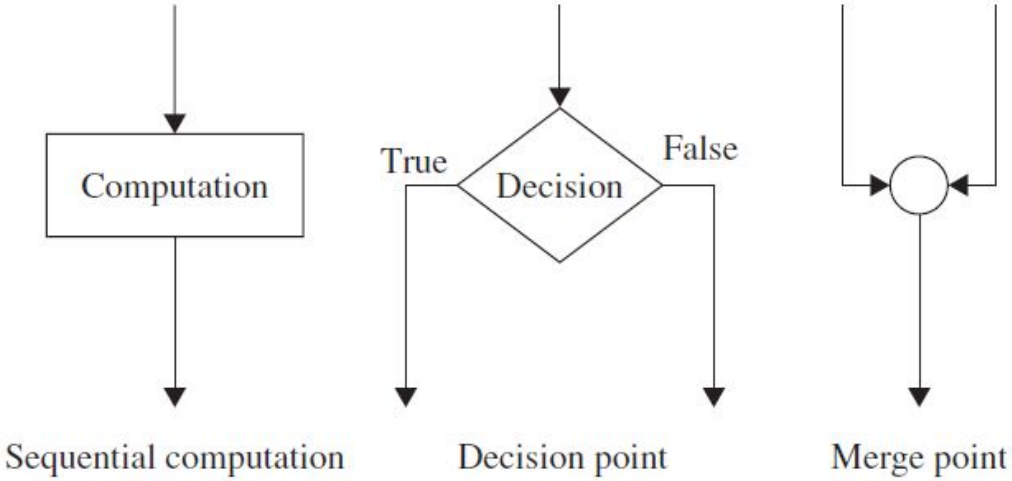
Create a CFG (Control Flow Graph)

\ A graphical representation of the statements and conditions of the program



CFG – Control Flow Graph



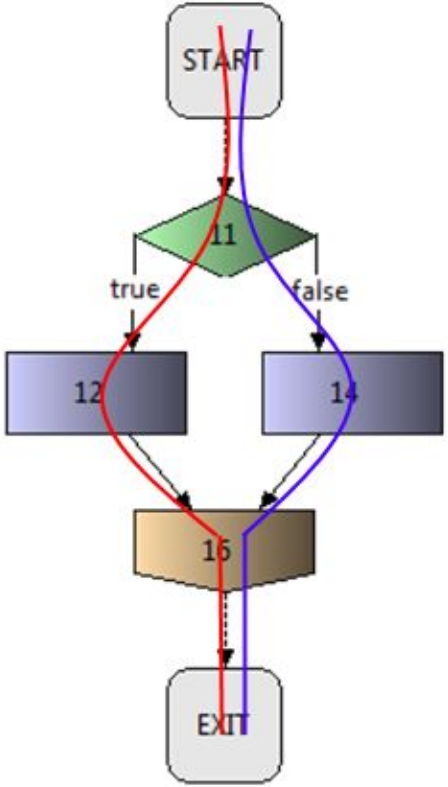


CFG – Control Flow Graph

Path examples:

- 11(T)-12-16
- 11(F)-14-16

```
9   public static int foo(int x) {
10      int res;
11      if (x > 0) {
12          res = 1;
13      } else {
14          res = 0;
15      }
16      return res;
17  }
```



