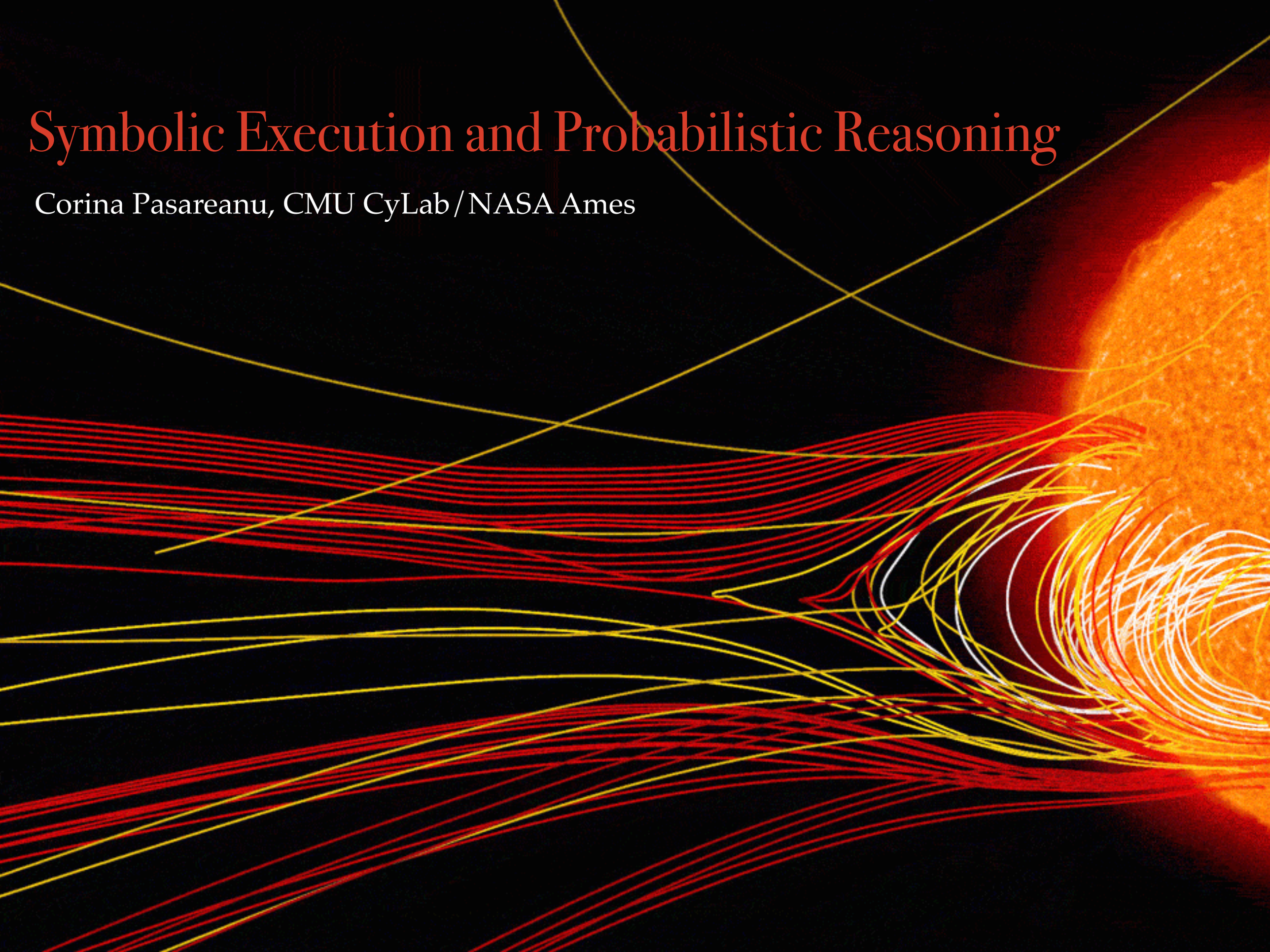
Symbolic Execution and Probabilistic Reasoning

Corina Pasareanu, CMU CyLab / NASA Ames



Software Safety and Security

- Software systems become more pervasive and complex
- Increased need for techniques and tools that ensure safety and security of software systems
- Research directions:
  - automated verification techniques
  - application at all phases of software development
  - theoretical foundations and practical tools

Symbolic Execution

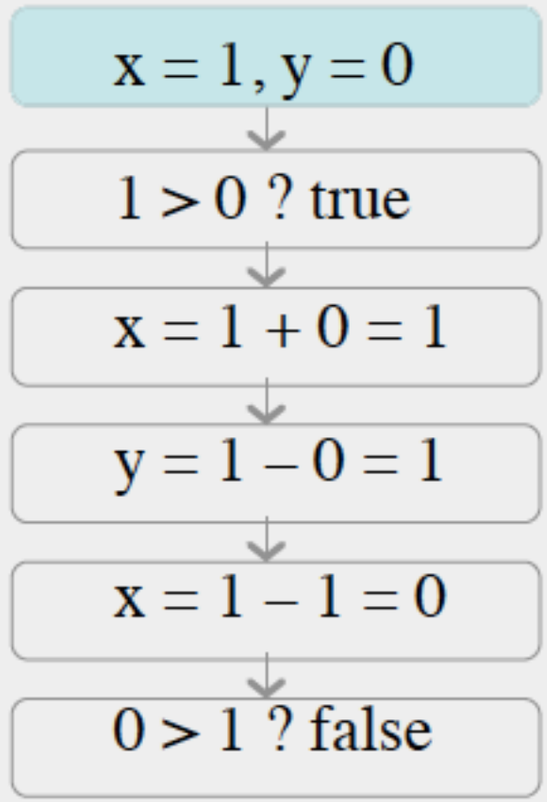
- Systematic program analysis technique — King [Comm. ACM 1976], Clarke [IEEE TSE 1976]
- Executes programs on symbolic inputs — represent multiple concrete inputs
- Path conditions — conditions on inputs following same program path
  - Check satisfiability using off-the-shelf solvers (Z3) – explore only feasible paths
  - Solve path conditions: obtain test inputs
- Bounded execution
- Many applications: test-case generation, error detection, …
- Many tools: SAGE, DART, KLEE, Pex, BitBlaze …
- Symbolic PathFinder

