

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. $\{ \text{True} \} x = -5 \{ x == -5 \}$ and $\{ x == -5 \} x = x + 3 \{ x == -5 + 3 == -2 \}$
 $\rightarrow \{ \text{True} \} x = -5; x = x + 3 \{ x == -2 \}$

2. $\{ \text{True} \} x = -5; x = x + 3 \{ x == -2 \}$ and $\{ x == -2 \} \text{if } (x < 0) \text{ then } x = -x \{ x == |-2| == 2 \}$
 $\rightarrow \{ \text{True} \} x = -5; x = x + 3; \text{if } (x < 0) \text{ then } x = -x \{ x == 2 \} \rightarrow \text{Proof Complete!}$

- 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 need 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

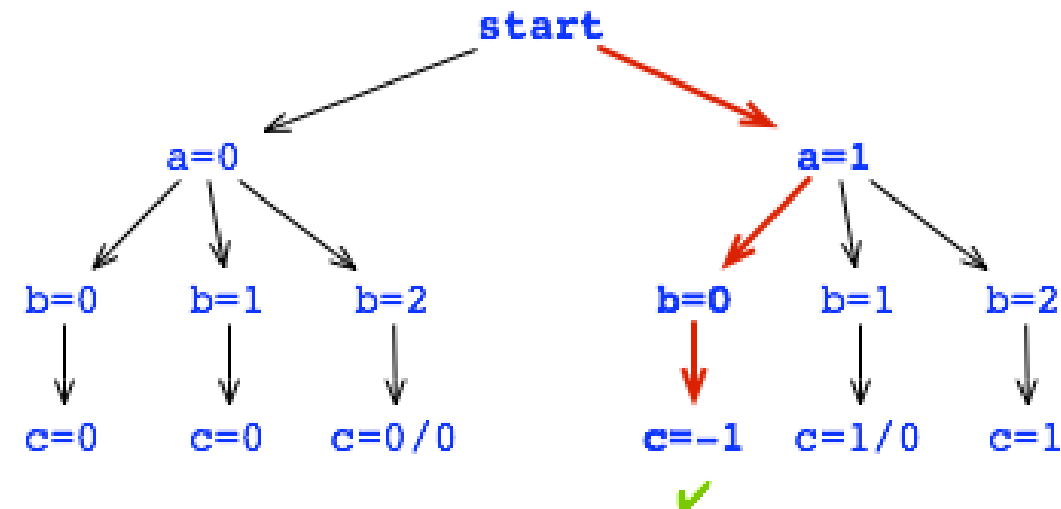
- Similarity
 - Model checking also tests a property, not an output value
- Difference
 - With stochastic testing, we tested (a few) randomized input values
 - With model checking, all states are checked *exhaustively*

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!



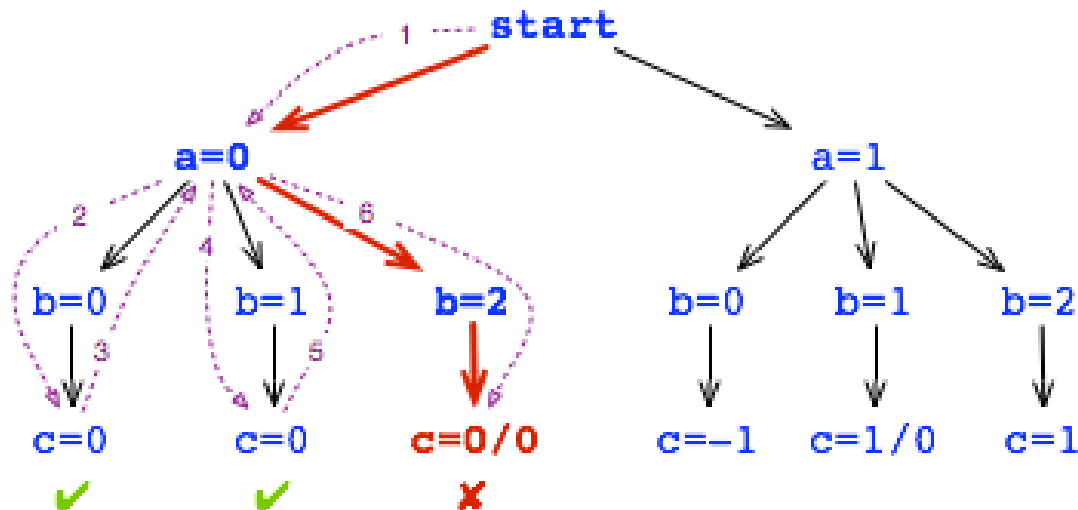
- ① `Random random = new Random();`
- ② `int a = random.nextInt(2);`
- ③ `int b = random.nextInt(3);`
- ④ `int c = a / (b + a - 2);`

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!