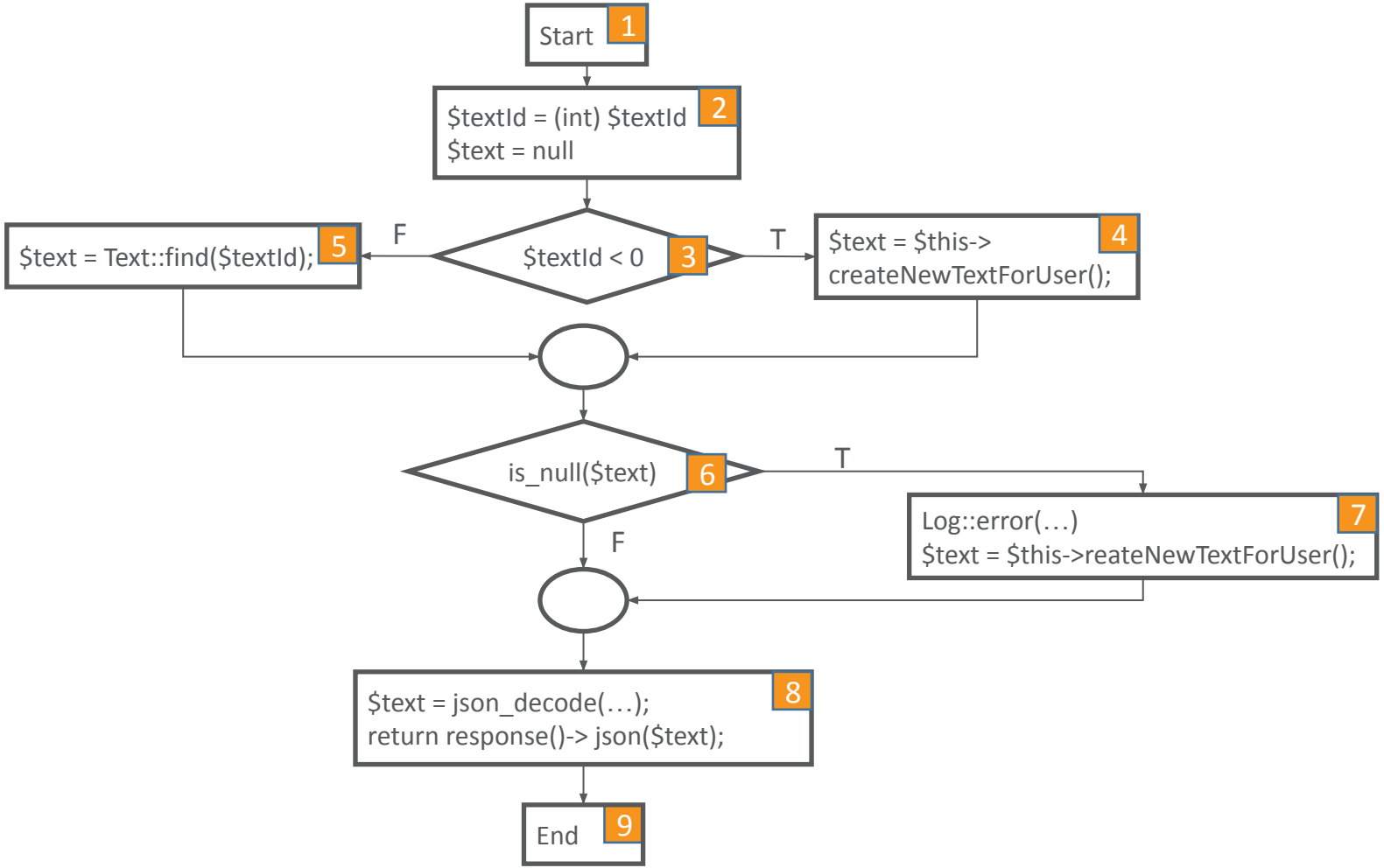
CFG – Example 1

```
public function getTextById($textId) {
    $textId = (int)$textId;
    $text = null;
    if ($textId < 0)
        $text = $this->createNewTextForUser();
    else
        $text = Text::find($textId);

    if (is_null($text)) {
        Log::error( message: "Can't find text with id " . $textId . ". Returning new default text.");
        $text = $this->createNewTextForUser();
    }

    $text = json_decode($text->text_state);
    return response()->json($text);
}
```

Language is PHP



SQ - Control Flow

CFG – Example 2

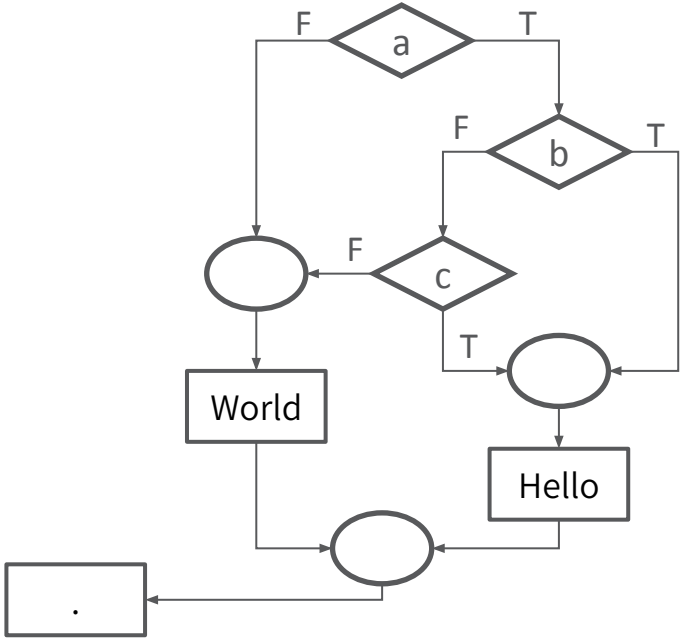
Conditions with && or || are splitted to two rhombuses.

&& : If the first condition is false — the control will not pass to the second argument.

(Similarly for ||)

For example:

```
if(a && (b || c)) {
  System.out.print("Hello");
} else {
  System.out.print("World");
}
System.out.println(".");
```

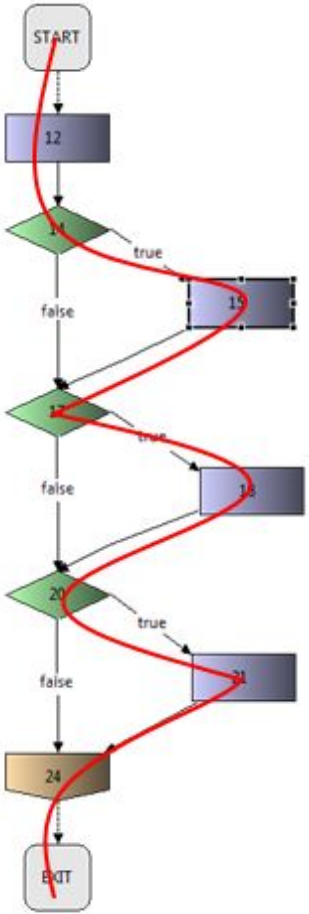


Control Flow Testing

<u>Second step - select a path</u>

Possible path select strategy:

- Every statement is executed at least once
- Not all conditions are evaluated to true and false



Infeasible Path

\ Generation of test input data: path □ input

\ **Feasible** path

\ **Infeasible** path

\ ABCEFG is an infeasible path

A ⟶ Place Order;

B ⟶ If Order Amount< 500;

C ⟶ then Print "No Discount";

D ⟶ Else Print "Applicable for 25% discount";

E ⟶ If Order Amount>1000;

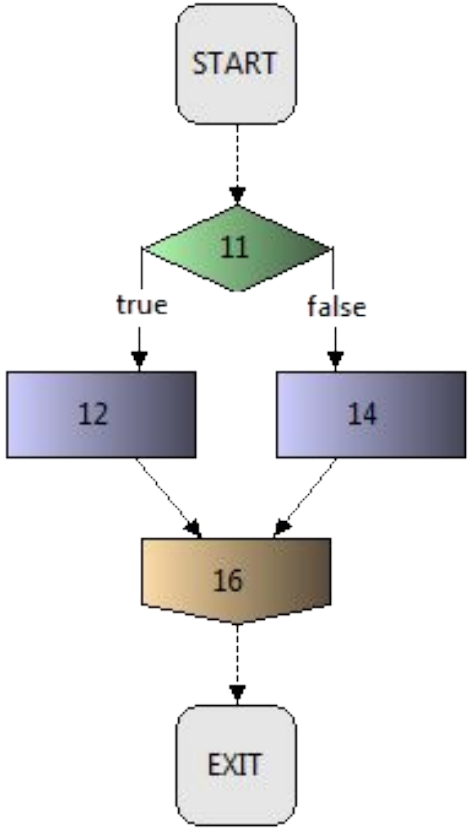
F ⟶ then Print "Applicable for 40% discount";

G ⟶ End

Control Flow Testing

What values of x will lead to which path?

```
9   public static int foo(int x) {
10      int res;
11      if (x > 0) {
12          res = 1;
13      } else {
14          res = 0;
15      }
16      return res;
17  }
```



If x <= 0 then it will be 11 14 16, otherwise the path is infeasible

… For now we don't care about feasibility/infeasibility

Path selection criteria

\ Execute all paths?

  \ Usually not feasible…

\ Select path based on criteria (defined later). Advantages:

  \ All program constructs related to the criteria are exercised at least once

  \ Know the program features that have been tested and those not tested

  \ Generate minimum input data for each selected path

Well-known path selection criteria

\ Select **all** paths

\ Select paths to achieve complete **statement** coverage

\ Select paths to achieve complete **branch** coverage

\ Select paths to achieve **predicate** coverage

All-Path Coverage Criterion

Number of possible paths ?

```
private static File a;
private static File b;
private static File c;

private static int openfiles() {
    int i = 0;

    if((a = new File( pathname: "file1")).exists()) {
        i++;
    }
    if((b = new File( pathname: "file2")).exists()) {
        i++;
    }
    if((c = new File( pathname: "file3")).exists()) {
        i++;
    }

    return i;
}
```

All-Path Coverage Criterion

Existence of file1	Existence of file2	Existence of file3
No	No	No
No	No	Yes
No	Yes	No
No	Yes	Yes
Yes	No	No
Yes	No	Yes
Yes	Yes	No
Yes	Yes	Yes

All-Path Coverage Criterion

Existence of file1	Existence of file2	Existence of file3
No	No	No
Yes	Yes	No
Yes	Yes	Yes

Small number of conditional statements
can lead to
a *complex* control flow

Statement Coverage Criterion

\ Assignments and conditions

\ Executing individual statements and observing the results

\ 100% statement coverage (executed at least once)

\ Complete statement coverage – good enough?

Weakest coverage criterion (minimum required)

Statement Coverage Criterion

Since every path covers many nodes,

only few paths can cover all the nodes

How do we select few feasible paths for full coverage?

```
private static File a;
private static File b;
private static File c;

private static int openfiles() {
    int i = 0;

    if((a = new File( pathname: "file1")).exists()) {
        i++;
    }
    if((b = new File( pathname: "file2")).exists()) {
        i++;
    }
    if((c = new File( pathname: "file3")).exists()) {
        i++;
    }

    return i;
}
```

