CS 569
Selected Topics in Software Engineering: Program Analysis & Evaluation

Symbolic and Concolic Testing

Oregon State University, Winter 2024

DEMO for Paper Presentation

Automatically Generating Precise Oracles from Structured Natural Language Specifications

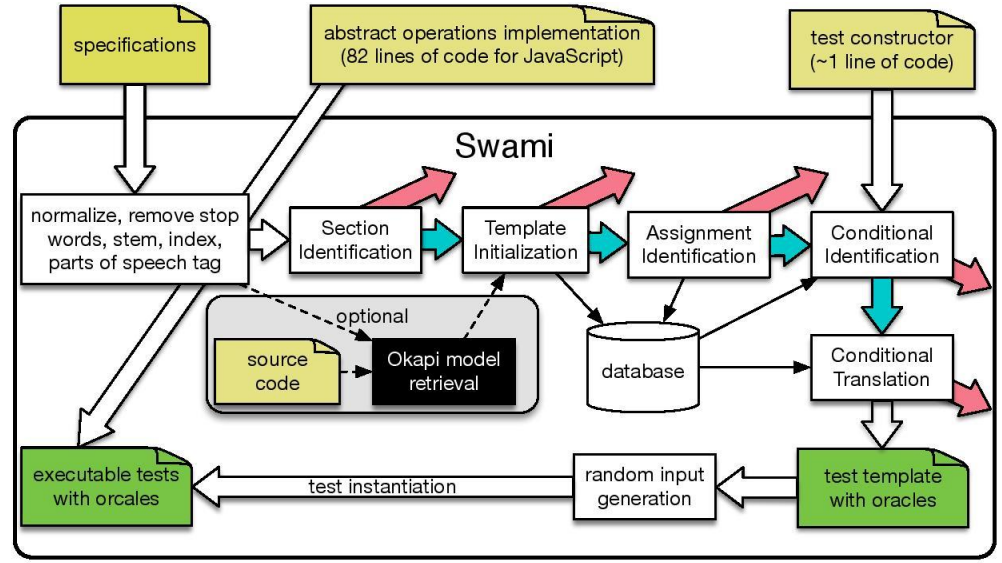
Manish Motwani and Yuriy Brun, ICSE' 2019

Practice your talk by recording your presentation
See: https://www.youtube.com/watch?v=GBwrvtrg1Ug

Swami

Structured Informal specification

- encode testable behavior
- Abstract Operations
- Implicit Operations
- Oracles embedded in Conditionals
- Assignments using local variables
- Ambiguous and deprecated

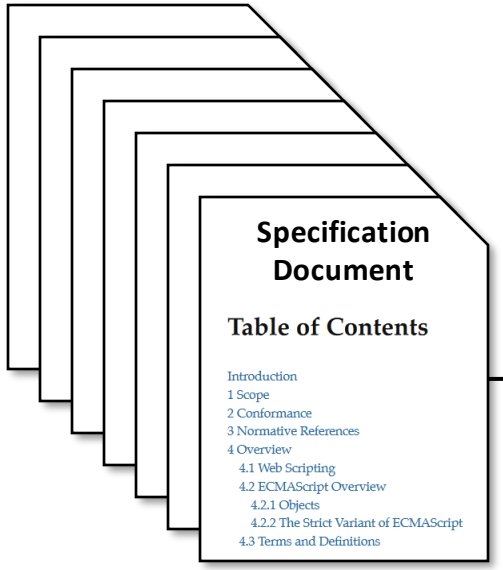


Automatically generate executable tests (inputs with oracles) for **Exceptions and Boundary conditions** from structured informal specifications

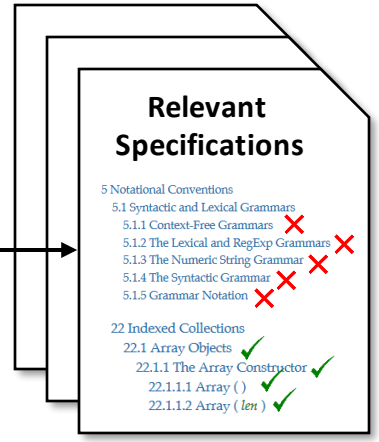
Executable Test

- Test inputs
- Test oracles

Step1: Identify specifications which encode testable behavior

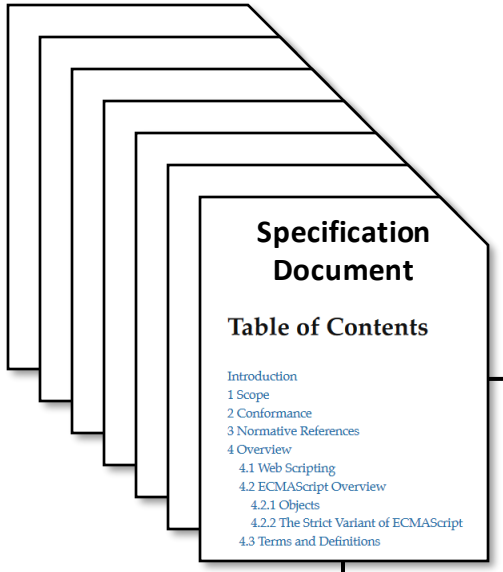


Rule-based approach



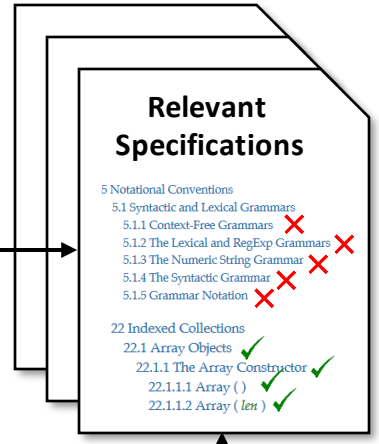
Rules are regular expressions composed of POS tags, keywords, and wild card characters

Step1: Identify specifications which encode testable behavior

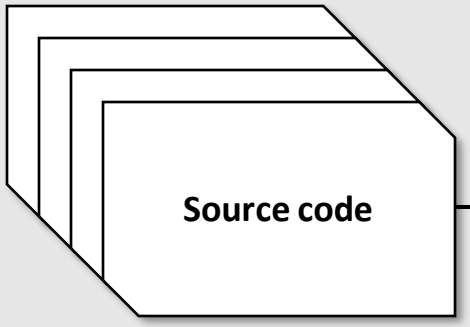


Rule-based approach

Heading RE: [CD new* NN LRB NN.* RRB]
Body RE: [If .* return .*] [if .* throw .* exception]



when the format of specification document is unknown

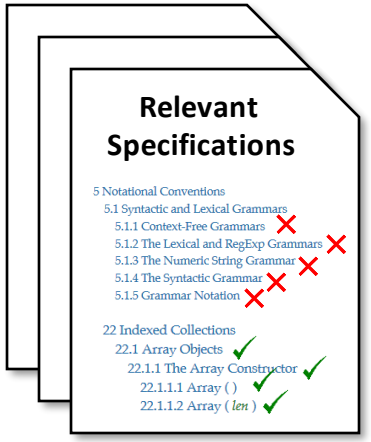


Information Retrieval-based approach

OKAPI model

section ID	matched Java class	similarity score
15.4.2.2	ScriptRuntime.java	0.37
15.4.2.2	Interpreter.java	0.31
15.4.2.2	BaseFunction.java	0.25
15.4.2.2	ScriptableObject.java	0.24
15.4.2.2	NativeArray.java	0.21

Example specification encoding testable behavior



21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Header RE:
CD new* NN LRB NN.* RRB

Body RE:
If .* throw .* exception

Body RE:
If .* return .*

Step2: Extract **method signature** from specification heading and initialize Test Template

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)

```
function test_< method name >(thisObj,<[ method args ]>) {}
```

```
function test_string_prototype_startswith(thisObj,searchString,position) {}
```

```
new String(thisObj).startsWith(searchString, position);
```

12. Otherwise, return **false**.

Initialized Test Template

Method invocation code

Step3: Identify and parse **Assignments** to store the local variables and their values

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Variable	Value
O	RequireObjectCoercible(this value)
S	ToString(O)
isRegExp	IsRegExp(searchString)
searchStr	ToString(searchString)
pos	ToInteger(position)
len	length of S
start	min(max(pos,0),len)
searchLength	length of searchStr

Step4: Identify and parse **Conditionals** to populate the conditional templates

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Step4: Identify and parse **Conditionals** to populate the conditional templates

Exception

```
if (<condition>) {
 try {
    var output = <method invocation>;
    return;
 }catch(e){
    <test constructor>(true, (e instance of <expected error>));
    return;
 }
}
```

Boundary Condition

```
if (<condition>) {
     var output = <method invocation>;
     <test constructor>(output, <expected output>);
     return;
}
```

3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Step4: Identify and parse **Conditionals** to populate the conditional templates

Exception

```
if (<condition>) {
 try {
    var output = <method invocation>;
    return;
 }catch(e){
    <test constructor>(true, (e instance of <expected error>));
    return;
 }
}
```

Boundary Condition

```
if (<condition>) {
     var output = <method invocation>;
     <test constructor>(output, <expected output>);
     return;
}
```

4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of length *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Exception oracle

Boundary condition oracle

Step4: Identify and parse **Conditionals** to populate the conditional templates

```
if (<condition>) {
 try {
    var output = <method invocation>;
    return;
 }catch(e){
    <test constructor>(true, (e instance of <expected error>));
    return;
 }
}
```

Exception

From step2

Input by developer

4. If *isRegExp* is **true**, throw a **TypeError** exception.

Exception oracle

```
if (isRegExp is true){
    try{
      var output = new String(thisObj).startsWith(searchString, position);
      return;
    }catch(e){
      assert.StrictEqual(true,(e instanceof TypeError));
      return;
    }
 }
```

Step4: Identify and parse **Conditionals** to populate the conditional templates

Boundary Condition

```
if (<condition>) {
    var output = <method invocation>;
    <test constructor>(output, <expected output>);
    return;
}
```

21.1.3.20 String.prototype.startsWith (*searchString* [, *po*

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefined**, this step produces the value 0.)
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *sta* ength *searchLength* is the same as the full element sequence of *searchStr*, return **true**.
12. Otherwise, return **false**.

Boundary condition oracle

```
if (searchLength+start is greater than len){
    var output = new String(thisObj).startsWith(searchString, position);
    assert.strictEqual(output, false);
    return;
 }
```

Step5: Recursively substitute **local variables** and **implicit operations**

```
if (isRegExp is true){
   try{
     var output = new String(thisObj).startsWith(searchString, position);
     return;
   }catch(e){
     assert.StrictEqual(true,(e instanceof TypeError));
     return;
   }
 }
```

```
  if (searchLength+start is greater than len){
      var output = new String(thisObj).startsWith(searchString, position);
      assert.strictEqual(output, false);
      return;
   }
```

Variable	Value
O	RequireObjectCoercible(this value)
S	ToString(O)
isRegExp	IsRegExp(searchString)
searchStr	ToString(searchString)
pos	ToInteger(position)
len	length of S
start	min(max(pos,0),len)
searchLength	length of searchStr

Method Arguments:

thisObj
searchString
position

Step5: Recursively substitute **local variables** and **implicit operations**

```
if (IsRegExp(searchString) === true){
   try{
     var output = new String(thisObj).startsWith(searchString, position);
     return;
   }catch(e){
     assert.StrictEqual(true,(e instanceof TypeError));
     return;
   }
 }
```

```
  if (ToString(searchString).length +
       Math.min(Math.max(ToInteger(position), 0),
       ToString(RequireObjectCoercible(thisObj)).length) >
       ToString(RequireObjectCoercible(thisObj)).length){
      var output = new String(thisObj).startsWith(searchString, position);
      assert.strictEqual(output, false);
      return;
   }
```

Variable	Value
O	RequireObjectCoercible(this value)
S	ToString(O)
isRegExp	IsRegExp(searchString)
searchStr	ToString(searchString)
pos	ToInteger(position)
len	length of S
start	min(max(pos,0),len)
searchLength	length of searchStr

Method Arguments:

thisObj
searchString
position

Step6: Add conditionals to the initialized test template and check if it compiles

```
function test_string_prototype_startswith(thisObj,searchString,position) {
    if (IsRegExp(searchString) === true){
       try{
         var output = new String(thisObj).startsWith(searchString, position);
         return;
       }catch(e){
         assert.StrictEqual(true,(e instanceof TypeError));
         return;
       }
     }

}
```

Implement **Abstract Operations** (100 lines JS code)

```
function IsRegExp(argument){
      return (argument instanceof RegExp);
}
…
```

```
function test_string_prototype_startswith(thisObj,searchString,position) {
   if (IsRegExp(searchString) === true){
      try{
        var output = new String(thisObj).startsWith(searchString, position);
        return;
      }catch(e){
        assert.StrictEqual(true,(e instanceof TypeError));
        return;
      }
    }
}
```

Implement **Abstract Operations** (100 lines JS code)

```
function IsRegExp(argument){
      return (argument instanceof RegExp);
}
…
```

Abstract Operations

```
function test_string_prototype_startswith(thisObj,searchString,position) {
   if (IsRegExp(searchString) === true){
      try{
        var output = new String(thisObj).startsWith(searchString, position);
        return;
      }catch(e){
        assert.StrictEqual(true,(e instanceof TypeError));
        return;
      }
    }

     if (ToString(searchString).length +
          Math.min(Math.max(ToInteger(position), 0),
          ToString(RequireObjectCoercible(thisObj)).length) >
          ToString(RequireObjectCoercible(thisObj)).length){
        var output = new String(thisObj).startsWith(searchString, position);
        assert.strictEqual(output, false);
        return;
      }

}
```

Test Template encoding Oracles

Step7: Instantiating test template by generating test inputs using random input generation

```
function IsRegExp(argument){
      return (argument instanceof RegExp);
}
…
```

Abstract Operations

```
function test_string_prototype_startswith(thisObj,searchString,position) {
   if (IsRegExp(searchString) === true){
      try{
        var output = new String(thisObj).startsWith(searchString, position);
        return;
      }catch(e){
        assert.StrictEqual(true,(e instanceof TypeError));
        return;
      }
    }

     if (ToString(searchString).length +
          Math.min(Math.max(ToInteger(position), 0),
          ToString(RequireObjectCoercible(thisObj)).length) >
          ToString(RequireObjectCoercible(thisObj)).length){
        var output = new String(thisObj).startsWith(searchString, position);
        assert.strictEqual(output, false);
        return;
      }

}
```

Test Template encoding Oracles

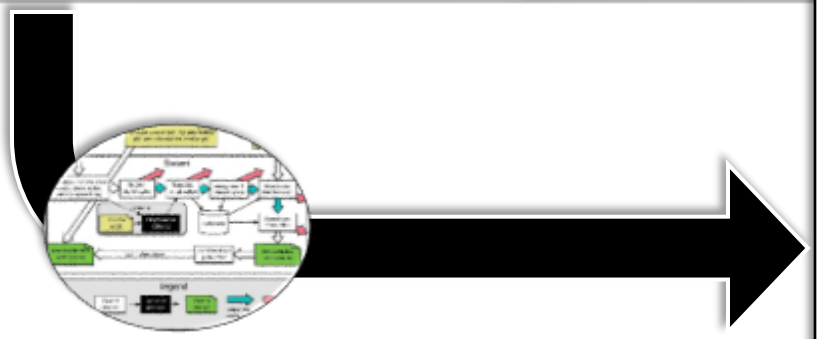
- **Total number of inputs**: 3
- **Heuristic**: String method => *thisObj* should be a valid string
- **Number of test inputs to be generated**: 1000

```
test_string_prototype_startswith("Y3I9", "E0RS6GU078", 894);
test_string_prototype_startswith("T82LL6", 572, false);
test_string_prototype_startswith("XU6W0", "J3A", Infinity);
test_string_prototype_startswith("W5E74X0R", null, NaN);
...
```

21.1.3.20 String.prototype.startsWith (*searchString* [, *position*])

The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *isRegExp* be ? IsRegExp(*searchString*).
4. If *isRegExp* is **true**, throw a **TypeError** exception.
5. Let *searchStr* be ? ToString(*searchString*).
6. Let *pos* be ? ToInteger(*position*). (If *position* is **undefin**
7. Let *len* be the length of *S*.
8. Let *start* be min(max(*pos*, 0), *len*).
9. Let *searchLength* be the length of *searchStr*.
10. If *searchLength*+*start* is greater than *len*, return **false**.
11. If the sequence of elements of *S* starting at *start* of len
    searchStr, return **true**.
12. Otherwise, return **false**.



Swami

Executable Test with Oracles

```
function IsRegExp(argument){
      return (argument instanceof RegExp);
}
```

Abstract Operations

```
function test_string_prototype_startswith(thisObj,searchString,position) {
   if (IsRegExp(searchString) === true){
      try{
        var output = new String(thisObj).startsWith(searchString, position);
        return;
      }catch(e){
        assert.StrictEqual(true,(e instanceof TypeError));
        return;
      }
    }
     if (ToString(searchString).length +
          Math.min(Math.max(ToInteger(position), 0),
          ToString(RequireObjectCoercible(thisObj)).length) >
          ToString(RequireObjectCoercible(thisObj)).length){
         var output = new String(thisObj).startsWith(searchString, position);
         assert.strictEqual(output, false);
         return;
      }
}
```

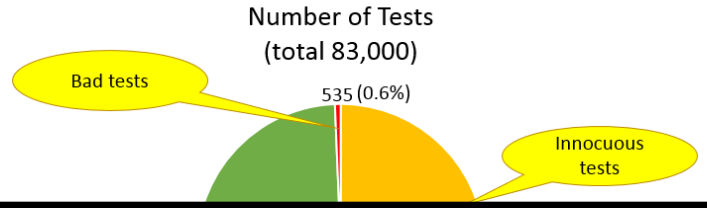
Test Template encoding Oracles

```
test_string_prototype_startswith("Y3I9", "E0RS6GU078", 894);
test_string_prototype_startswith("T82LL6", 572, false);
test_string_prototype_startswith("XU6W0", "J3A", Infinity);
test_string_prototype_startswith("W5E74X0R", null, NaN);
...
```

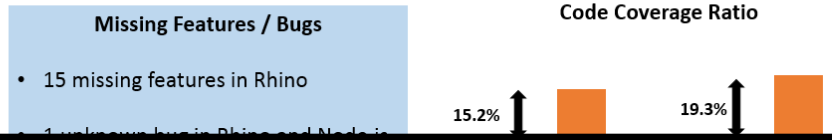
Test Inputs

Evaluation: A reminder

Swami-generated tests are precise to the specification



Swami covers more code and identifies features and bugs missed by developer-written tests



On JavaScript specification, Swami

- is 98.4% precise to the specification,
- improves developer-written and generated test suites, and
- Identifies 1 new bug and 15 missing features in Rhino,
  1 new bug in Node.js, and
  18 semantic ambiguities in the spec

Swami tests complement EvoSuite-written tests

Swami tests complement EvoSuite-written tests

QUIZ: Randoop, Korat, and Swami

Which of the following statements are true for each test generation technique:

	Randoop	Korat	Swami
Uses type information to guide test generation.	✓	✓	✓
Each test is generated independently of the past test.			✓
Generates tests deterministically.		✓	
Suited to test method sequences.	✓		
Avoids generating redundant tests.	✓	✓	

Symbolic Execution

- Reasoning about behavior of program by "executing" it using symbolic values
- Originally proposed by James King (1976, CASM) and Lori Clarke (1976, IEEE TSE)
- Became practical around 2005 because of advances in constraint solving (SMT solvers)

Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
            y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```

Concrete Execution

Suppose a = b = c = 1

x = y = z = 0

If(a) = true

x = -2

If (b > 5) = false

assert(-2 + 0 + 0 != 3) = true

Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
            y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```

Symbolic values

Symbolic Execution

$a = a_0$ $b = b_0$ $c = c_0$

Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
            y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```