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

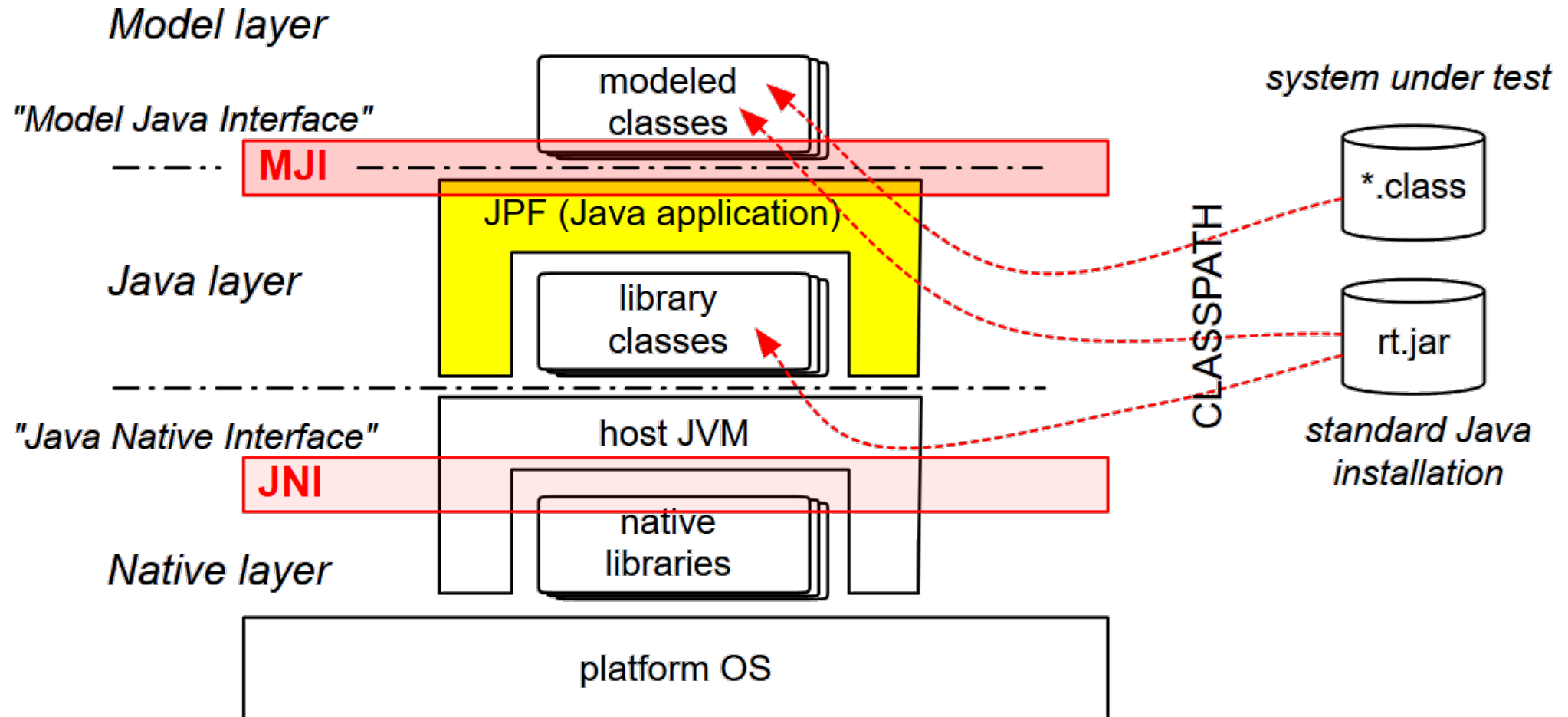
Java Path Finder (JPF)

A Model Checker for Java

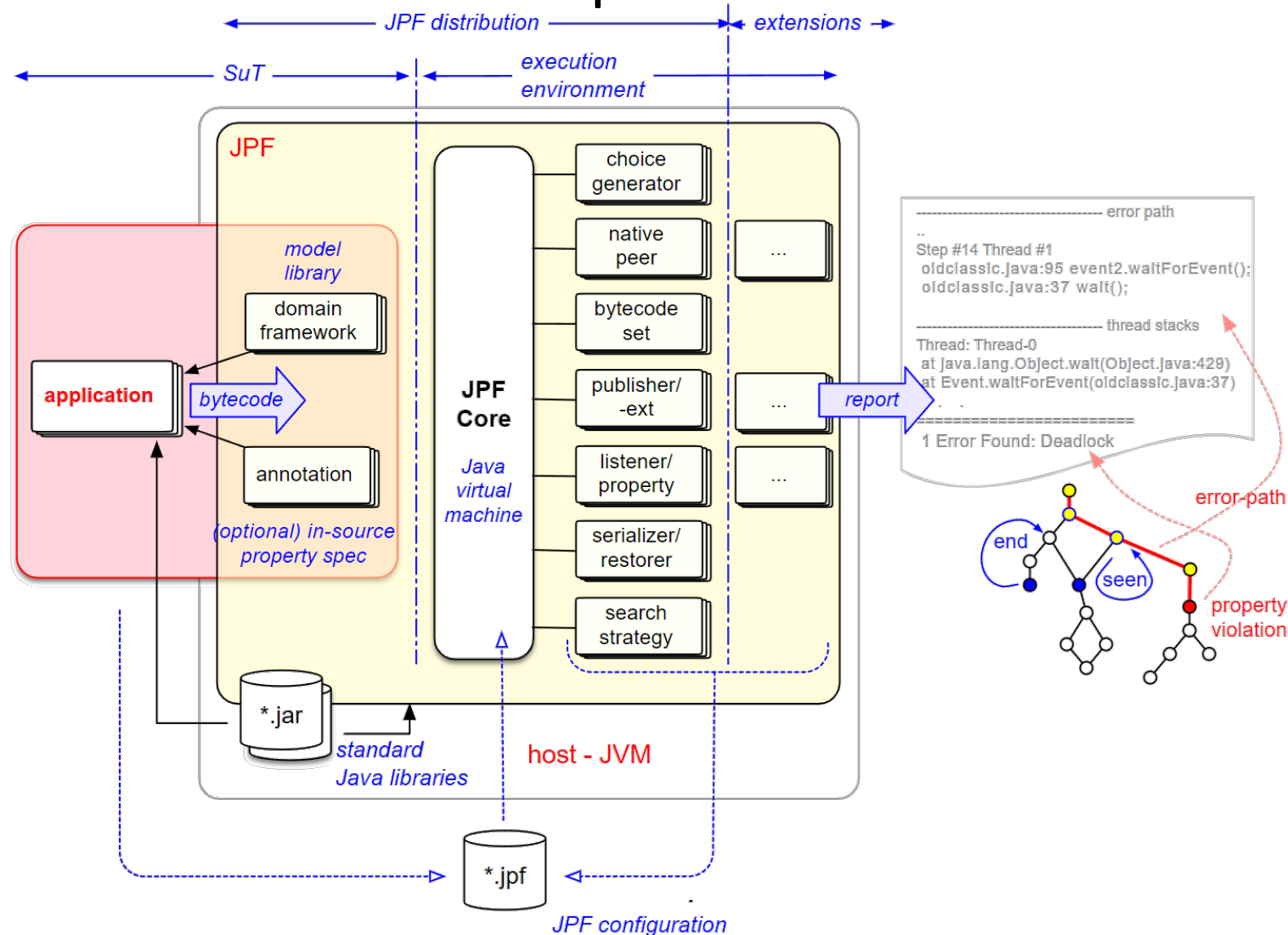
Java Path Finder (JPF)