Statement Coverage Criterion

\ Select short paths

\ Then continuously choose an increasingly potential longer length

\ Select arbitrary complex paths

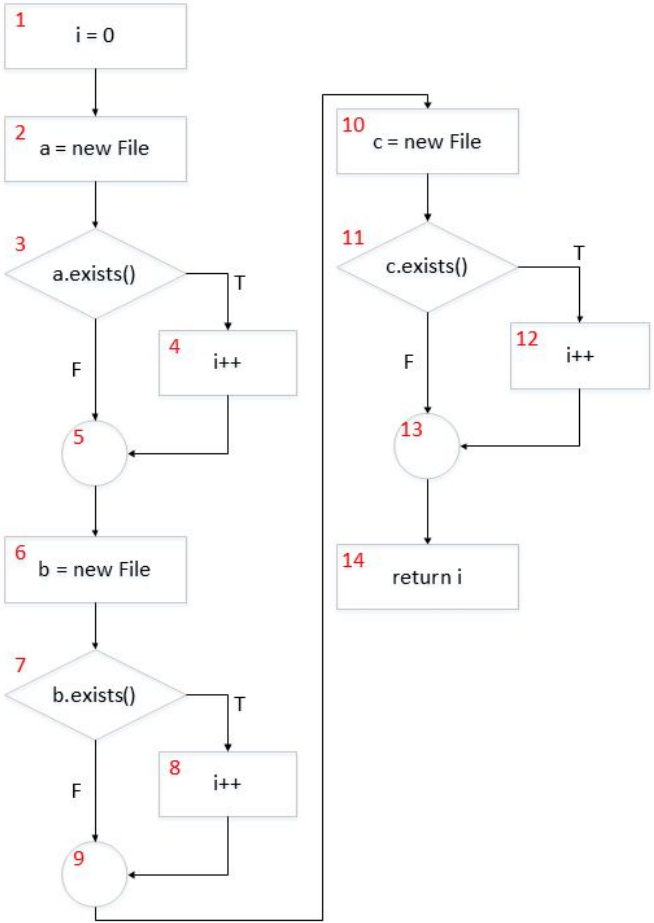
For example:

Start with 1-2-3-5-6-7-8-10-11-13-14

Then, add branches that keep the other statements

The minimal set of paths that covers statement coverage criterion:

\ 1-2-3-4-5-6-7-8-9-10-11-12-13-14

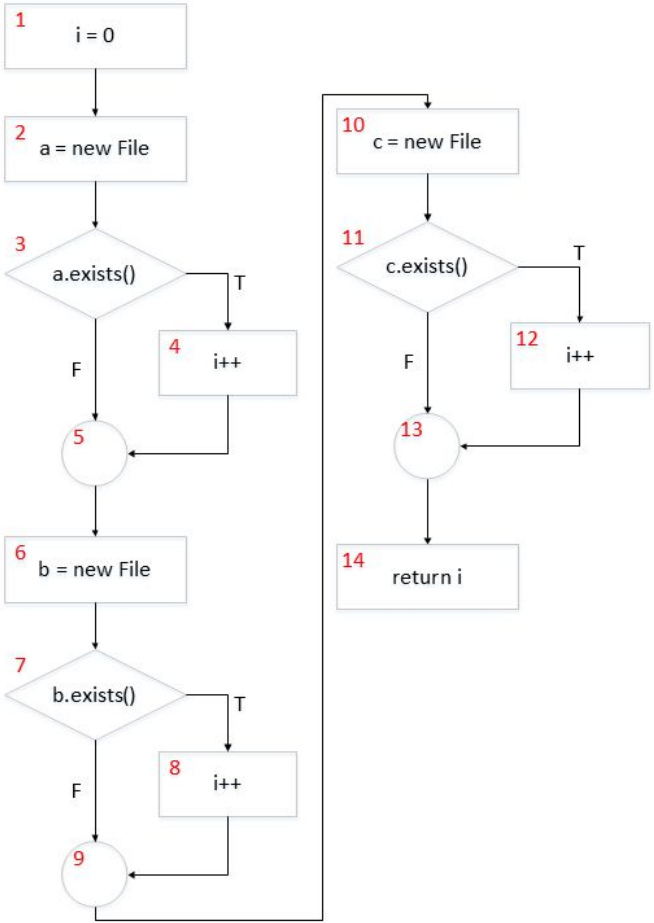


Branch Coverage Criterion

\ Branch □ an outgoing edge from a node

\ Covering a branch at least once □ 100% branch coverage

\ 1-2-3-4-5-6-7-8-9-10-11-12-13-14 □ missing branches



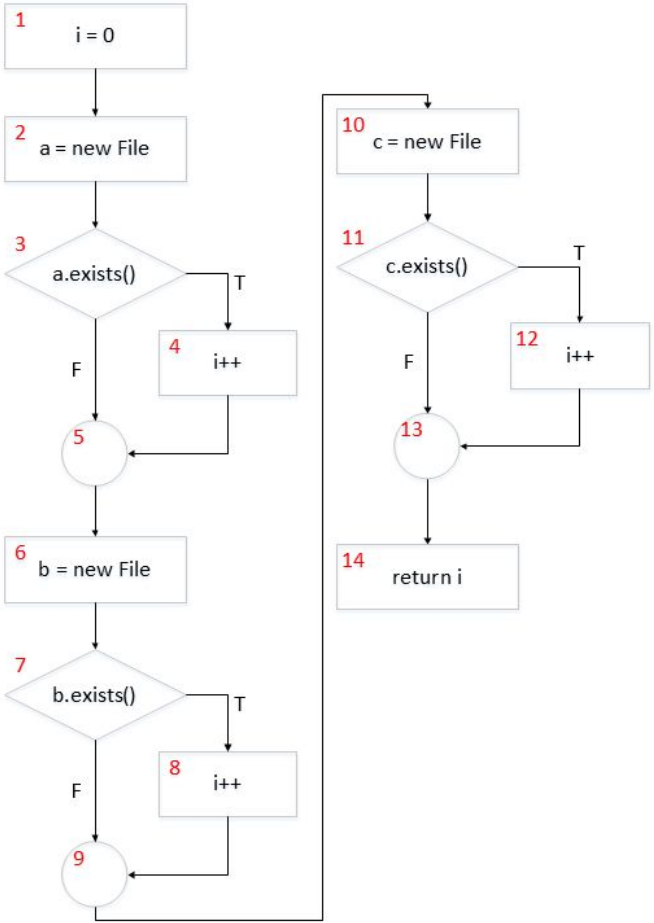
Branch Coverage Criterion

What paths should we select?