Symbolic values

Symbolic Execution

$a = a_0 \quad b = b_0 \quad c = c_0$

$x = y = z = 0$

Example

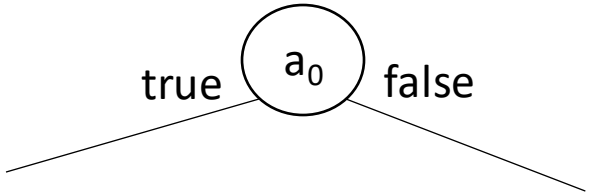
```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
           y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```

Symbolic values

Symbolic Execution

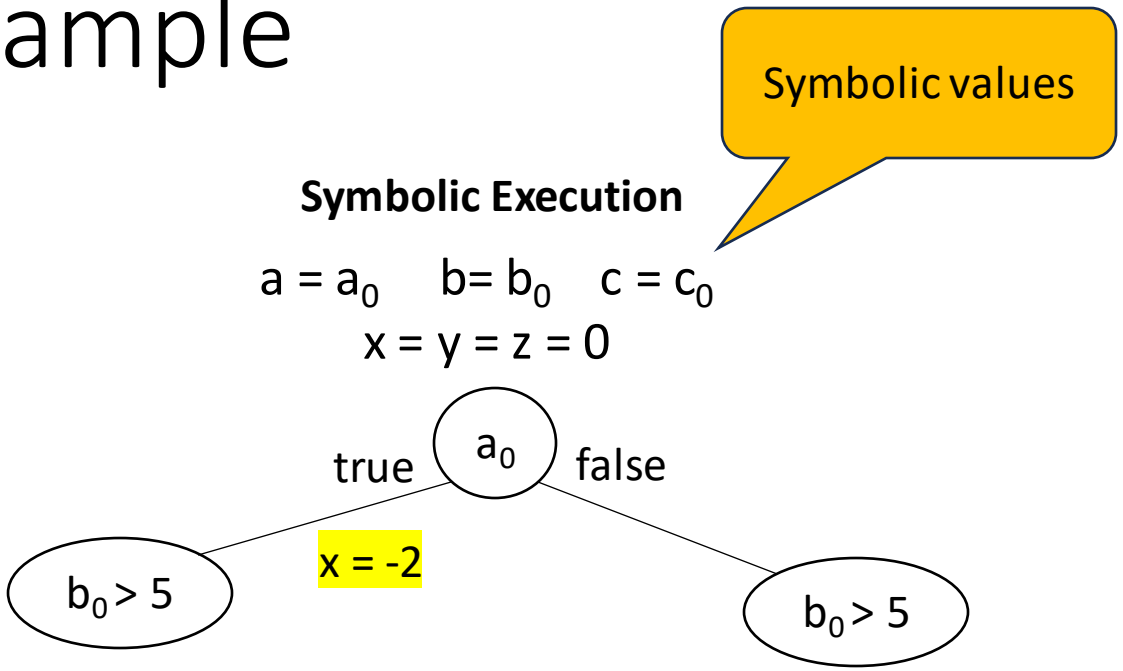
$a = a_0$ $b = b_0$ $c = c_0$

$x = y = z = 0$



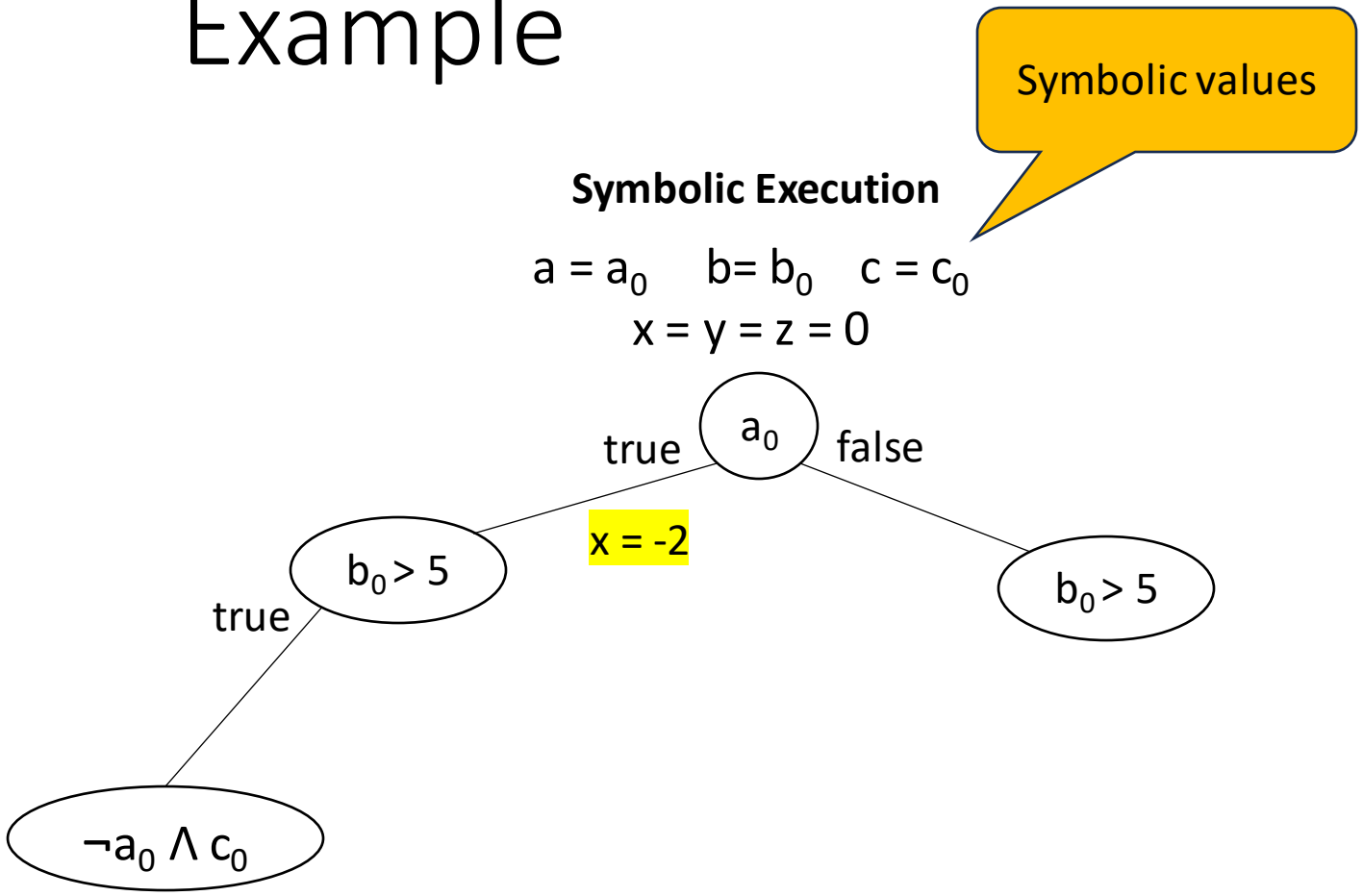
Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
           y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```



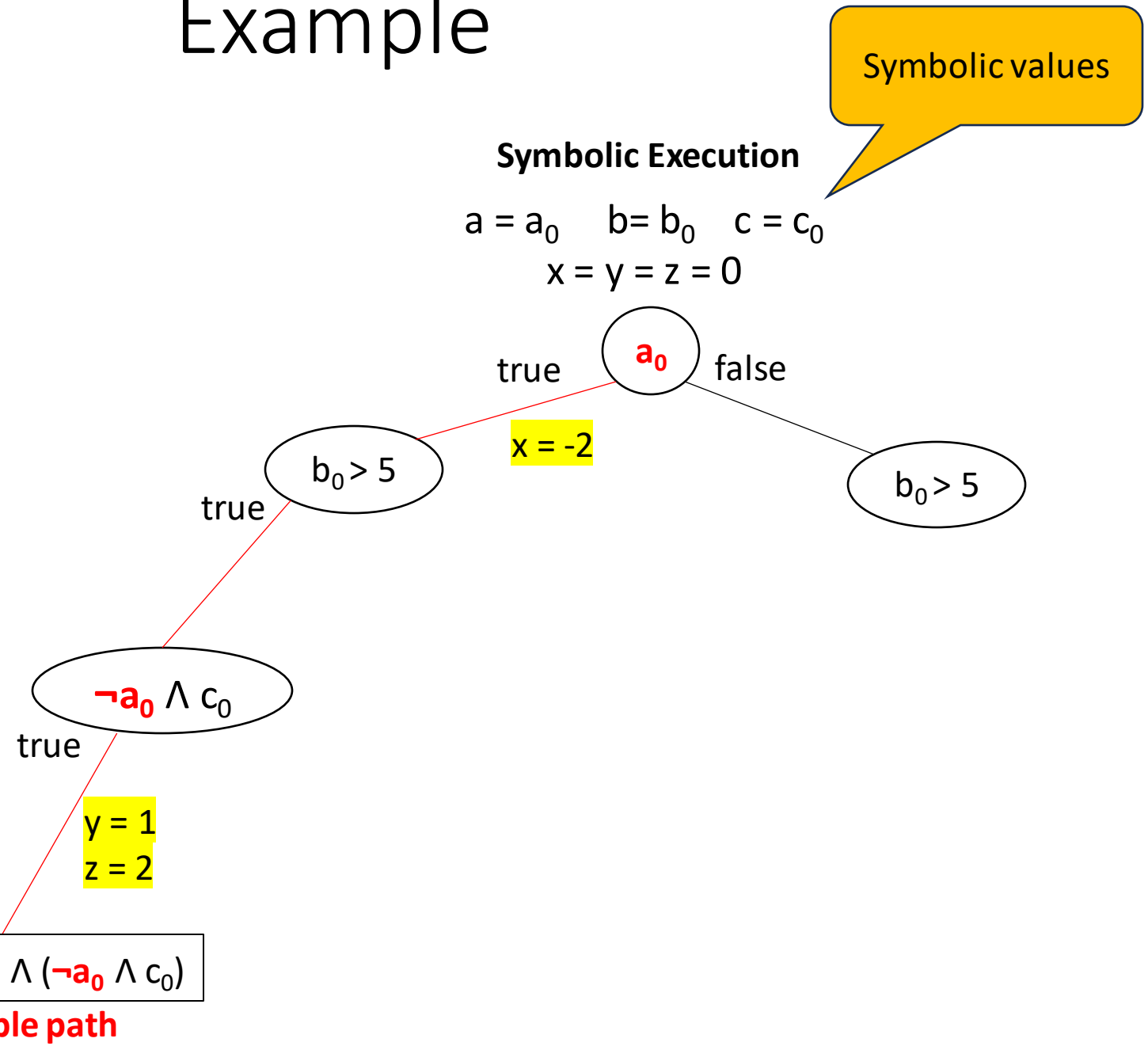
Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
           y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```



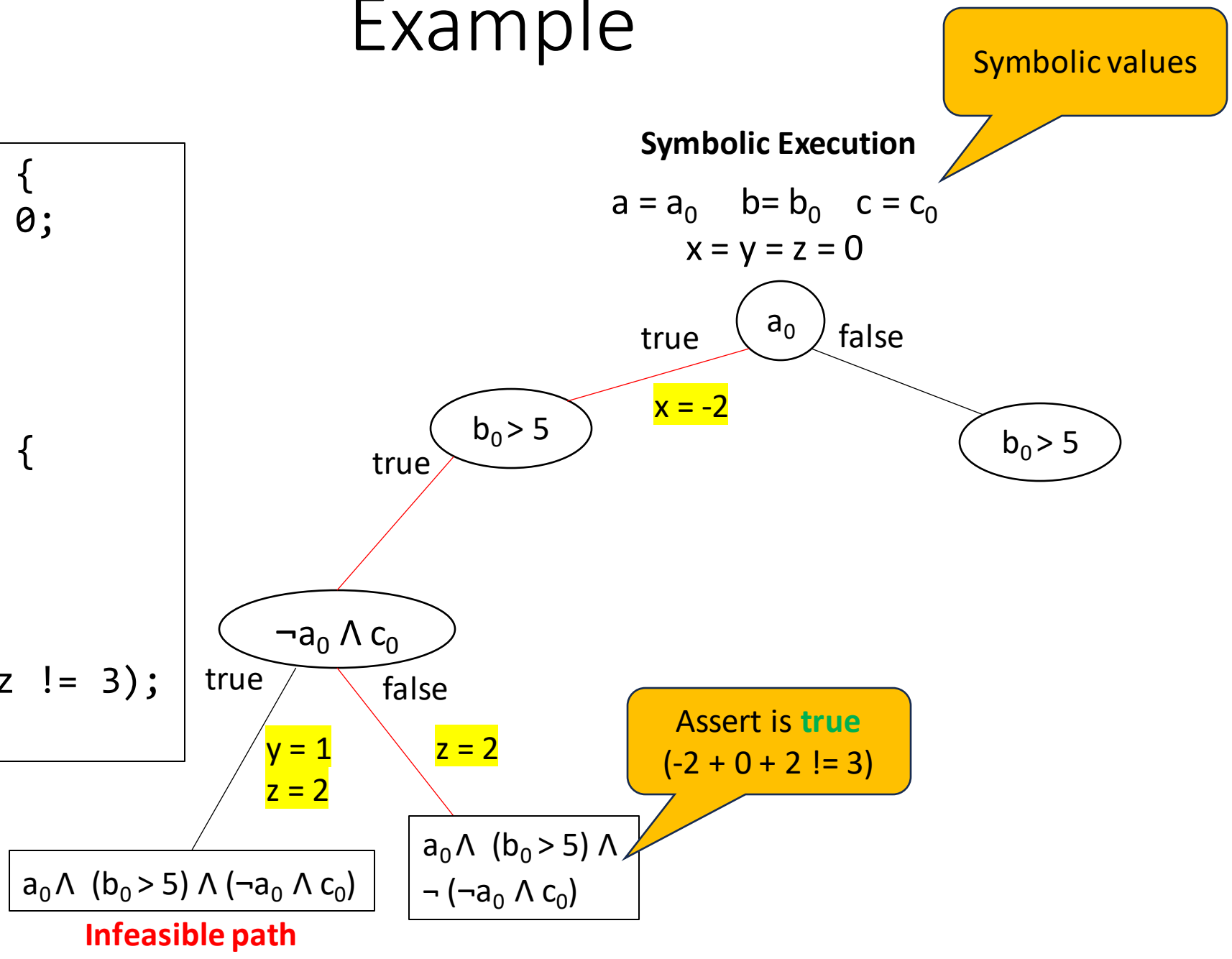
Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
            y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```



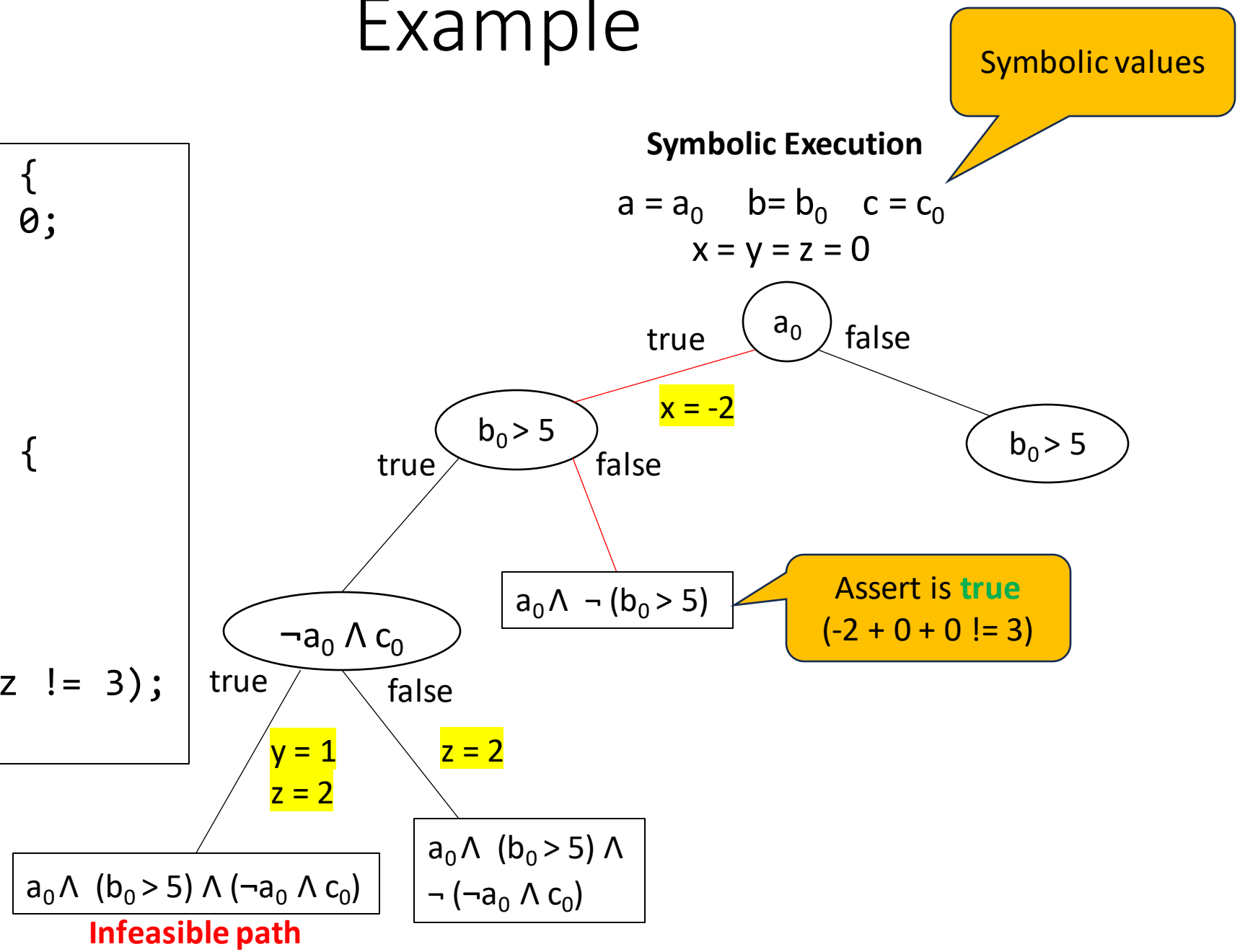
Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
            y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```



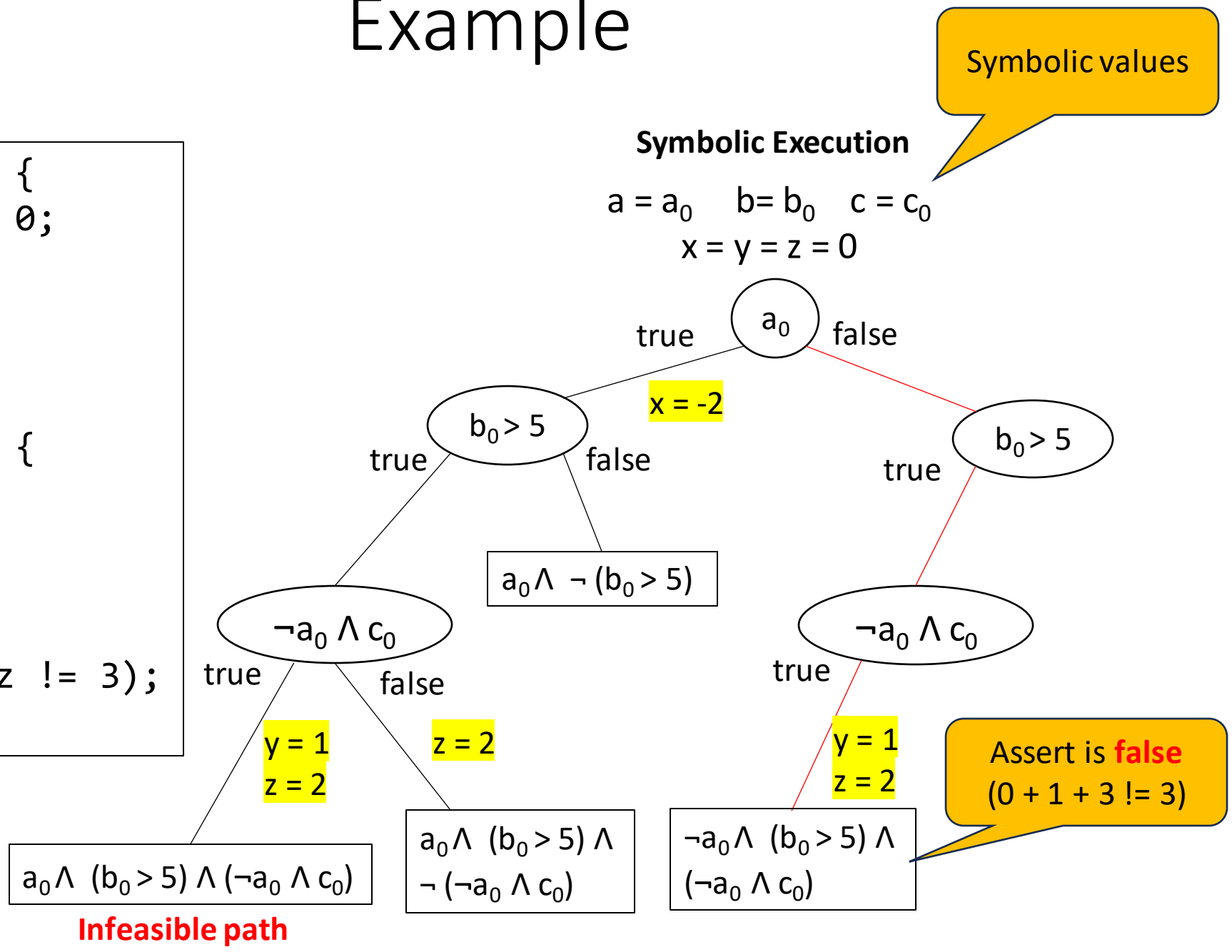
Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
            y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```



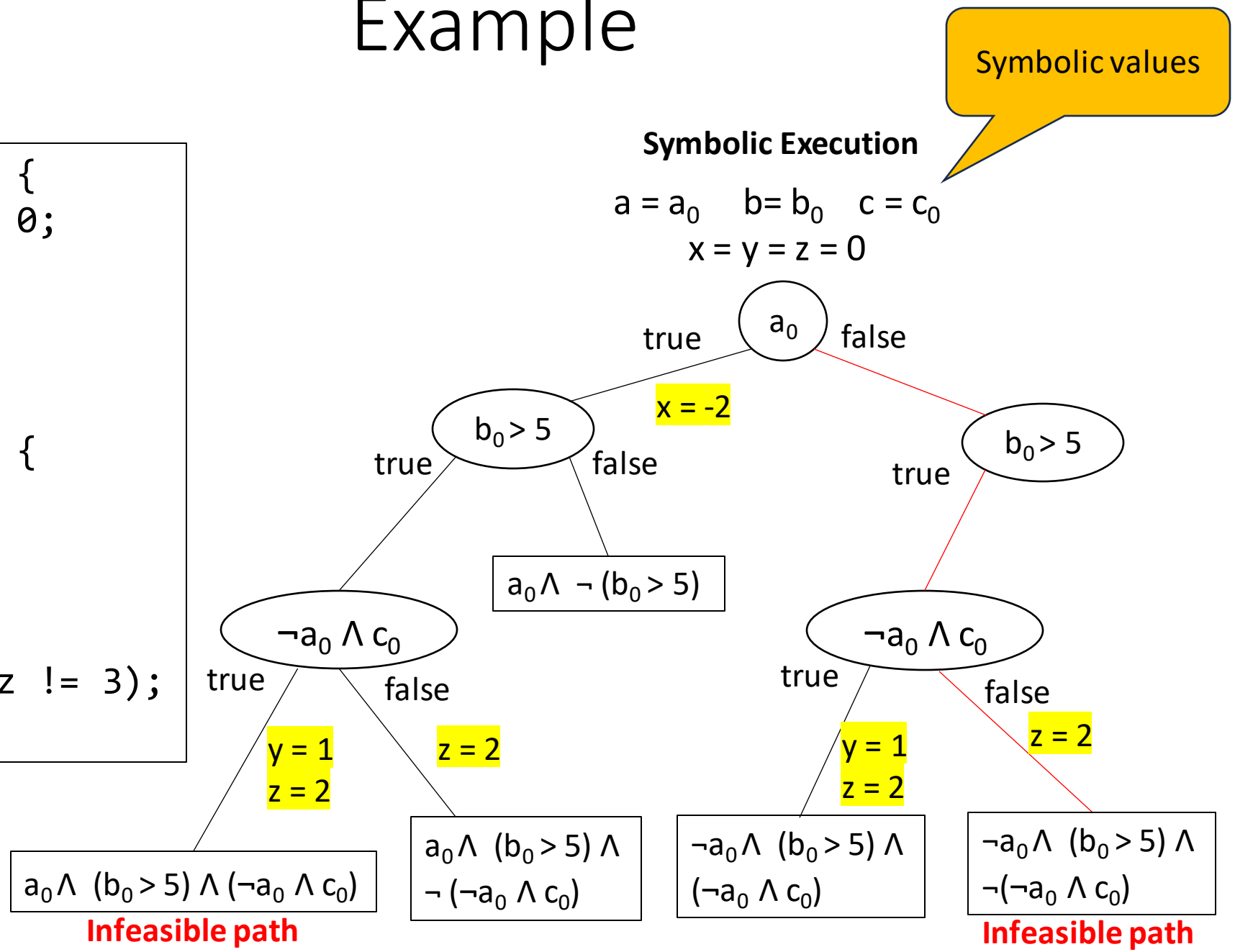
Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
            y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```



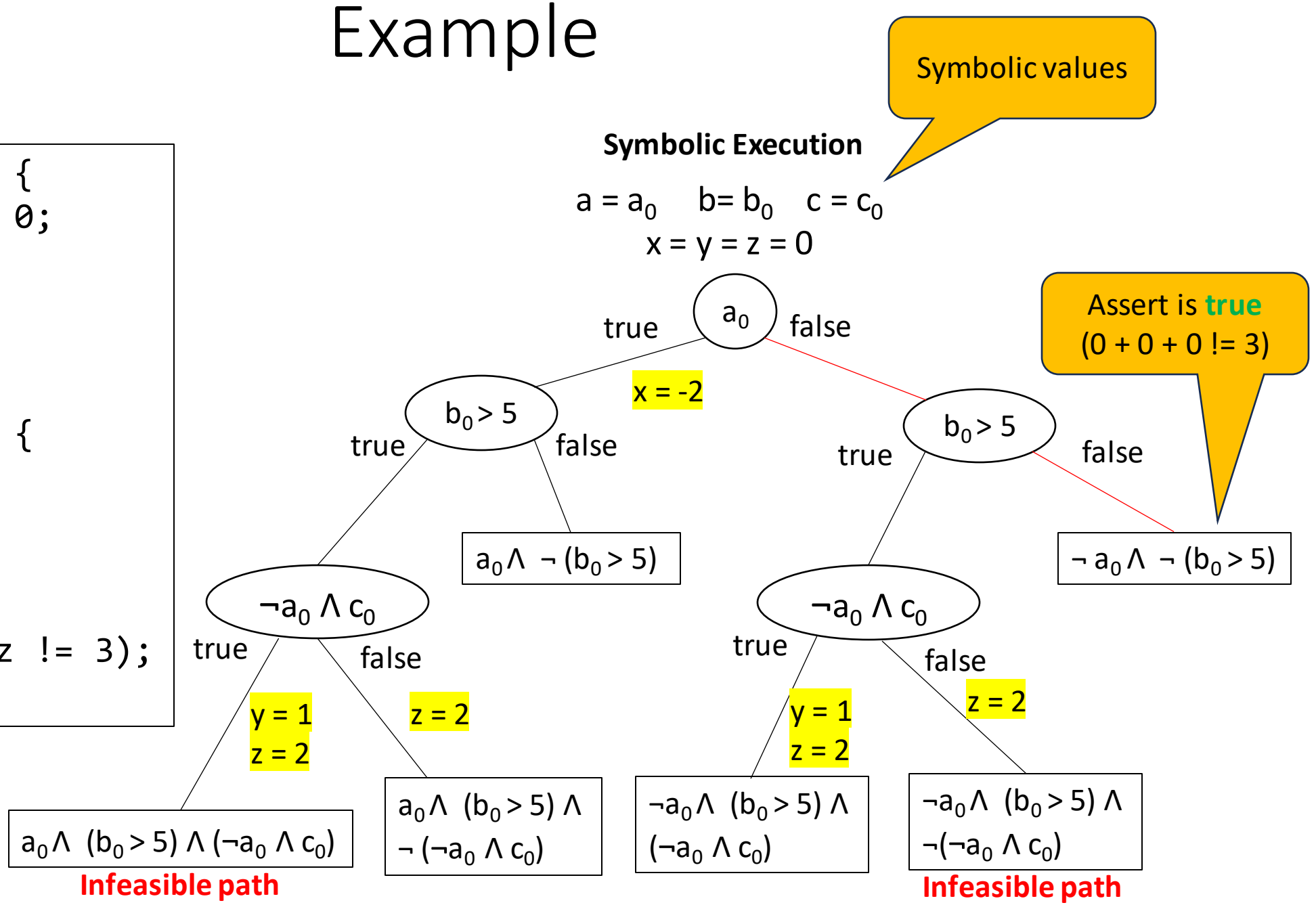
Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
            y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```



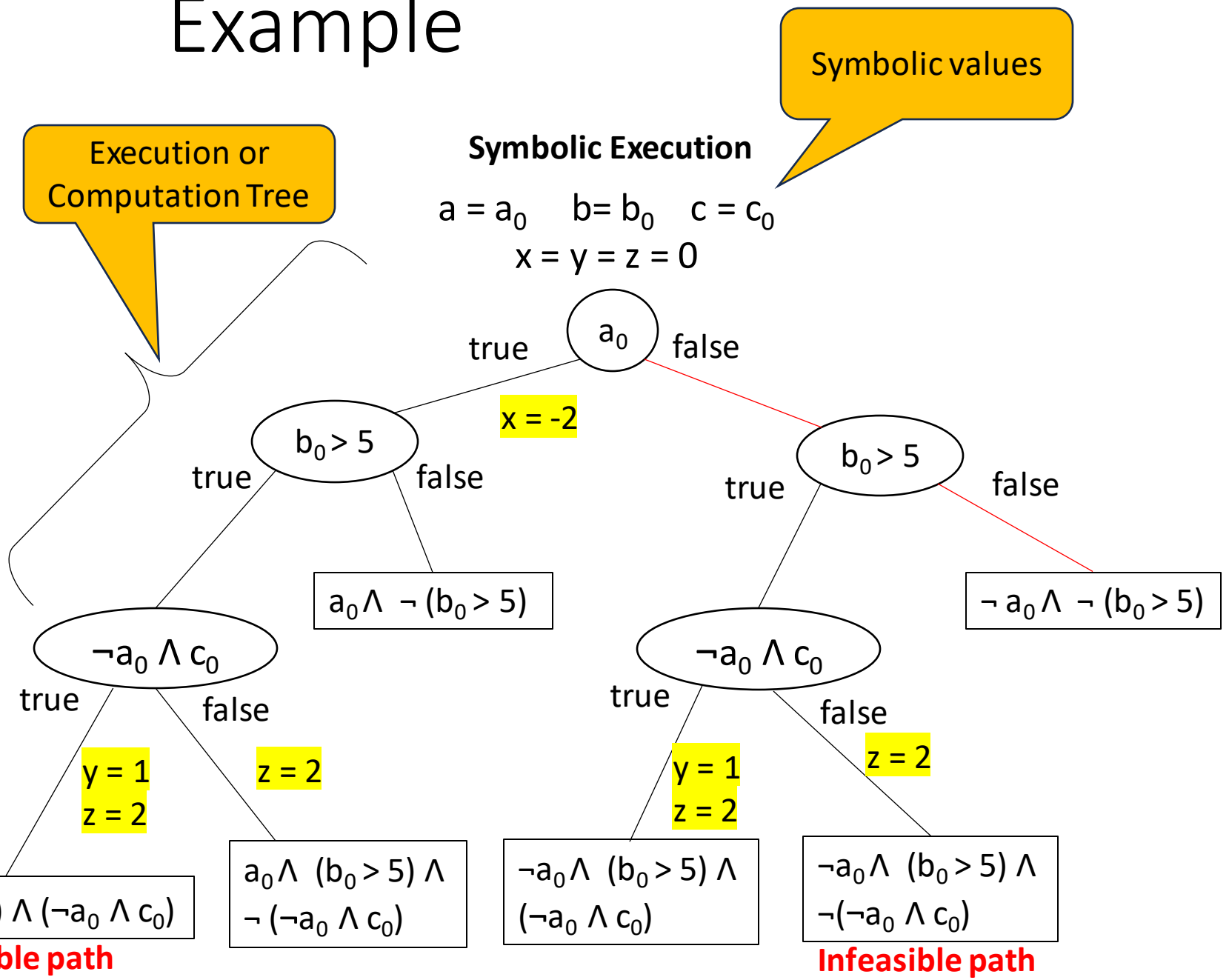
Example

