

CS1632: Static Analysis, Part 2

Wonsun Ahn

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- Linters
- Bug finders
- **Formal verification**

Formal Verification

- Proving one or the other about a program:
 - Program has no defect
 - Program has defects (and find all of them)
- What!?



... with some caveats 😊

Methods of Formal Verification

- Theorem Proving

- Deducing postcondition from precondition through math

Caveat

- Model Checking

- Given a finite state model of a system, exhaustively checking all the states to see if model meets a given specification

Caveat

Theorem Proving

Deducing postcondition from precondition through math

Hoare Logic Theorem Proving

- Hoare Logic: Proves a Hoare Triplet through mathematical deduction
- Hoare Triplet: {Precondition} Program {Postcondition}
 - Meaning: Given Precondition and Program, Postcondition is always true
- Examples of Hoare Triplets:
 - { true } $x = 5$ { $x == 5$ }
 - { $x == y$ } $x = x + 3$ { $x == y + 3$ }
 - { $x == a$ } if ($x < 0$) then $x = -x$ { $x == |a|$ }
 - { $x < 0$ } while ($x != 0$) $x = x - 1$ { Infinite loop } \leftarrow No such triple!
- Proves increasingly complex triplets through composition

Hoare Logic Syntax

English	Formal
false	\perp
true	\top
not p	$\neg p$
p and q	$p \wedge q$
p or q	$p \vee q$
p implies q	$p \Rightarrow q$
p iff q	$p \Leftrightarrow q$
for all x , p	$\forall x. p$
there exists x such that p	$\exists x. p$

- Idea is to use this syntax to prove with pen and paper your program is correct
- Sounds unappealing? 😞
 - Many programmers would agree!
- There exist “theorem provers” that automate mundane parts of proving
 - But needs human assistance at difficulties
 - Example difficulty: reasoning about recursive data structures (lists, trees, ...)

Theorem Proving Advantages

- Can prove large programs with infinite states
 - Remember this Hoare triplet?
`{ x < 0 } while (x!=0) x := x-1 { Infinite loop }`
 - Model checker will have trouble because it has an infinite number of states
 - But a human or machine theorem prover can tell there is an infinite loop!
- Leads programmer to a deeper understanding of the program
 - After spending weeks proving the program is correct, a natural outcome
 - But really, it does lead to some fundamental insights about your program

Theorem Proving Disadvantages

- Requires (a lot of) human involvement
 - Every time a theorem prover encounters difficulty humans have to step in
 - Requires many hours of highly skilled labor to complete a proof
 - Humans also make mistakes
- Proofs can be obscenely long
 - In one report by Motorola, a proof was 25 MB long (more than 100 pages)
 - Beyond the comprehension limits of a normal human being

Industry Reception

- Advocates want a “formal methods guru” on every project team
 - The education required to produce a “formal methods guru” is very different from the education of a typical software engineer
- Naturally, industry is resistant
 - Used in niche markets where correctness is paramount
 - Some embedded systems, cryptography libraries, OS kernels (seL4)
- Industry would like a “push button” solution
 - something that Model Checking provides!

Model Checking

Given a finite state model of a system, exhaustively checking whether this model meets a given specification

The Model Checking Problem

- Does **implementation** **satisfy** **specification** ?
- **Implementation** is also called a **system model**.
 - System model can be just your source code
 - Or some abstract model like a state machine
- **Specification** is also called a **system property**.
 - The same "property" in property-based testing

Examples of System Properties

- Assertions (invariants)
 - Embedded in source code or part of property-based unit test
- Memory related properties
 - No out of bounds array accesses
 - No null references
 - No leaks, double-free, access after free (in C/C++)
- Thread related properties
 - No data races when threads access shared data
- Security related properties
 - No write of private data to insecure public channels

Comparison with Property-Based Testing

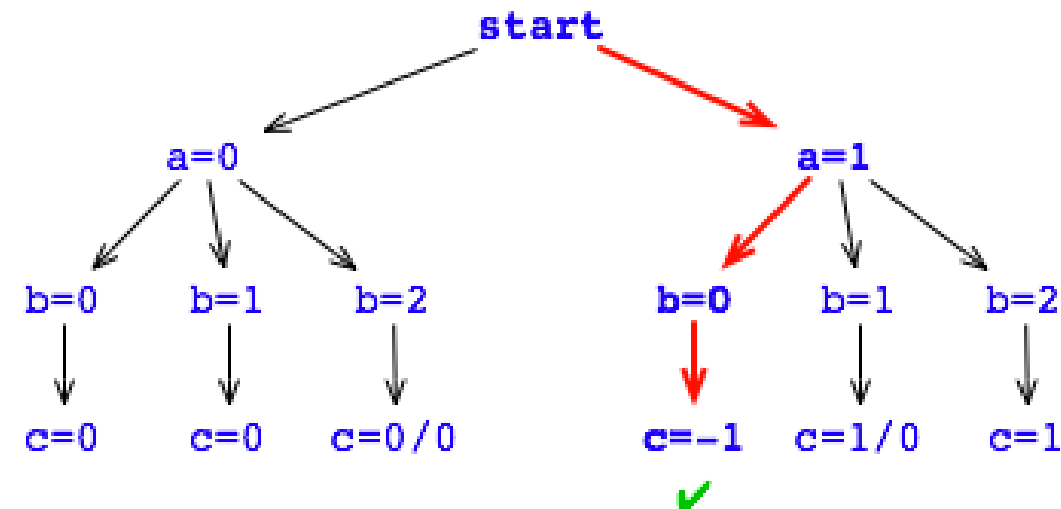
- Similarity
 - Model checking also tests a property, not an output value
- Difference
 - With stochastic testing, we tested (a few) randomized input values
 - With model checking, all states are checked *exhaustively*

Stochastic Testing (on a Single Trial)

Given this code:

```
int a = random.nextInt(2);  
int b = random.nextInt(3);  
int c = a / (b + a - 2);
```

If unlucky and not all paths are covered after all trials, bug may never be found!



