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 postconditions from preconditions 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 (x is a variable, A and B are constants):
  - { True } x = A { x == A }
  - { x == A } x = x + B { x == A + B }
  - { x == A } if (x < 0) then x = -x { x == |A| }
  - { True } if (x < 0) then x = -x { x >= 0 }

Proof is done by composing Hoare Logic Triplets

- Suppose we wanted to prove the following assertion passes:

```
x = -5;
x = x + 3;
if (x < 0) then x = -x;
assertEquals(2, x);          // prove this passes
```

- Composition of Hoare Logic Triplets:
  1. { True } x = -5 { x == -5 } and { x == -5 } x = x + 3 { x == -5 + 3 == -2 }
     → { True } x = -5; x = x + 3 { x == -2 }
  2. { True } x = -5; x = x + 3 { x == -2 } and { x == -2 } if (x < 0) then x = -x { x == |-2| == 2 }
     → { True } x = -5; x = x + 3; if (x < 0) then x = -x { x == 2 } → `Proof Complete!`

- Proof can be generated by human or automated theorem prover

Theorem Proving Advantages

- Can prove large programs with many (infinite) states
  - Model checker needs to visit each state to verify property is true
  - E.g. to prove { True } if (x < 0) then x = -x { x >= 0 }
    → Model checker needs to verify postconditions by visiting all values of x

- 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
  - Automated theorem provers often needs human assistance
    (e.g. They have trouble reasoning about data structures like lists, trees, etc.)
  - Highly skilled people with formal methods training is needed

- Automated proofs can be obscenely long
  - In one report by Motorola, a proof was 25 MB long (more than 100 pages)
  - Hard for humans to comprehend and double check the proof

Industry Reception

- Used only in niche markets where correctness is paramount
  - Mission critical systems, cryptography libraries, OS kernels
  - Proof for seL4 OS microkernel: https://github.com/seL4/l4v

- 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 derived from source code
- Specification is also called a system property.
    - The same “property” in property-based testing

Examples of System Properties

- 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 dataraces when threads access shared data
- User assertions (invariants)
  - Embedded in source code or part of property-based unit test

Comparison with Stochastic Testing

- Similarity
  - Model checking also tests a property, not an output value

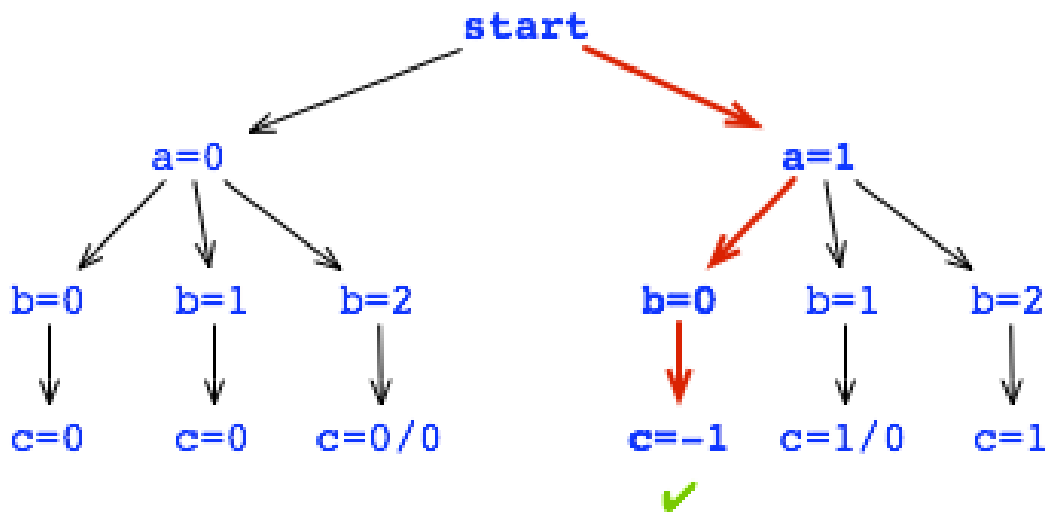
- Difference
  - With stochastic testing, we tested (a few) randomized input values
  - With model checking, all states are checked *exhaustively*