```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
            y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```



Example

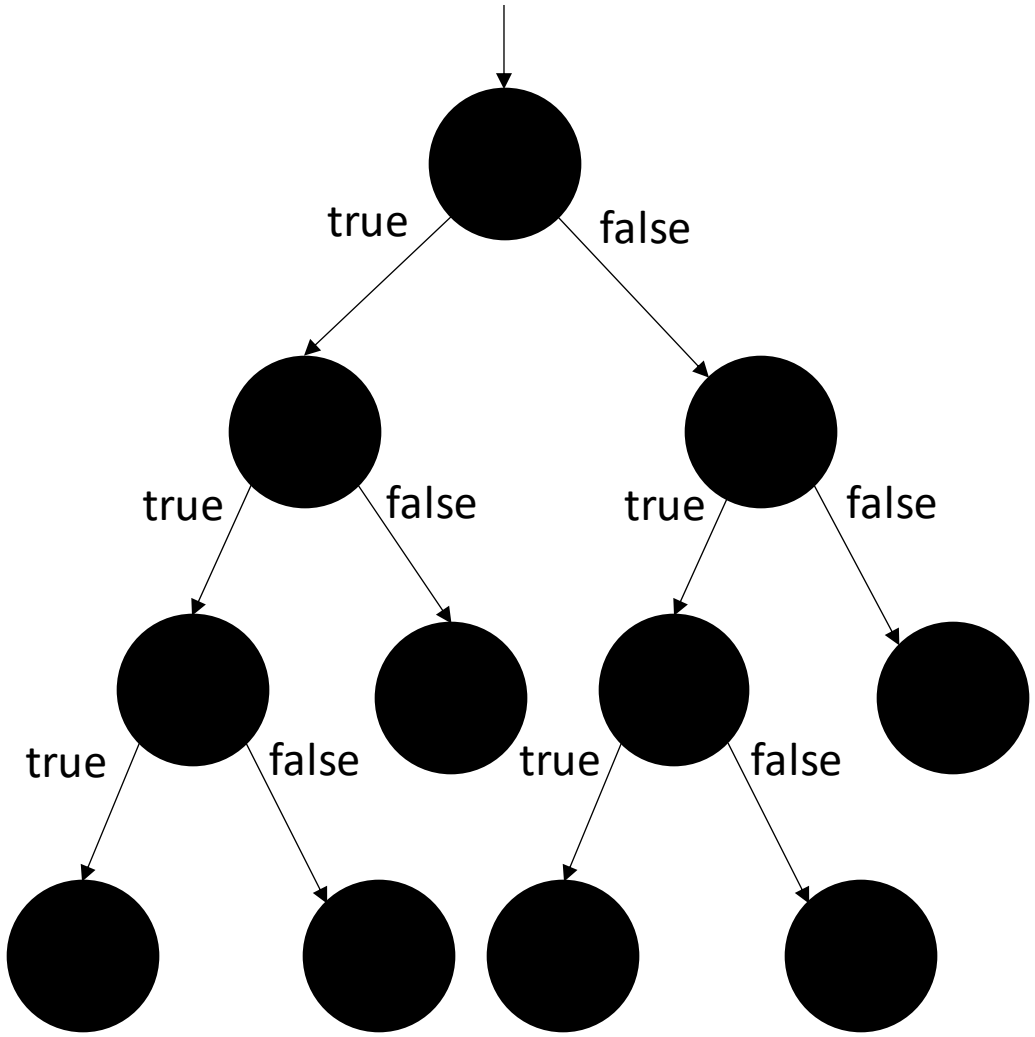
```
function f(a, b, c) {
    var x = y = z = 0;
    if (a) {
       x = -2;
    }
    if (b > 5) {
       if (!a && c) {
            y = 1;
       }
      z = 2;
    }
    assert(x + y + z != 3);
}
```



Execution Tree

All possible execution paths

- **Binary tree**
- **Nodes**: conditional statements
- **Edges**: execution of program on non-conditional statements
- Each path in tree represents equivalence class of inputs.



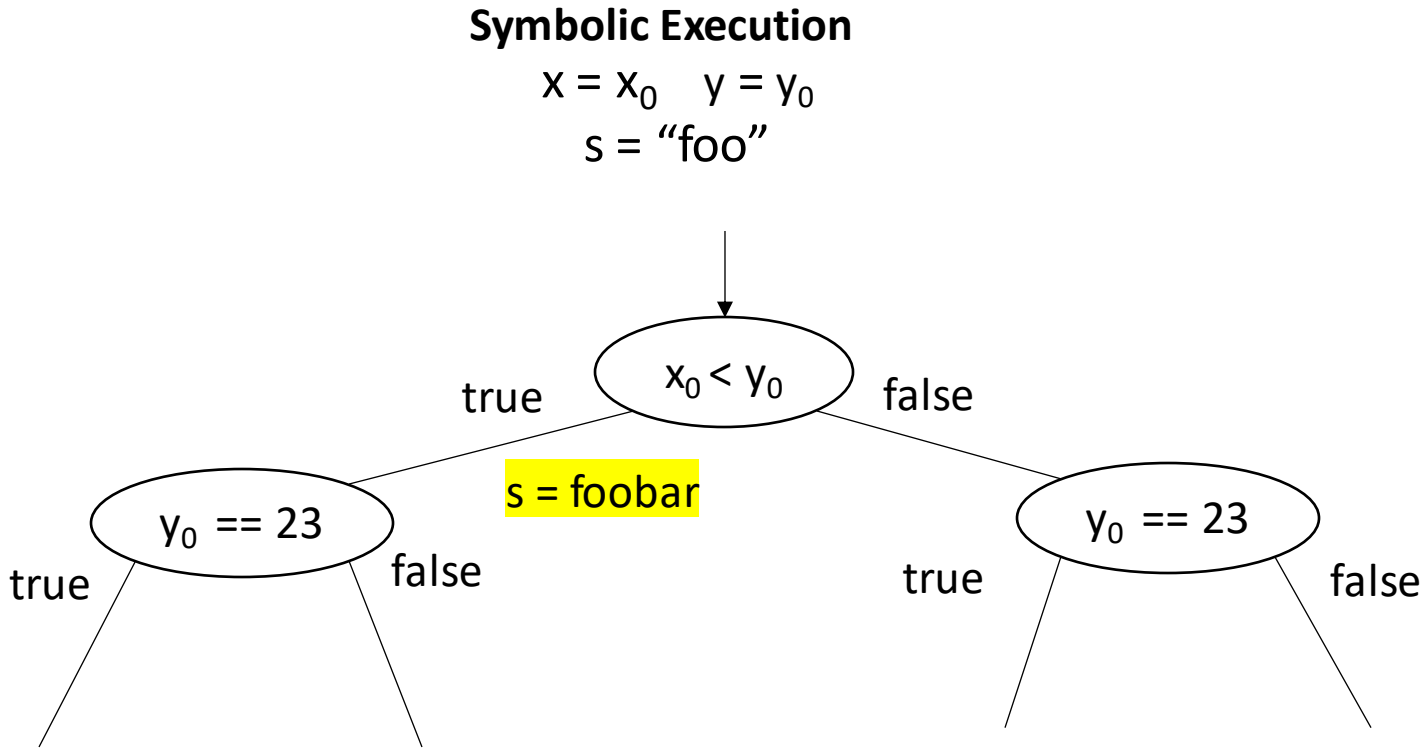
QUIZ: Execution Tree

```
function f(x,y) {
    var s = “foo”;
    if (x < y) {
       s += “bar”;
        console.log(s);
    }
    if (y == 23) {
       console.log(s);
}
```

How many nodes and edges are there in the execution tree of the above function?

QUIZ: Execution Tree

```
function f(x,y) {
    var s = “foo”;
    if (x < y) {
       s += “bar”;
        console.log(s);
    }
    if (y == 23) {
        console.log(s);
}
```

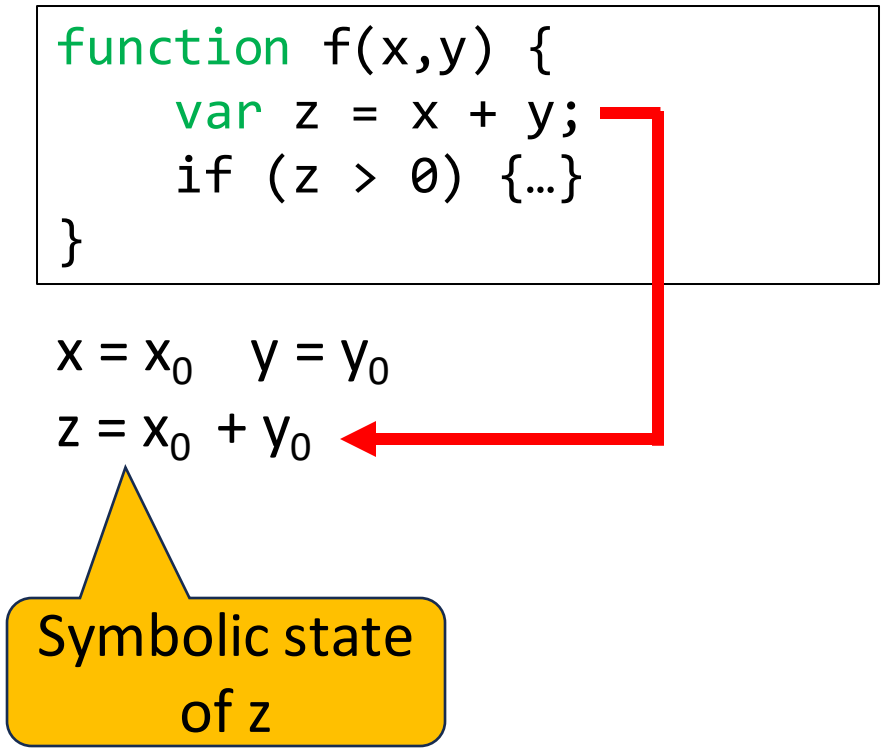


How many nodes and edges are there in the execution tree of the above function?