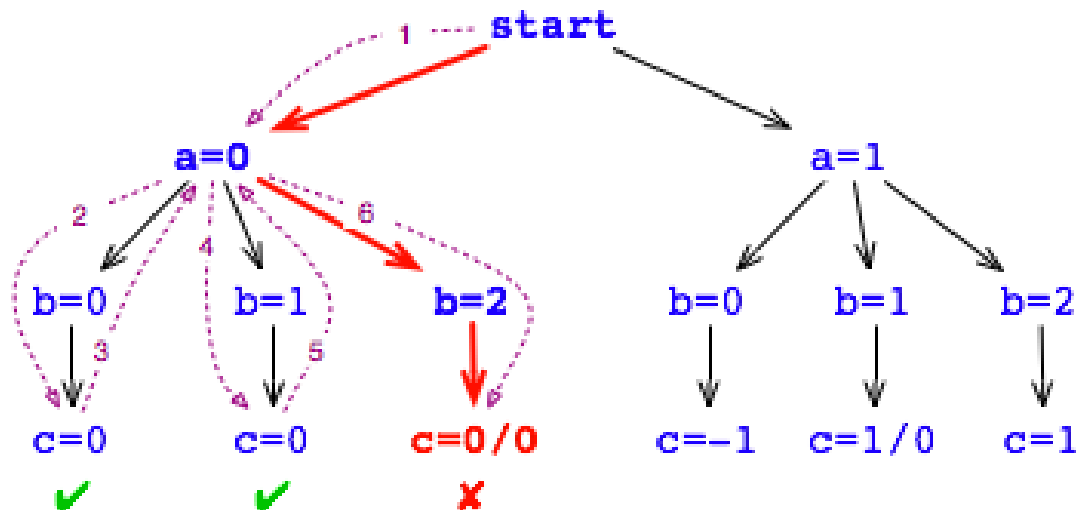
- ① `Random random = new Random();`
- ② `int a = random.nextInt(2);`
- ③ `int b = random.nextInt(3);`
- ④ `int c = a / (b + a - 2);`

Model Checking

Given this code:

```
int a = random.nextInt(2);  
int b = random.nextInt(3);  
int c = a / (b + a - 2);
```

Bug is always found!
(through exhaustive searching)
If none found, guaranteed correct!



① `Random random = new Random();`

② `int a = random.nextInt(2);`

③ `int b = random.nextInt(3);`

④ `int c = a / (b + a - 2);`

State Explosion Problem

- Did you notice? You may end up with *a lot* of states.
- Single reason preventing wide adoption of model checking
 - May run into memory limitations (can't contain entire state graph)
 - May run into time limitations (can't explore entire graph within allotted time)
 - Especially a big problem for sizable programs (> 100,000 lines of code)

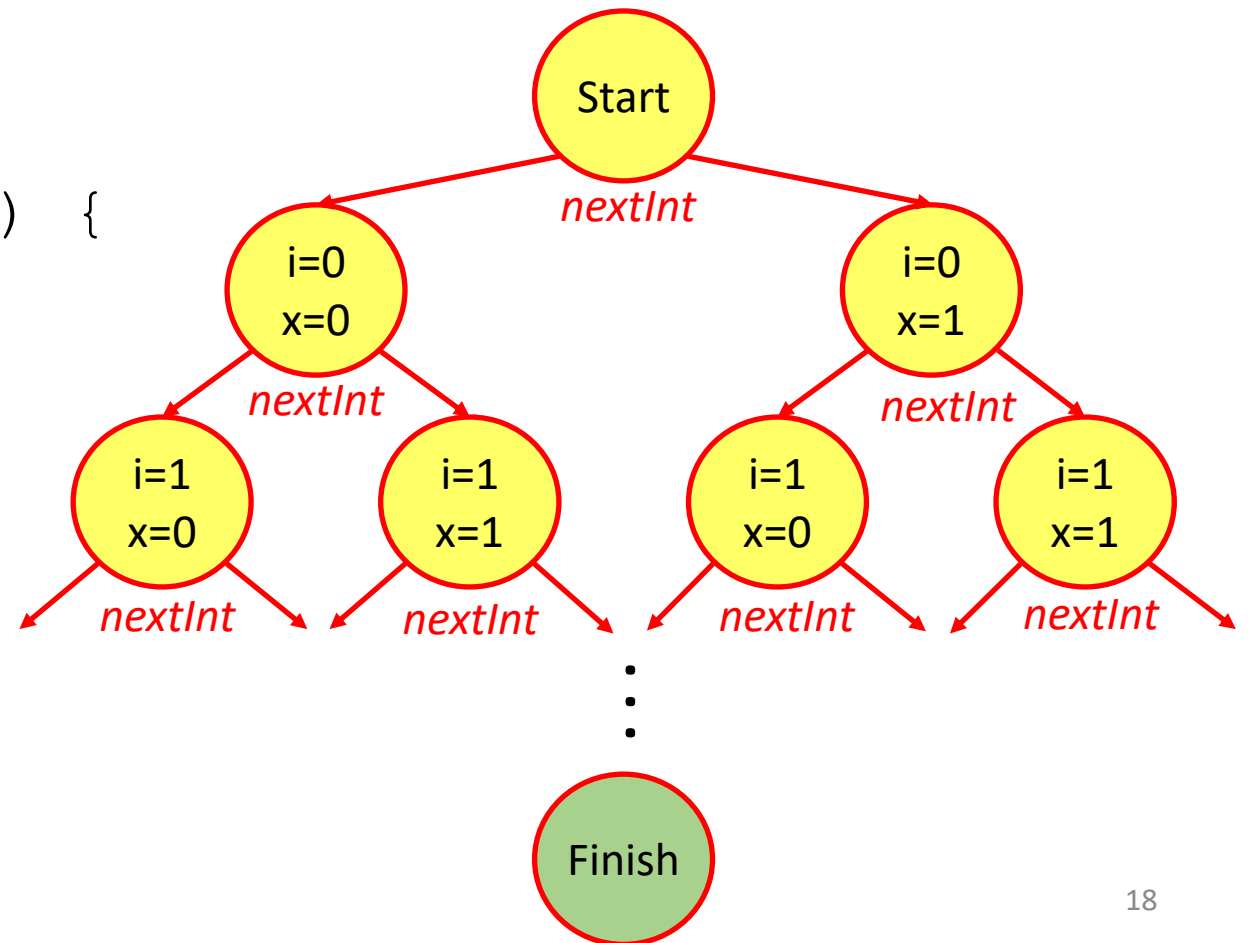
State Explosion Problem: Example 1

- Number of states explodes exponentially with each call to `random.nextInt`

- Given below code:

```
for(int i = 0; i < N; i++) {  
    x = rand.nextInt(2);  
    assert x < 2;  
}
```