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)`

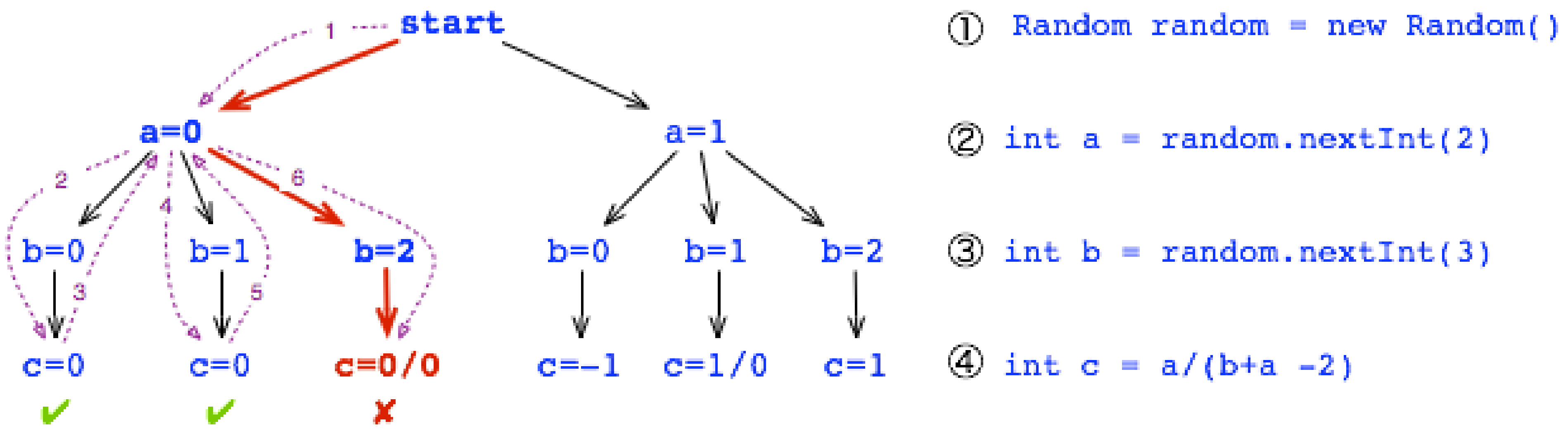
④ `int c = a/(b+a -2)`

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!



State Explosion Problem

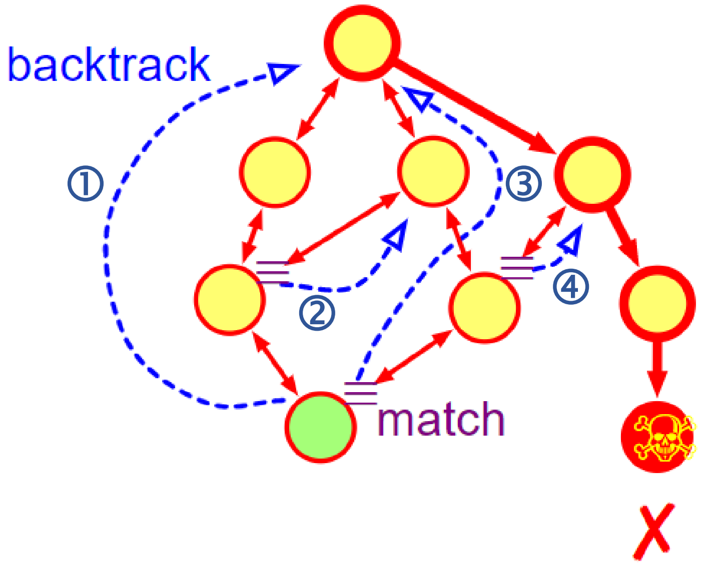
- Non-trivial programs have many more states in their Finite State Machines
  - May run into memory limitations (can’t contain entire state graph)
  - May run into time limitations (can’t explore entire graph within allotted time)