3 Nodes ands 7 Edges

Symbolic Values and Symbolic States

- Unknown values of variables e.g., user inputs are kept symbolically
- Symbolic State maps program variables to their symbolic values



Path Conditions

Quantifier-free formula over the symbolic inputs that encode all branch decisions taken so far

```
function f(x,y) {
    var z = x + y;
    if (z > 0) {…}
}
```

$x = x_0 \quad y = y_0$

$z = x_0 + y_0$

$(x_0 + y_0) > 0$

Path condition

Satisfiability of Formulas

- Determine whether a path is feasible
  - Check if the path condition is satisfiable
- Done by powerful SMT/SAT solvers
  - SAT = Satisfiability
  - SMT = Satisfiability Modulo Theory
  - E.g., Z2, Yices, STP
- For satisfiable formula, solvers also provide concrete solution
- Examples
  - $(x_0 + y_0) > 1$: SAT $x_0 = 1; y_0 = 1$
  - $(x_0 + y_0) < 0 \wedge (x_0 - 1) > 5 \wedge (y_0 > 0)$: UNSAT

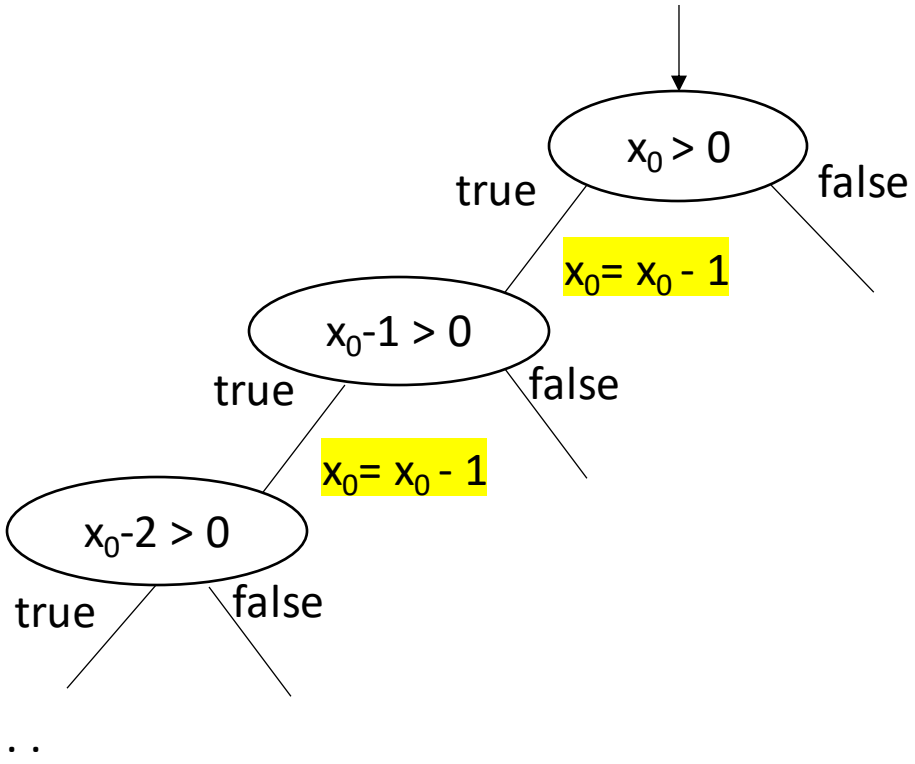
Applications of Symbolic Execution

- General goal: reason about the behavior of program
- Basic applications
  - Detect infeasible paths
  - Generate test inputs
  - Find bugs and Vulnerabilities
- Advanced applications
  - Generate program invariants
  - Prove program equivalence
  - Debugging
  - Automatic Program Repair

Problems of Symbolic Execution

- Loops and Recursions: infinite execution tree

```
function f(a) {
    var x = a;
    while (x > 0) {
       x--;
    }
}
```



Problems of Symbolic Execution

- Loops and Recursions: infinite execution tree
- Path explosion: number of paths is exponential in the number of conditionals

```
function f(a) {
    if (x) {…}
    if (y) {…}
    …
}
```

How many paths are there in the execution tree of this function?

Problems of Symbolic Execution

- Loops and Recursions: infinite execution tree
- Path explosion: number of paths is exponential in the number of conditionals

```
function f(a) {
    if (x) {…}
    if (y) {…}
    …
}
```

How many paths are there in the execution tree of this function?

$2^2 = 4$

1. $x \wedge y$
2. $x \wedge \neg y$
3. $\neg x \wedge y$
4. $\neg x \wedge \neg y$

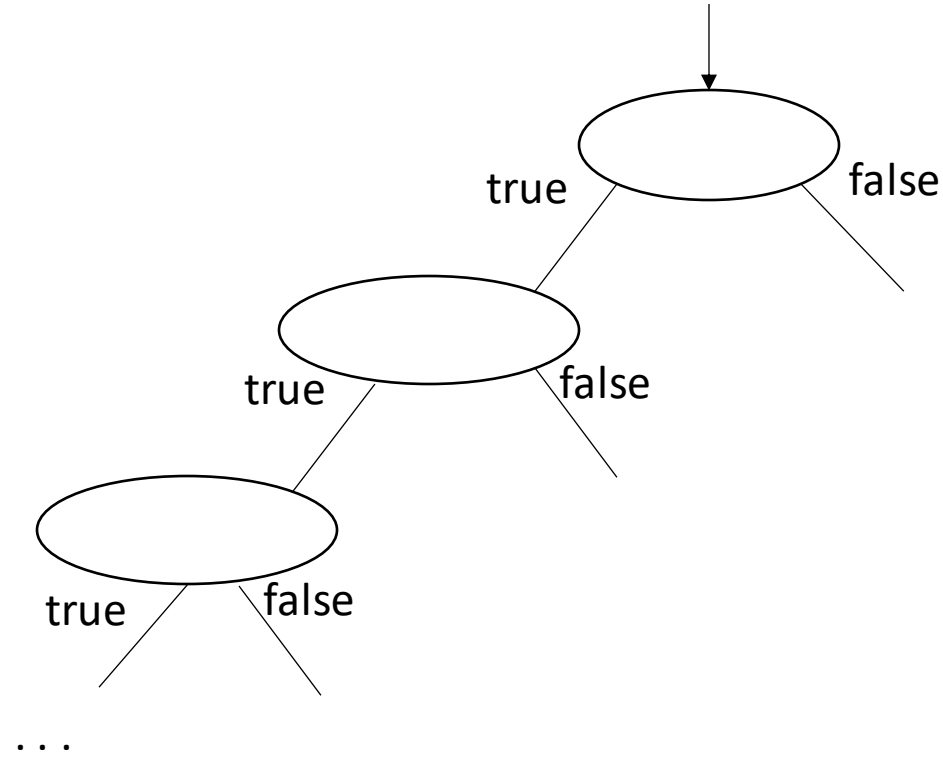
Problems of Symbolic Execution

- Loops and Recursions: infinite execution tree
- Path explosion: number of paths is exponential in the number of conditionals
- Environmental modeling: dealing with native/system/library calls
- Solver limitations: dealing with complex path conditions
- Heap modeling: symbolic representation of data structures and pointers

Dealing with Large Execution Trees

Heuristically select which branch to explore next

- Select at random
- Select based on coverage
- Prioritize based on distance to "interesting" program locations
- Interleaving symbolic execution with random testing



Dealing with Environment

- Program behavior may depend on parts of system not analyzed by symbolic execution engine
- E.g., native APIs, interaction with network, file system access

```
var fs = require(“fs”);
var content =
fs.readFileSync(“/tmp/foo.txt”);
if (content == “bar”) {
        …
}
```

How to reason about the environment?

Dealing with Environment

- Program behavior may depend on parts of system not analyzed by symbolic execution engine
- E.g., native APIs, interaction with network, file system access
- Solution: Model the file system (implemented by KLEE)
  - If all arguments are concrete, forward to OS
  - Otherwise, provide models that can handle symbolic files
    - **Goal**: explore all possible legal interactions with the environment

```
var fs = require("fs");
var content =
fs.readFileSync("/tmp/foo.txt");
if (content == "bar") {
       …
}
```

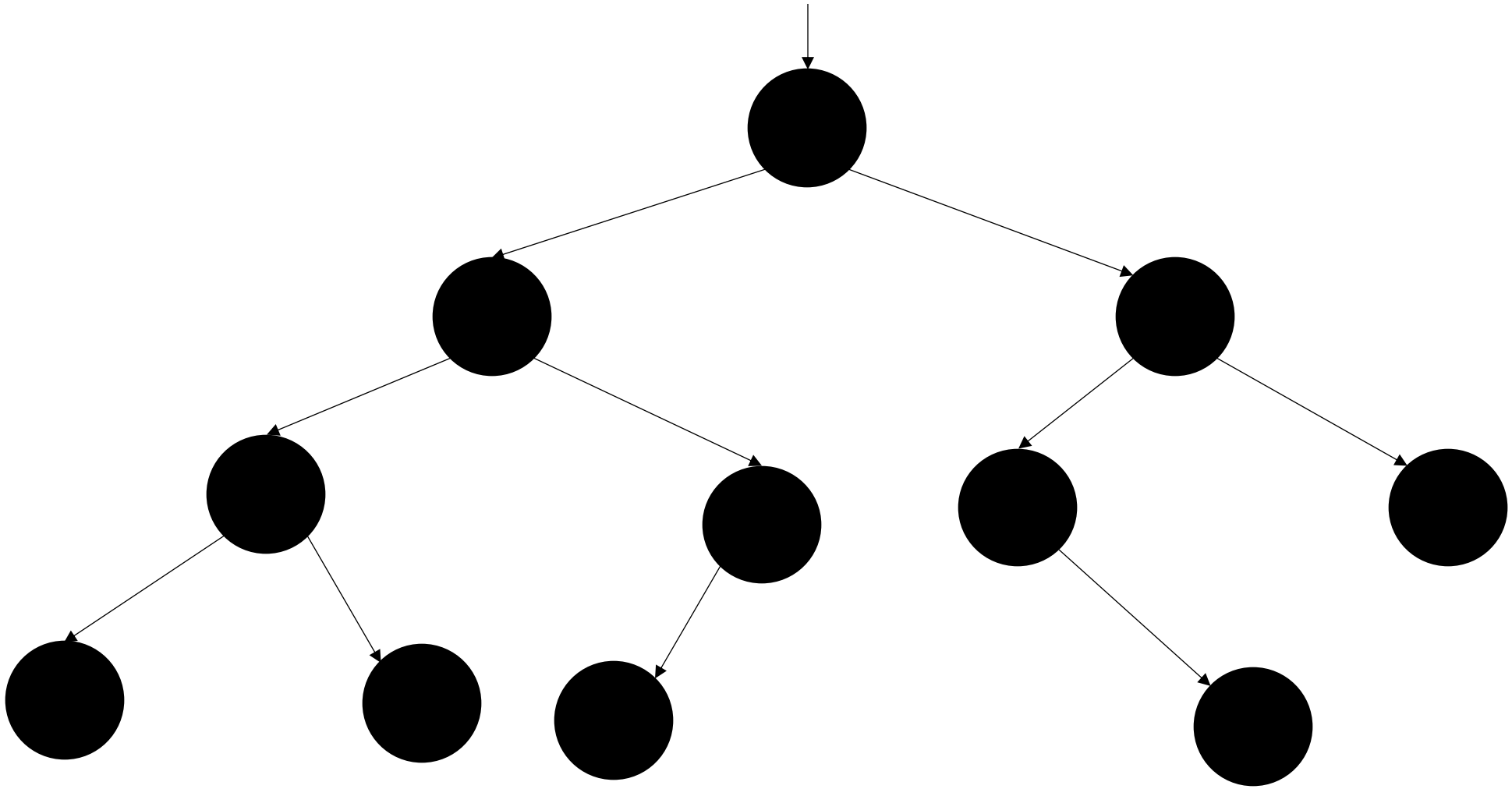
```
var fs = {
  readFileSync:  function(file){
  // doesn't read actual file system, but
  // models its effect for symbolic file name
 }
}
```

Combined Approach

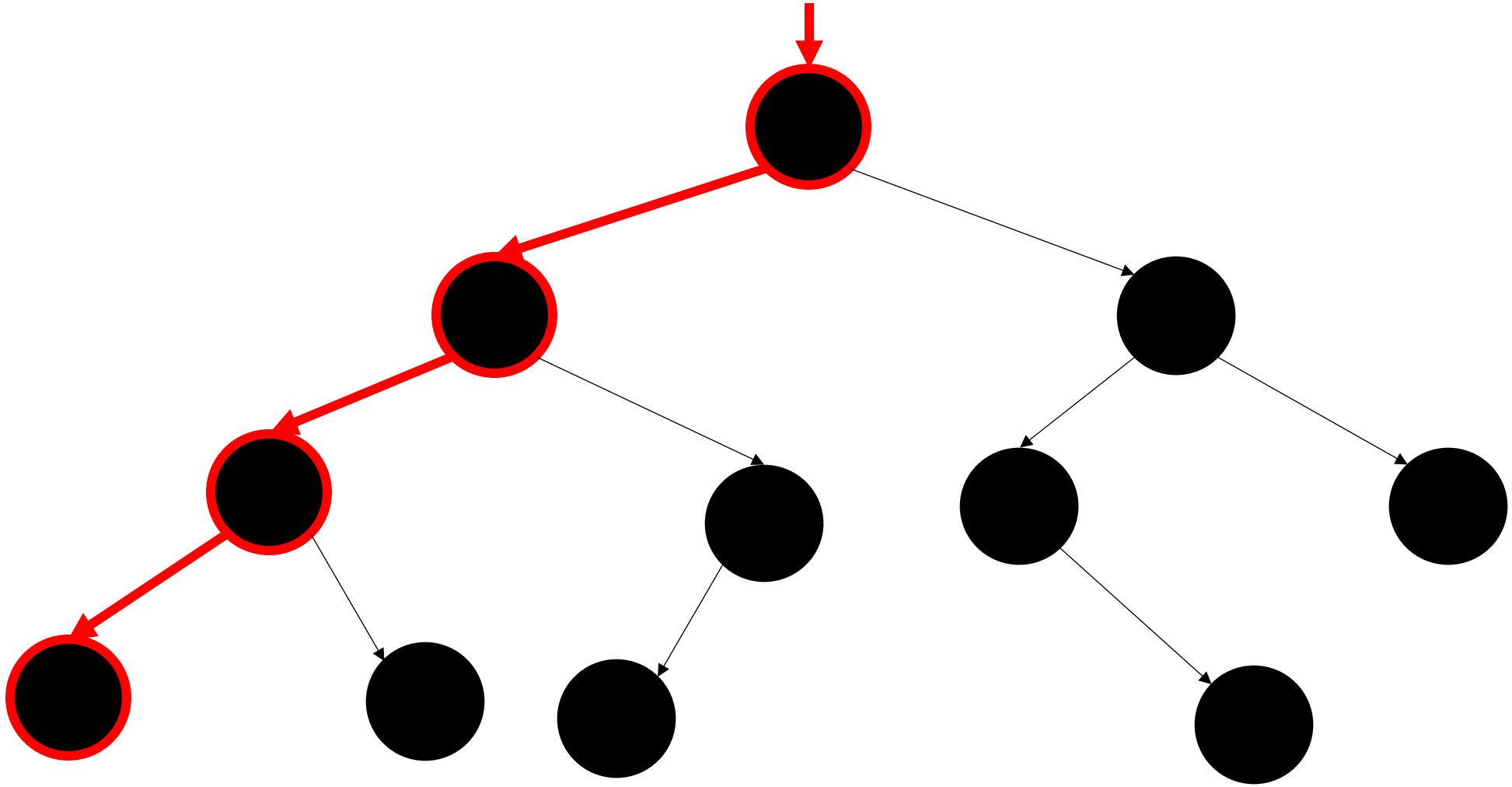
Concrete + Symb**olic** = **Concolic** Testing or
Dynamic Symbolic Execution (DSE)

- Start with random input values
- Keep track of both concrete and symbolic values
- Use concrete values to simplify symbolic constrains to help solver