- Potential number of states = 2^{N+1}



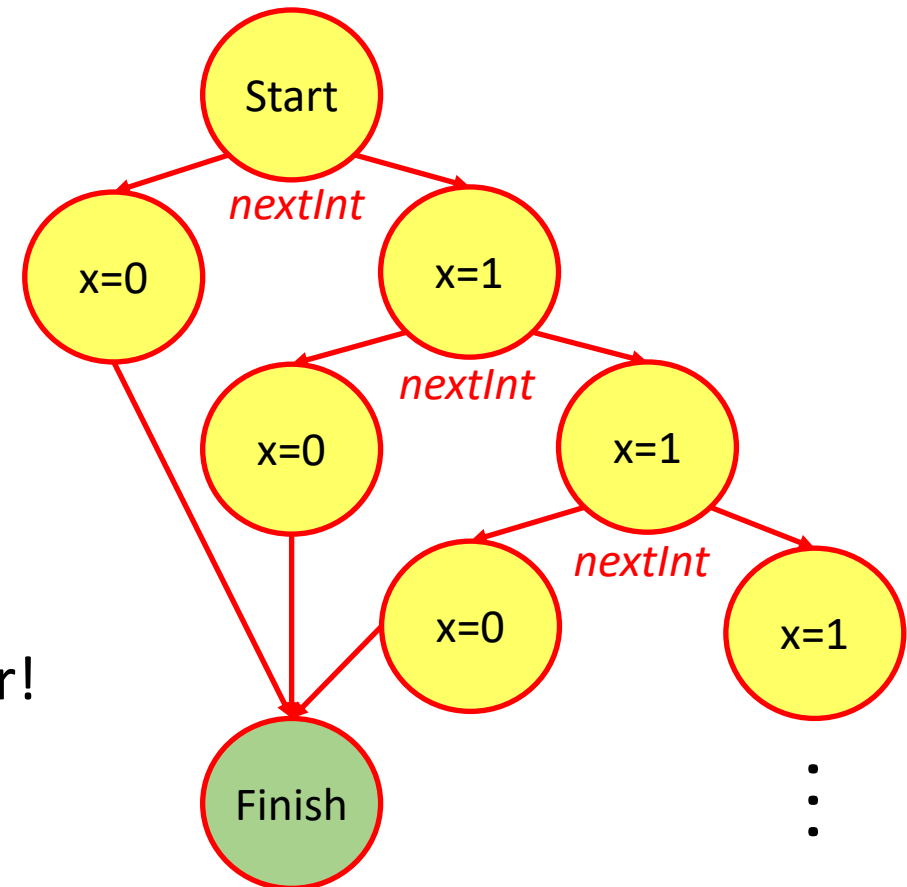
State Explosion Problem: Example 2

- You might even end up with an infinite number of states!

- Given below code:

```
while(true) {
    x = rand.nextInt(2);
    if(x == 0) break;
    assert x < 2;
}
```

- Model checker can potentially go on forever!
 - Will keep creating states to the right

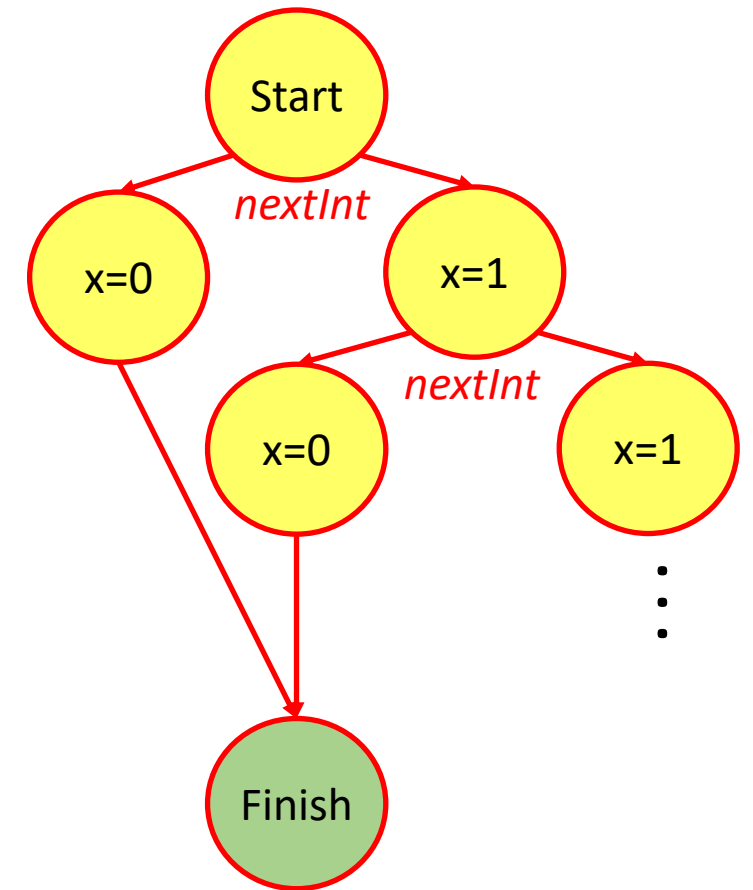


Solution: Match & Backtrack

- If you were paying attention, you might have noticed ...
- Sometimes we encounter identical states already visited:

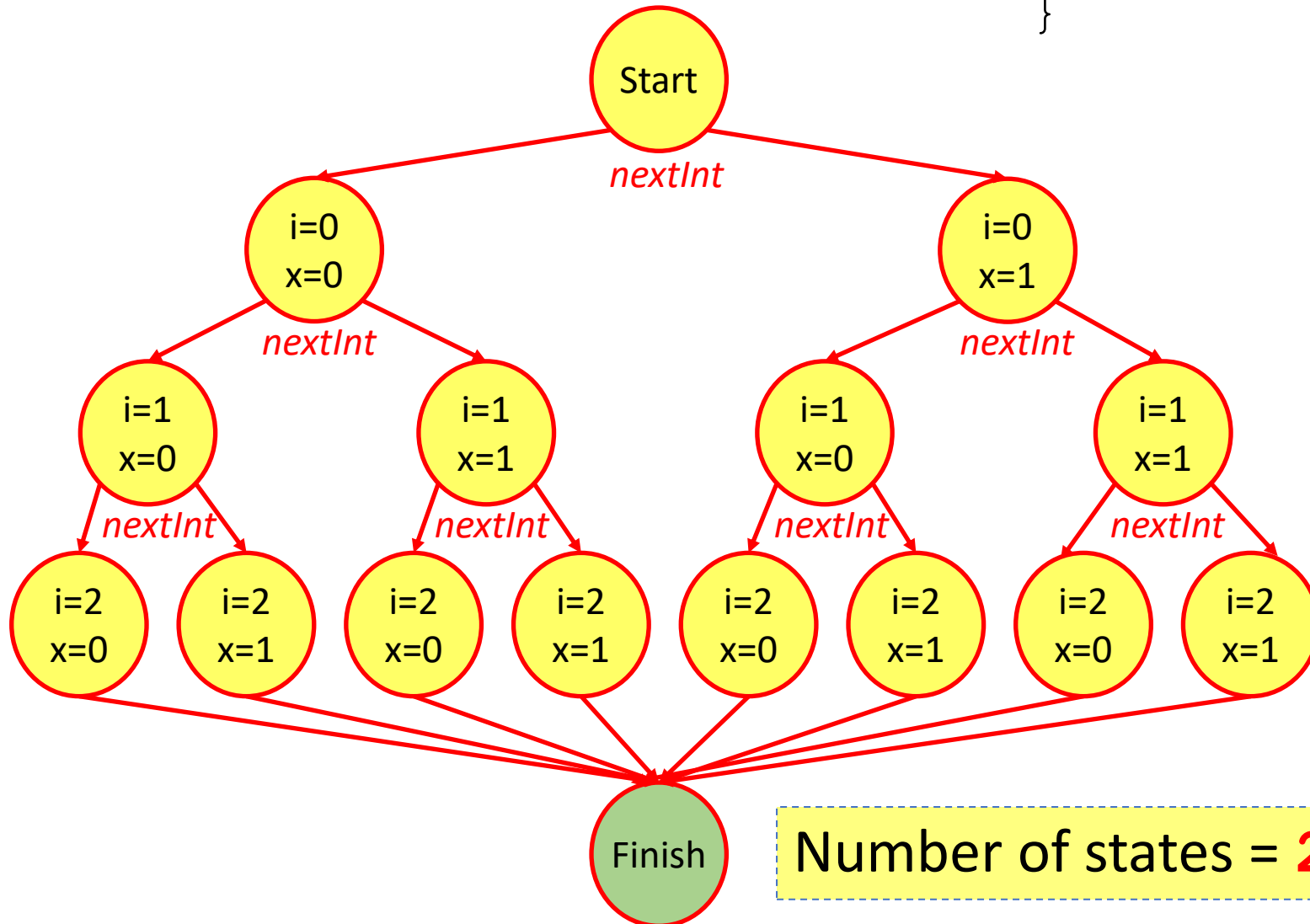
```
while(true) {  
    x = rand.nextInt(2);  
    if(x == 0) break;  
}
```

→ If you do, do not create a new state, backtrack!
- Backtrack: going to closest previous state with unexplored transition
 - When next state matches a previously visited state
 - When you reach the finish state
- Let's assume we visit states in Depth First Search (DFS) order

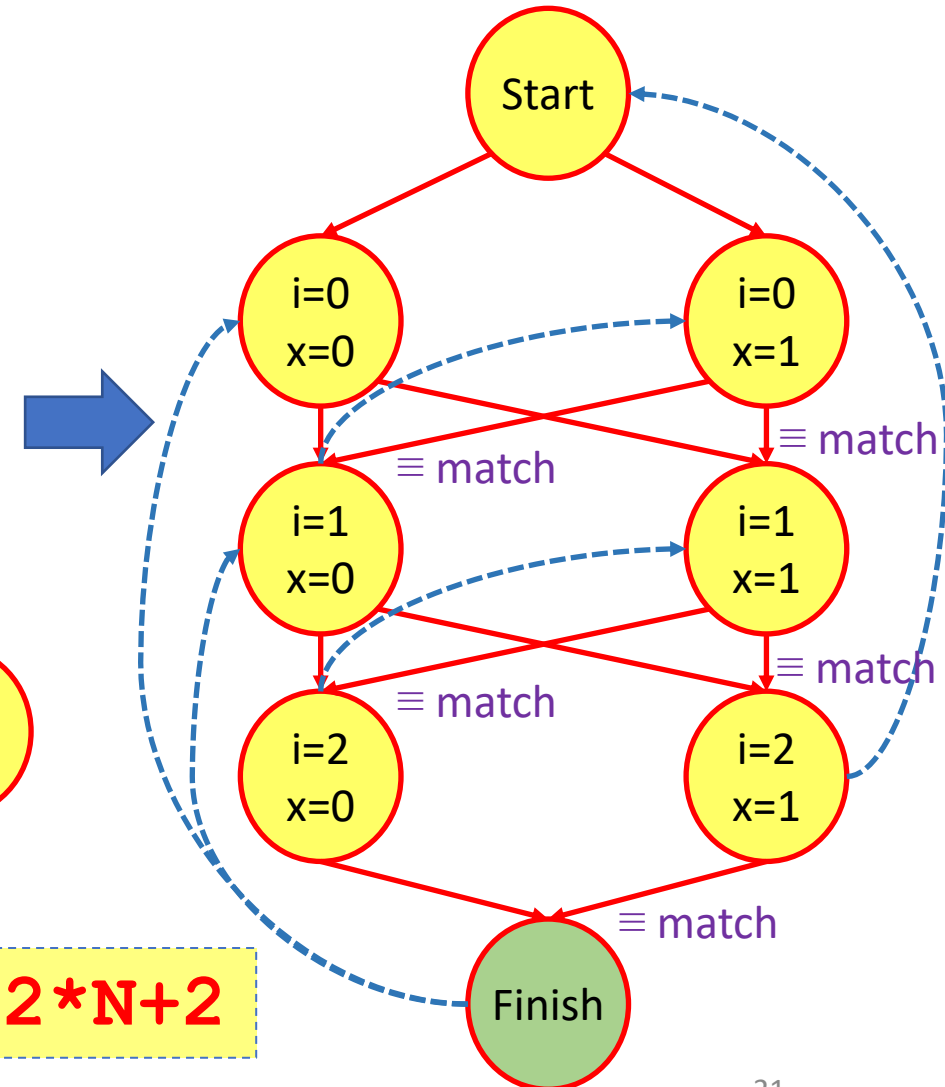


Match & Backtrack on

```
for(int i = 0; i < 3; i++) {  
    x = rand.nextInt(2);  
}
```