  → This is called the **State Explosion Problem**

- Single reason preventing wide adoption of model checking

- **State matching & backtracking** alleviates the problem but not all

State Exploration with Matching & Backtracking



Circles: Program states
Arrows: State transitions

- State divergence happens when there is a *choice*
  - Value from random number generation or user input
  - Choice of thread to run among multiple threads
- **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

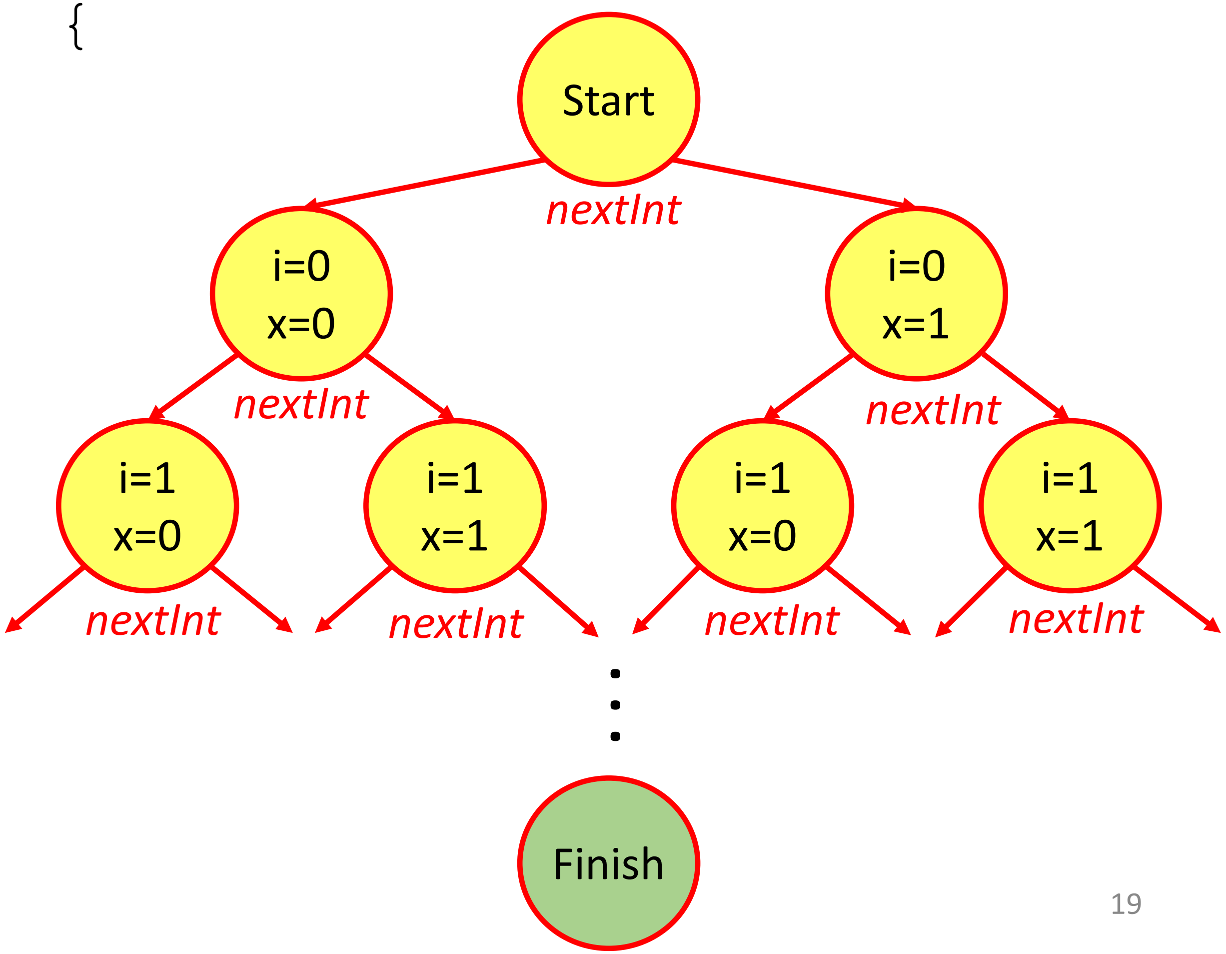
→ Ensures **each unique state is visited only once**