Example Concrete Execution

Code that swaps 2 integers

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Concrete Execution Path

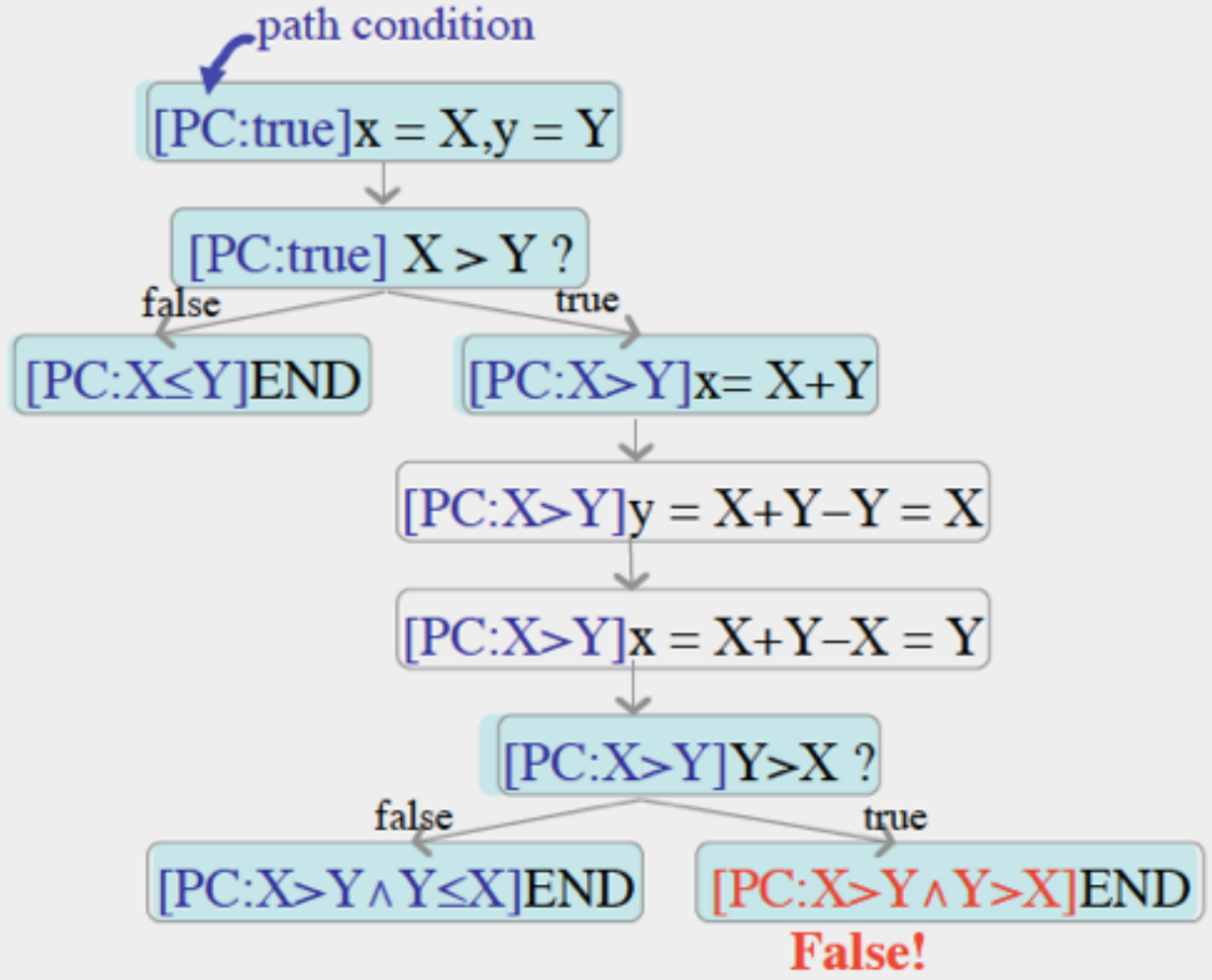


Example Symbolic Execution

Code that swaps 2 integers

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Symbolic Execution Tree



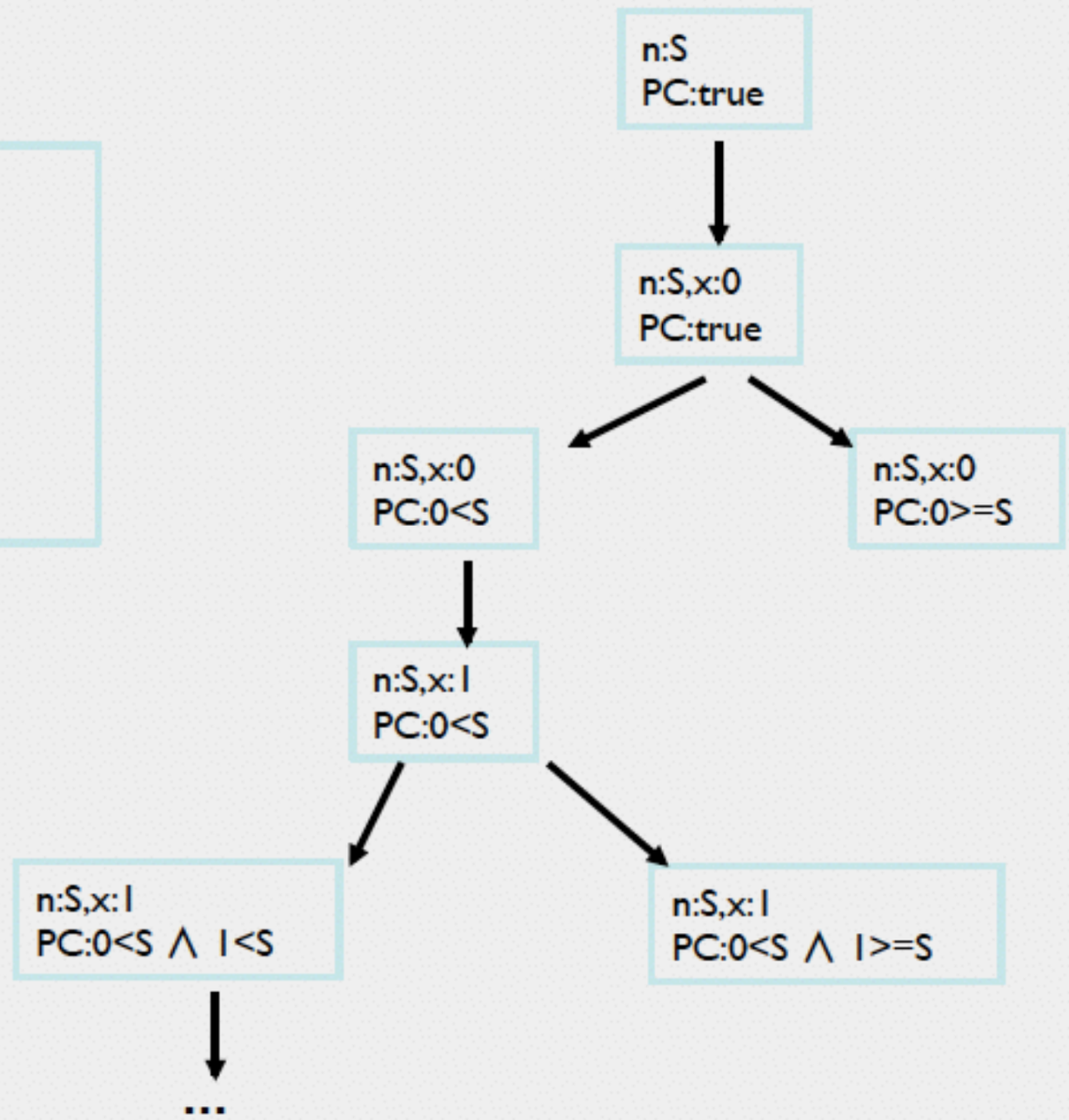
Solve PCs: obtain test inputs

Loops

example code

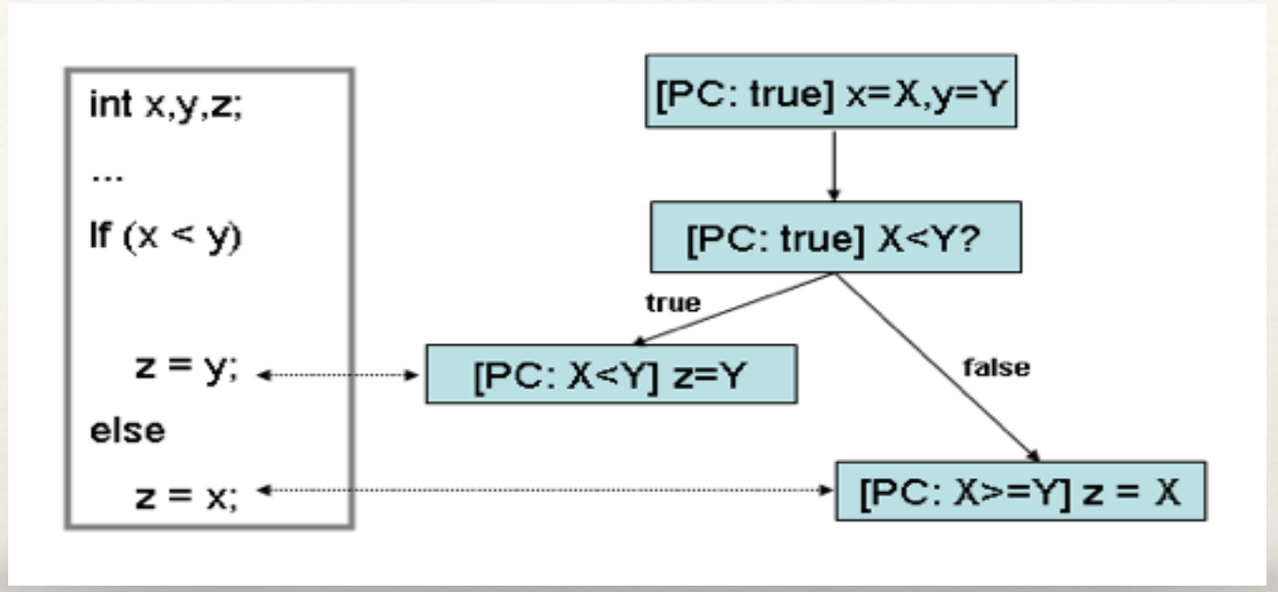
```
void test(int n) {
    int x = 0;
    while(x < n)
      x = x + 1;
}
```

infinite symbolic execution tree

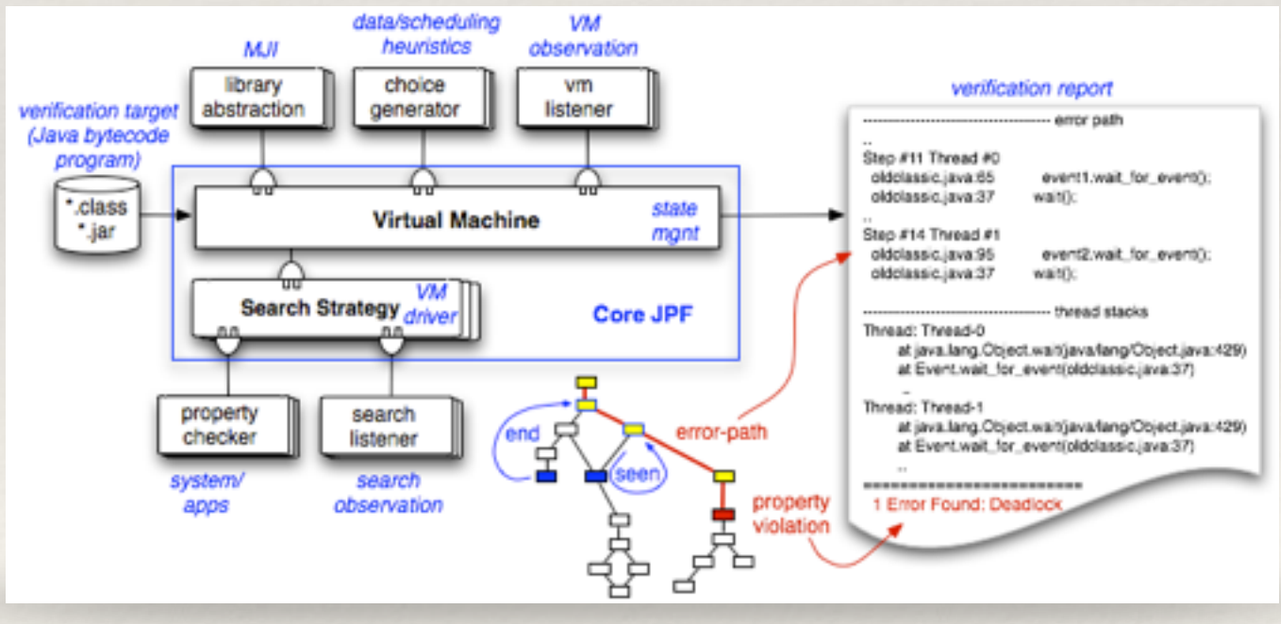


Symbolic PathFinder

- Symbolic execution tool for Java bytecode; **open-sourced**
- **Lazy initialization** for input data structures and arrays
- Handles **multi-threading** and **string** operations
- Supports **quantitative** reasoning
- Comes with **library** models



Java PathFinder tool-set



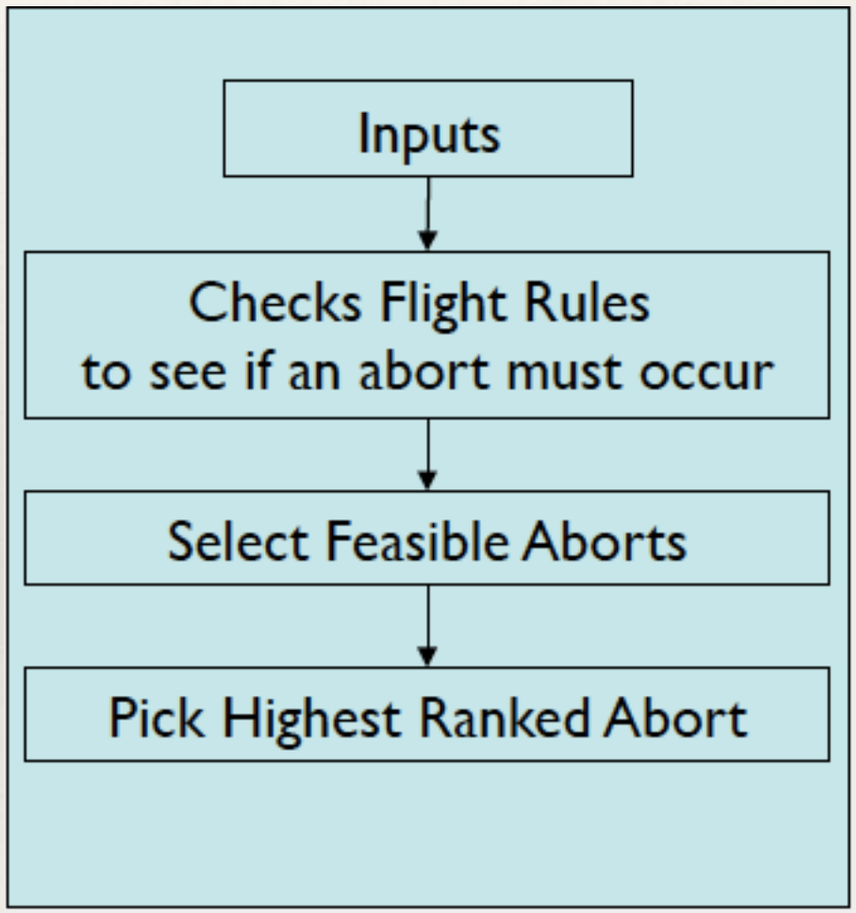
https://github.com/SymbolicPathFinder/jpf-symbc

Test Generation and Bug Finding

NASA Applications

- NASA control software: onboard abort executive (OAE) [ISSTA'08]
  - manual testing: time consuming ~ 1 week
  - guided random testing could not obtain full coverage
  - SPF generated ~200 tests to obtain full coverage **<1min**
  - Flight rules covered 27/27
  - Aborts covered 7/7
  - Size of input: 27 values/test case
- Found major bug in new version