- 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

App executes on top of Java Path Finder JVM



Java Path Finder is a special Java Virtual Machine



JPF VM enumerates all random states

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

Not shown, but also generates a
“trace” of random values chosen

```
-bash-4.2$ ./runJPF.sh Random.jpf  
JavaPathfinder core system v8.0 (C) 2005-2014 United States Government.  
...  
===== error 1  
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty  
java.lang.ArithmeticException: division by zero  
    at Rand.main(Rand.java:34)  
...
```

Where in code
exception happened

JPF VM enumerates all thread interleavings

```
-bash-4.2$ ./runJPF.sh Interleaving.jpf
```

```
JavaPathfinder core system v8.0 (C) 2005-2014 United States  
Government.
```

```
...
```

```
===== search started
```

```
main main Thread-1 Thread-1  
main Thread-1 main Thread-1  
main Thread-1 Thread-1 main  
Thread-1 main main Thread-1  
Thread-1 main Thread-1 main  
Thread-1 Thread-1 main main
```

Able to explore all interleavings and
generate all possible outputs!

Java Path Finder Usage

Tips on how to use JPF

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 → **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

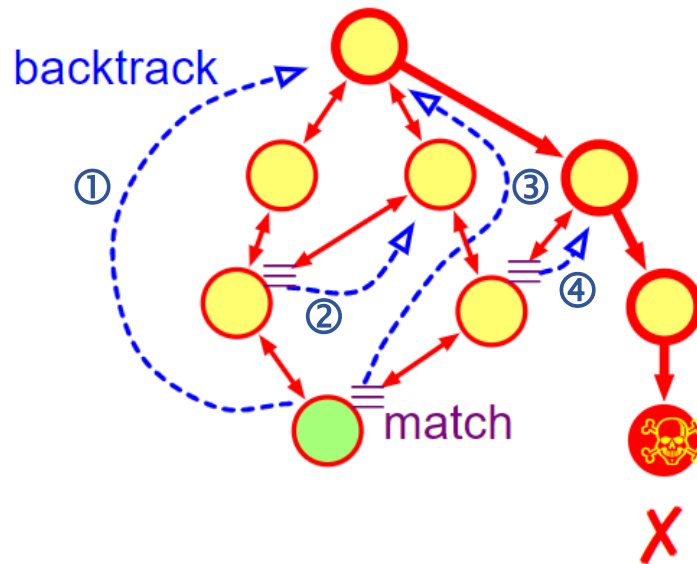
State Space Explosion

When it happens and how to deal with it

State Space Explosion Problem

- Some programs may have (close to) an infinite states in their 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 Space 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

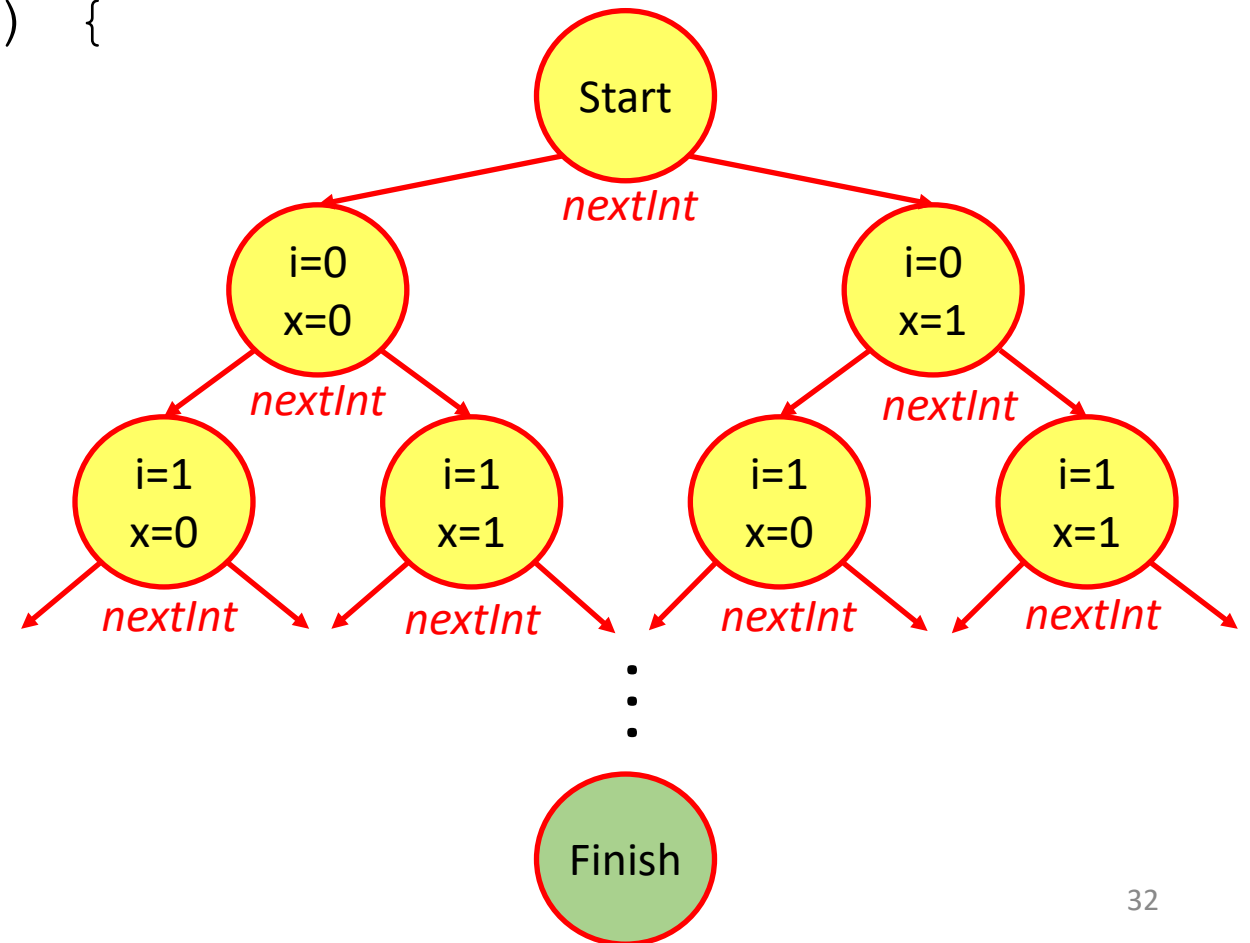
- State divergence happens when there is a *choice*
 - Value from random number generation or user input
 - Choice of thread to run among multiple threads
 - **Match**
 - When next state matches a previously visited state
 - Backtrack to not repeat work
 - **Backtrack**
 - On reaching terminal state 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:

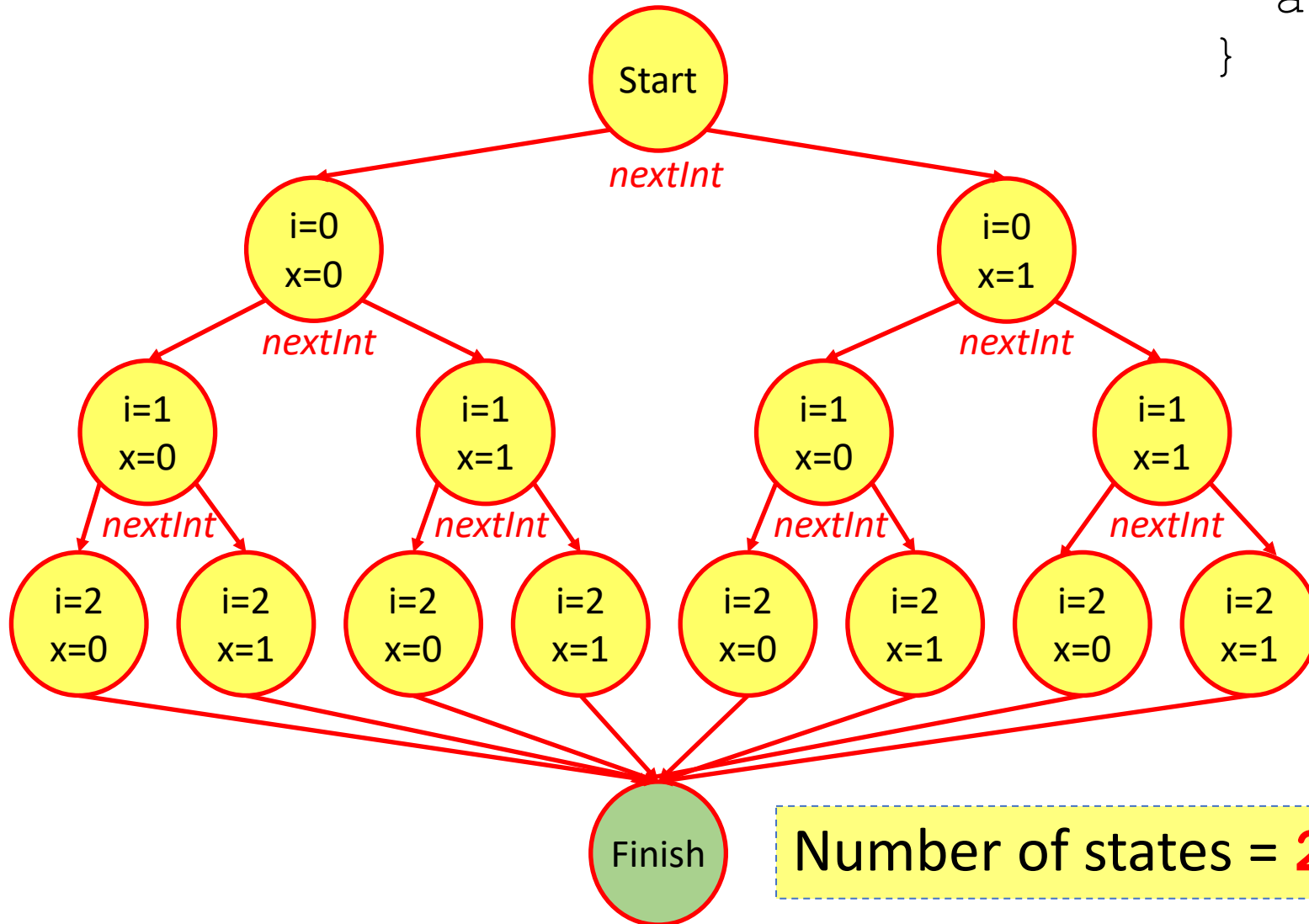
```
for(int i = 0; i < N; i++) {  
    x = rand.nextInt(2);  
    assert x < 2;  
}
```