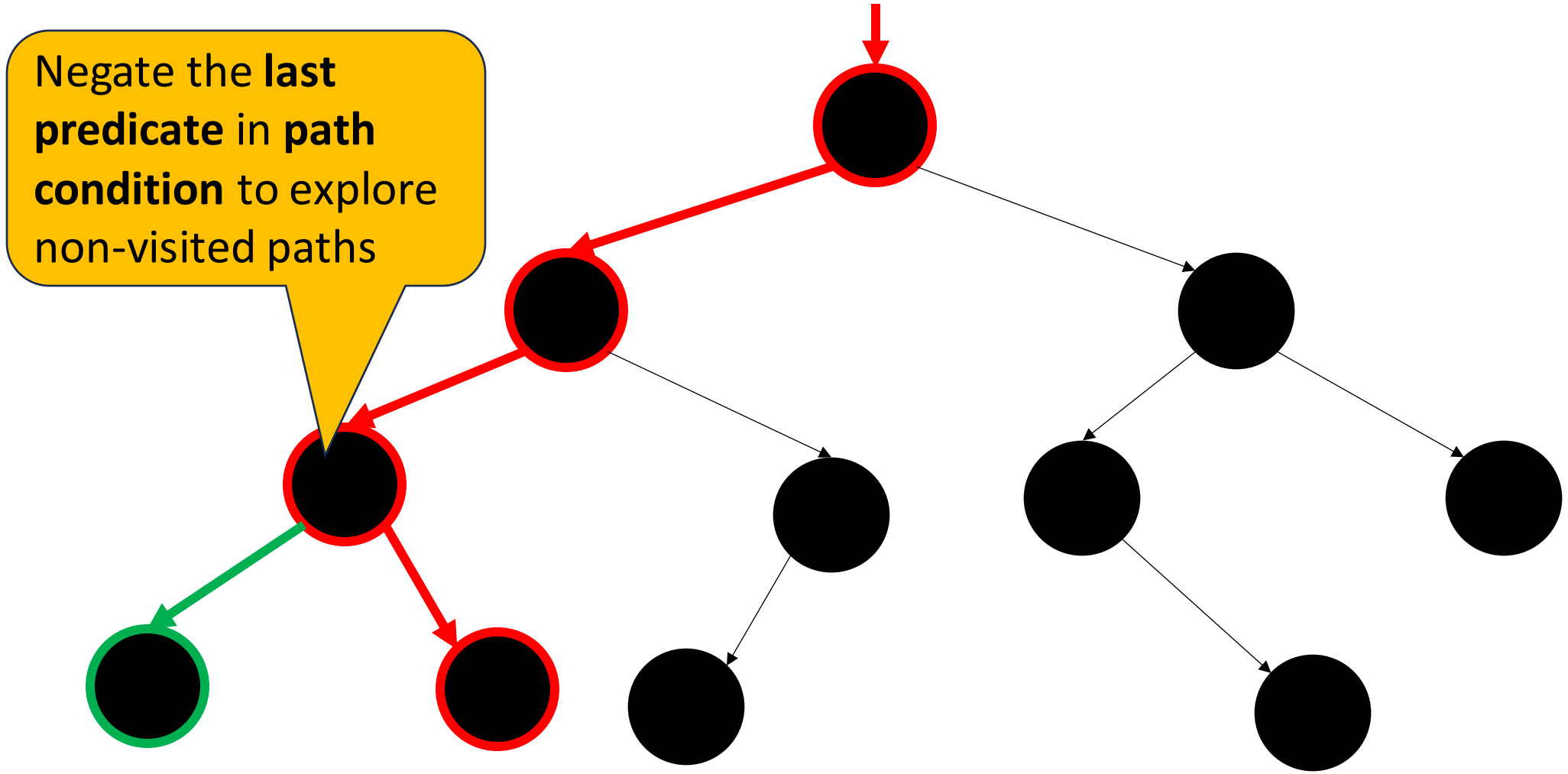
Exploring the Execution Tree



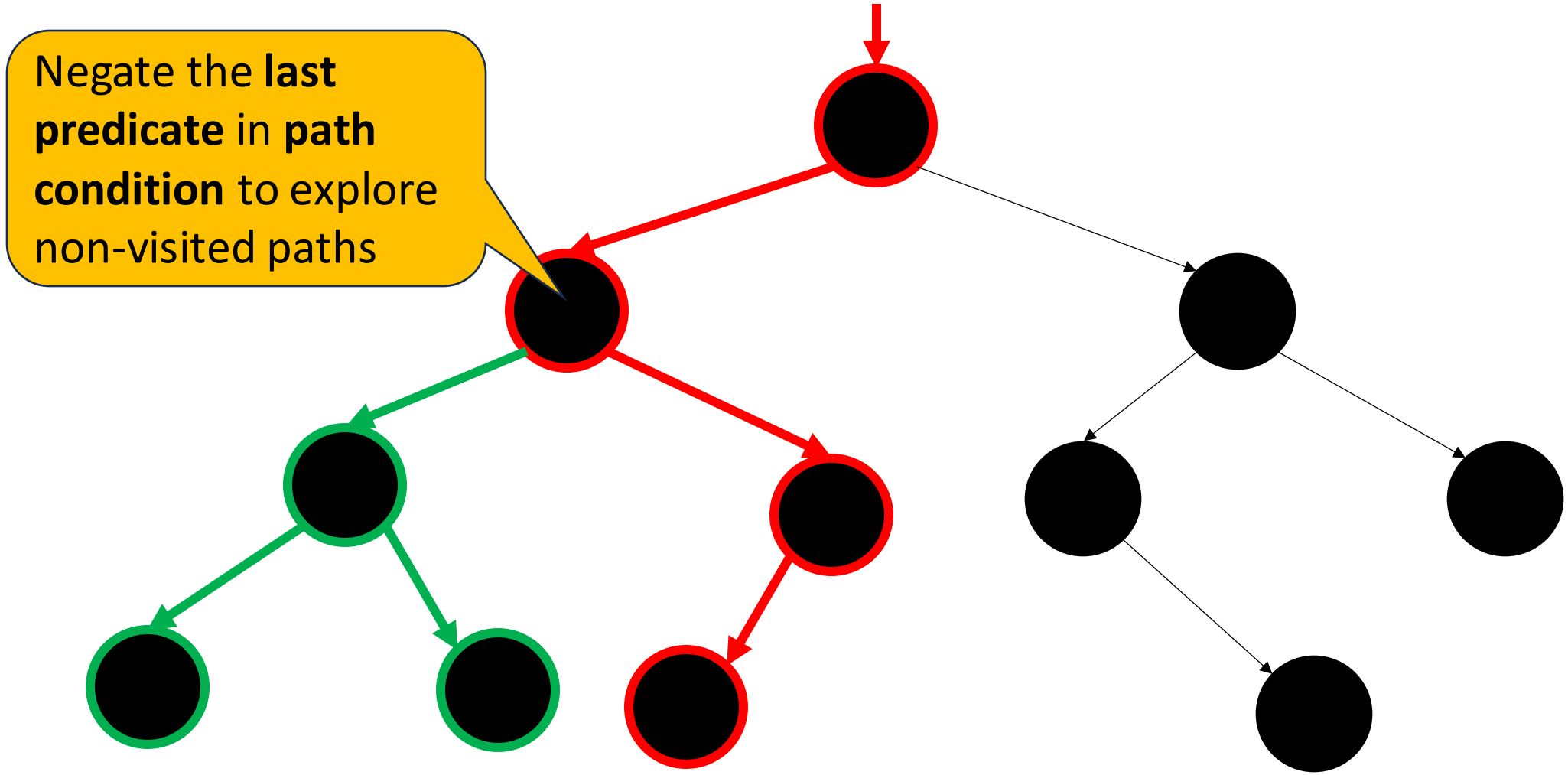
Exploring the Execution Tree



Exploring the Execution Tree

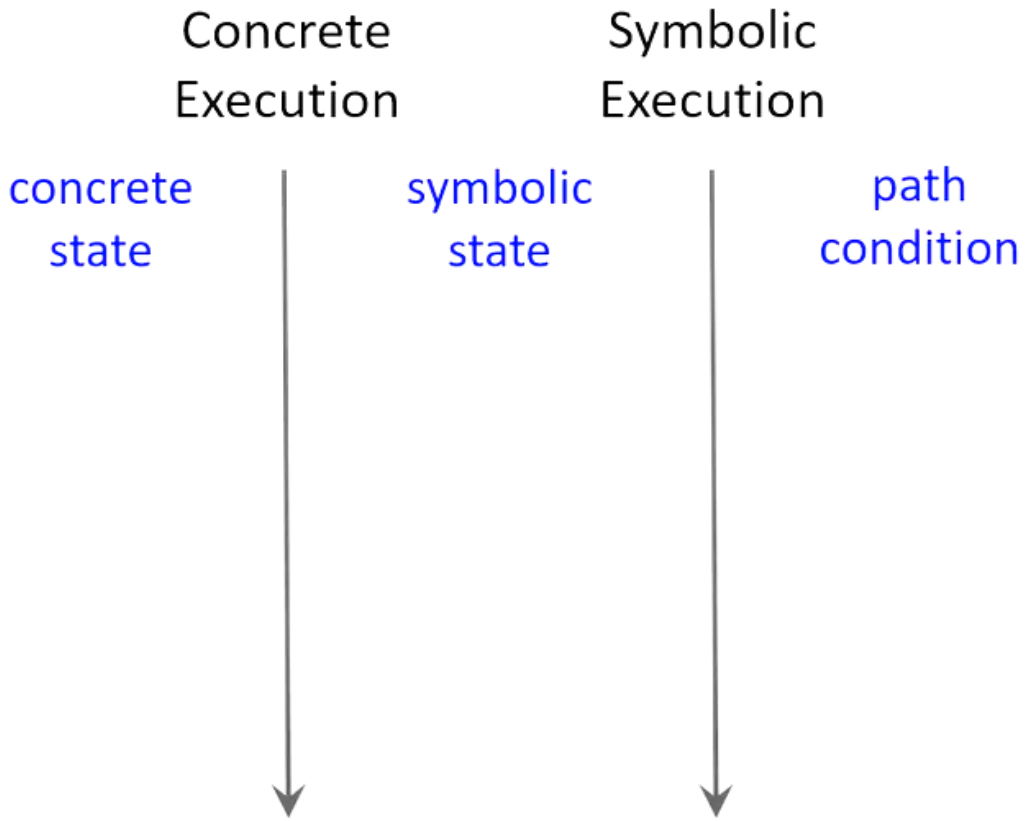


Exploring the Execution Tree



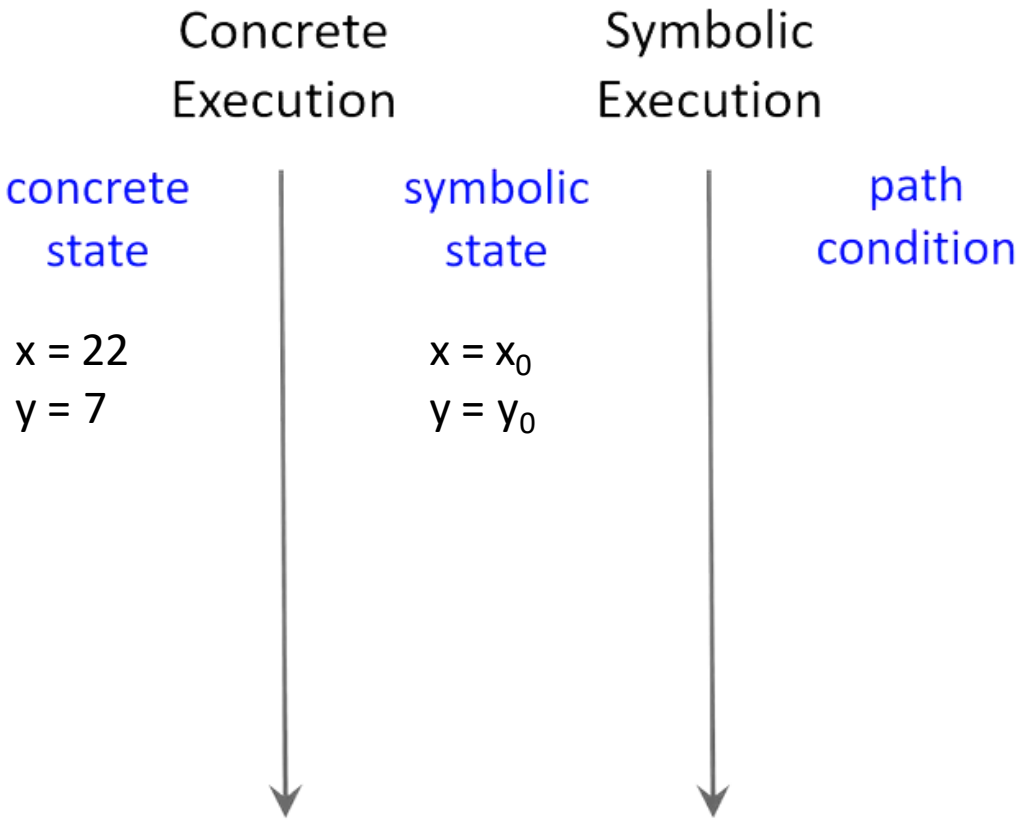
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



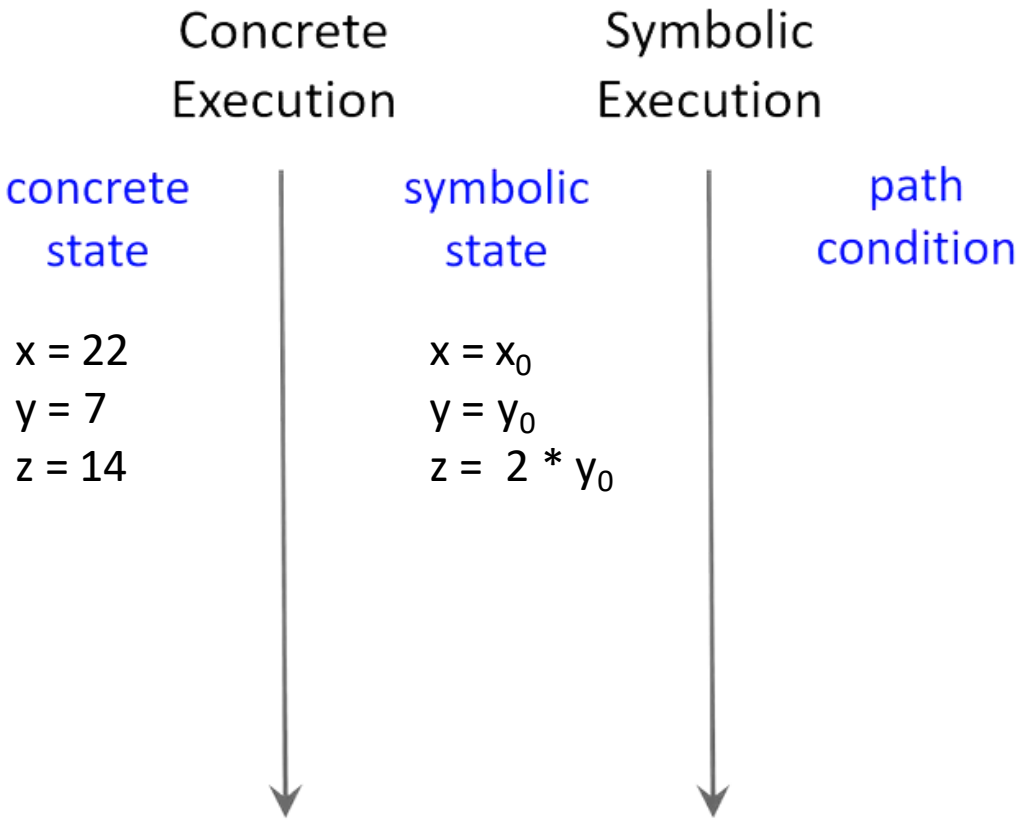
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



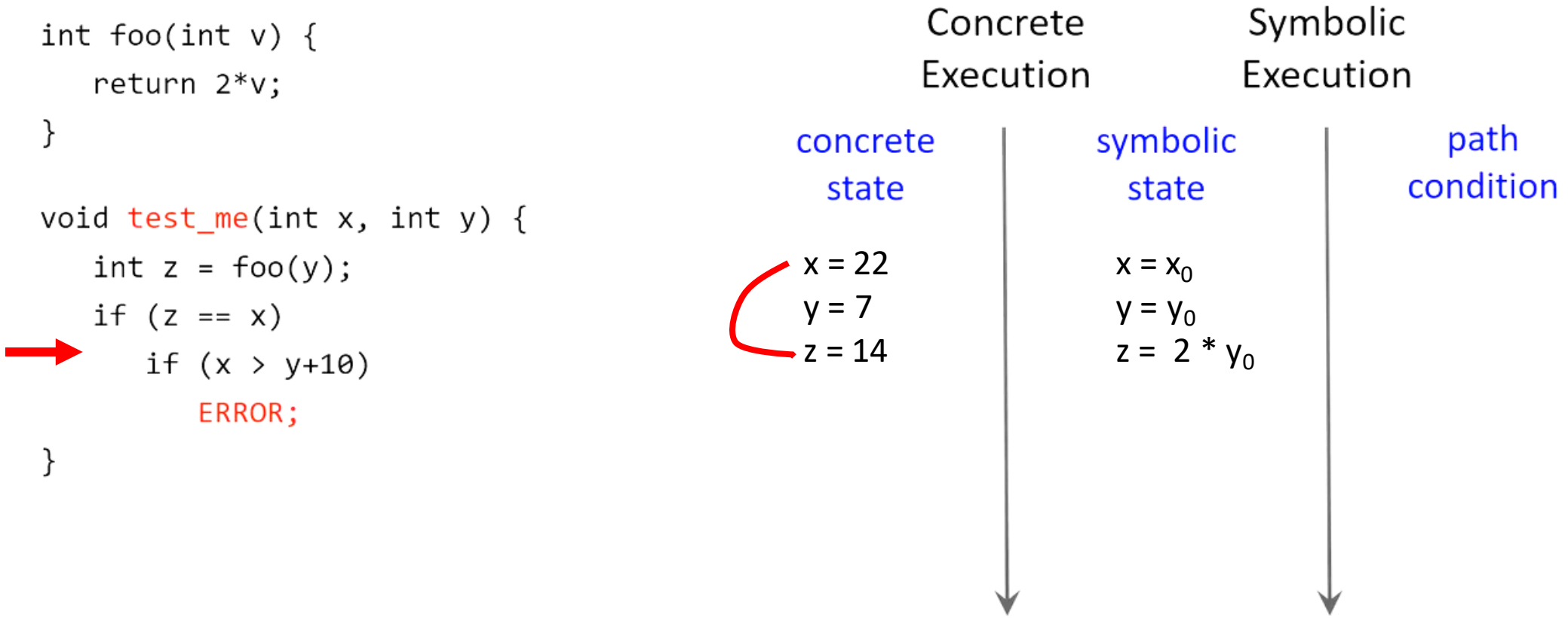
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



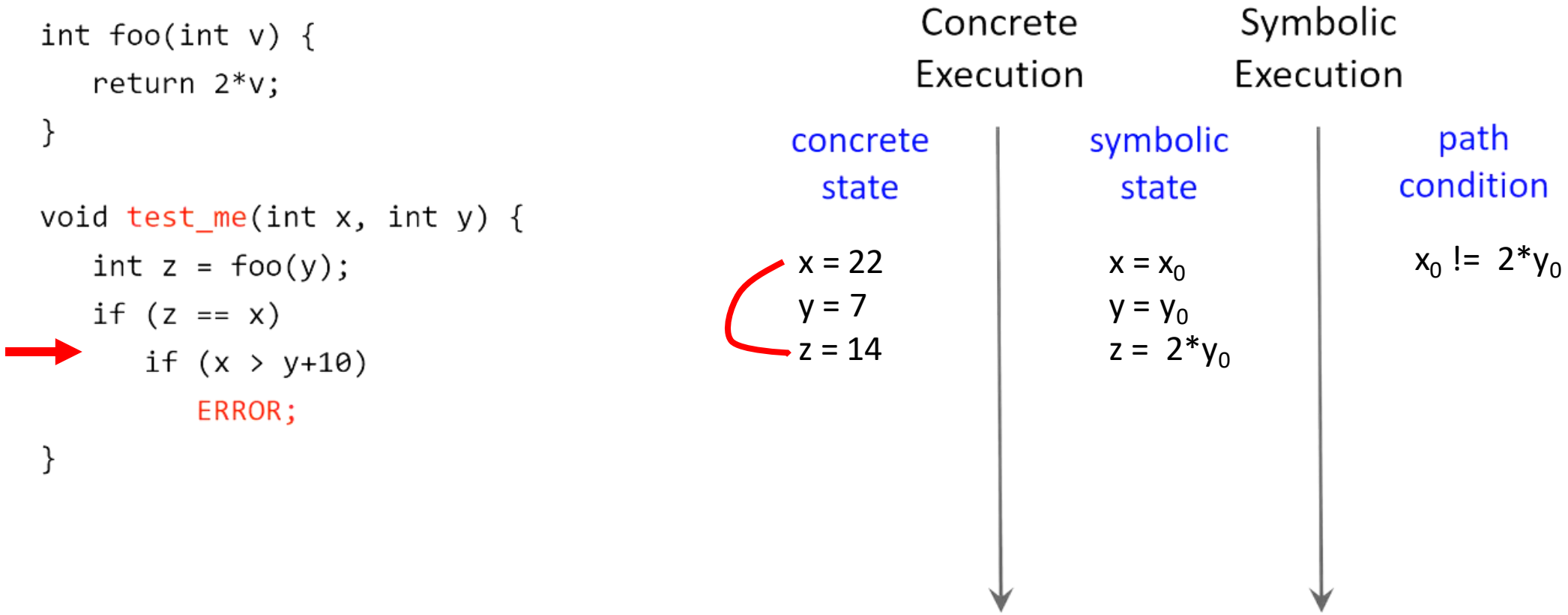
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



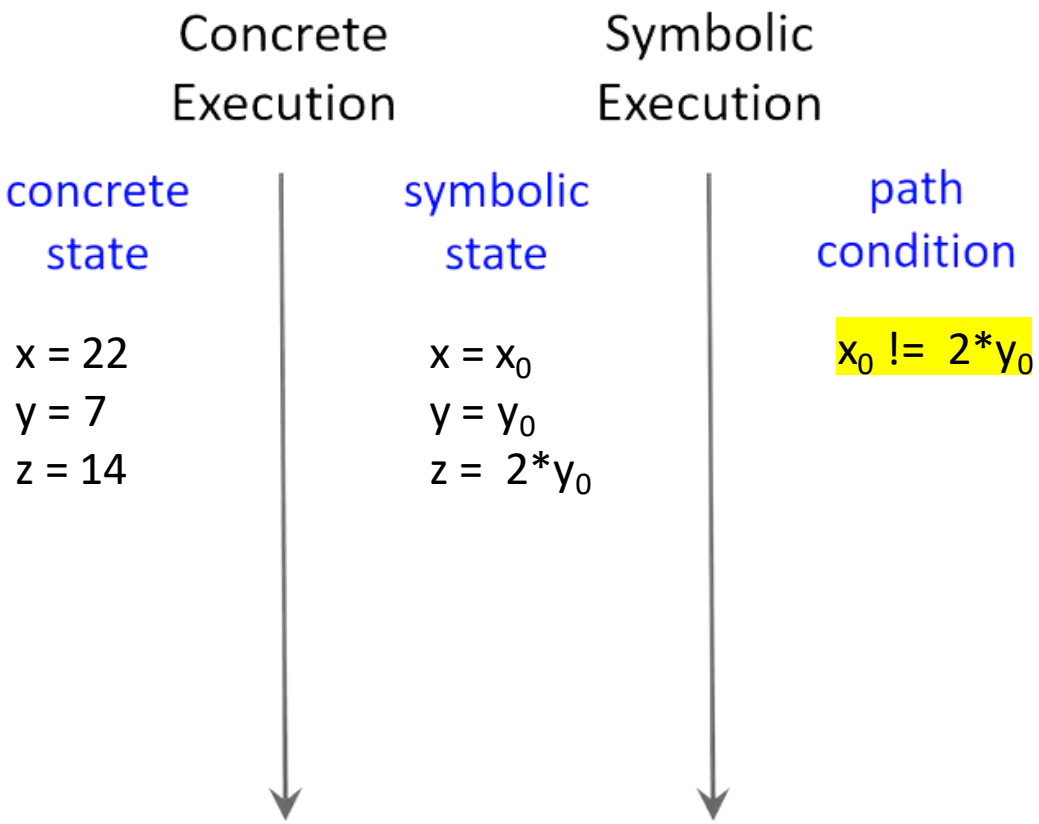
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



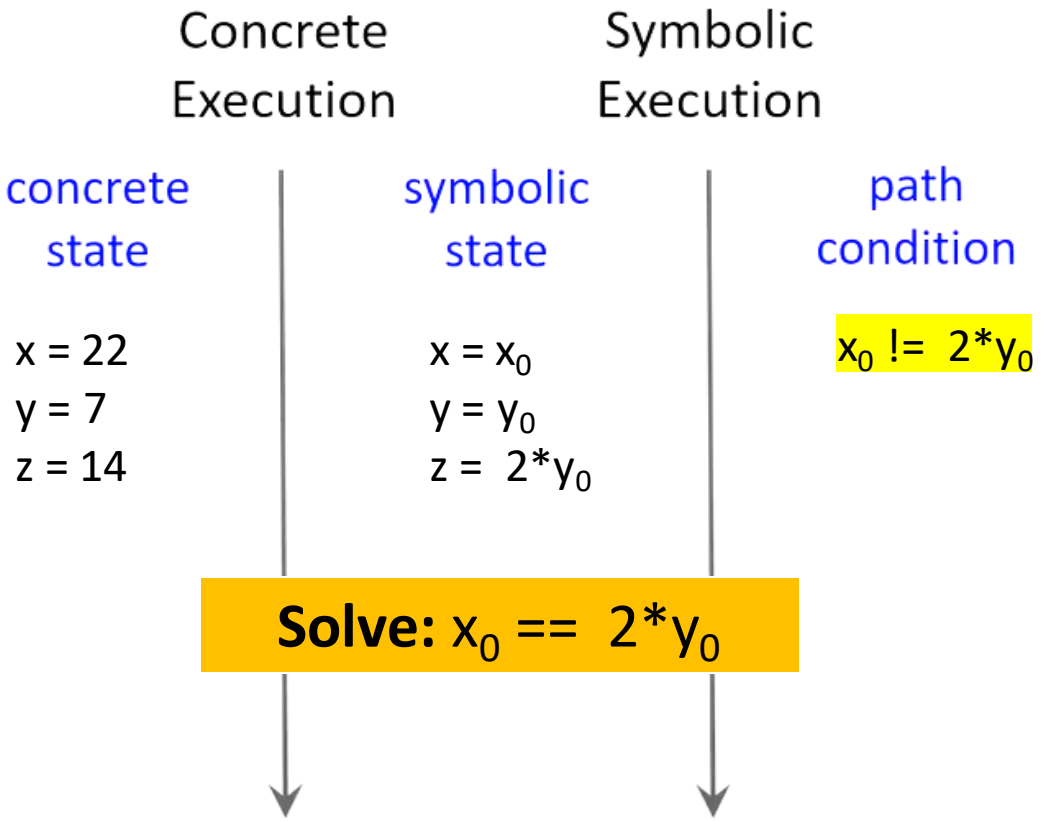
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