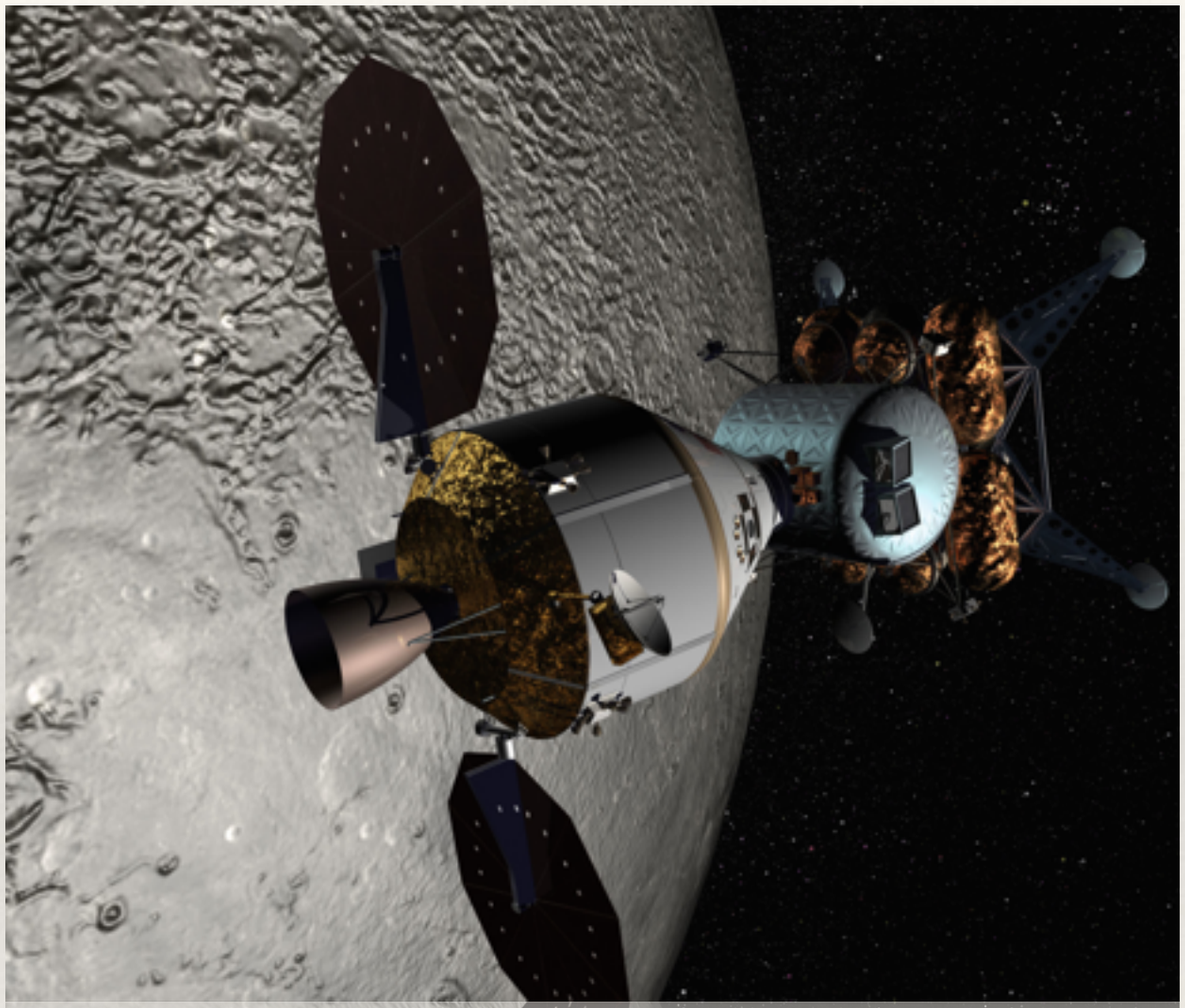
OAE structure



Test Generation and Bug Finding

NASA Applications

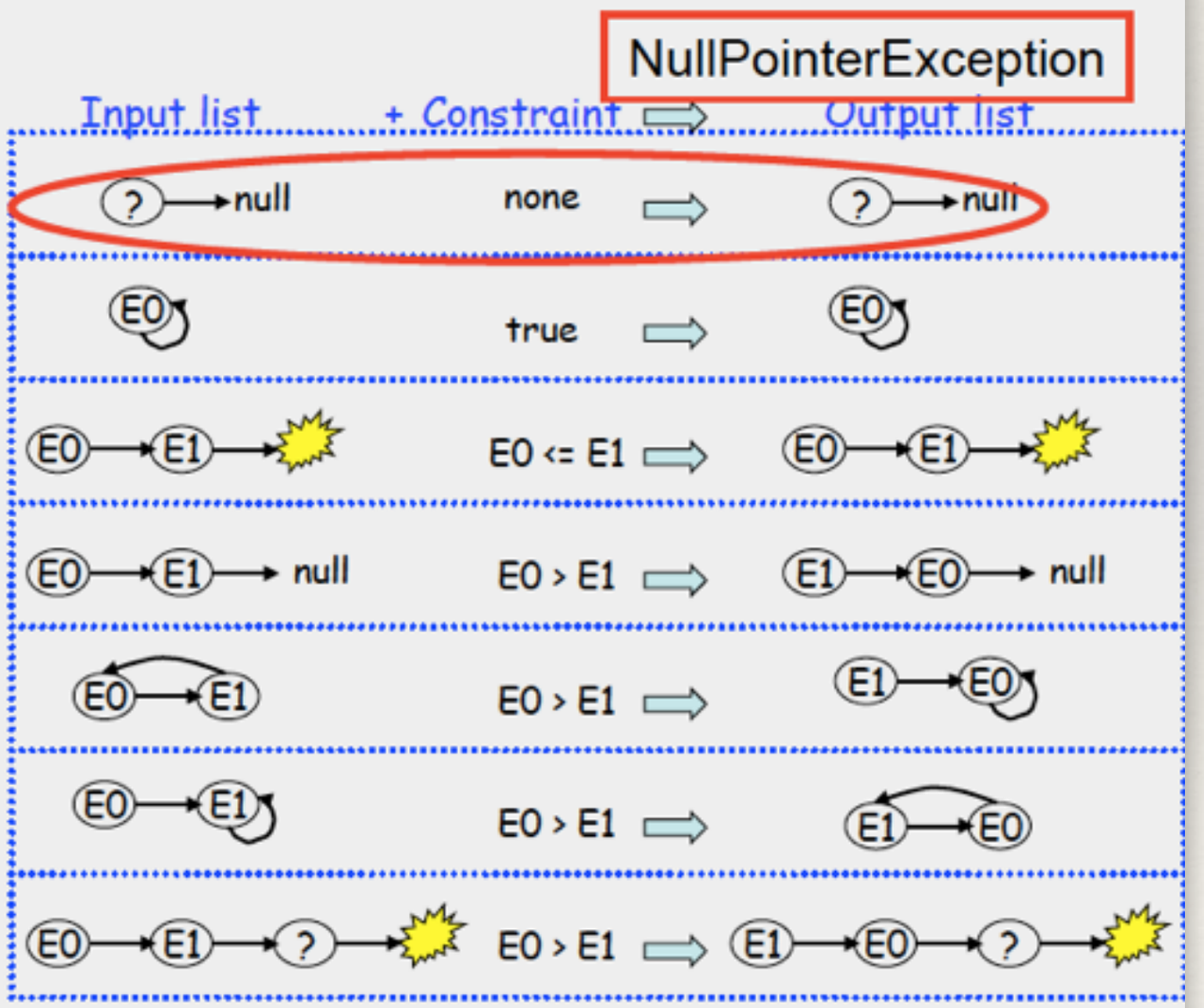
- Polyglot [ISSTA'11, NFM'12]
  - analysis and test generation for UML, Stateflow and Rhapsody models
  - pluggable semantics for different state chart formalisms
  - analyzed MER arbiter, Ares-Orion communication
  - current work: CoCoSim — extensible verification framework for Simulink/ Stateflow
- Tactical Separation Assisted Flight Environment (T-SAFE) [NFM'11, ICST'12]
  - integration with Coral heuristic solver for complex mathematical constraints



Handling Data Structures

- Lazy initialization [TACAS'03,ISSTA'04] — nondeterminism handles aliasing

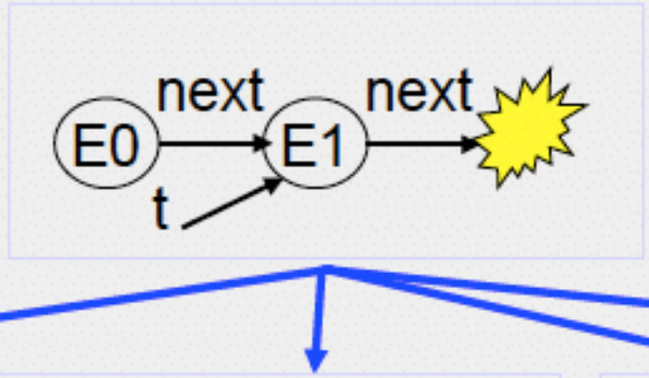
```
class Node {
    int elem;
    Node next;

    Node swapNode() {
        if (next != null)
            if (elem > next.elem) {
                Node t = next;
                next = t.next;
                t.next = this;
                return t;
            }
        return this;
    }
}
```

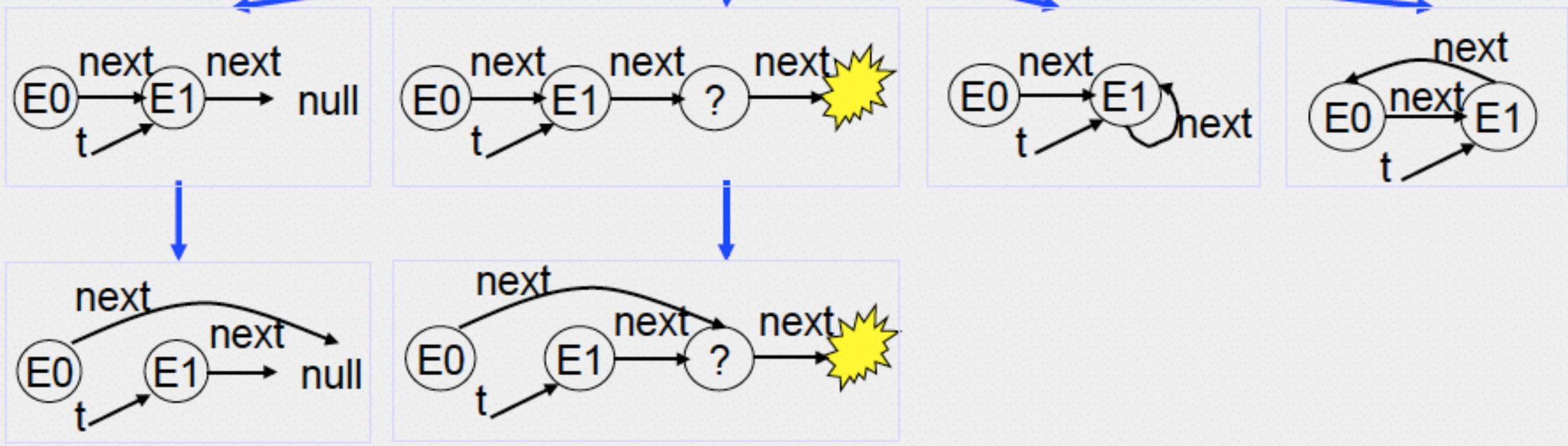


Lazy Initialization

consider executing

next = t.next;





Dynamic Symbolic Execution/Concolic Testing

- collect symbolic constraints **during** concrete executions
- DART = Directed Automated Random Testing
- Concolic = Concrete/symbolic testing

- P. Godefroid, K. Sen and many many others …
- very popular, simple to implement