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

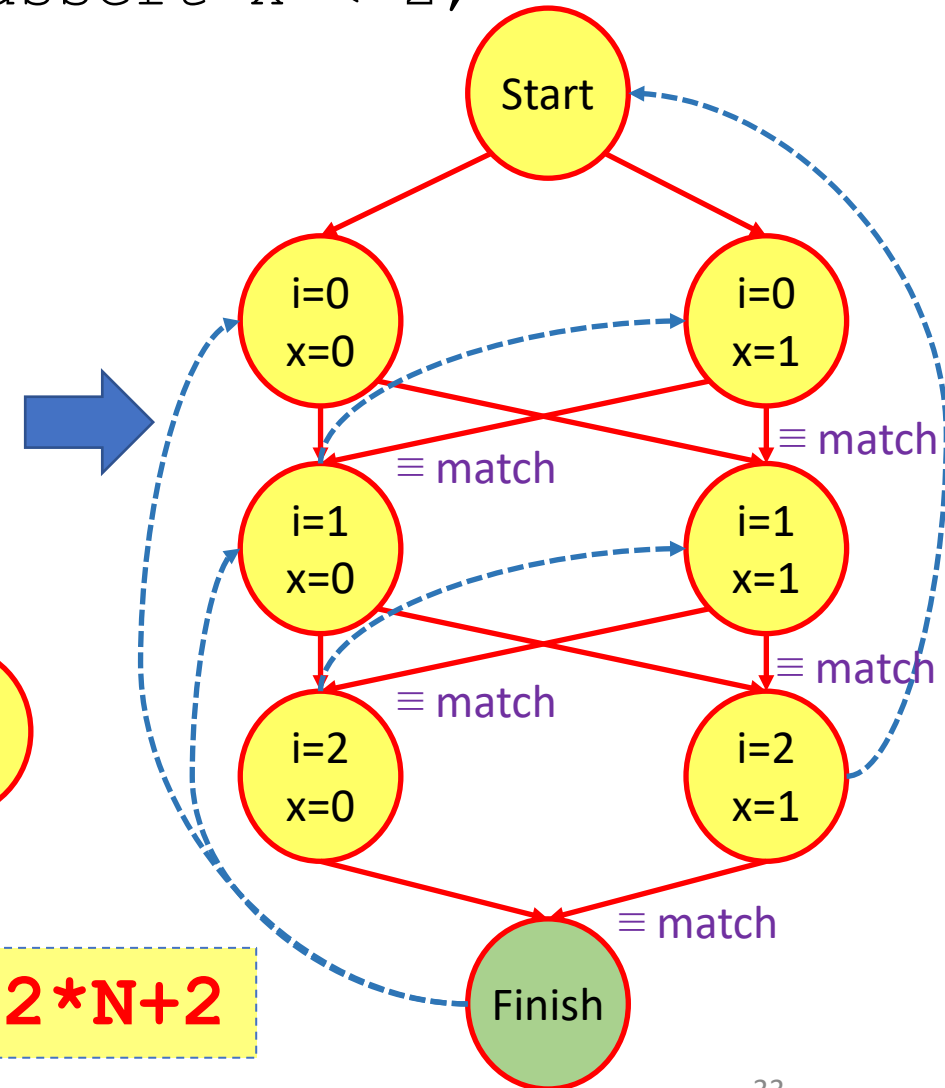


Matching & Backtracking on

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



Number of states = $2 * N + 2$

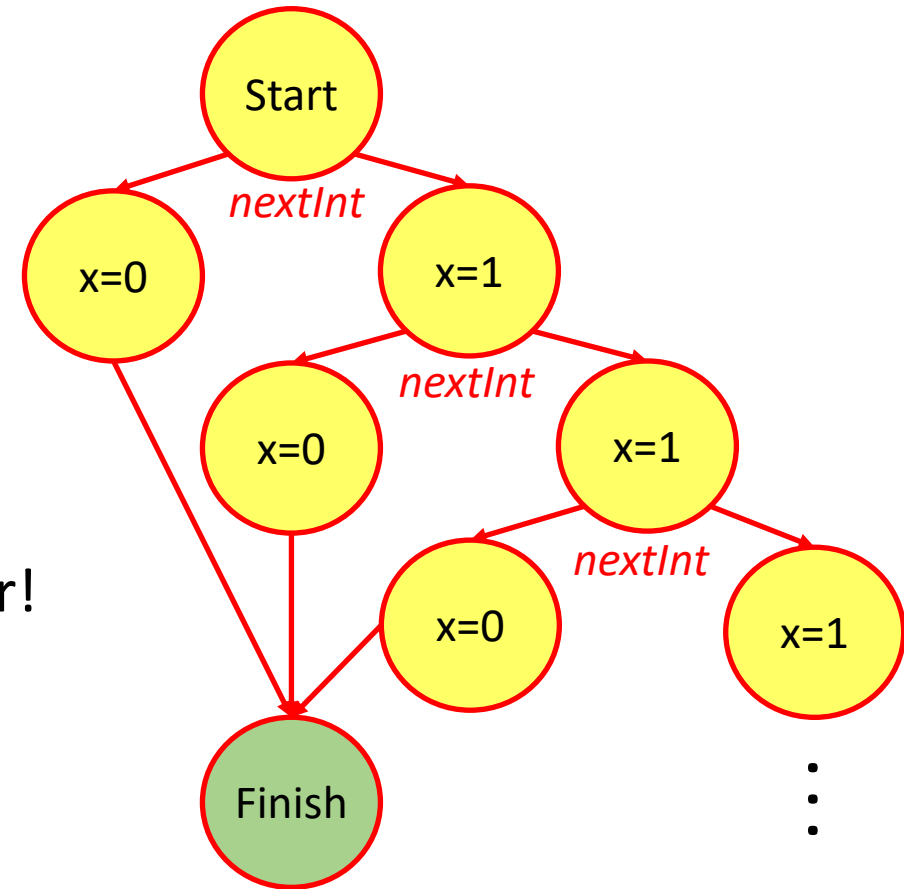


Matching & Backtracking: Example 2

- Given below code:

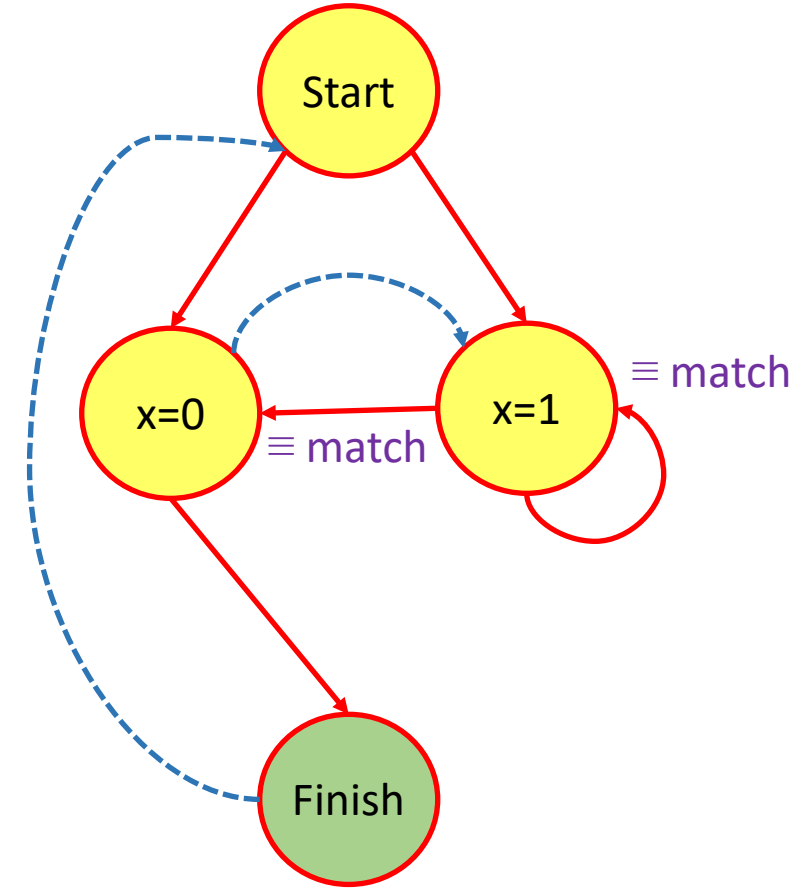
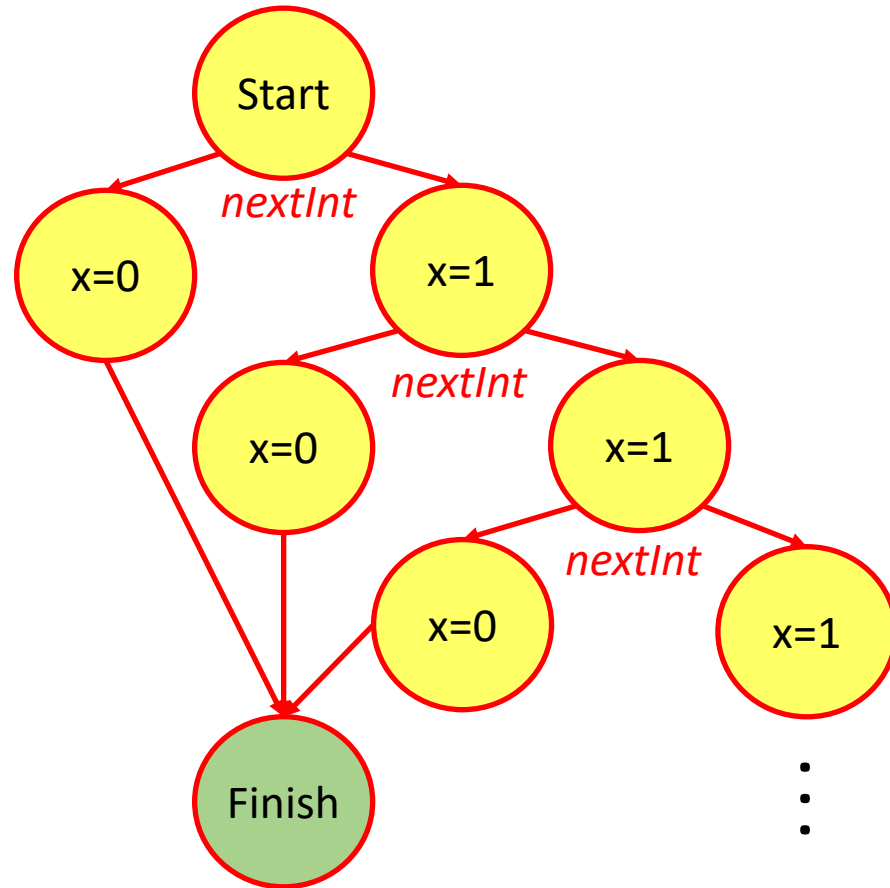
```
while(true) {  
    x = rand.nextInt(2);  
    if(x == 0) break;  
    assert x < 2;  
}
```