Matching & Backtracking: Example 1

- Given below code:

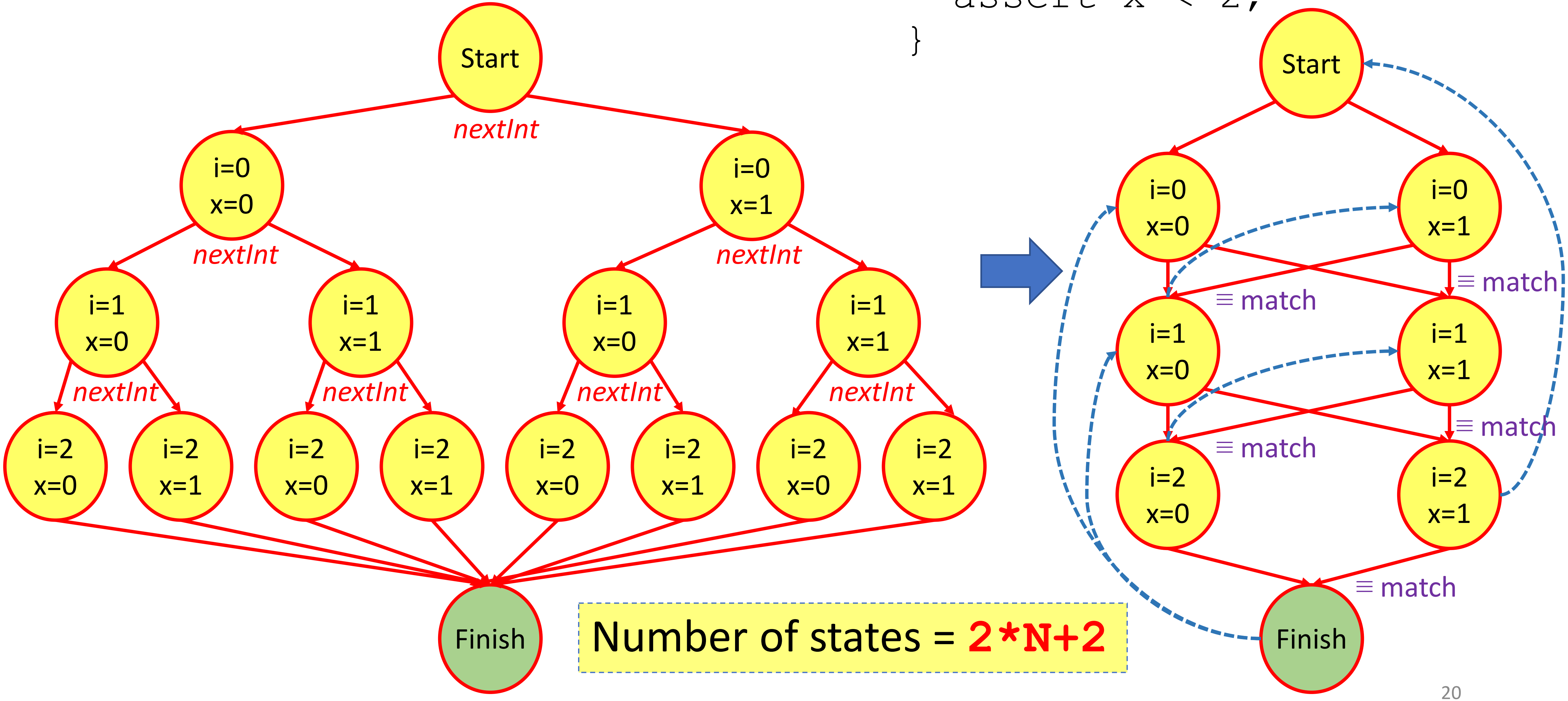
```
for(int i = 0; i < N; i++) {
   x = rand.nextInt(2);
   assert x < 2;
}
```