- **1-2-3-4-…**
- **1-2-3-5-…**
- **…-7-8-…**
- **…-7-9-…**
- **…-11-12-…**
- **…-11-13-…**

The minimal set of paths that covers statement coverage criterion:

- 1-2-3-4-5-6-7-8-9-10-11-12-13-14
- 1-2-3-5-6-7-9-10-11-13-14

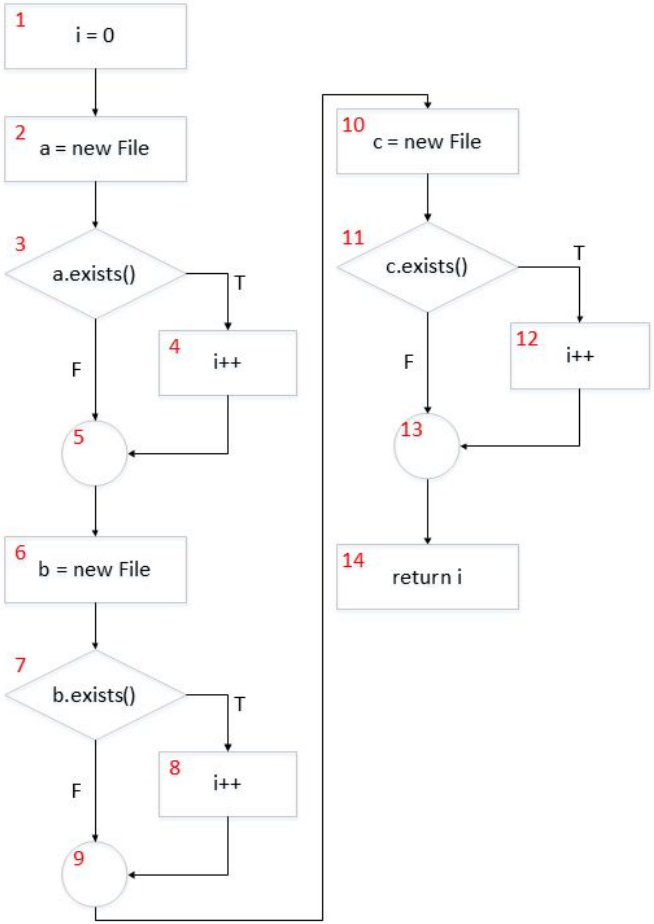


Branch Coverage Criterion

\ what paths should we select?

- \ **1-2-3-4-…**
- \ **1-2-3-5-…**
- \ **…-7-8-…**
- \ **…-7-9-…**
- \ **…-11-12-…**
- \ **…-11-13-…**

\ Problems?



Branch Coverage Criterion

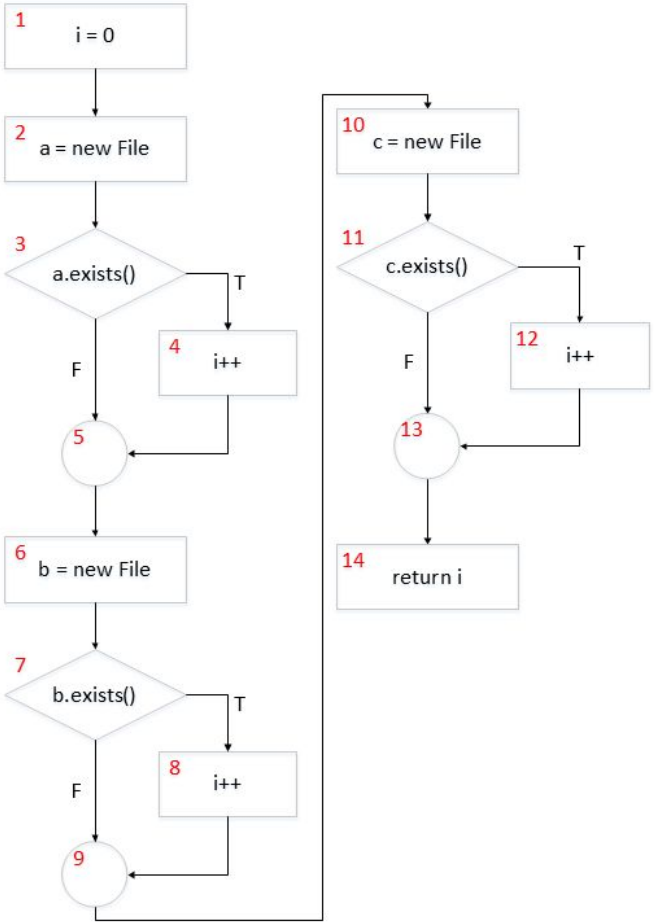
\ what paths should we select?