Dynamic Symbolic Execution/Concolic Testing

	Concrete Execution	Symbolic Execution	Path Constraint
int x, y;			
if (x > y) {			
x = x + y;			
y = x – y;			
x = x – y;			
if (x > y)			
assert false;			
}			

x = 0, y = 0

create symbolic variables x, y

Dynamic Symbolic Execution/Concolic Testing

Concrete Execution	Symbolic Execution	Path Constraint
int x, y;	create symbolic variables x, y	
if (x > y) {		x ≤ y
x = x + y;		
y = x – y;		
x = x – y;		
if (x > y)		
assert false;		
} ← x = 0, y = 0		

Solve: !(x≤y)

Solution: x=1, y=0

Dynamic Symbolic Execution/Concolic Testing

	Concrete Execution	Symbolic Execution	Path Constraint
int x, y;			
if (x > y) {	x = 1, y = 0	create symbolic variables x, y	
x = x + y;			
y = x – y;			
x = x – y;			
if (x > y)			
assert false;			
}			

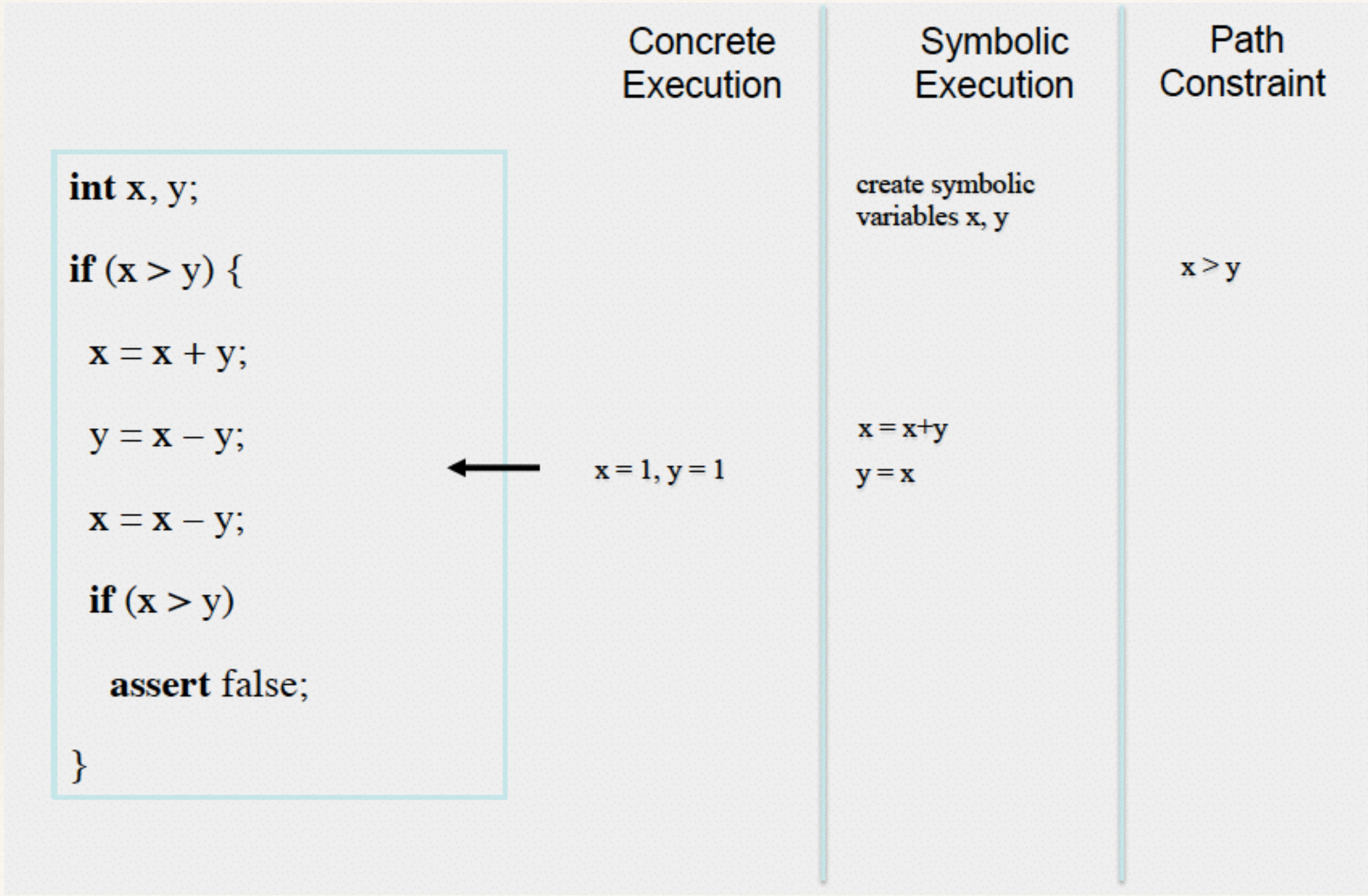
Dynamic Symbolic Execution/Concolic Testing

	Concrete Execution	Symbolic Execution	Path Constraint
int x, y;			
		create symbolic variables x, y	
if (x > y) {			x > y
	← x = 1, y = 0		
x = x + y;			
y = x – y;			
x = x – y;			
if (x > y)			
assert false;			
}			

Dynamic Symbolic Execution/Concolic Testing

	Concrete Execution	Symbolic Execution	Path Constraint
int x, y;		create symbolic variables x, y	
if (x > y) {			x > y
x = x + y;	x = 1, y = 0	x = x+y	
y = x – y;			
x = x – y;			
if (x > y)			
assert false;			
}			

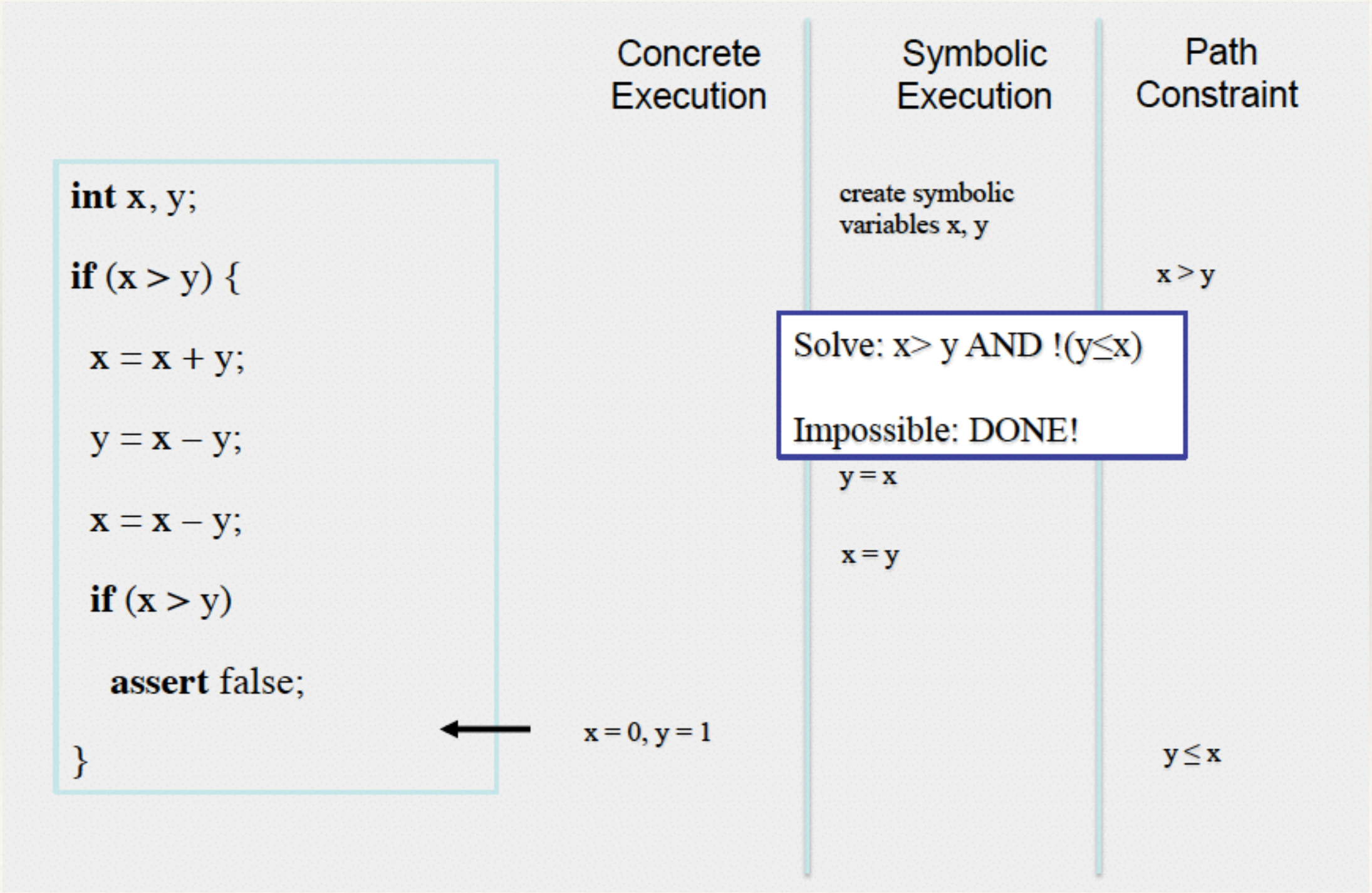
Dynamic Symbolic Execution/Concolic Testing



Dynamic Symbolic Execution/Concolic Testing

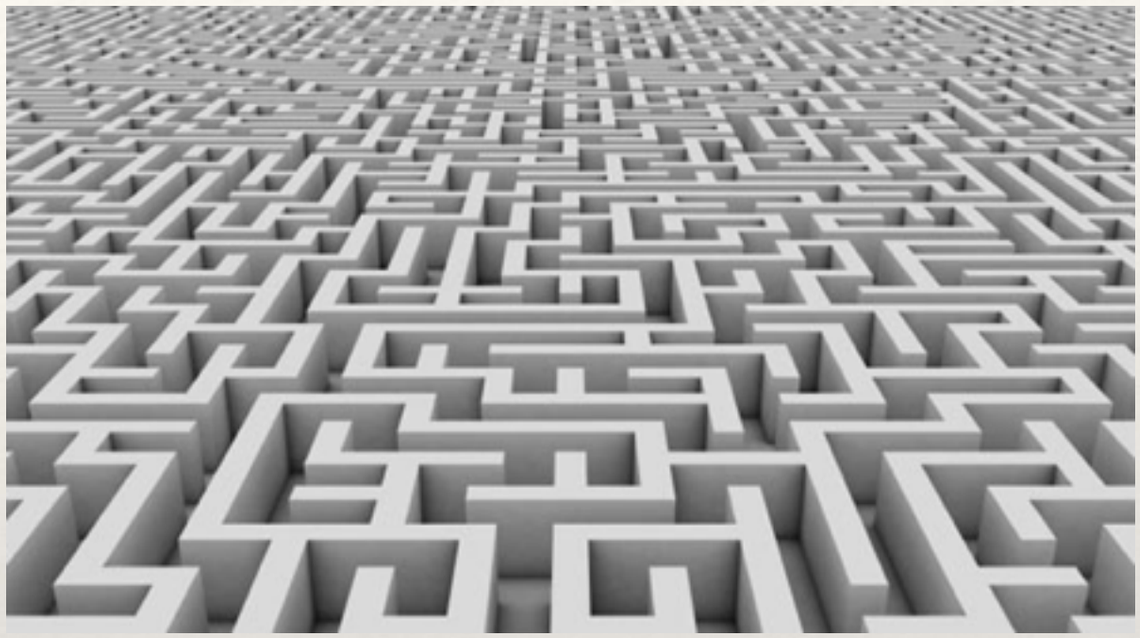
	Concrete Execution	Symbolic Execution	Path Constraint
int x, y;		create symbolic variables x, y	
if (x > y) {			x > y
x = x + y;			y = x
y = x – y;			x = y
x = x – y;	x = 0, y = 1		
if (x > y)			
assert false;			
}			