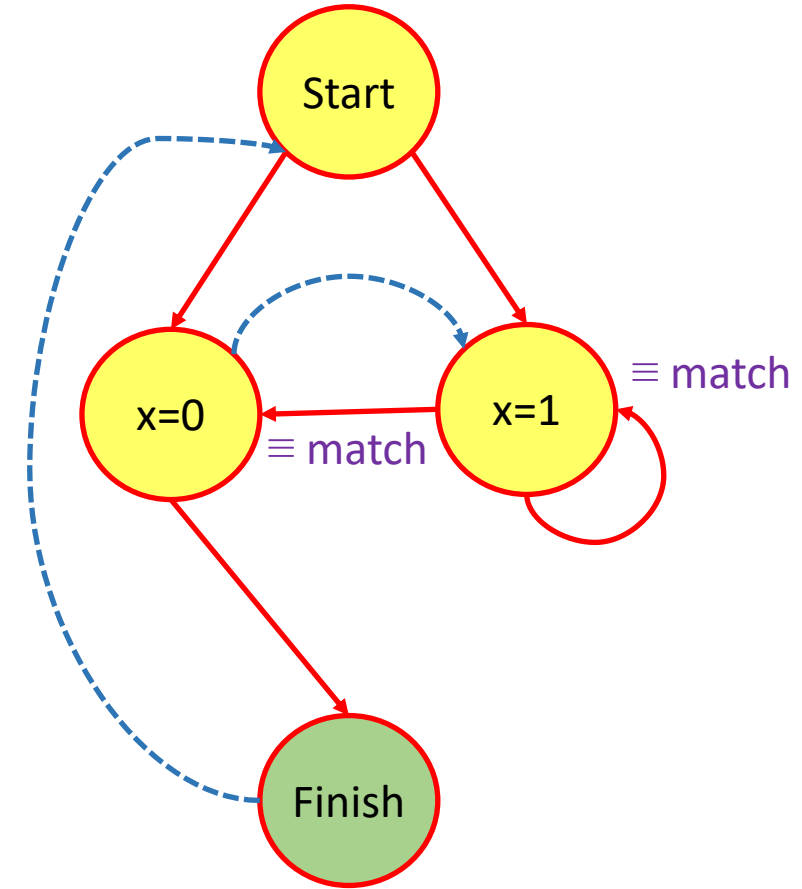
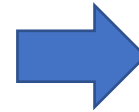
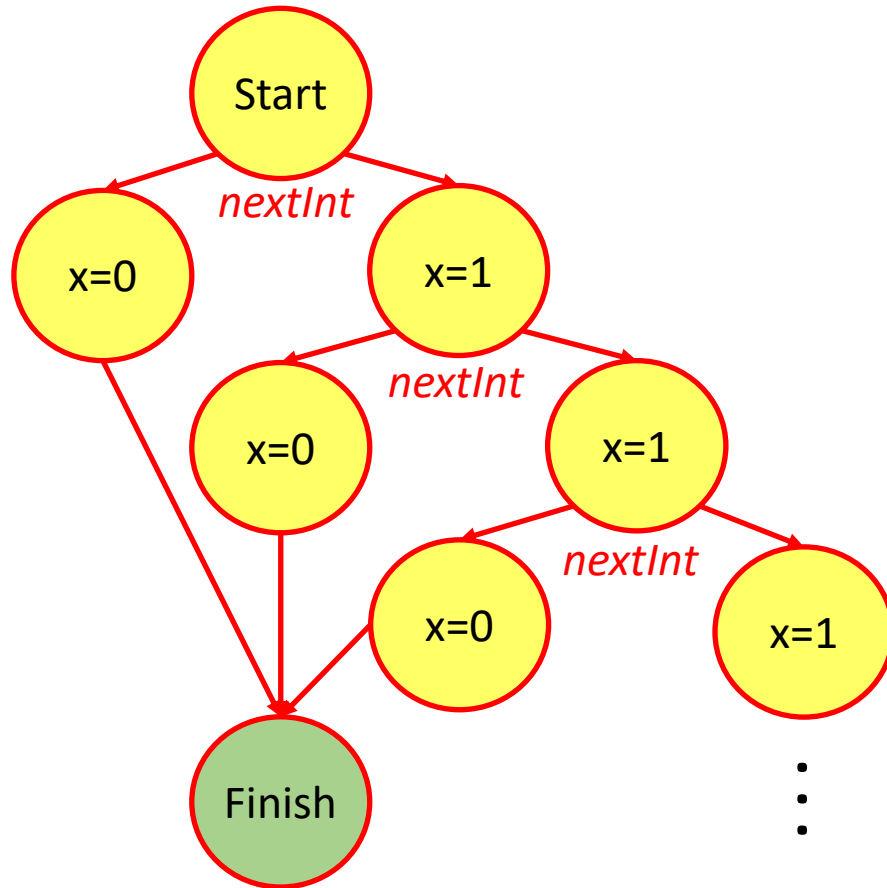


Number of states = $2*N+2$



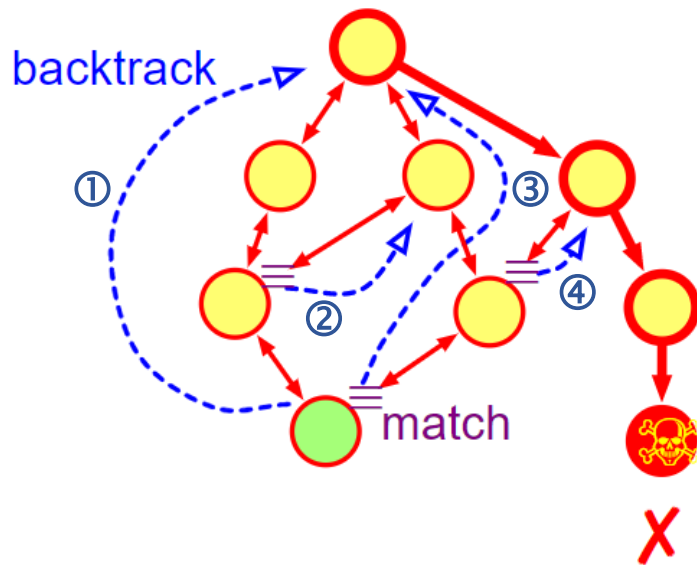
Match & Backtrack on

```
while(true) {  
    x = rand.nextInt(2);  
    if(x == 0) break;  
}
```