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



Matching & Backtracking on

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



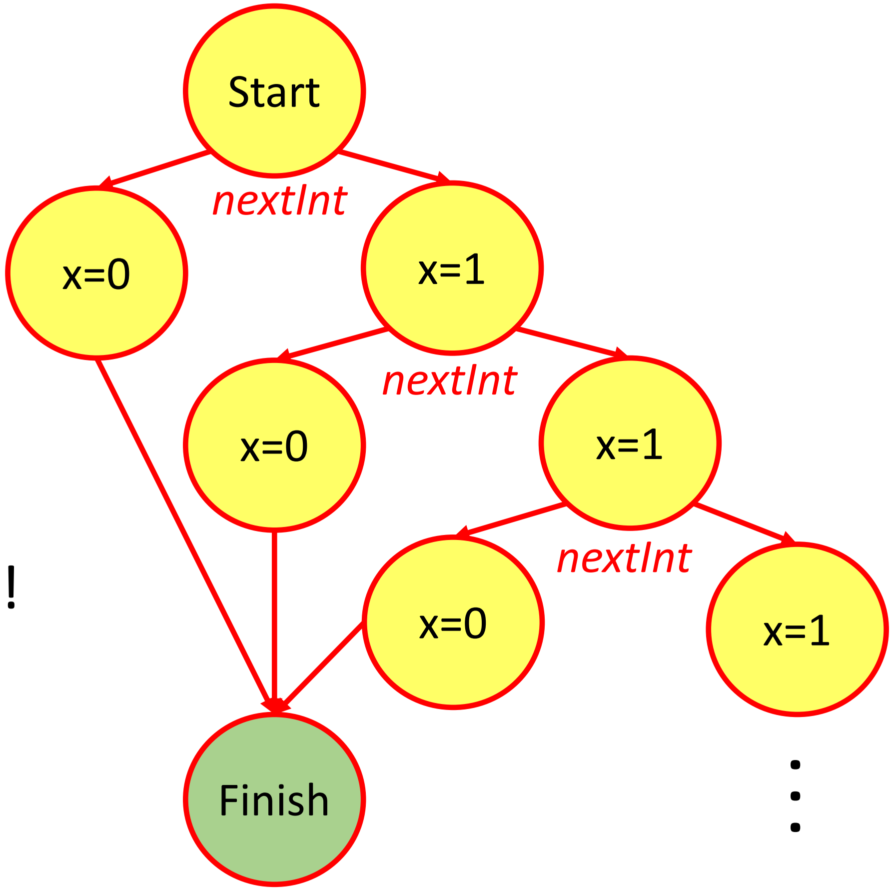
Number of states = **2*N+2**

Matching & Backtracking: Example 2

- Given below code:

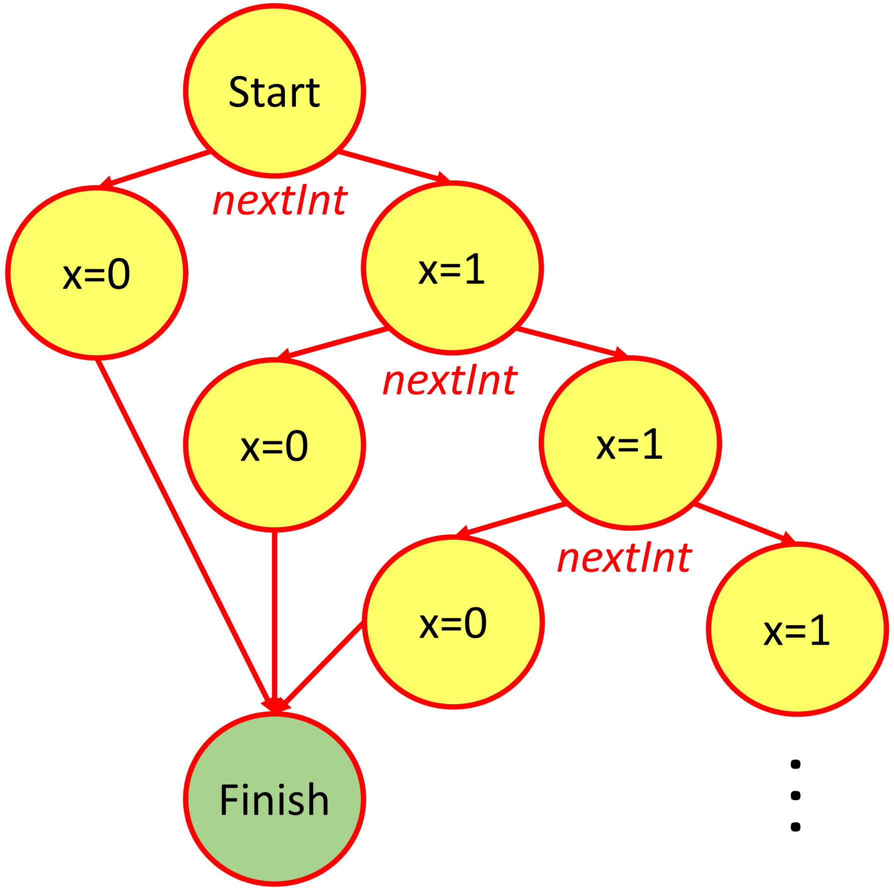
```
while(true) {
   x = rand.nextInt(2);
   if(x == 0) break;
   assert x < 2;
}
```

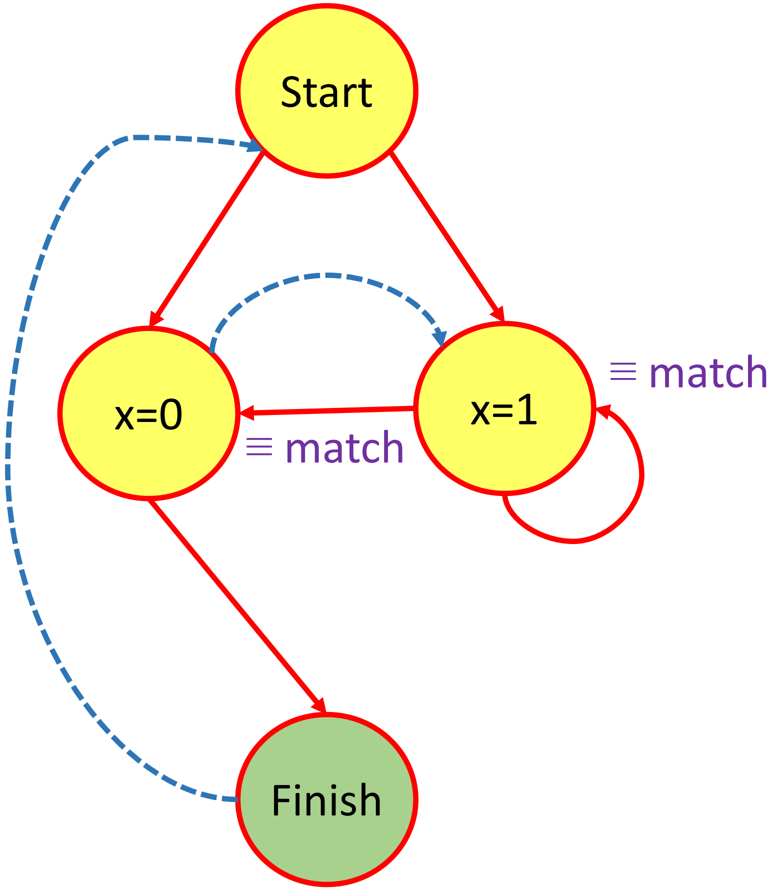
- Model checker can potentially go on forever!
  - Will keep creating states to the right