Dynamic Symbolic Execution/Concolic Testing



Complexity Analysis

- Problem
  - Estimate the worst-case complexity of programs
- Applications
  - Finding vulnerabilities related to denial-of-service attacks
  - Guiding compiler optimizations
  - Finding and fixing performance bottlenecks in software



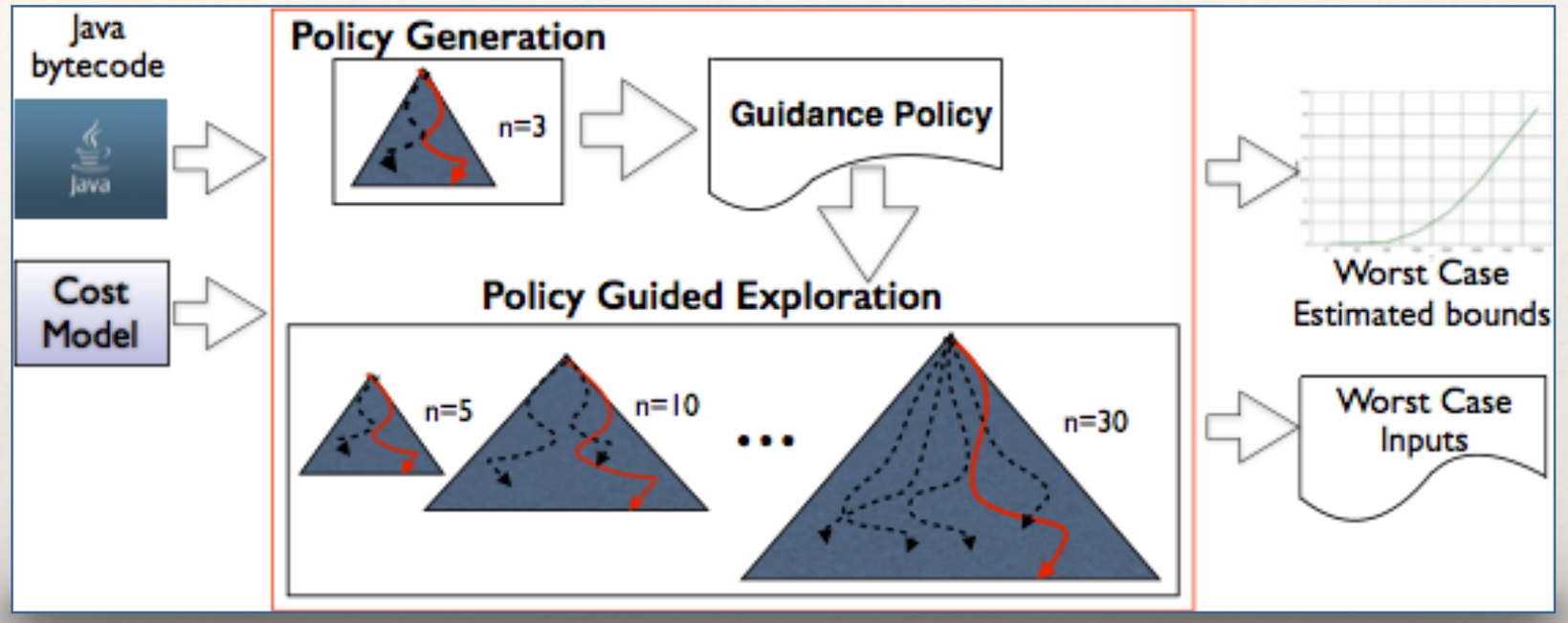
DARPA STAC

Symbolic Complexity Analysis

- Computes inputs that expose worst-case behavior
- Computes bounds on worst-case complexity
- Simple approach
  - Perform symbolic execution over the program — compute cost of each path
  - Return the path with largest cost
  - Scalability issues
- Symbolic execution guided by path policies [ICST'17]
  - Encode choices along worst-case path
  - Intuition: worst-case behavior for small input can predict worst-case behavior for larger input

Guided Symbolic Execution

- Policy Generation
  - Exhaustive symbolic execution at small input size(s)
  - Compute path with largest cost
  - Build policy based on decisions taken along that path
- Policy Guided Execution
  - Symbolic execution for increasing input sizes
  - Explore only paths that conform with policy
  - For each input size compute path (and input) with largest cost



- Function fitting
  - Computes estimate of worst-case behavior as a function of input size
  - Gives lower bounds on worst-case complexity for any size

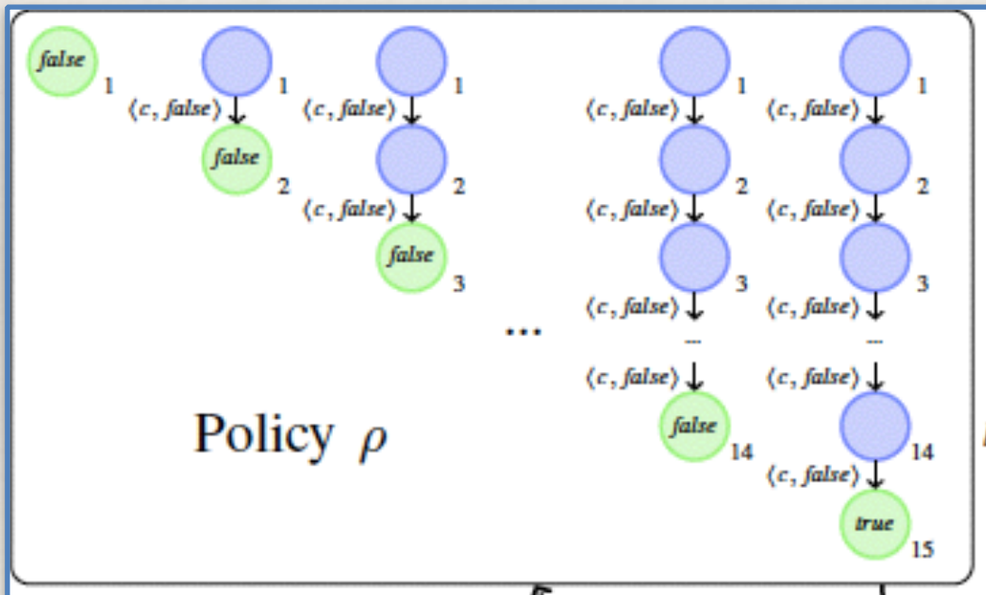
Path Policies

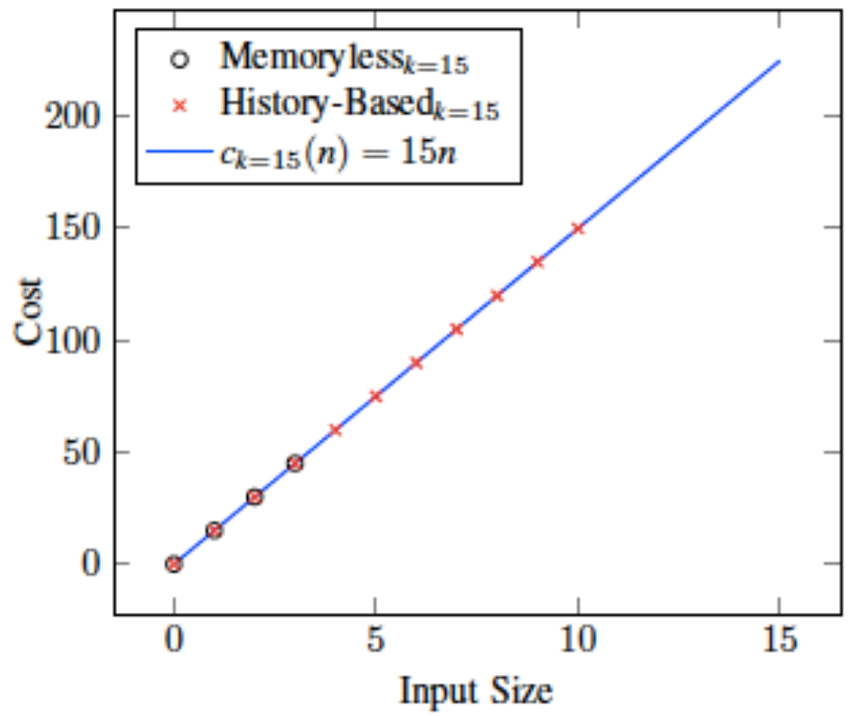
- Decide which branch to execute for the conditions in the program
  - Similar to e.g. [Burnim et al. ICSE'09, Zhang et al. ASE'11]
- **New**
  - History aware: take into account the history of choices made along a path to decide which branch to execute next
    - Context preserving: the decision for each condition depends on the history computed with respect to the enclosing method
- Symbolic execution, guided by policies, can reduce to exploring a single path regardless of input size
- Scales far beyond non-guided symbolic execution and outperforms previous techniques
- **Theoretical guarantee:** when policies are "unified", worst-case path policy is eventually found
  - Unification over policies obtained for successive small inputs
  - For each condition: take union over decisions specified by each policy

Example

```
....
7 Entry findEntry(String o, ....) {
8    for(Entry e = l; e!=null; e=e.next) {
9       if (e.key.equals(o)) {
10         return e;
11      }
12   }
....
16   return null;
17 }
```