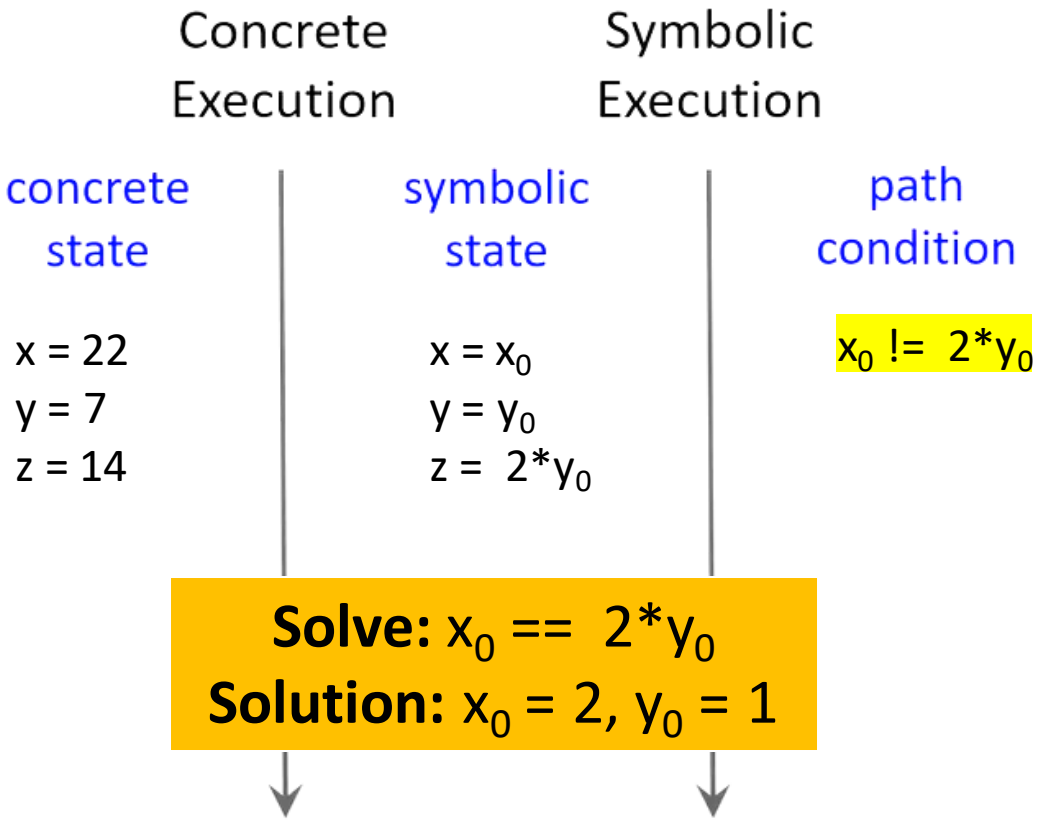
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



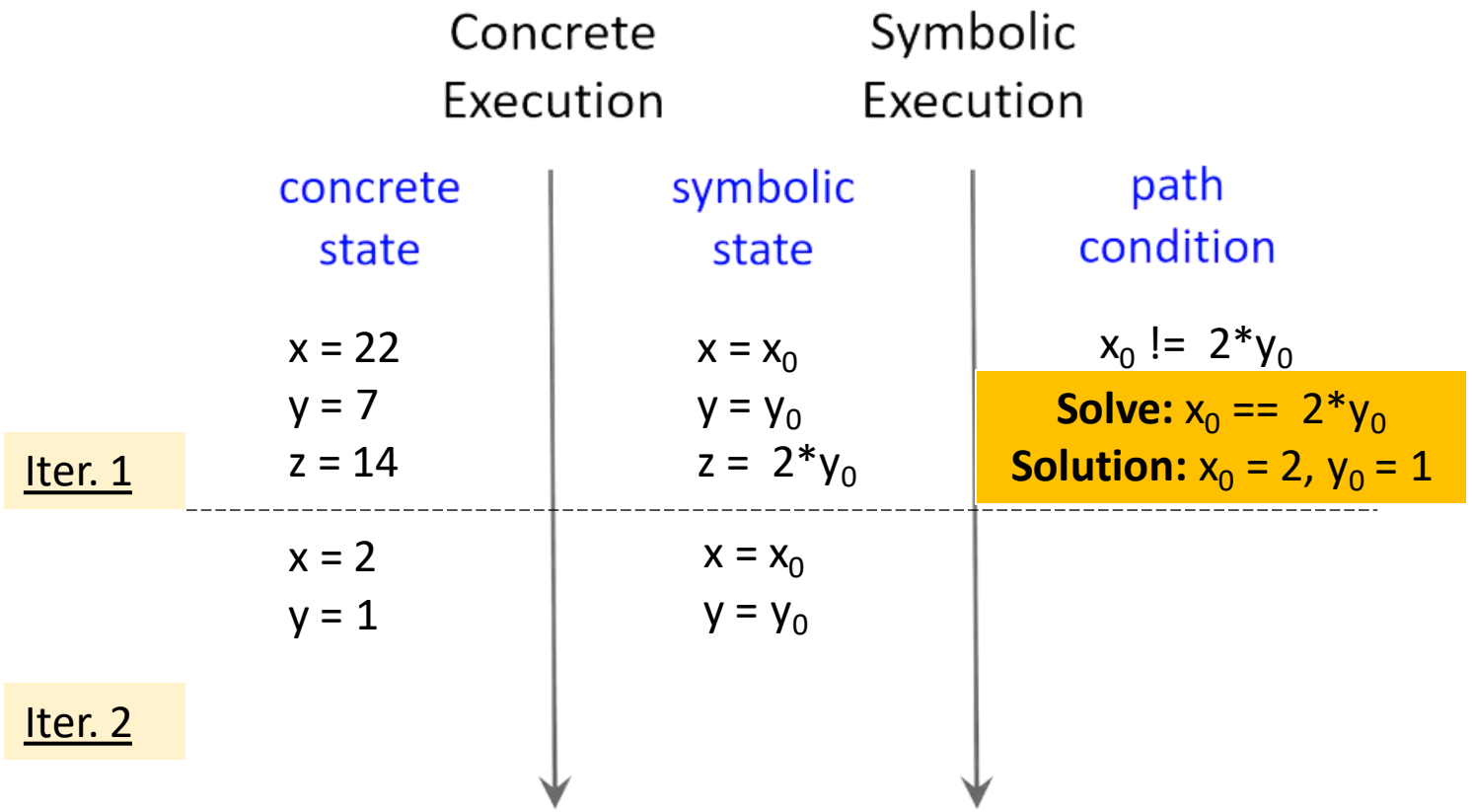
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



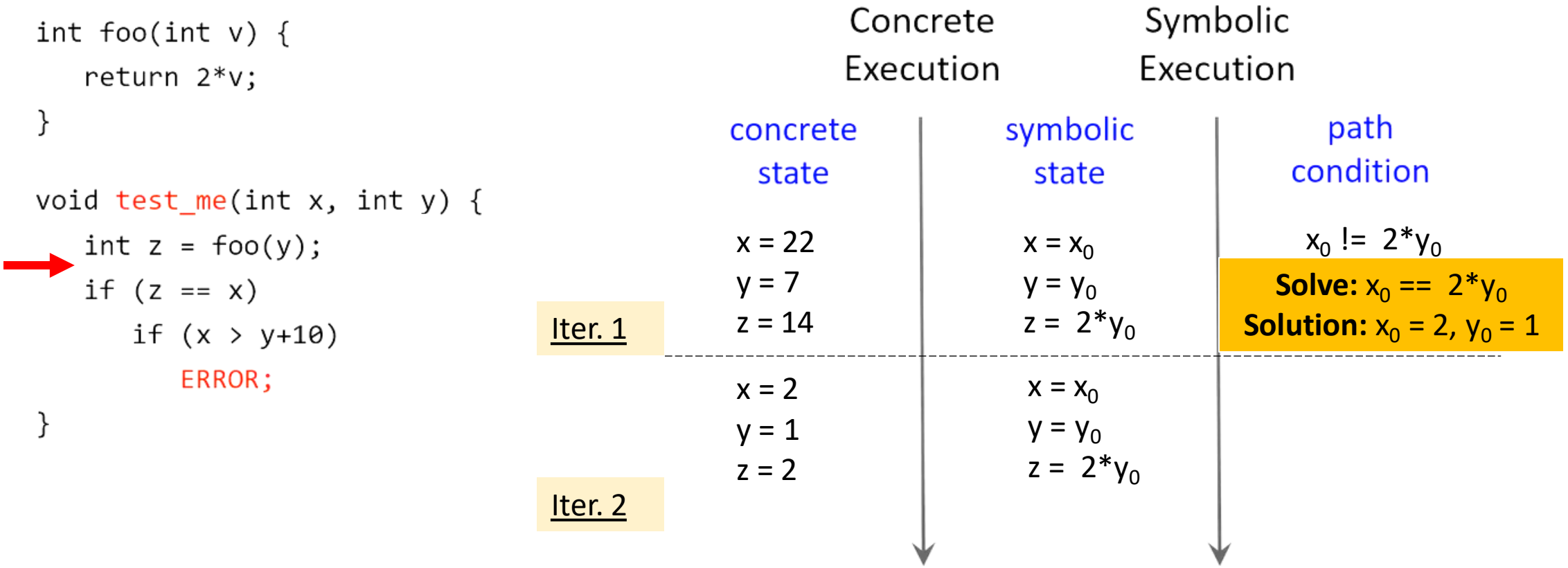
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



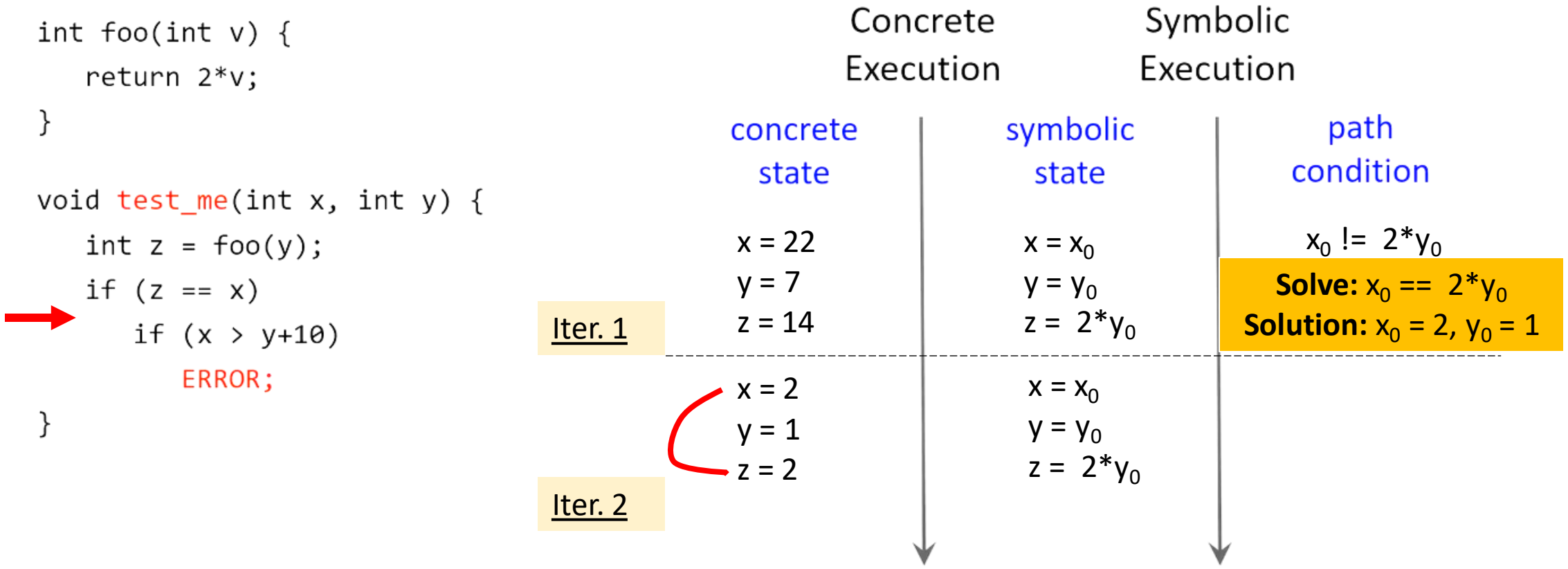
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



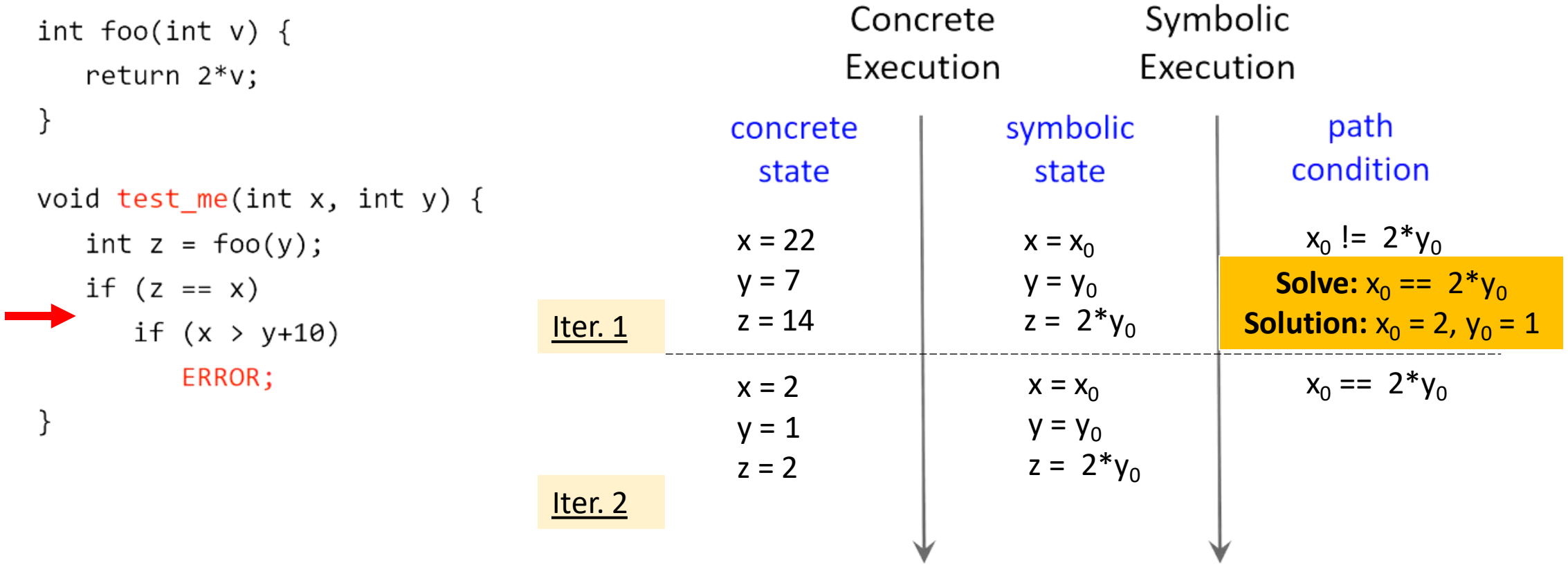
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



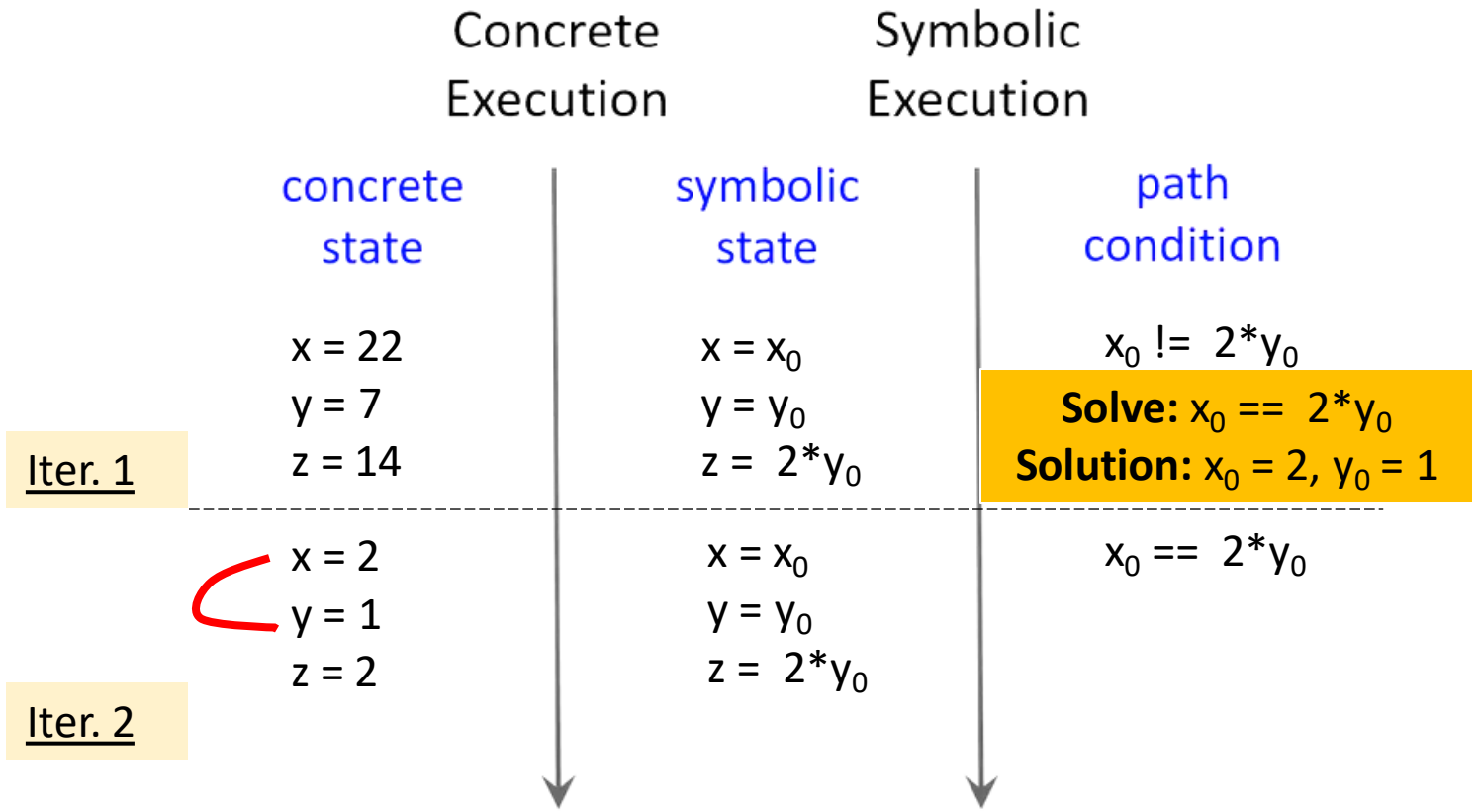
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



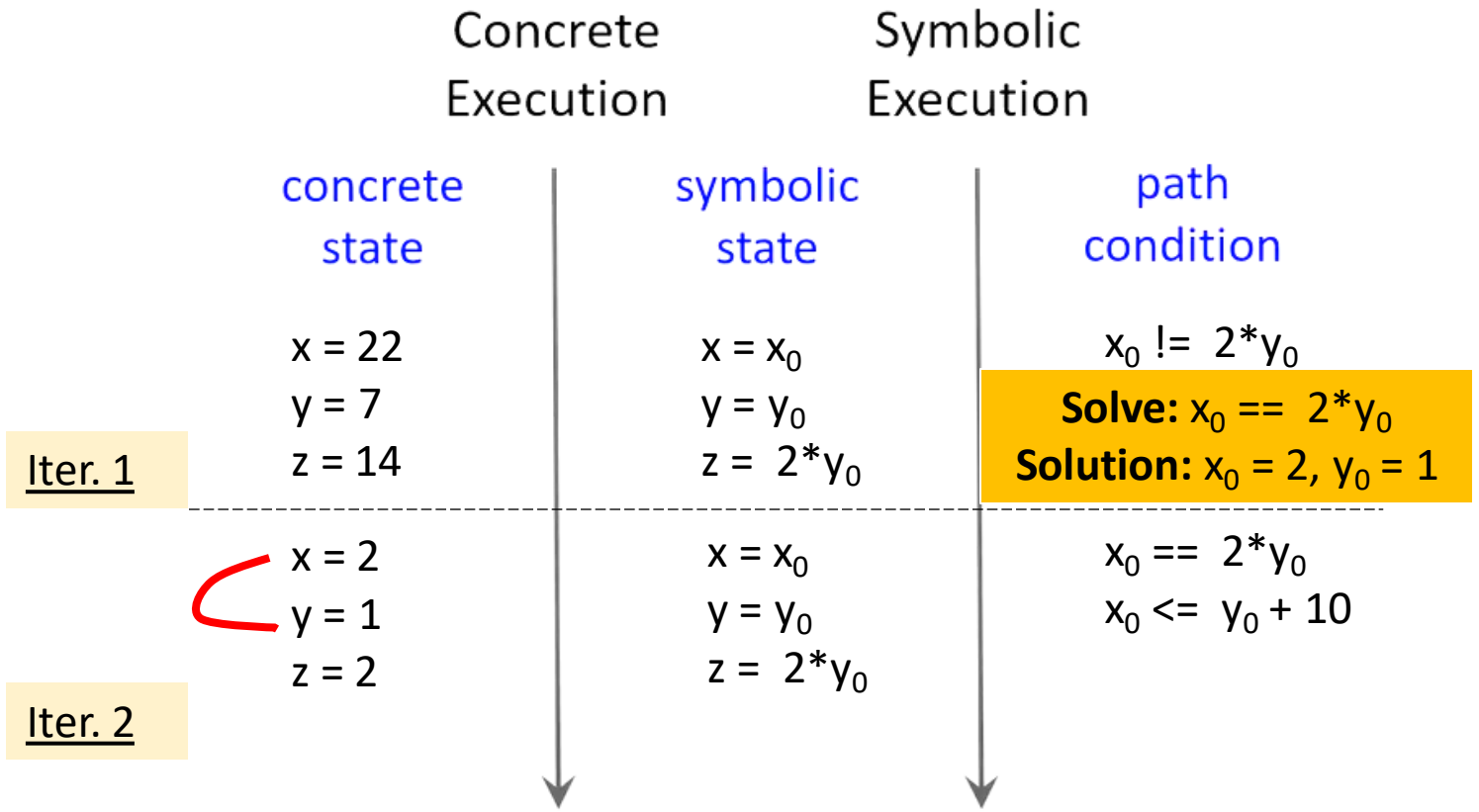
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