- \ **1-2-3-4-…**
- \ **1-2-3-5-…**
- \ **1-2-…-7-8-…**
- \ **1-2-…-7-9-…**
- \ **1-2-…-11-12-…**
- \ **1-2-…-11-13-…**

\ Problems?

- \ Covers only branches and not all combinations.

  What if the bug is in a specific combination?



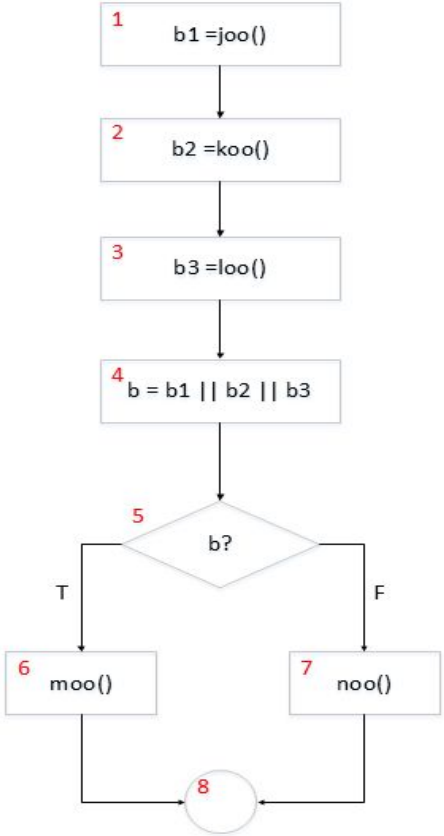
Predicate Coverage Criterion

Take a look at this code…

```
private static boolean b;
private static boolean b1;
private static boolean b2;
private static boolean b3;

private static void foo() {
    b1 = joo();
    b2 = koo();
    b3 = loo();

    b = b1 || b2 || b3;
    if (b) {
        moo();
    } else {
        noo();
    }
}
```



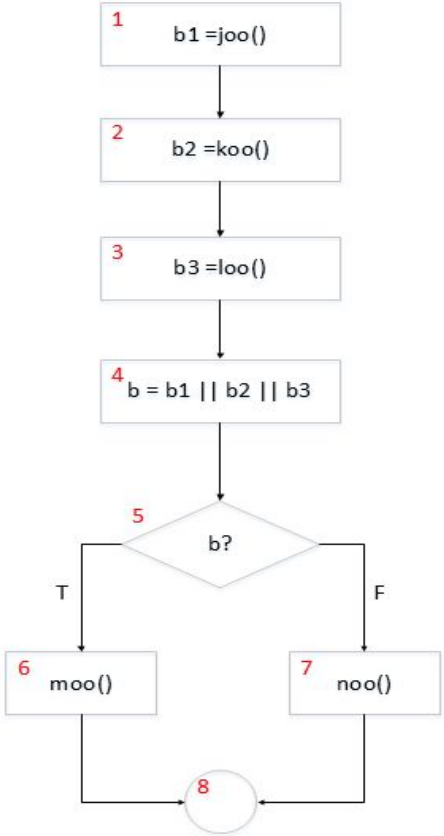
Predicate Coverage Criterion

\ How many test cases do we need to get full **statement** coverage?

\ How many test cases do we need to get full **branch** coverage?

\ What did we miss?

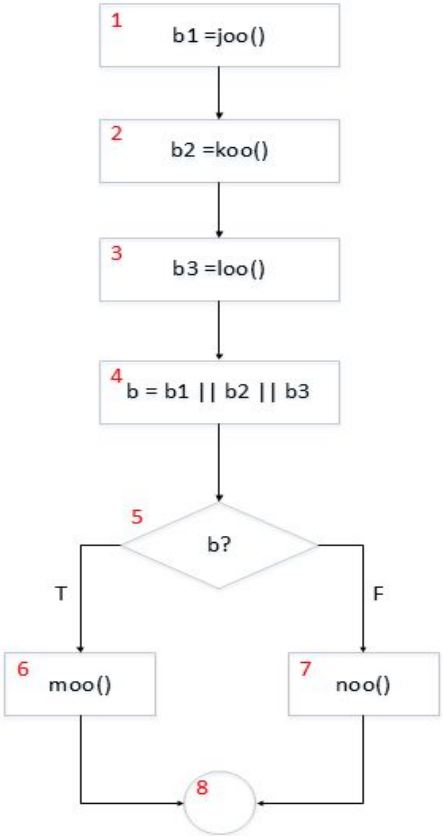
Test case	b1	b2	b3	b
1	T	F	F	T
2	F	F	F	F