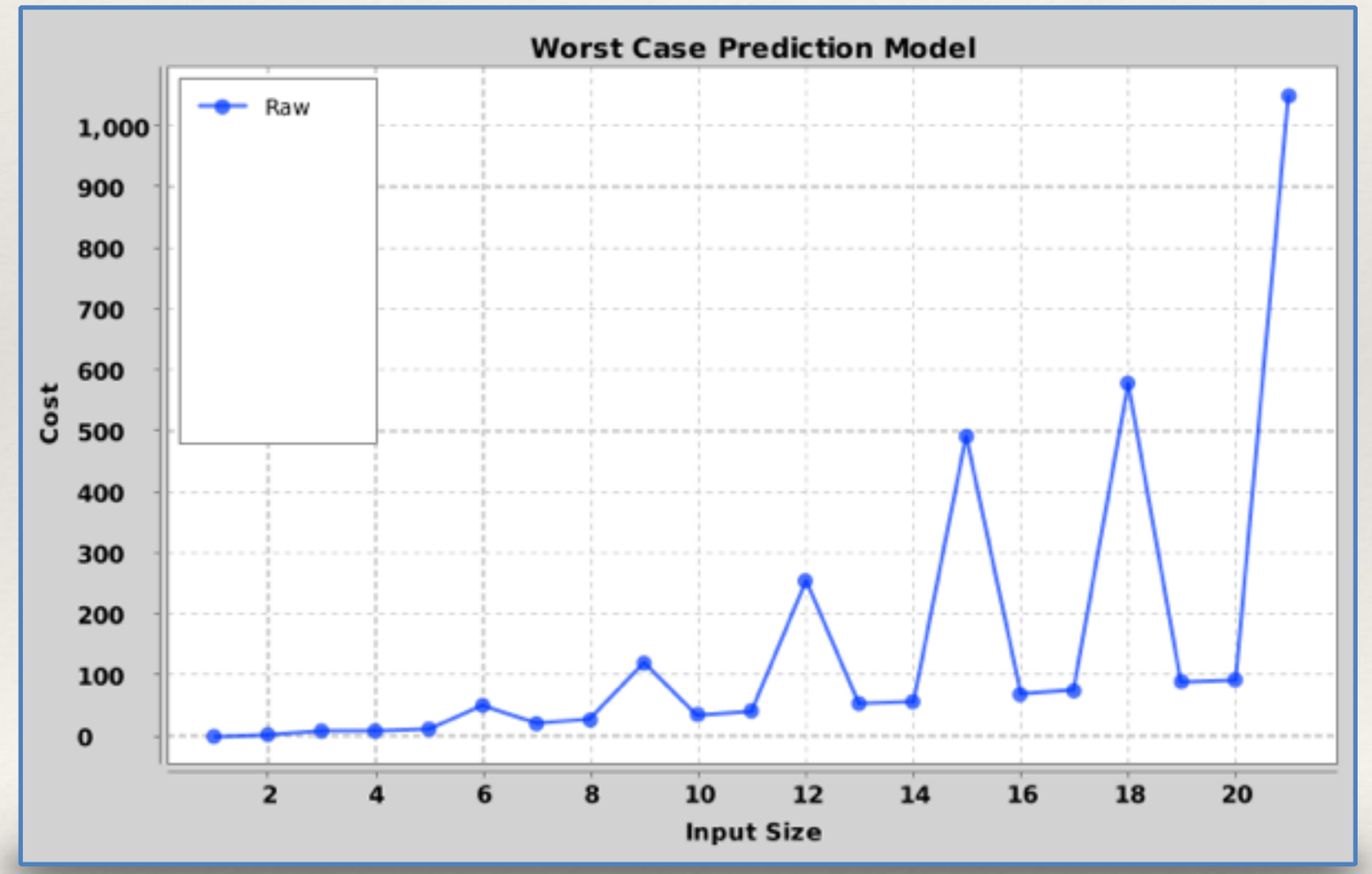
```
18 class String {
19 char[] value;
20 // ...
21  public boolean equals(Object oObj) {
22    // ...
23    String o = (String) oObj;
24    if (val.length == o.val.length) {
25       for(int i=0; i<val.length; i++) {
26          if (val[i]!=o.val[i])
27             return false;
28       }
29       return true;
30    }
31    return false;
32  }
33 }
```





Example Application: TextCruncher Sort

- Text processing application with various filters, e.g. *WordCount, NGramScore*
- Found vulnerability in sorting algorithm
- Triggered by files with 3 x n different words: 6000 words: 5 min; 6001 words: few secs.



Vulnerability: exponential for lists of length n × 3

From ISSTAC project

DARPA STAC

Probabilistic Reasoning

- Extension of symbolic execution with probabilistic reasoning [ICSE'13,PLDI'14]
  - Computes the probability of a target event, under an input distribution
- Model counting over symbolic constraints
  - Latte, Barvinok -- integer linear constraints, finite domain

Probabilistic Reasoning

- E.g. assuming uniform distribution,
- Compute path conditions that lead to target event
- Count the number of input values that satisfy the corresponding path conditions
- Divide it by the size of the input domain (D)

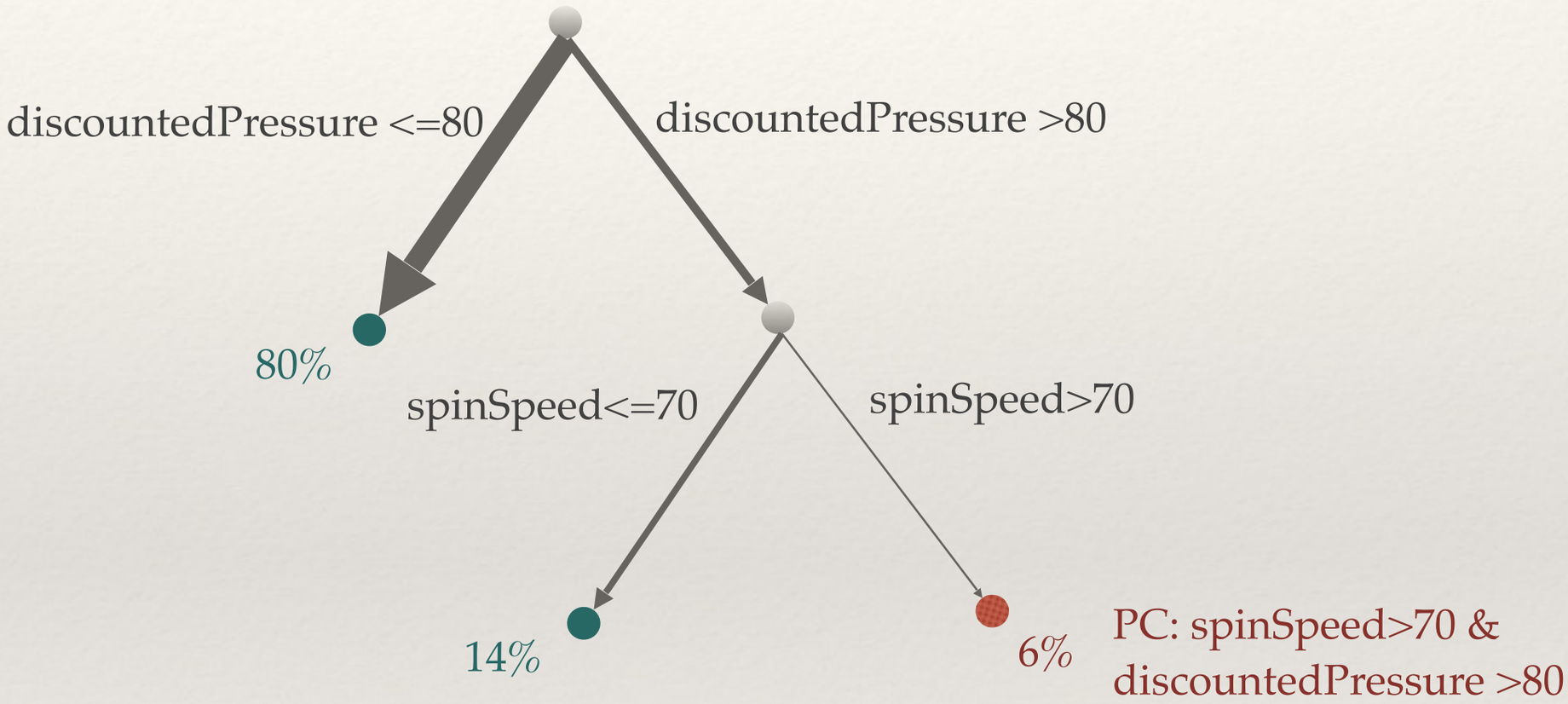
Probability of event e:

$$p(e) = \frac{1}{\sharp D} \sum_i \sharp PC_i$$

PC_i **leads to** e.

Example

input domain 100 x 100



Pr(Fail) = #(PC)/#D

= #(spinSpeed>70 & discountedPressure >80)/D

= 30 x 20/10000 = 6%

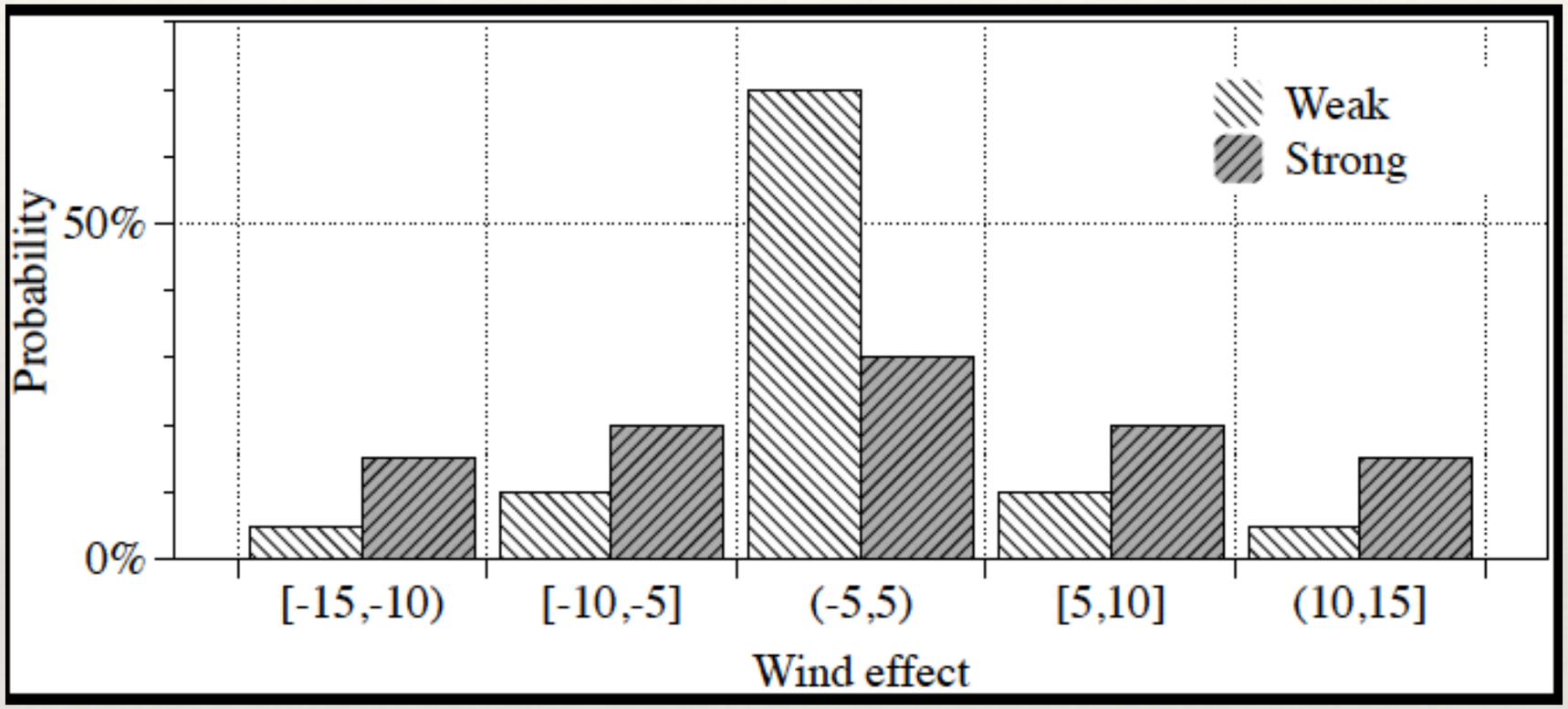
Software Reliability

- Probability of successful termination under stochastic environment assumptions
- Perform bounded symbolic execution: results in three sets of paths
  - Success PC^s: lead to successful termination
  - Fail PC^f: lead to failure
  - Grey PC^g: "don't know"
- For given usage profile UP: $Pr(Fail \mid UP) = Pr(PC^f s \mid UP)$, e.g. for uniform UP:

  $Pr(Fail) = \#(PC^f)/D = \#(spinSpeed>70 \ \& \ discountedPressure>80)/D = 30 \times 20/10000 = 6\ \%.$
- $Pr(Success \mid UP)$ and $Pr(Grey \mid UP)$ are computed similarly