Number of states = 4

Efficient State Exploration with Backtracking



Circles: Program states
Arrows: State transitions

- State divergence happens when there is a *choice* (e.g. random number generation)
- **Match**
 - When next state matches a previously visited state
 - Backtrack to not repeat work
- **Backtrack**
 - On reaching terminal state or when there is a match
 - Go to closest previous state with unexplored transition

Efficient State Exploration with Backtracking

```
Hashtable states_seen;
```

```
Stack pending;
```

```
pending.push(initial_state);
```

```
while(!pending.empty()){
```

```
    current = pending.pop();
```

```
    if(current in states_seen)
```

```
        continue; // match! Backtrack.
```

```
    check current for correctness;
```

```
    states_seen.insert(current);
```

```
    for transition T in current {
```

```
        successor = execute transition T on current;
```

```
        pending.push(successor);
```

```
    }
```

```
}
```


Still, State Space can be Infinite (or Too Big)

- Even with backtracking, state space can be infinite (or too big)
- Cannot prove program correct if checker does not terminate in time
 - But can still find many defects in the process!
- Checker tries to test as many behaviors as possible (within given time)
 - By prioritizing states that test new behaviors, using heuristics
 - That way, more likely to find defects

What is Program State?

- Definition of program state depends on programming language
 - What is the state that your program can access and modify?
- C/C++ programs: essentially your entire memory space!
- Java programs: abstract state maintained by the Java Virtual Machine
 - JVM maintains a stack (for local variables) and a heap (for objects)
 - That state is *much* smaller than your entire memory space
- Choice of programming language is important for model checking too
 - As well as for compiler static analysis
 - Memory managed languages like Java or C# tend to fair better

Java Path Finder (JPF)

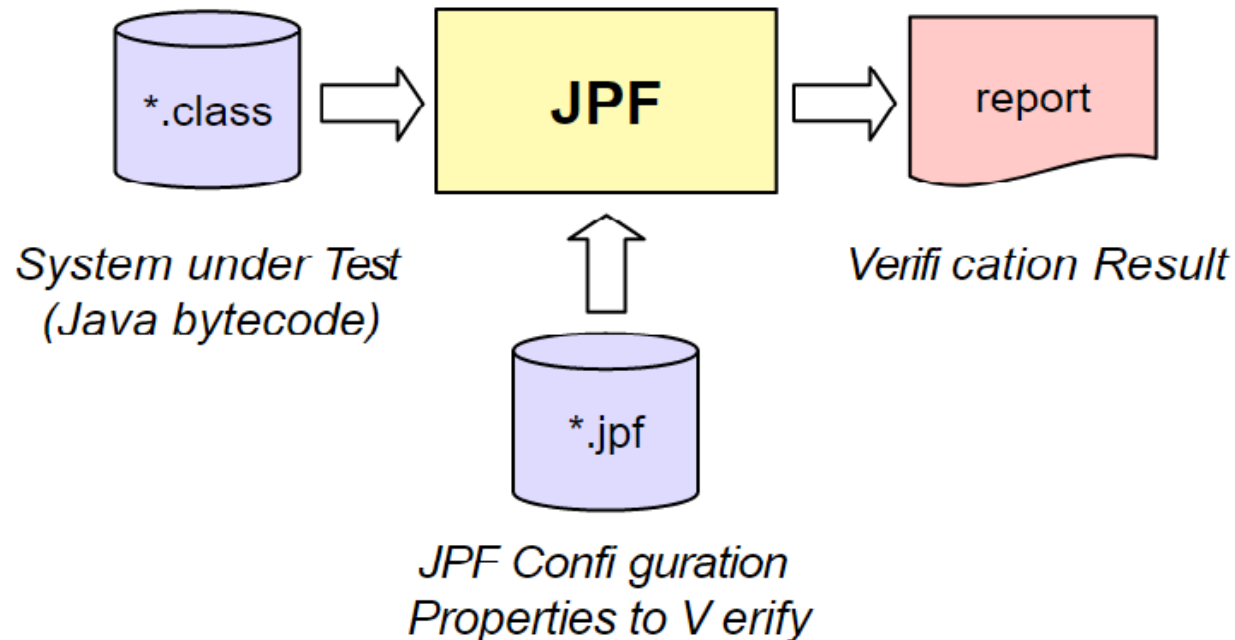
A Model Checker for Java

Java Path Finder (JPF)

- A model checker for Java: Uses all the principles we learned
 - Called Path Finder because it explores all possible paths in a program
 - Including paths that go through interleavings in a multithreaded program (!)
- Developed and maintained by NASA
 - For the purpose of rigorously model checking code for their space missions
- Open Source / Apache License Version 2
- Released 2010, still actively maintained and extended

Java Path Finder: How to Run

- Set main method class in the configuration (.jpf) file:
`target = TestRunner`
- JPF checks target class according to configuration, generates report



Java Path Finder: Example Configuration File

Target class main method to run. In this case, we are invoking JUnit through the TestRunner.

target = TestRunner

target.args =

If set to true, enumerates all possible values returned by Random.nextInt.

If set to false, the original Random.nextInt is called to generate an actual random number.

cg.enumerate_random = true

On property violation, print the error, the choice trace, and the Java stack snapshot

report.console.property_violation=error,trace,snapshot

If true, prints program output as JPF traverses all possible paths

vm.tree_output = true

Java Path Finder: Example Report

JavaPathfinder core system v8.0 - (C) 2005-2014 United States Government. All rights reserved.