- 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);
```

```
    }
```

```
}
```

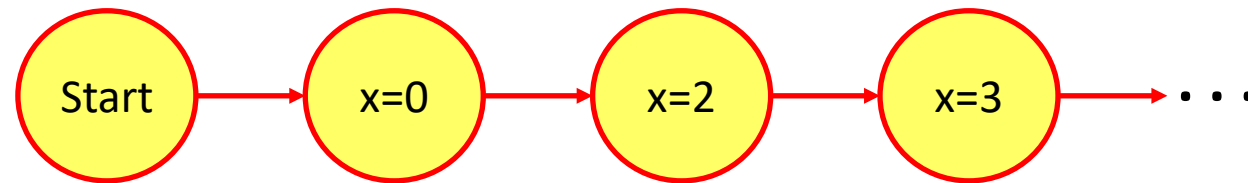
State Space Explosion Sometimes is Unavoidable

- 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

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
 - JPF VM **does not execute the `@Before` method** (it didn't exist at that point)

Pitfall: Expecting JPF VM to execute @Before

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

    @Before public void setUp() {
        // Only executed on host virtual machine (JPF VM doesn't exist yet)
        // Host VM simply assigns x = 0 (doesn't know any better)
        x = Verify.getInt(-5, 5);
    }

    @Test public void testSquare() {
        if (verifyNoPropertyViolation() == false) {
            // This is the host virtual machine so return immediately.
            return;
        }
        // Bug! Test case is tested against x == 0 only.
        int y = x * x;
        assertTrue(y > 0);
    }
}
```

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

Backup

Holy Grail of Formal Verification

- Soundness
 - If no defect is reported, then the program does not fail
 - No false negatives
- Preciseness
 - If a defect is reported, then the program does fail
 - No false positives
- Termination
 - The verification terminates

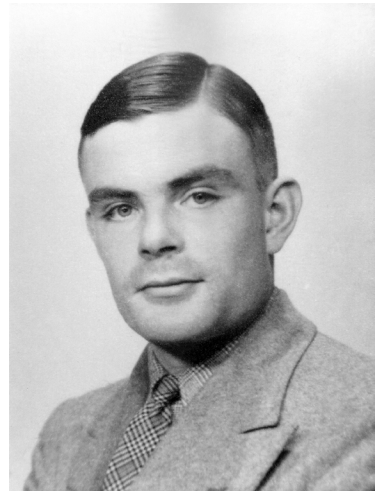
It is impossible to achieve the holy grail in general!

Formal Verification is Undecidable

$x \in \text{Variable}$

$P \in \text{Program} = \text{assert } x \mid x++ \mid x-- \mid$
 $P_1 ; P_2 \mid \text{if } x \text{ then } P_1 \text{ else } P_2 \mid \text{while } x \text{ } P$

- Assertion checking for even this simple language is undecidable!
- “The **Halting Problem** cannot be solved for *all* possible programs (for a Turing-complete language)” - *Alan Turing (1936)*
- Silver-lining: But for *some (most)* programs it can be solved



Concrete and Abstract Model Checking

- Concrete model checking
 - States in model are actual concrete program states
 - As in, your program stack and your program heap
- Abstract model checking
 - States in model are some abstraction of actual program states
 - Abstraction is done in hopes of reducing the state explosion problem
 - Typically tradeoffs accuracy for efficiency

Concrete and Abstract Model Checking

- Concrete model checking
 - States in model are actual concrete program states
 - As in, your program stack and your program heap
- Abstract model checking
 - States in model are some abstraction of actual program states
 - Abstraction is done in hopes of reducing the state explosion problem
 - Typically tradeoffs accuracy for efficiency

Abstract Model Checking

- Requires an intermediate description of abstract model
 - Describes the system at a high level
 - Throws away implementation details
- Good for checking designs, rather than implementations
 - Success stories: hardware circuits, cache-coherence protocols
- Problem: Specifying an abstract model is HARD for large systems
 - What you check is not what you run!
 - As the system evolves model has to be updated
 - Manual extraction of abstract model can miss or introduce errors

Automatically Extracting the Model

- Statically analyze the code to generate a model
 - Models usually mimic the implementation

Murphi abstract model

```
Rule "PI Local Get (Put)"
  1: Cache.State = Invalid
    & ! Cache.Wait
  2: & ! DH.Pending
  3: & ! DH.Dirty ==>
    Begin
  4: Assert !DH.Local;
  5: DH.Local := true;
  6: CC_Put(Home, Memory);
  EndRule;
```

Flash Memory Driver Implementation

```
void PILocalGet(void) {
    // ... Boilerplate setup
  2  if (!hl.Pending) {
  3    if (!hl.Dirty) {
  4!    // ASSERT(hl.Local);
        ...
  5    hl.Local = 1;
  6    PI_SEND(F_DATA, F_FREE, F_SWAP,
              F_NOWAIT, F_DEC, 1);
}
```

Automatic Model Extraction

- Examples
 - FeaVer : C program -> Promela (SPIN) model
 - Bandera: Java -> Bandera model
- Features
 - Sophisticated property-driven slicing techniques
 - Can throw away state unrelated to property that is being proved
- Problems
 - Not all primitives are available in the modeling language
 - Pointers, dynamic object creation, dynamic threads, exceptions
 - A precise-enough slice could be as large as the program itself

Concrete and Abstract Model Checking

- Concrete model checking
 - States in model are actual concrete program states
 - As in, your program stack and your program heap
- Abstract model checking
 - States in model are some abstraction of actual program states
 - Abstraction is done in hopes of reducing the state explosion problem
 - Typically tradeoffs accuracy for efficiency

Concrete Model Checking

- Code as the model – directly execute the code!
- Concrete model checkers
 - Verisoft (C/C++) – Bell Laboratories
 - CBMC (C/C++) – Oxford University
 - Java Path Finder (Java) – NASA

Language and Concrete Model Checking

- Not many programming languages have a dedicated model checker
- Fortunately, you can convert most languages to Java bytecode
 - JavaScript, Python, Ruby, Lua, ...
 - Even (for a limited set of) C / C++
 - And then, you can model check the bytecode using a JVM

State Space Reduction Techniques

- State space can be infinite (or very large)
 - Try exploring as many behaviors as possible (likely you can't explore all)
 - Focus on precision (finding defects accurately)
- Techniques to reduce memory consumption due to state space
 - State collapsing
 - Heuristic state approximation
 - Hash compaction
 - Heap canonicalization
 - Symbolic execution