- $Pr(Fail \mid UP)+Pr(Success \mid UP)+Pr(Grey \mid UP)=1$
- $Rel = Pr(Success \mid UP)$
- $Confidence = 1 - Pr(Grey \mid UP)$ ("1" means that analysis is complete)

Usage Profiles



- Summarize succinctly hundreds of hours of operation/simulation
- UPs can be seen as "pre-conditions"
- Arbitrary UPs – handled through discretization
- Continuous input distributions [FSE'15]

Computing with usage profiles

- Usage profile: set of pairs $<c_i, p_i>$
- c_i — usage scenario, constraint on inputs
- p_i — probability that the input is in c_i

$$Rel = Pr^s(P) = \sum_i Pr(PC_i^s \mid UP) =$$

$$= \sum_i \sum_j Pr(PC_i^s \mid c_j) \cdot p_j = \sum_i \sum_j \frac{\sharp(PC_i^s \wedge c_j)}{\sharp(c_j)} \cdot p_j$$

Model Counting

- Latte, Barvinok: integer linear constraints, finite domain — Polynomial in number of variables and constraints
  - Omega Lib used for algebraic simplifications
  - Optimizations: independence, caching
- Research on
  - model counting for data structures [SPIN'15],
  - strings [FSE'16] — ABC Solver (UC Santa Barbara)
  - non-linear constraints [NFM'17]

Model Counting for Data Structures

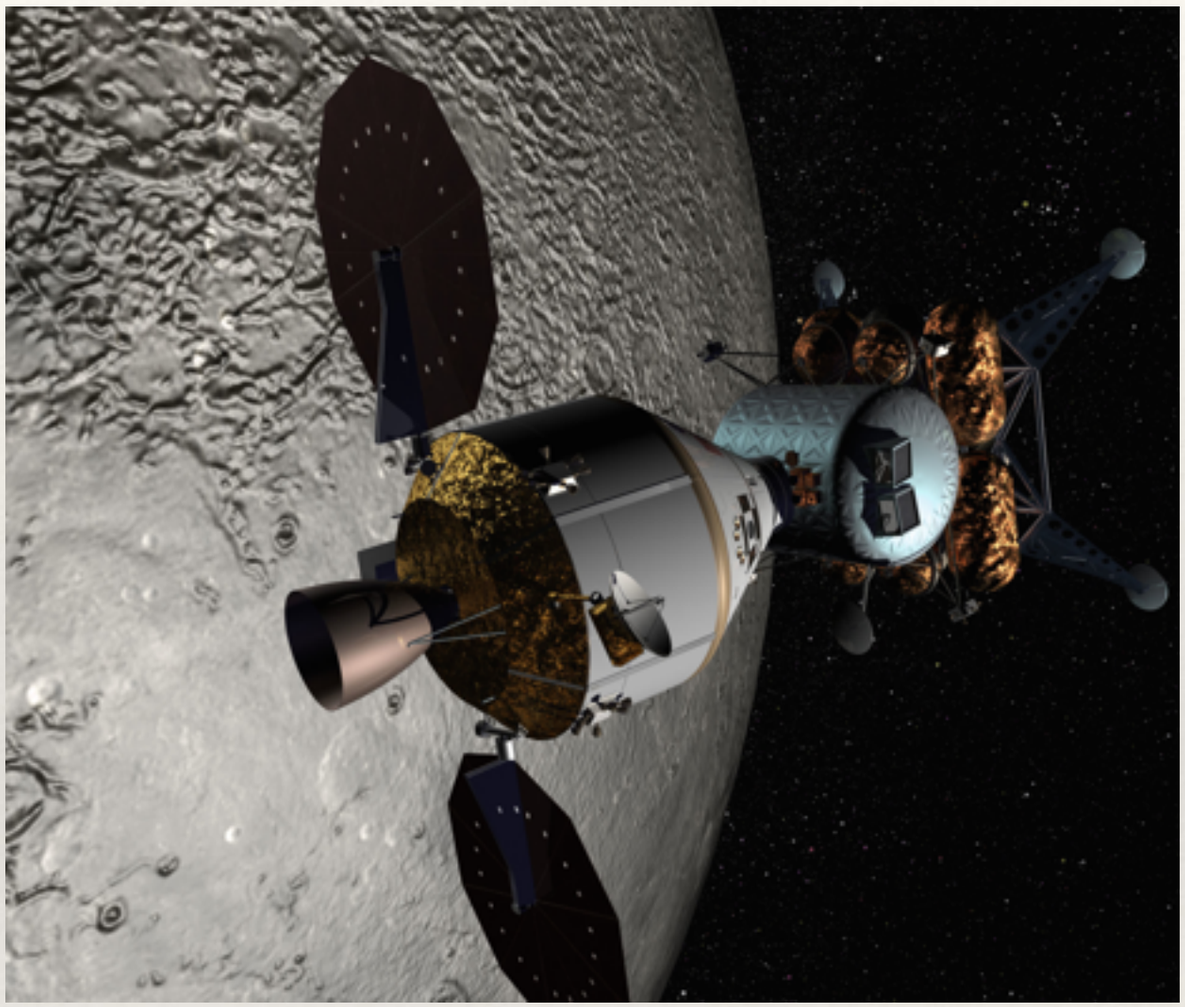
- SPF performs lazy initialization
- Computes Heap PC
- Explicit enumeration using Korat (MIT)
- Complex predicates
  - E.g. “acyclic lists of integers with size smaller than the largest contained value”
- Computationally expensive

Multi-threading

- Enumerate all possible schedules (using model checking, partial order reduction)
  - Compute best/worst "reliability"
  - Report best/worst schedule
  - Useful for debugging
- Tree-like schedules [ASE'15]
  - Monte-Carlo sampling of symbolic paths
  - Reinforcement learning used to iteratively compute schedules

Application: Onboard Abort Executive

- NASA control software
  - Mission aborts
  - 3754 paths, 36 input sensors
  - 30 usage scenarios
  - Execution time: 20.5 sec
  - Checking for “no aborts”
  - Rel > 0.9999999



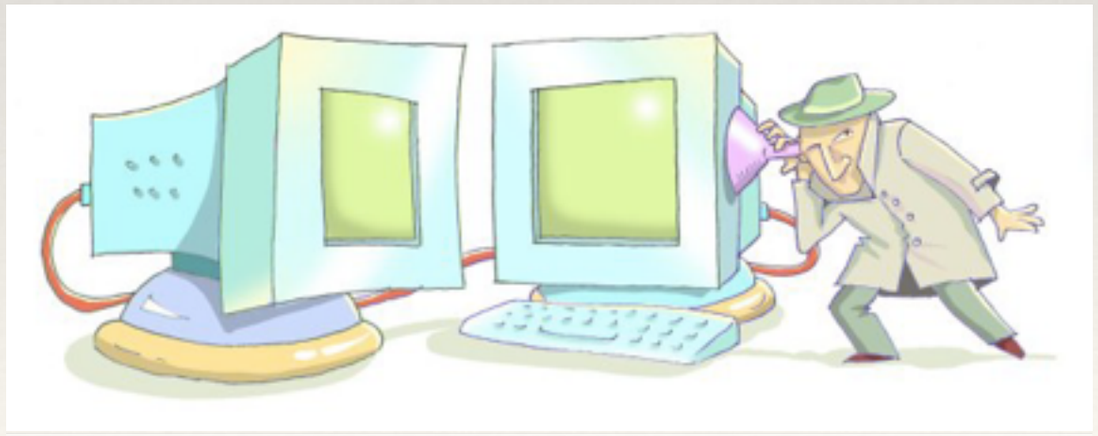
Side-Channel Analysis

- Side-channel attacks
  - recover secret inputs to programs from non-functional characteristics of computations
  - time or power consumption, number of memory accesses or size of output files

```
boolean verifyPassword(byte [] input,
                       byte [] password) {
  for ( int i = 0; i < SIZE; i++) {
   if (password[ i ] != input[ i ])
     return false ;
   Thread.sleep(25L);
  }
  return true;
}
```

low
high

- An attack on "main" channel: exponential
- On "side channel": linear



Side-Channel Analysis

- Non-interference — too strict
- Quantitative Information-Flow Analysis (QIF) to determine information leakage

- Perform symbolic execution (high and low symbolic)
- Collect all symbolic paths — each path leads to an observable

- Side channels produce a set of "observables" that partition the secret
- *Cost model* for observables: execution time, number of packets sent/received over network, etc.

$$\mathcal{O} = \{o_1, o_2, ...o_m\},$$

Quantifying Information Leakage

Channel Capacity

$$CC(P) = log_2(|\mathcal{O}|)$$

Shannon Entropy

$$\mathcal{H}(P) = -\sum_{i=1,m} p(o_i)\log_2(p(o_i))$$

Computing Shannon Entropy

$$\mathcal{H}(P) = -\sum_{i=1,m} p(o_i)\log_2(p(o_i))$$

- Use symbolic execution and model counting

the probability of observing o_i is:

$$p(o_i) = \frac{\sum_{cost(\pi_j)=o_i} \sharp(PC_j(h,l))}{\sharp D}$$

Password Example

```
// 4-bit input and password; D=256
boolean verifyPassword(byte [] input,
                       byte [] password){
  for(int i = 0; i < SIZE; i++){
   if (password[i]!=input[i])
     return false ;
   Thread.sleep(25L);
  }
  return true;
}
```

```
// 4-bit input and password; D=256
boolean verifyPassword(byte [] input,
                       byte [] password){
  boolean matched=true;
  for(int i = 0; i < SIZE; i++){
   if (password[i]!=input[i])
     matched=false ;
   else
     matched=matched;
   Thread.sleep(25L);
  } return matched; }
```

Corrected!

- 5 paths
  - *h[0]!=l[0]* returns false: *128 values*
  - *h[0]=l[0] & h[1]!=l[1]* returns false: *64 values*
  - *h[0]=l[0] & h[1]=l[1] & h[2]!=l[2]* returns false: *32 values*
  - *h[0]=l[0] & h[1]=l[1] & h[2]=l[2] & h[3]!=l[3]* returns false: *16 values*
  - *h[0]=l[0] & h[1]=l[1] & h[2]=l[2] & h[3]=l[3]* returns true: *16 values*

Observable is time: *H=1.875*

Observable is output: *H=0.33729*

Maximizing Leakage

```
void example(int lo, int hi) {
 if(lo<0){
  if(hi<0) cost=1;
  else if(hi<5) cost=2;
  else cost=3;
 }
 else {
  if(hi>1) cost=4;
  else cost=5;
 }
}
```

- using symbolic low value over-approximates leakage
- example: 5 possible observables; **lo<0: 3 observables**, lo≥0: 2 observables

- Goal: find low input that maximizes number of observables (channel capacity)
- Shows most powerful "attack" in one step
- Shows most vulnerable program behavior

Maximizing Leakage using MaxSMT

```
void example(int lo, int hi) {
 if(lo<0){
  if(hi<0) cost=1;
  else if(hi<5) cost=2;
  else cost=3;
 }
 else {
  if(hi>1) cost=4;
  else cost=5;
 }
}
```

$$C_1 :: (l < 0 \wedge h_1 < 0)$$
$$C_2 :: (l < 0 \wedge h_2 \geq 0 \wedge h_2 < 5)$$
$$C_3 :: (l < 0 \wedge h_3 \geq 5)$$
$$C_4 :: (l \geq 0 \wedge h_4 > 1)$$
$$C_5 :: (l \geq 0 \wedge h_5 \leq 1)$$