===== system under test

TestRunner.main() → **Main method you are testing**

===== search started: 3/27/20 12:38 AM

PROGRAM OUTPUT → **Output of your program if any**

===== results

no errors detected → **Errors will be listed if there are exceptions or assertion failures**

===== statistics

elapsed time: 00:00:06 → **Time elapsed for testing in hours:mins:seconds**

states: new=4155,visited=3529,backtracked=7684,end=467

... → **States created while model checking**

===== search finished: 3/27/20 12:38 AM

Verify API: Enumerating Input Values

- Suppose you want to prove the following main method correct:

```
public static void main(String[] args) {  
    int x = Integer.parseInt(args[0]);  
    int y = Integer.parseInt(args[1]);  
    int diff = x - y;  
    if (x > y) assert diff > 0;  
    if (x < y) assert diff < 0;  
    System.out.println(x + " - " + y + " = " + diff);  
    return;  
}
```

- And you want to prove it correct for a set of command line arguments

Verify API: Enumerating Input Values

- Verify meaning: verify program for the specified set of input values

```
public static void main(String[] args) {  
    int x = Verify.getInt(3, 5);  
    int y = Verify.getIntFromList(4, 6);  
    int diff = x - y;  
    if (x > y) assert diff > 0;  
    if (x < y) assert diff < 0;  
    System.out.println(x + " - " + y + " = " + diff);  
    return;  
}
```

- In terms of semantics, very similar to Random value enumeration

Verify API: Java Path Finder Report

JavaPathfinder core system v8.0 - (C) 2005-2014 United States Government. All rights reserved.

===== system under test

JPFTester.main()

===== search started: 3/27/20 12:38 AM

3 - 4 = -1

3 - 6 = -3

4 - 4 = 0

4 - 6 = -2

5 - 4 = 1

5 - 6 = -1

→ **Output of:**

x = Verify.getInt(3, 5)

y = Verify.getIntFromList(4, 6)

===== results

no errors detected

...

===== search finished: 3/27/20 12:38 AM

Java Path Finder with JUnit

Running JUnit on top of Java Path Finder

Leveraging JPF for JUnit Testing

- Instead of writing test cases for one or two input values
 - Enumerate a large set of input values and all their combinations
 - Use Verify API for this purpose
- For a nondeterministic program with random value generation
 - Enumerate all random values to prove test case always passes
 - Just set “cg.enumerate_random = true” in the .jpf configuration file
- For a nondeterministic program due to thread interleavings
 - Enumerate all possible thread interleavings (!)

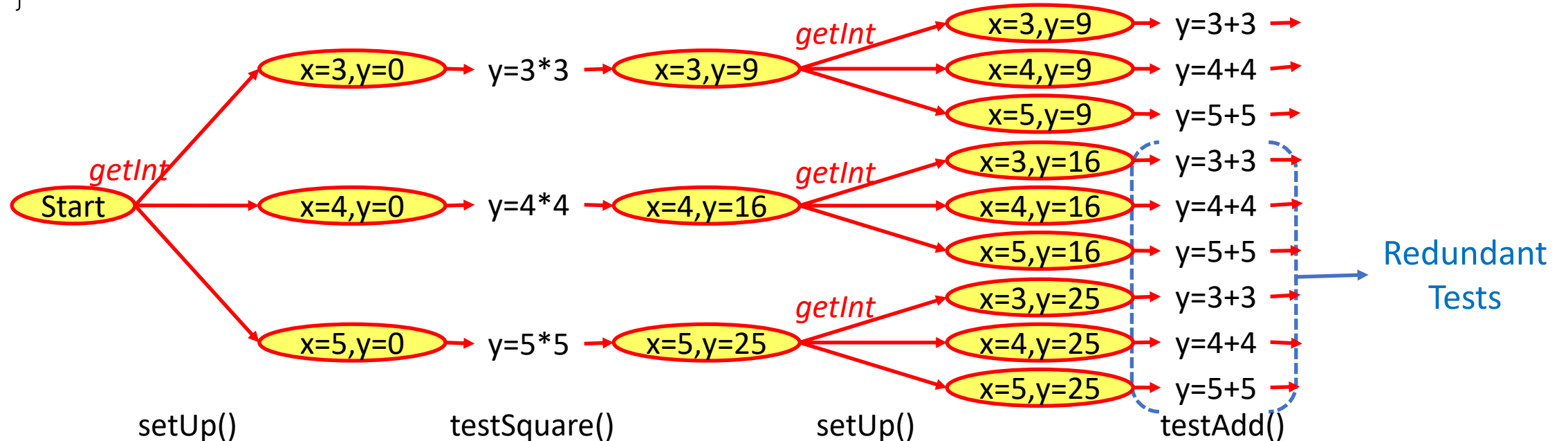
(Possible) JUnit Class for JPF Testing

```
public class testArithmetic {  
    private int x = 0;  
    private int y = 0;  
  
    @Before public void setUp() {  
        x = Verify.getInt(3, 5);  
    }  
  
    @Test public void testSquare() {  
        y = x * x;  
        assert y > 0;  
    }  
  
    @Test public void testAdd() {  
        y = x + x;  
        assert y == 2 * x;  
    }  
}
```

→ Correct but leads to a lot of unnecessary testing!

@Before Re-enumerates x Before Every Test

```
@Before public void setUp() {  
    x = Verify.getInt(3, 5);  
}
```