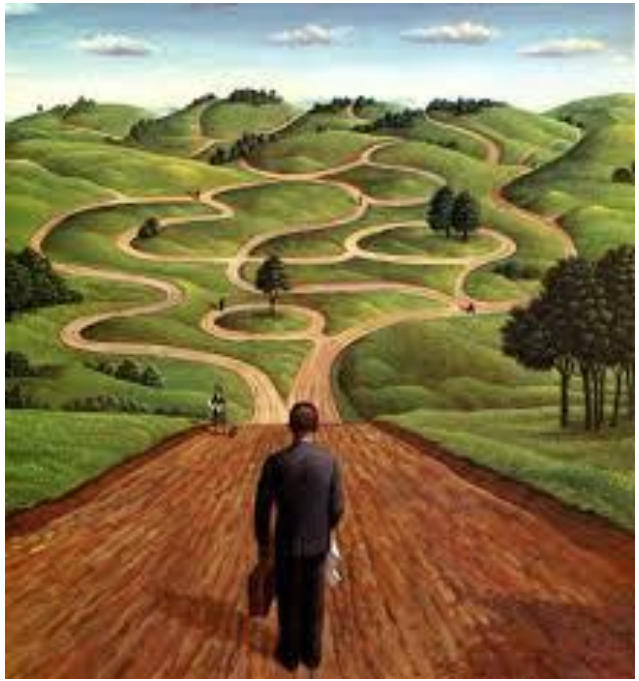
Predicate Coverage Criterion

\ Predicate coverage = all predicate options
\ (not the same as all possible paths)
\ To get full predicate coverage:

Test case	b1	b2	b3	b
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	T
5	F	T	T	T
6	F	T	F	T
7	F	F	T	T
8	F	F	F	F



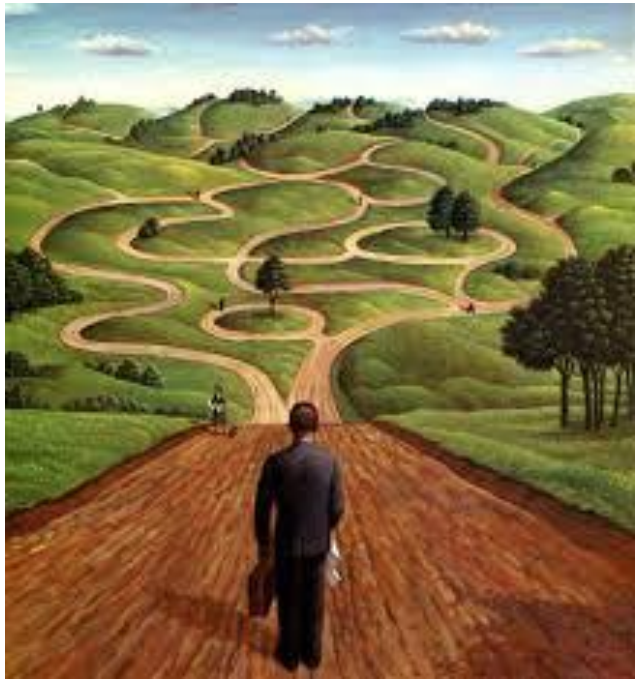
Test Input Generation



After selecting the paths, we need to select input values.

How to select input values in a way that the selected path is executed when we execute the program?

Test Input Generation



Symbolic execution:

Analyze how the function's input vector affects the paths and use it to find the desired inputs

Symbolic execution - terminology

\ Input vector

- \ Function input
- \ Globals and constants
- \ Files (or contents)
- \ Network connections

\ What is the input vector in this function?

```
private static File a;
private static File b;
private static File c;

private static int openfiles() {
    int i = 0;

    if((a = new File( pathname: "file1")).exists()) {
        i++;
    }
    if((b = new File( pathname: "file2")).exists()) {
        i++;
    }
    if((c = new File( pathname: "file3")).exists()) {
        i++;
    }

    return i;
}
```

Symbolic execution - terminology

\ Input vector

\ Function input

\ Globals and constants

\ Files (or contents)

\ Network connections

\ What is the input vector in this function?

Input vector: all data entities read by the function whose values must be fixed prior to entering

```
private static File a;
private static File b;
private static File c;

private static int openfiles() {
    int i = 0;

    if((a = new File( pathname: "file1")).exists()) {
        i++;
    }
    if((b = new File( pathname: "file2")).exists()) {
        i++;
    }
    if((c = new File( pathname: "file3")).exists()) {
        i++;
    }

    return i;
}
```

Symbolic execution - terminology

\ **Predicate**

\ a logical function evaluated at a decision point (e.g., exists())

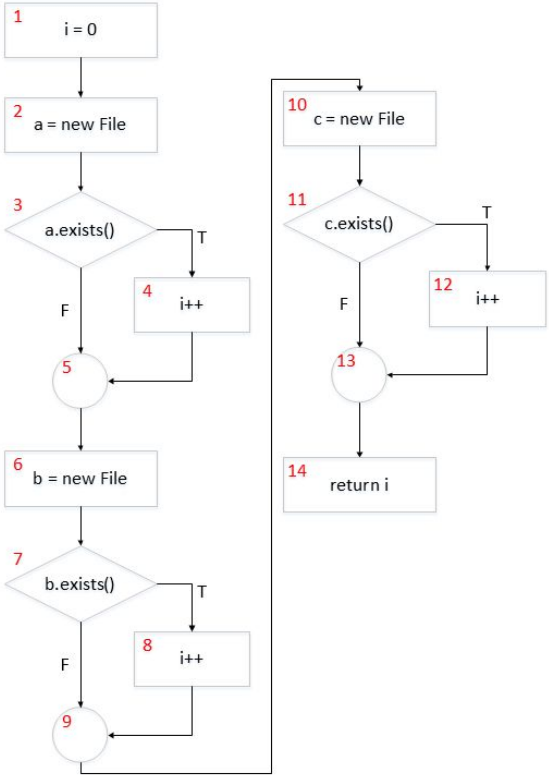
\ **Path predicate**

\ the set of predicates associated with a path

\ Example - for path 1-2-3(F)-5-6-7(T)-8-9-10-11(F)-13

the *path predicate* is composed of elements of the input vector:

a.exists() ≡ F, b.exists() ≡ T, c.exists() ≡ F



Symbolic execution - terminology

Predicate Interpretation

input values <a, b, c> and a vector of local variables <i>

\ Local variables are not visible outside a function but are used to:

- \ hold intermediate results
- \ point to array elements
- \ control loop iterations

Local variables play no roles in selecting inputs that force the paths to execute!