Match & Backtrack on

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





Number of states = **4**

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);
      }
}
```

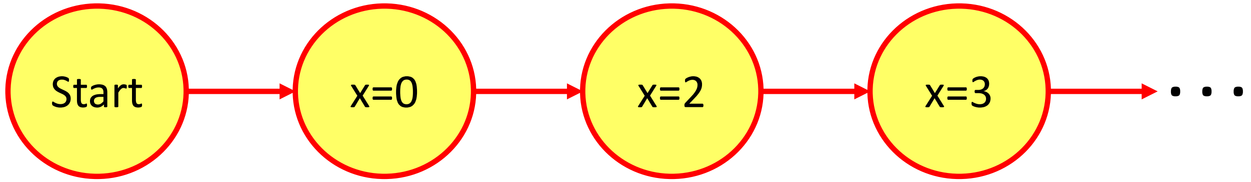
State Space Explosion Sometimes is Unavoidable

- Even with state matching, sometimes state space explodes
- May have to filter out part of program state that causes explosion
  - Human has to tag variables or objects as filtered
  - May result in parts of program state that are not verified
  - But these parts are typically not the parts that require rigorous verification (Involve event counters, statistics, logs, etc.)

State Explosion Even with State Matching

- Given below code:

```
x = 0;
while(true) {
  x = x + 1;
  assert x > 0;
}
```



- The value of x keeps incrementing at every iteration, creating a new state
  - Results in state explosion, making it impossible to model check!
  - Variables like these are typically stat counters, which can be filtered out with minimal loss

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

Java Path Finder (JPF)

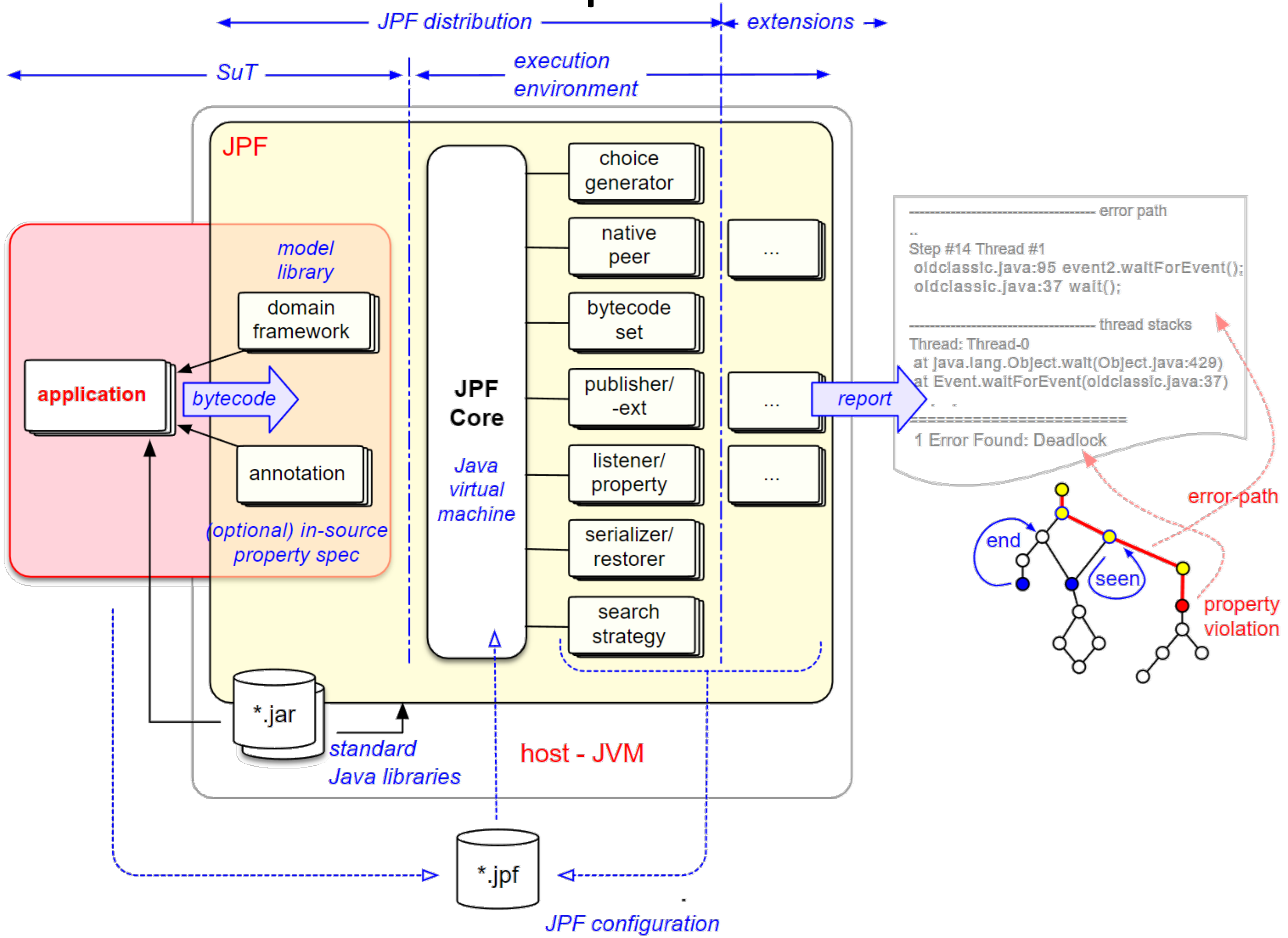
A Model Checker for Java

Java Path Finder (JPF)

- A model checker for Java: Uses all the principles we learned
  - Called Path Finder because it explores all paths in finite state machine
  - Concrete model checker: system model is the Java Virtual Machine
    → No need to translate source code into abstract model

- Developed and maintained by NASA
  - To model check code for their space missions
  - Open Source / Apache License Version 2
  - Released 2010, still actively maintained and extended

Java Path Finder is a special Java Virtual Machine



Java Path Finder: Example Configuration File