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- Heap canonicalization
- Symbolic execution

State collapsing

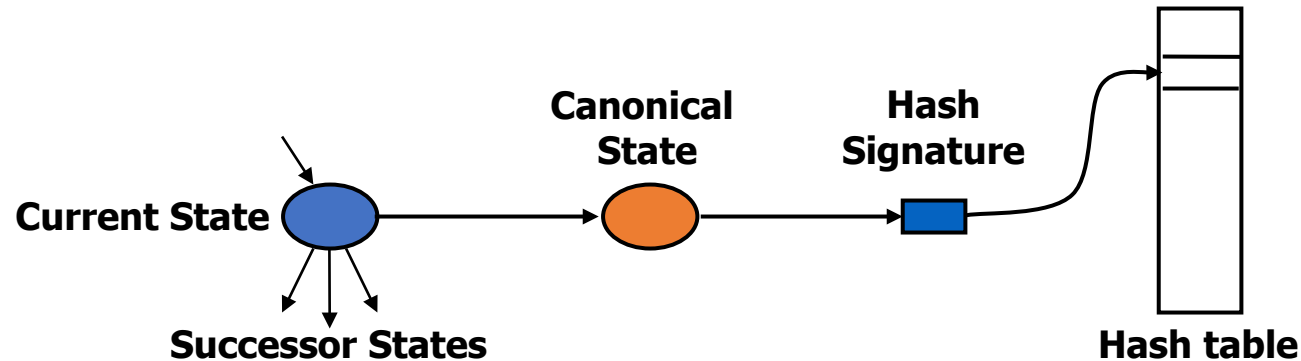
- Typically a state transition involves changing very small state
 - Updating a local variable in the stack
 - Creating a new object on the heap
- Instead of storing the entire state each time in hash table ...
- Store the delta (change) from the previous state in hash table

State Space Reduction Techniques

- State collapsing
- **Heuristic state approximation**
- Hash compaction
- Heap canonicalization
- Symbolic execution

Heuristic state approximation

- Explore one out of a (large) set of equivalent states
- Canonicalize (unify) states before hashing



- Example: suppose an int value has two equivalence classes
 - When hashing to check for match with an already visited state, Unify all values in equivalence class to one chosen value
 - Leads to a drastic reduction in visited states
 - Can also lead to missed defects – **unsound**

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- **Hash compaction**
- Heap canonicalization
- Symbolic execution

Hash compaction

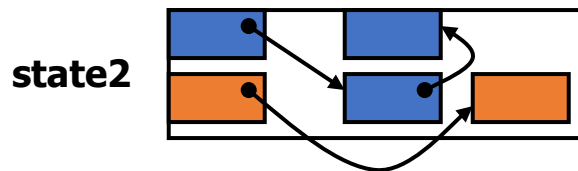
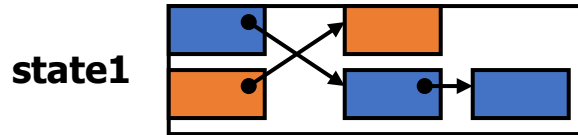
- Only store hash of state in the hash table
 - Do not store the actual state
 - A state match is determined solely by an equality check on the hash
- Might miss defects due to hash collisions – **unsound**
 - Two states that are different may be stored as the same hash
 - Means some states will not be visited as a result
- But orders of magnitude memory savings
 - Can compact 100 kilobyte state to 4-8 bytes!

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- **Heap canonicalization**
- Symbolic execution

Heap canonicalization

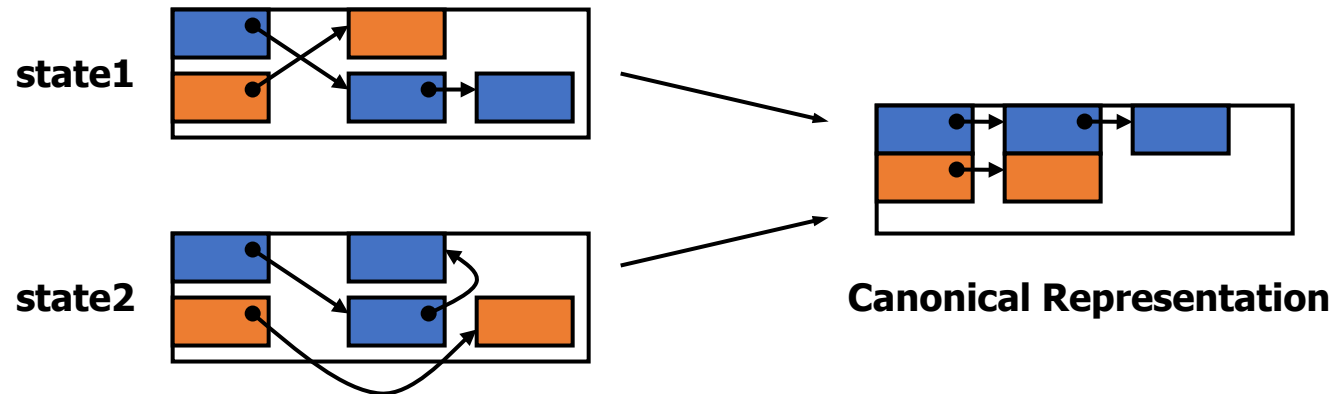
- Problem: two logically equivalent program states appear different because of differences in heap layout
- Example:



- Are the two states logically different?
 - No! But appears different due to different reference pointer values.

Heap canonicalization

- Solution: Canonicalize heap to unify layout
- Example:



- Canonical layout can be found by doing a fixed traversal of heap
 - DFS: Depth first search, or BFS: Breadth first search
- Note: can do it incrementally on each heap modification w/o full traversal

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- Heap canonicalization
- Symbolic execution

To be continued ...