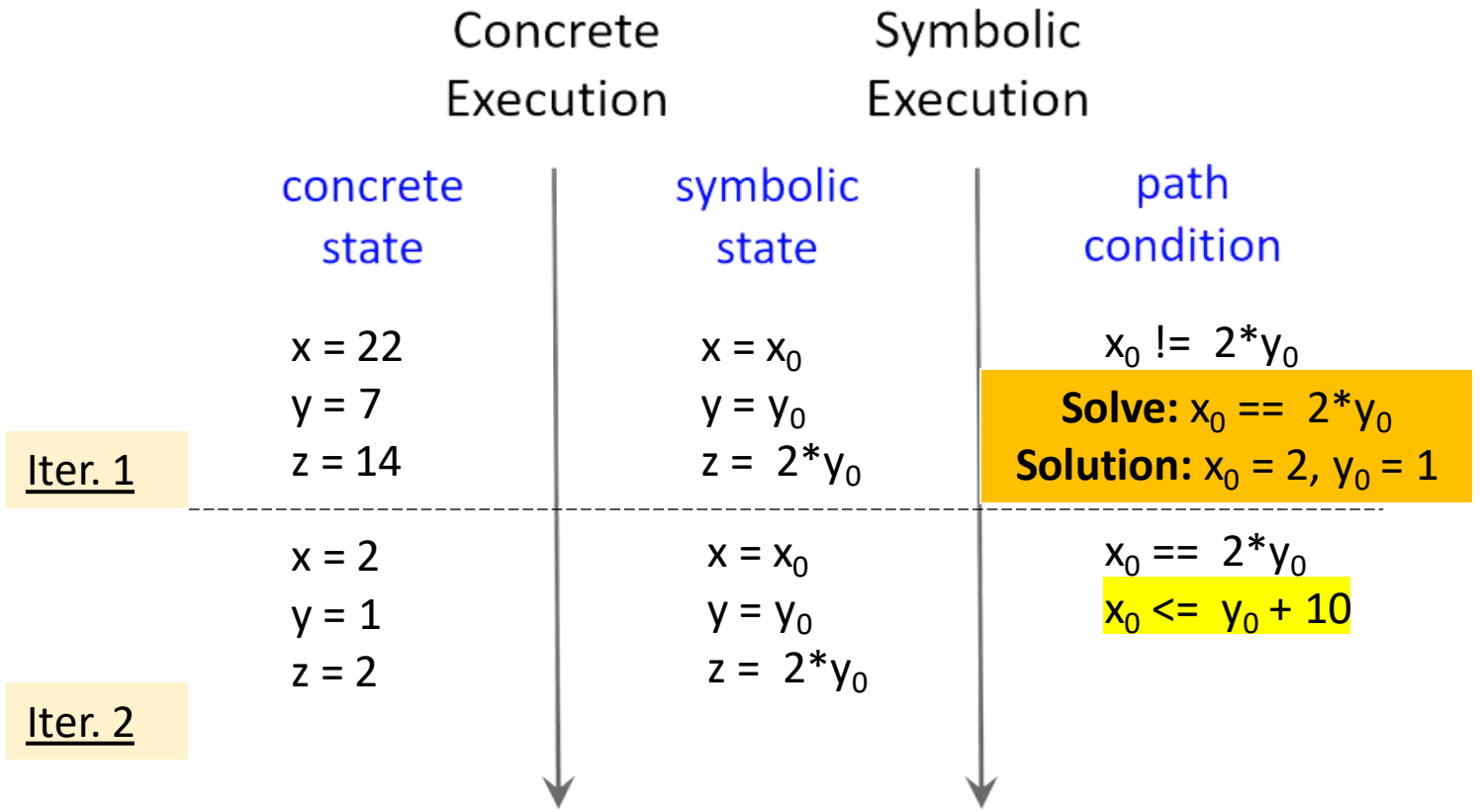
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

	Concrete Execution: concrete state	Symbolic Execution: symbolic state	Symbolic Execution: path condition
Iter. 1	x = 22 y = 7 z = 14	x = x_0 y = y_0 z = $2*y_0$	x_0 != $2*y_0$ **Solve:** x_0 == $2*y_0$ **Solution:** x_0 = 2, y_0 = 1
Iter. 2	x = 2 y = 1 z = 2	x = x_0 y = y_0 z = $2*y_0$	x_0 == $2*y_0$ x_0 <= y_0 + 10

Solve: (x_0 == $2*y_0$) Λ (x_0 > y_0 + 10)

DSE: An Illustrative Example

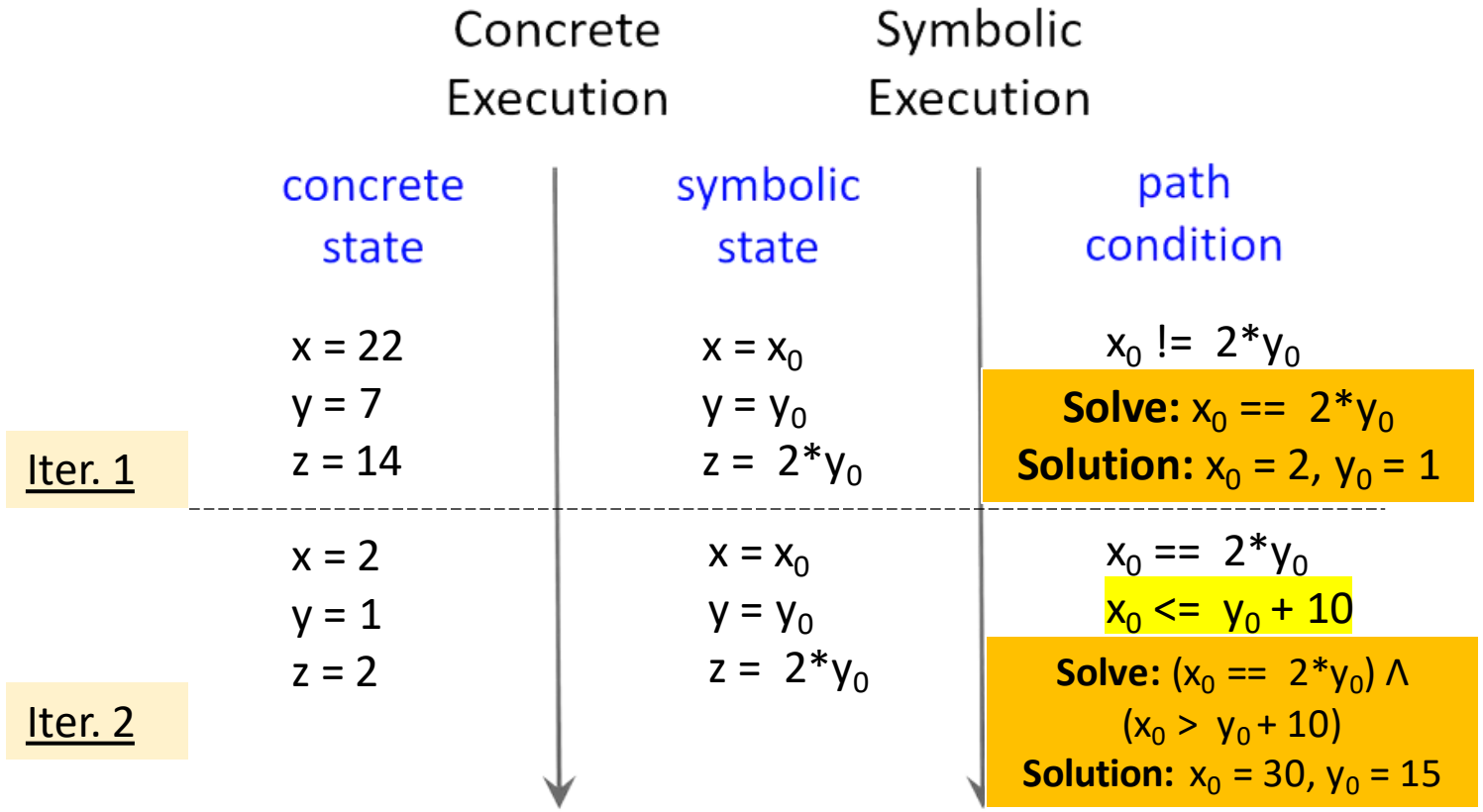
```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

	Concrete Execution: concrete state	Symbolic Execution: symbolic state	Symbolic Execution: path condition
Iter. 1	x = 22 y = 7 z = 14	x = x_0 y = y_0 z = 2*y_0	x_0 != 2*y_0 **Solve:** x_0 == 2*y_0 **Solution:** x_0 = 2, y_0 = 1
Iter. 2	x = 2 y = 1 z = 2	x = x_0 y = y_0 z = 2*y_0	x_0 == 2*y_0 x_0 <= y_0 + 10

Solve: (x_0 == 2*y_0) Λ (x_0 > y_0 + 10)
Solution: x_0 = 30, y_0 = 15

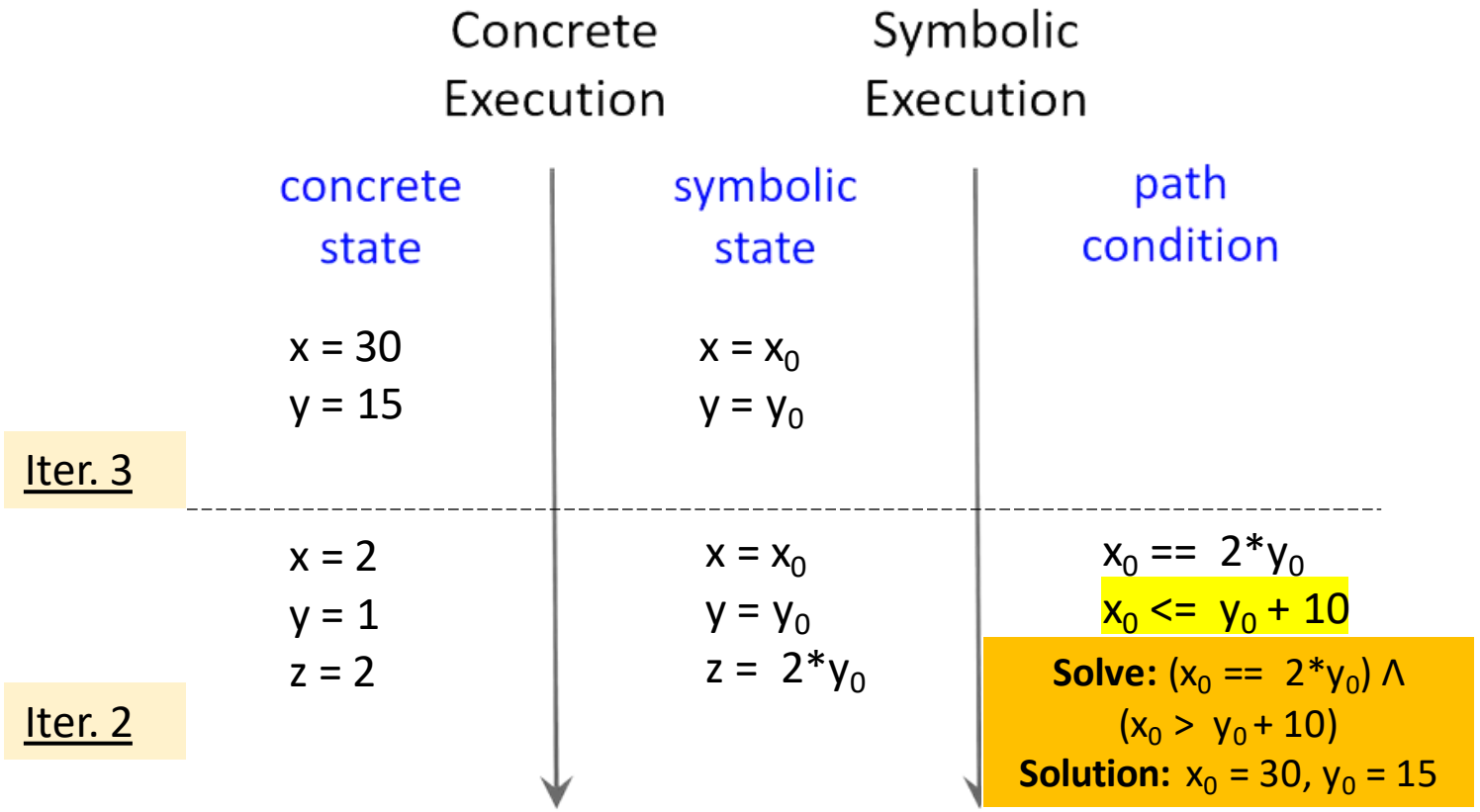
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



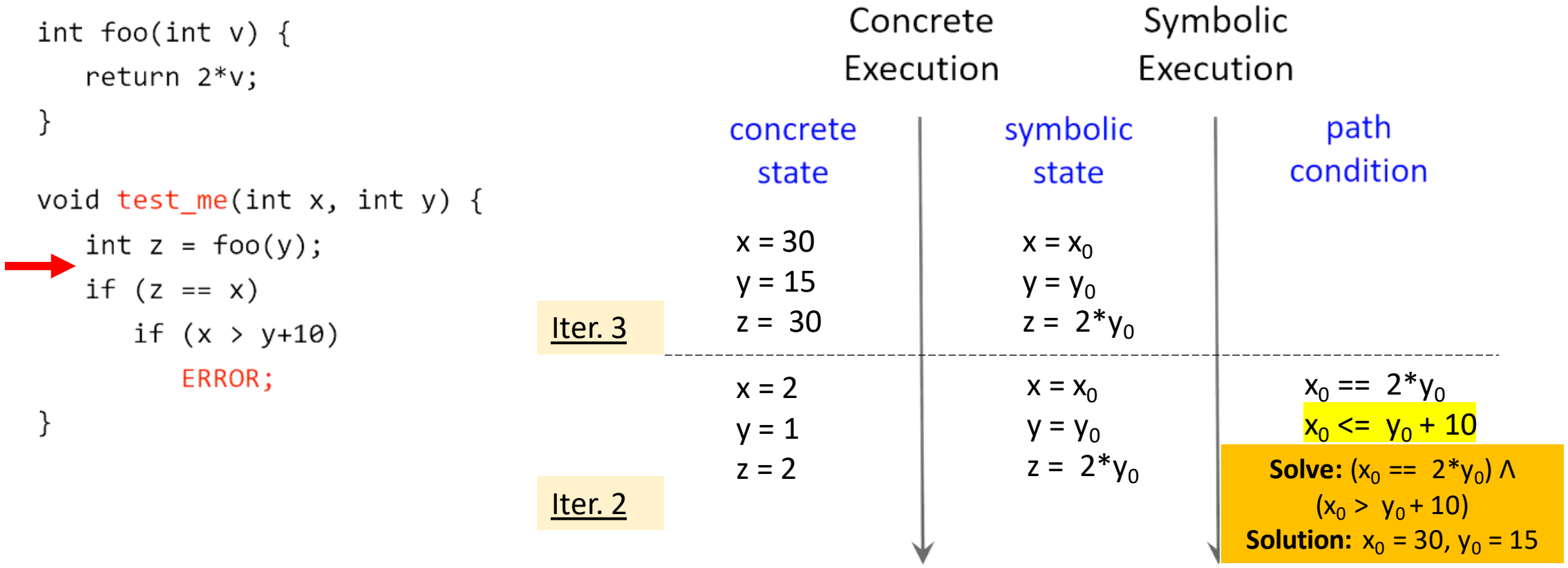
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



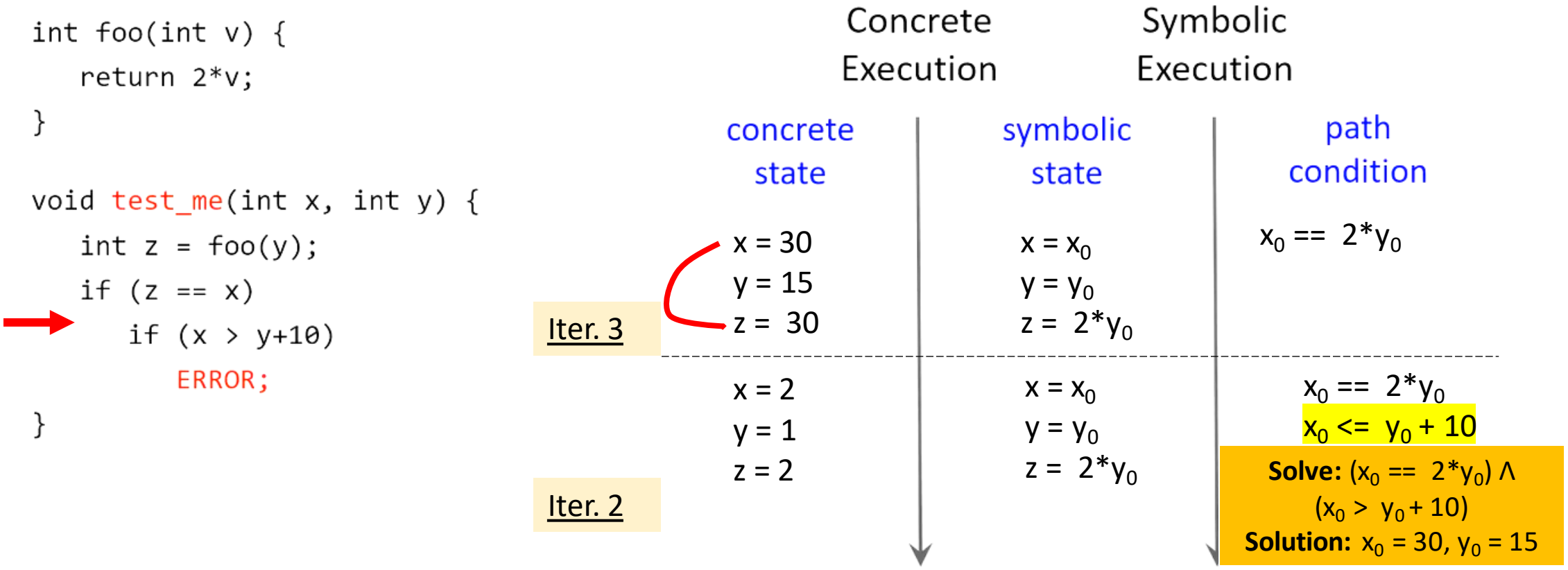
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



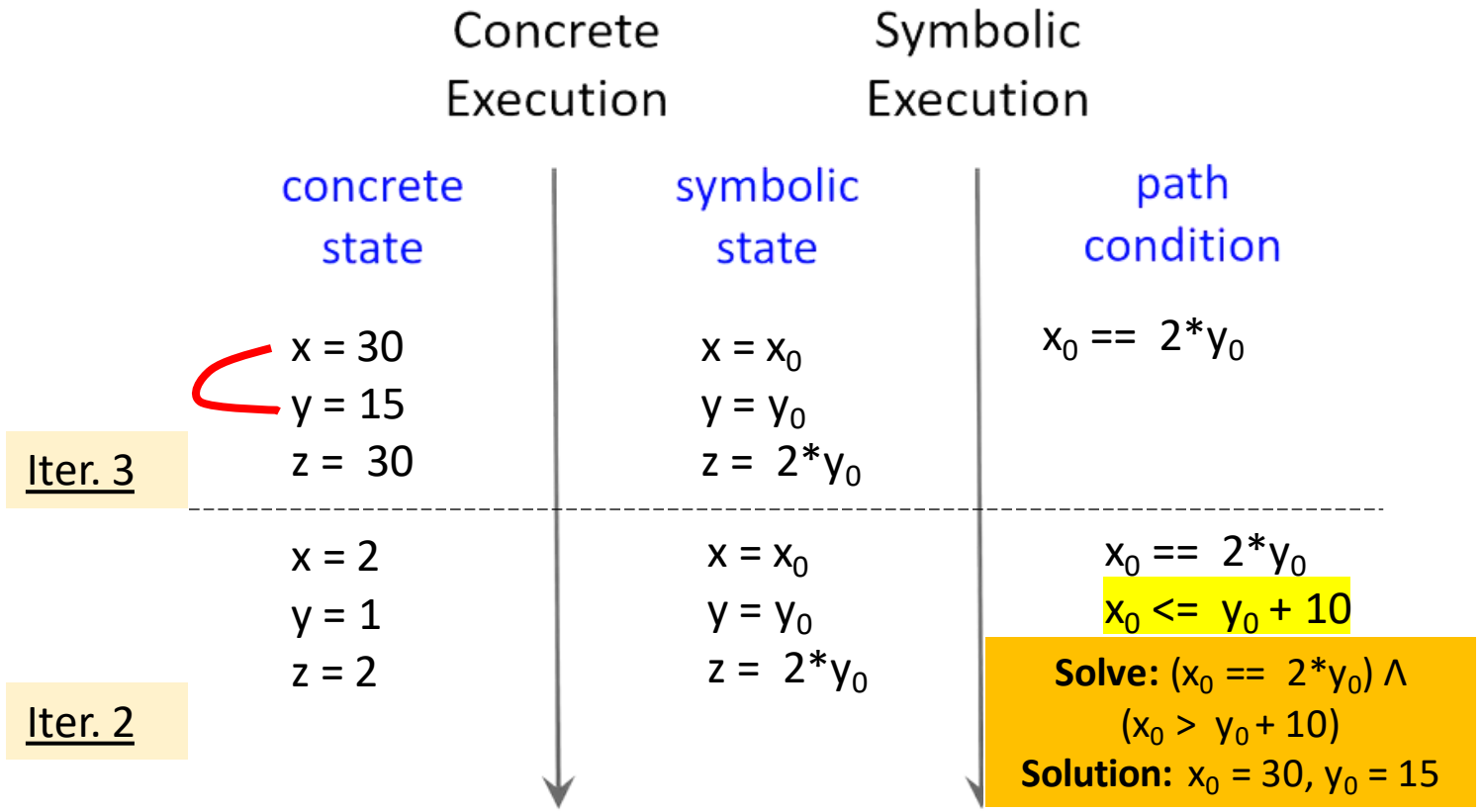
DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



DSE: An Illustrative Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

	Concrete Execution: concrete state	Symbolic Execution: symbolic state	path condition
Iter. 3	x = 30 y = 15 z = 30	x = x_0 y = y_0 z = 2*y_0	x_0 == 2*y_0 x_0 > y_0 + 10
Iter. 2	x = 2 y = 1 z = 2	x = x_0 y = y_0 z = 2*y_0	x_0 == 2*y_0 x_0 <= y_0 + 10

Solve: (x_0 == 2*y_0) ∧ (x_0 > y_0 + 10)
Solution: x_0 = 30, y_0 = 15

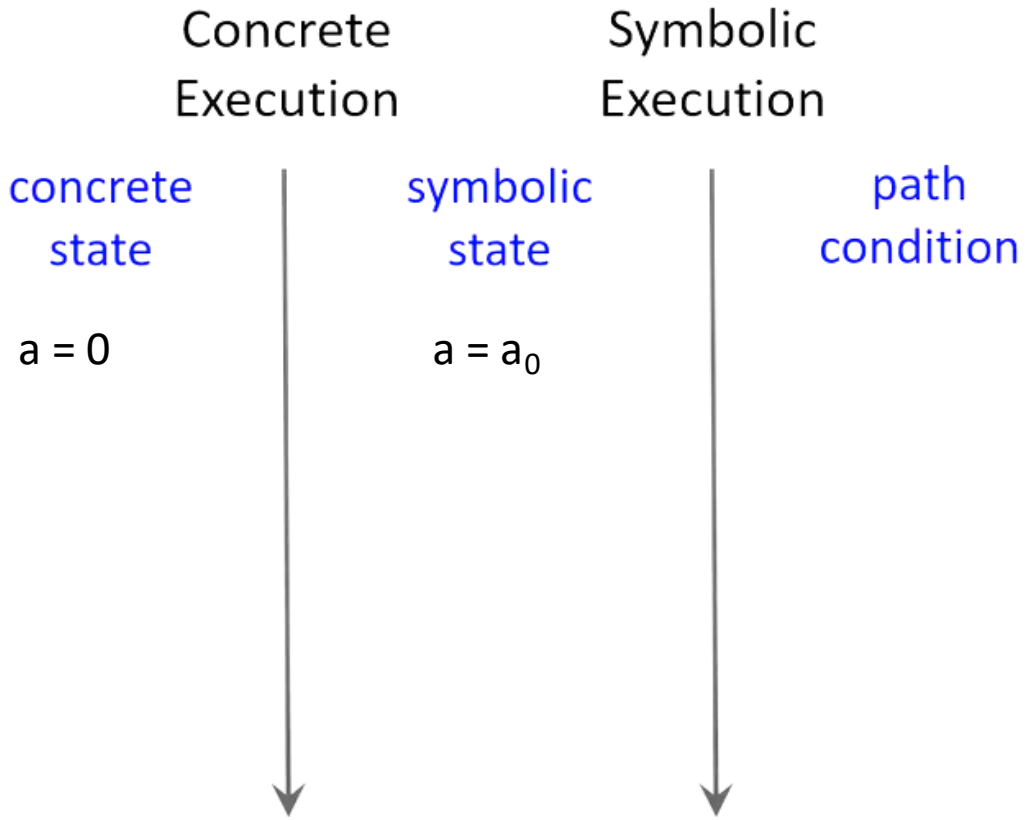
DSE Algorithm

Repeat until all paths are covered

- Execute program with concrete input ***i*** and collect symbolic constraints at branch points: ***C***
- Negate one constraint to force taking an alternative branch ***b'***: Constraint ***C'***
- Call constraint solver to find solution for ***C'***: new concrete input ***i'***
- Execute program with input ***i'*** to take branch ***b'***
- Check at runtime that ***b'*** is indeed taken otherwise: "divergent execution"

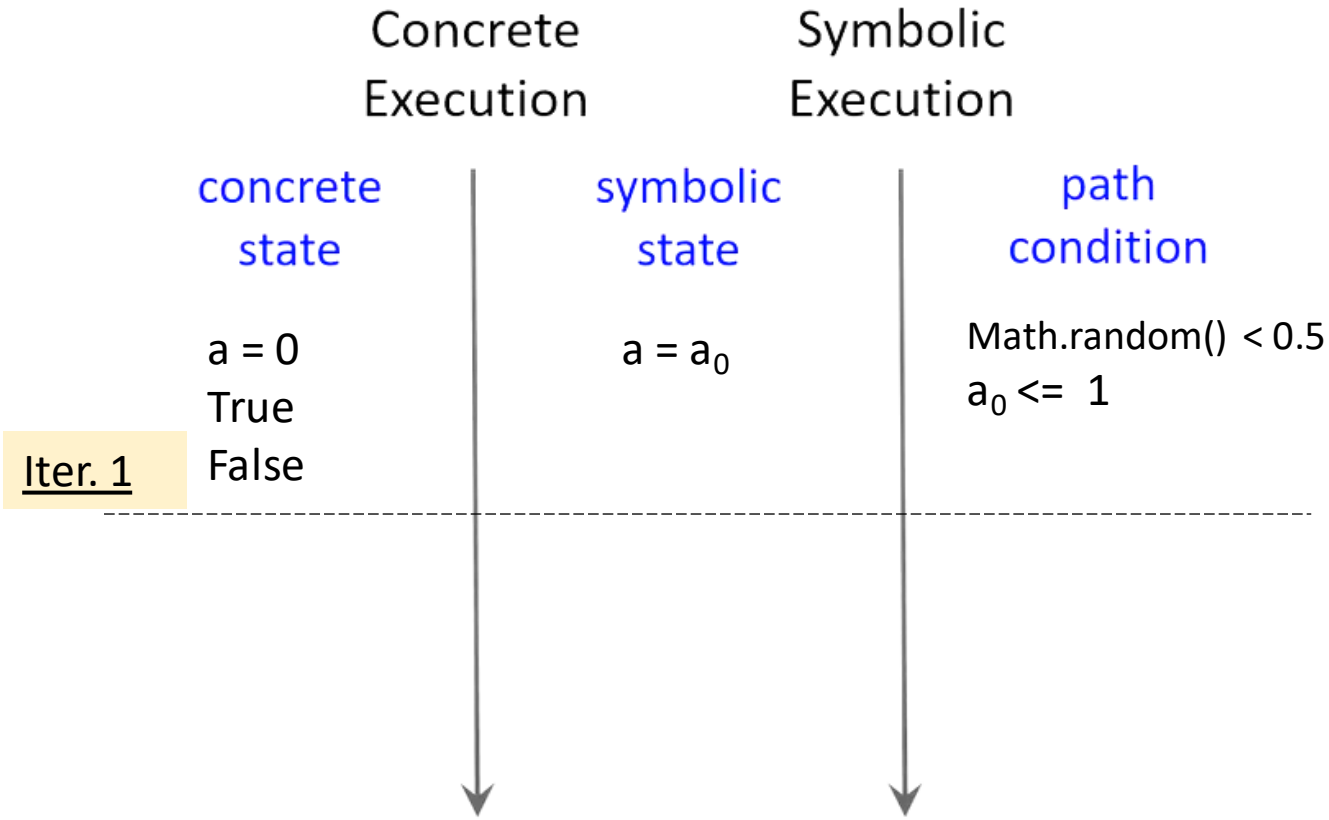
Divergent Execution Example