```
# Target class main method to run.  In this case, we are invoking JUnit through the TestRunner.
target = edu.pitt.cs.Rand
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
states:          new=4155,visited=3529,backtracked=7684,end=467
…
====================================================== search finished: 3/27/20 12:38 AM
```

Time elapsed for testing in hours:mins:seconds

States created while model checking

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

JUnit Testing with Java Path Finder

JPF is invoked on-demand by JUnit

- All JUnit tests are executed on the host Java virtual machine
  - All @Test, @Before, @After methods executed on the host JVM
- For @Test methods that you want to run using JPF,
  call **`verifyNoPropertyViolation()`** at beginning of method

- Semantics of **`verifyNoPropertyViolation()`** :
  - Creates a new JPF virtual machine that starts executing the @Test method
  - Results in two virtual machines executing the @Test method
    - **Host** virtual machine: returns **false** on verifyNoPropertyViolation() call
    - **JPF** virtual machine: returns **true** on verifyNoPropertyViolation() call

Example JUnit Test using JPF

```
public class ArithmeticTest {
  private int x = 0;

  public void setUp() {
    x = Verify.getInt(-5, 5);
  }

  @Test public void testSquare() {
    if (verifyNoPropertyViolation() == false) {
      // This is the host virtual machine so return immediately.
      return;

    }
    setUp();  // Call setUp() on the JPF virtual machine.
    int y = x * x;
    assertTrue(y > 0);
  }
  …
}
```

Pitfall: JUnit Test using JPF

```
public class ArithmeticTest {
  private int x = 0;

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

  @Test public void testSquare() {
    if (verifyNoPropertyViolation() == false) {
      // This is the host virtual machine so return immediately.
      return;

    }
    // Bug! @Before executed on host, not on JPF virtual machine!
    int y = x * x;
    assertTrue(y > 0);
  }
  …

}
```