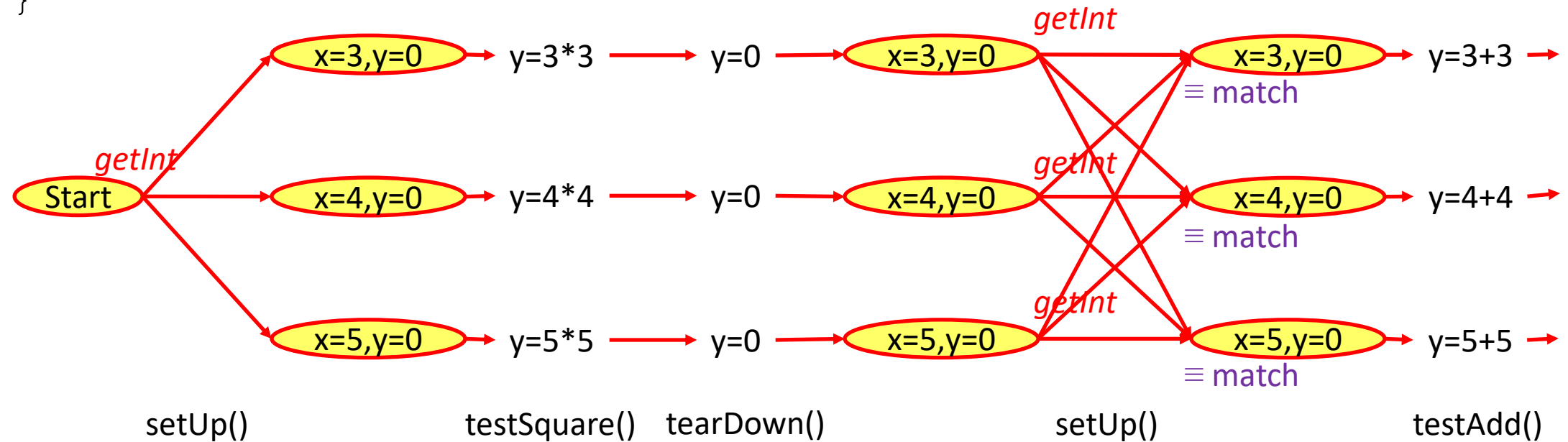


(Fixed) JUnit Class for JPF Testing

```
public class testArithmetic {  
    private int x = 0;  
    private int y = 0;  
  
    @Before public void setUp() {  
        x = Verify.getInt(3, 5);  
    }  
  
    @After public void tearDown() {  
        y = 0; // Return to initial state before next setup()  
    }  
  
    @Test public void testSquare() {  
        y = x * x;  
        assert y > 0;  
    }  
  
    @Test public void testAdd() {  
        y = x + x;  
        assert y == 2 * x;  
    }  
}
```

Proper Teardown Allows State Matching

```
@After public void tearDown() {  
    y = 0;  
}
```



- But why re-enumerate x each time and perform matching? → A lot of wasted work!

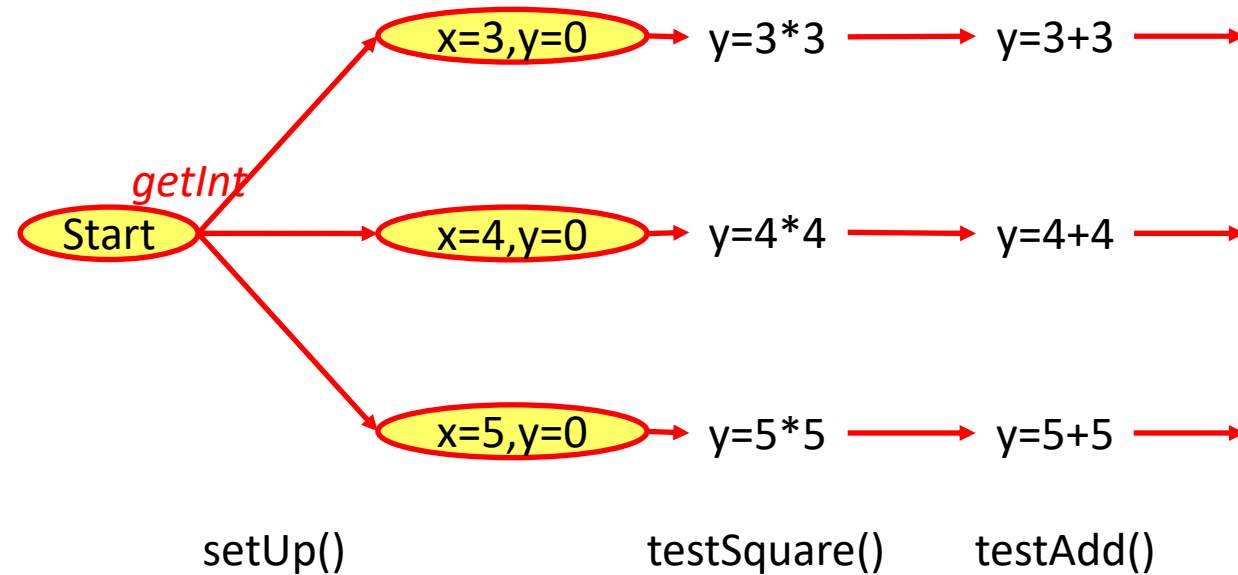
(Better) JUnit Class for JPF Testing

```
public class testArithmetic {  
    private int x = 0;  
    private int y = 0;  
  
    @BeforeClass public static void setUp() {  
        x = Verify.getInt(3, 5);  
    }  
  
    @Test public void testSquare() {  
        y = x * x;  
        assert y > 0;  
    }  
  
    @Test public void testAdd() {  
        y = x + x;  
        assert y == 2 * x;  
    }  
}
```

→ @BeforeClass is called only once before all the tests in the class

@BeforeClass Enumerates x Only Once

```
@BeforeClass public static void setUp() {  
    x = Verify.getInt(3, 5);  
}
```



- Much simpler and more efficient, don't you agree?

Interpreting JPF Report with JUnit Testing

JavaPathfinder core system v8.0 - (C) 2005-2014 United States Government. All rights reserved.

===== system under test

TestRunner.main() → **JUnit test runner**

===== search started: 3/27/20 12:38 AM

testCase1: expected:<0> but was:<1>
testCase2: expected:<"one"> but was:<"two">

Output of JUnit:
Where you will find the error messages

===== results

no errors detected → **JUnit itself does not have any errors so it will always say this**

...

===== search finished: 3/27/20 12:38 AM

What if there are Too Many States?

- State explosion can cause JPF to run for hours, or even forever
- Some mitigation techniques
 - Limit the range of input values you test using Verify API
 - Limit the part of the code you model check
 - E.g. check only the part that is critical (in terms of security, safety, ...)
 - E.g. check only the part that is hard to verify using conventional testing
 - Limit number of states created using @FilterField

Limiting State Creation Using @FilterField

```
public class WebServer {  
    // For statistics gathering purposes  
    @FilterField int pageCounter;  
  
    public void sendPage(String url) {  
        pageCounter++;  
        // Do the actual processing  
    }  
}
```

- `pageCounter` puts `WebServer` in a new state every time `sendPage` is called
 - Means state cannot be matched, even if state remains the same otherwise
 - Leads to a lot of unnecessary state creation
- `@FilterField` says, ignore `pageCounter` for the purposes of state matching