```
function f(a){
    if (Math.random() < 0.5){
        if (a > 1) {
            console.log
        }
    }
}
```



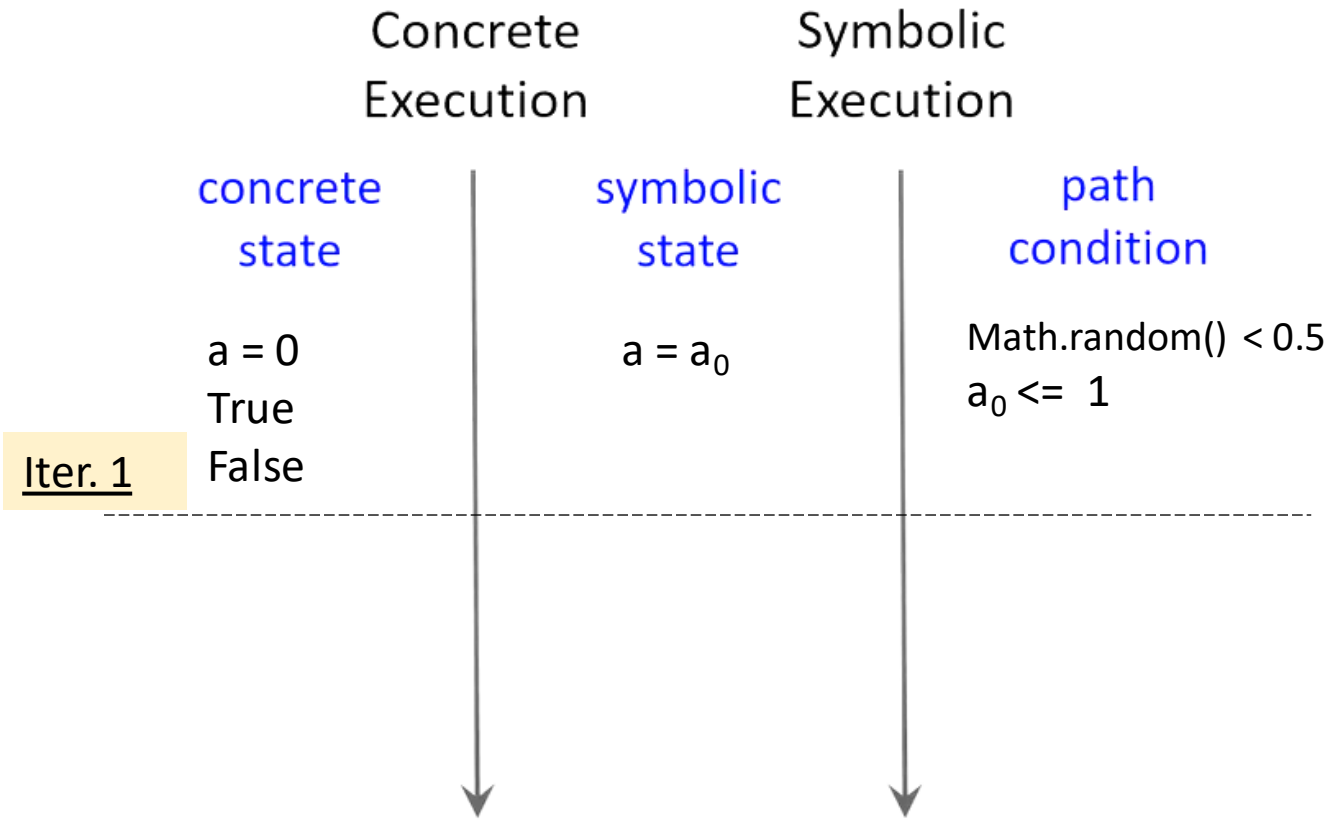
Divergent Execution Example

```
function f(a){
    if (Math.random() < 0.5){
        if (a > 1) {
            console.log
        }
    }
}
```



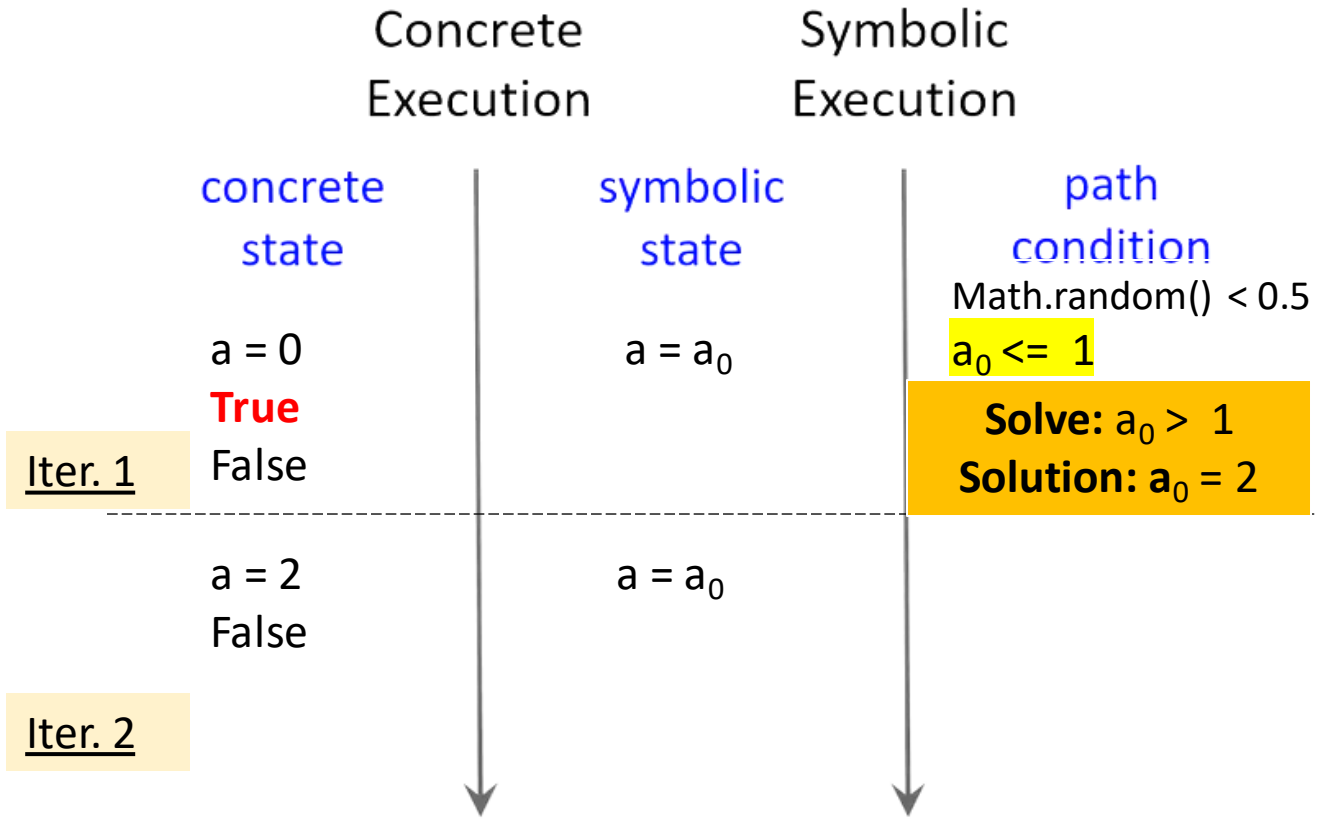
Divergent Execution Example

```
function f(a){
   if (Math.random() < 0.5){
      if (a > 1) {
            console.log
      }
   }
}
```

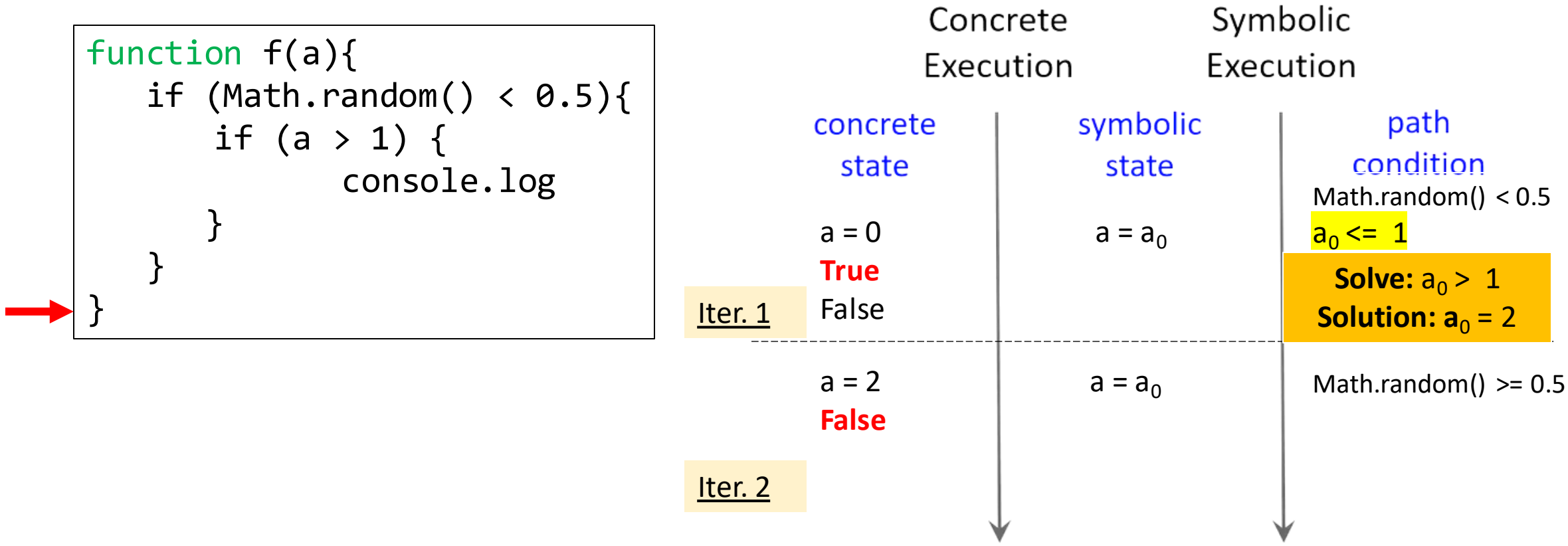
Divergent Execution Example

```
function f(a){
   if (Math.random() < 0.5){
      if (a > 1) {
            console.log
      }
   }
}
```



Divergent Execution Example

```
function f(a){
    if (Math.random() < 0.5){
        if (a > 1) {
            console.log
        }
    }
}
```



Benefits of Concolic Approach

When symbolic reasoning is impossible or impractical, fall back to concrete values

- Native/system API functions
- Operations not handled by solvers (e.g., floating point operations)

Implementations of Concolic Approach

- KLEE: LLVM (C family of languages – C, C++, Objective C, and Objective C++)
- PEX: .NET framework
- jCUTE: Java programs
- Jalangi: JavaScript programs
- SAGE and S2E: binaries (x86, ARM, …)

Implementations of Concolic Approach

SAGE: binaries (x86, ARM, …)

- Scalable Automated Guided Execution – developed at MSR
- Used by multiple product teams at Microsoft, run daily on cloud
- Found many security bugs in various Microsoft applications (MS Windows, MS Office, Media player, Bing, etc.)
- What makes it so useful?
  - Works on large applications => find bugs in multiple components
  - Focuses on input file fuzzing => makes it fully automated
  - Works on binaries => easy to deploy (does not depend on programming language or build process)

QUIZ (1/2): Properties of DSE

Assume that programs can have infinite execution trees. Which of the following statements are true of DSE applied to such programs?

1. DSE is guaranteed to terminate
2. DSE is complete: if it ever found an error, the program can trigger that error in some concrete execution
3. DSE is sound: if it terminates and did not find an error, the program, cannot trigger an error in any concrete execution

QUIZ (1/2): Properties of DSE

Assume that programs can have infinite execution trees. Which of the following statements are true of DSE applied to such programs?

1. DSE is guaranteed to terminate
2. ✓ DSE is complete: if it ever found an error, the program can trigger that error in some concrete execution
3. DSE is sound: if it terminates and did not find an error, the program, cannot trigger an error in any concrete execution

QUIZ (2/2): Properties of DSE

1. The testing approach of DSE is:
   (a) Automated, black-box (b) Automated, white-box
   (c) Manual, black-box (d) Manual, white-box

2. The input search of DSE is:
   (a) Randomized (b) Systematic

3. The static analysis of DSE is:
   (a) Flow-insensitive (b) Flow-sensitive (c) Path-sensitive

QUIZ (2/2): Properties of DSE

1. The testing approach of DSE is:
   (a) Automated, black-box (b) Automated, white-box ✓
   (c) Manual, black-box (d) Manual, white-box

2. The input search of DSE is:
   (a) Randomized (b) Systematic ✓

3. The static analysis of DSE is:
   (a) Flow-insensitive (b) Flow-sensitive (c) Path-sensitive ✓

Test Generation: The Bigger Picture

Why didn't automatic test generation become popular decades ago?

- Weak-type systems
  - Test generation relies heavily on type information
  - C, Lisp just didn't provide the needed types
- Contemporary languages lend themselves better to test generation
  - Java, UML
- Lack of computational power required (e.g., modern NLP techniques, SMT solvers)

What Have We Learned?

- Automatic test generation is a good idea
  - **Key**: avoid generating illegal and redundant tests and cover more parts of the program
  - Performs better for generating unit tests in strongly-typed languages
- Generating test oracles is harder than generating test inputs
- Being adopted in industry
  - Likely to become widespread in the future

Announcements

- Project Plan Presentation
  Due **next Wednesday, 02/14/2024 before class time**
  - Describe **exactly** what you have decided to do in project

- Paper Presentation
  Starts **in 1.5 weeks, Monday, 02/19/2024**
  - Groups presenting papers 1 and 3 should get feedback from me next week Wednesday during my office hours
  - In general, each group should get feedback on their presentation during my office hours in the week before you present.