MaxSMT solution: *Lo=-1* satisfies first 3 clauses

Leakage $\log_2(3)$=1.58 bits

- MaxSMT solving — generalization of SMT to optimization
  - given a set of weighted clauses
  - find solution that maximizes the sum of the weights of the satisfied clauses
- Assemble PCs that lead to same observable into "**clauses**" of weight "1"
- MaxSMT solution gives maximal assignment ⇒ largest number of observables
- Any other assignments lead to fewer observables

Multi-run Analysis

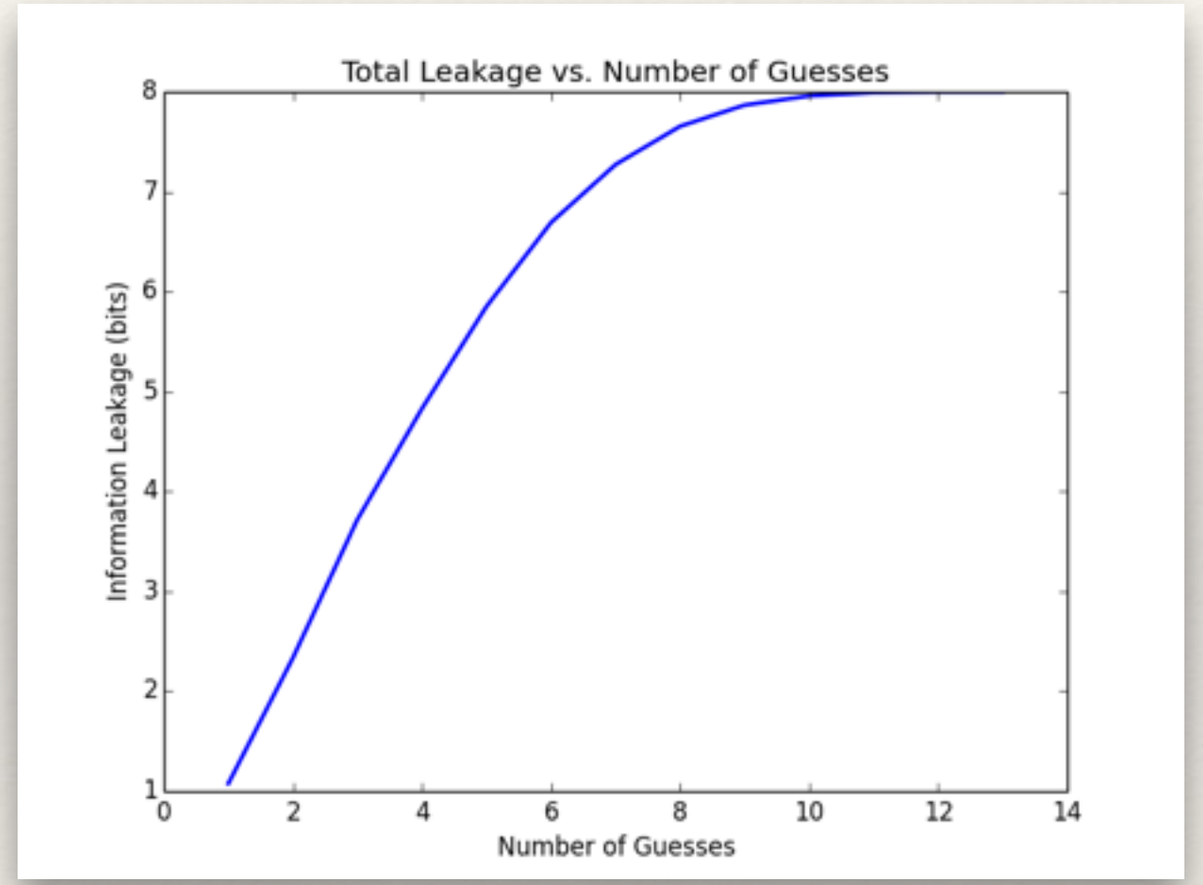
- The attacker learns the secret by observing multiple program runs
- Generalization to multiple-run side-channel analysis

$$P(h, l_1); P(h, l_2); ..P(h, l_k)$$

- An “observable” is a **sequence** of costs
- MaxSMT used to synthesize a sequence of public inputs that maximize leakage; non-adaptive attacks; greedy approach [CSF’16]
- Maximize Shannon leakage: parameterized model counting+ numerical optimization; adaptive attacks [CSF’17]
- Analysis of password examples and cryptographic functions
- Shown experimentally to perform better than previous approaches based on self composition or brute-force enumeration
- More work on side-channel analysis [ISSTA’18]

Results for Password Check

Results for 4 elements with 4 values (8 bits of information)

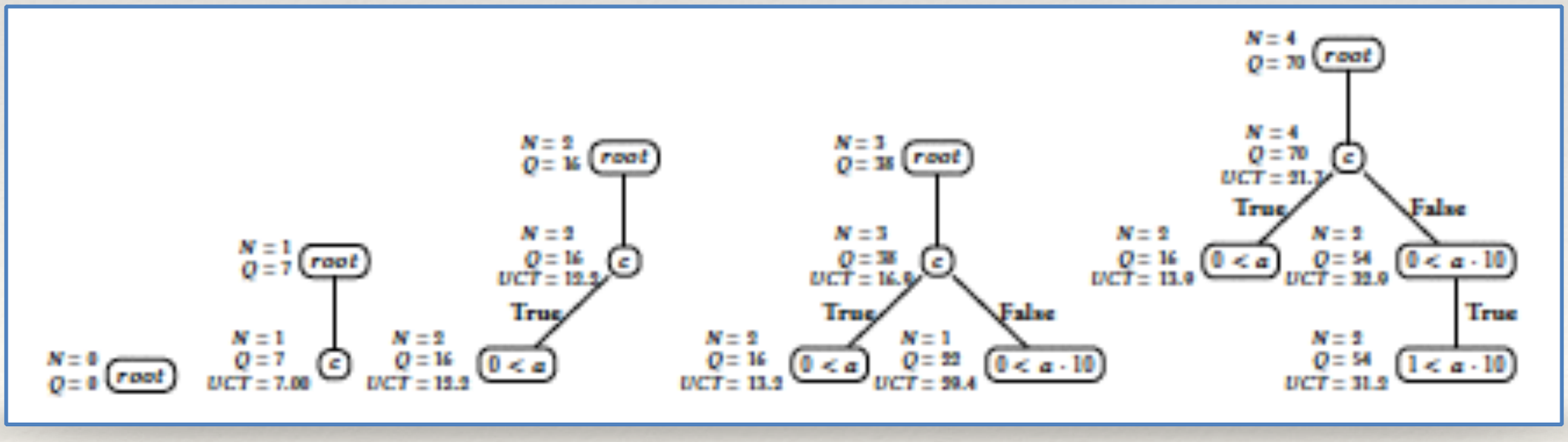


Timing Side Channel

Current/Future Work

Monte Carlo Tree Search For SW Analysis

- Monte Carlo Tree Search [SEFM'18]
  - Heuristic search algorithm; Iterative expansion of search tree to find optimal decisions
  - State-of-the art results in solving Go, board games, poker
  - Good for domains modeled as a tree
- Sampling along symbolic paths for increased scalability
  - Symbolic paths represent multiple concrete paths; Organized in tree
  - Optimize with respect to the longest path (highest reward)
- Aggressive pruning of state space
  - Speeds up analysis and guarantees convergence



Symbolic Execution and Fuzzing

- Fuzzing: random testing with some guidance
  - cheap
  - not good at finding “deep paths” that depend on complicated constraints
- Symbolic execution
  - expensive
  - good at finding deep paths
- Better together!
- See **Badger** talk at ISSTA’18 on Wednesday

Probabilistic Analysis for Autonomous Vehicles

SafeTugs

- Currently aircraft either needs to use their engines or be towed during departure/arrival ground operations
- Engines off is more efficient
- This project will focus on autonomous tugs for towing

Analysis

- Predictive analysis for safe surface and air operation [HLDVT'16]
  - Involves model inference from telemetry/simulation data
- "Simulation" environment using Symbolic PathFinder and probabilistic reasoning [PHS'16]
  - Planning phase generates a plan of tug movement on a grid (abstraction of the airport)
  - The plan is given as input to SPF; calculate how robust the plan is when the probabilities are changed
  - The output of our tool can be used to trigger dynamic re-planning during operation

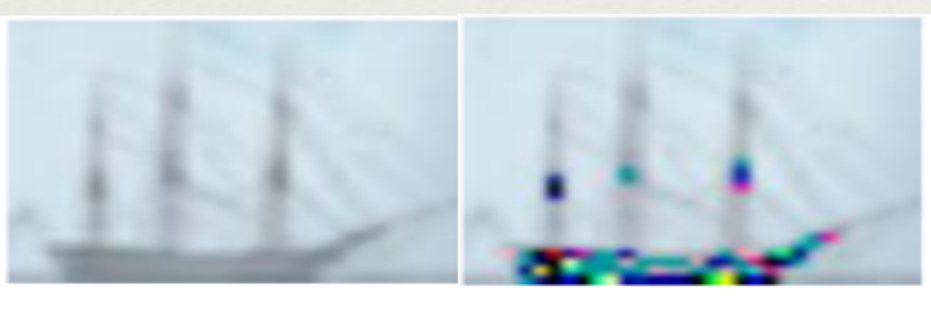
P**rogress**: see workshop talk on Thursday at TAV-CPS/IoT !

Checking Robustness of Deep Neural Nets

- Deep Learning
  - Machine learning that enables representation and modeling of complex non-linear relationships
  - Neural Networks (feed-forward, convolutional), Deep Belief Networks
- Application domains:
  - Pattern analysis, image classification, speech/audio recognition, perception modules in self-driving cars
  - High-dimensional, Classifiers are non-linear and potentially discontinuous
- Deep Neural networks are vulnerable to adversarial inputs:
  - given input x, find new input x′ that is "similar" to x but is assigned a class different from x by the network [Szegedy et. al. 2013]?
- Current research: use symbolic execution and k-means clustering for robustness check
  - $|x-x'| < d \Rightarrow F(x) = F(x')$; a counterexample is an adversarial input
  - w/ D. Gopinath (CMU)
  - **see ATVA'18 talk!**

Street sign

Birdhouse



ship truck

Quantification of Software Changes

- Programs evolve during development and maintenance
  - There is a need for detection and characterization of software changes
- Current techniques
  - Syntactic: diff, imprecise, leads to unnecessary maintenance work
  - Behavioral: check logical implication between behavioral abstractions: yes/no answers
- Quantitative representation of program change [ASE'15]

- Probability of reaching program events — how that evolves in time
  - rank program versions based on probability of failure
  - after bug fixing probability of failure should decrease
- Percentage of inputs affected by change
  - measurable delta between program versions
  - measurable effort to re-test
- Automated program repair:
  - rank repairs based on probability of success/failure

Conclusion

- Symbolic execution and its extension to probabilistic reasoning
- Applications in program analysis for safety and security
- Future directions …
  - Leakage computation for noisy side channels [CSF'18]
  - Distributed analysis— lots of opportunities for symbolic execution
  - Combinations with fuzzing
- Challenges
  - scalability; handling loops
  - non-linear numeric constraints, string constraints: constraint solving, (parametrized) model counting, MaxSMT

Thank you

Contact information: corina.s.pasareanu@nasa.gov,

[redacted]

pcorina@cmu.edu