Symbolic execution - terminology

symbolic substitution

Input vector: input **<x1, x2>**, the local variable **<y>** and the constants **<0, 7>**

```
public static int SymSub(int x1, int x2){
int y;
y = x2 + 7;
if (x1 + y >= 0)
     return (x2 + y);
else return (x2 - y);
}
```

x1 + y ≥ 0 □ **x1 + x2 + 7 ≥ 0** (symbolically substituting **y** with **x2 +7**)

terms of the input vector **<x1,x2>** and the constant vector **<0,7>**

Symbolic execution - terminology

\ **Predicate interpretation** □ symbolically substituting operations to express the predicates solely in terms of the input vector

\ **Path predicate expression**: an interpreted path predicate

- \ It is void of local variables and is solely composed of elements of the input vector
- \ Path forcing input values can be generated by solving the set of constraints in a path predicate expression

Symbolic execution - terminology

\ If the set of constraints cannot be solved □ **infeasible path** □ consider other paths in an effort to meet a chosen path selection criterion

\ **Generating Input Data from Path Predicate Expression**:

- \ We must solve the corresponding path predicate expression in order to generate input data which can force a program to execute a selected path
- \ Is it not easy…

Symbolic execution tools

\ Stanford's KLEE:

\ – http://klee.llvm.org/

\ NASA's Java PathFinder:

\ – http://javapathfinder.sourceforge.net/

\ JaCoCo — Java Code Coverage Library

\ Microsoft Research's SAFE

\ UC Berkeley's CUTE

\ EPFL's S2E

Symbolic execution

Symbolic execution is widely used in practice. Tools based on symbolic execution have found serious errors and security vulnerabilities in various systems:

- Network servers
- File systems
- Device drivers
- Unix utilities
- Computer vision code
- …

Symbolic execution

At any point during program execution, symbolic execution keeps two formulas:

symbolic store and a **path constraint**

Therefore, at any point in time, the symbolic state is described as the conjunction of these two formulas.

Symbolic Store

The values of variables at any moment in time are given by a function

Var is the set of variables

Sym is a set of symbolic values (Var → Sym)

σ_s is called a **symbolic store**

Example: σ_s: x→x0, y→y0

Semantics

- Arithmetic expression evaluation simply manipulates the symbolic values.
- Let σ_S: x → x0, y → y0
- Then, z = x + y will produce the symbolic store: x → x0, y → y0, z → x0+y0
- That is, we literally keep the symbolic expression x0+y0

Path Constraint

\ The analysis keeps a path constraint (pct) □ a formula

\ The formula is typically in a decidable logical fragment without quantifiers

\ At the start of the analysis, the path constraint is true

\ Evaluation of conditionals affects the path constraint, but not the symbolic store

Path Constraint Progress

Let σ_s : `x` $\mapsto$ `x0`, `y` $\mapsto$ `y0`

Let pct = `x0 > 10`

Lets evaluate: `if (x > y + 1) {5: … }`

At label 5, we will get the symbolic store σ_s . It does not change. But we will get an updated path constraint:

pct = `x0 > 10` $\wedge$ `x0 > y0 + 1`

Symbolic Execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Can you find the inputs that make the program reach the ERROR?

Lets execute this example with classic symbolic execution

Symbolic Execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

The read() functions read a value from the input and because we don't know what those read values are, we set the values of `x` and `y` to fresh symbolic values called `x0` and `y0`

pct is true because so far we have not executed any conditionals

σ_s : $x \mapsto x0$, $y \mapsto y0$

pct : true

Symbolic Execution - Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : true

Here, we simply executed the function twice() and added the new symbolic value for z.

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

We forked the analysis into 2 paths: the true and the false path. So we **duplicate** the state of the analysis.

This is the result if x = z:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0

This is the result if x != z:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 ≠ 2*y0

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

We can avoid further exploring a path if we know the constraint pct is **unsatisfiable**. In this example, both pct's are satisfiable so we need to keep exploring both paths.

This is the result if x = z:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0

This is the result if x != z:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 ≠ 2*y0

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Lets explore the path when x == z is true.
Once again we get 2 more paths.

This is the result if x > y + 10:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0
∧
x0 > y0+10

This is the result if x ≤ y + 10:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0
∧
x0 ≤ y0+10

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

So the following path reaches "**ERROR**".

This is the result if x > y + 10:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0
∧
x0 > y0+10

We can now ask the SMT solver for a satisfying assignment to the pct formula.

For instance, `x0 = 40, y0 = 20` is a satisfying assignment. That is, running the program with those concrete inputs triggers the error.

38