Symbolic Execution and Software Testing

Corina Pasareanu
Carnegie Mellon/NASA Ames
corina.s.pasareanu@nasa.gov

Overview

- "Classical" symbolic execution and its variants
  - Generalized symbolic execution
  - Dynamic and concolic testing
- Challenges
  - Multi-threading, complex data structures
  - Complex constraints
  - Handling loops, native libraries
- Scalability issues – path explosion problem
  - Compositional and parallel techniques
  - Abstraction
- Applications & Tools

Symbolic Execution

- King [Comm. ACM 1976] , Clarke [IEEE TSE 1976]
- Analysis of programs with unspecified inputs
  - Execute a program on symbolic inputs
- Symbolic states represent sets of concrete states
- For each path, build a path condition
  - Condition on inputs for the execution to follow that path
  - Check path condition satisfiability -- explore only feasible paths
- Symbolic state
  - Symbolic values/expressions for variables
  - Path condition
  - Program counter

Example: Standard 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

x = 1, y = 0

↓

1 > 0 ? true

↓

x = 1 + 0 = 1

↓

y = 1 – 0 = 1

↓

x = 1 – 1 = 0

↓

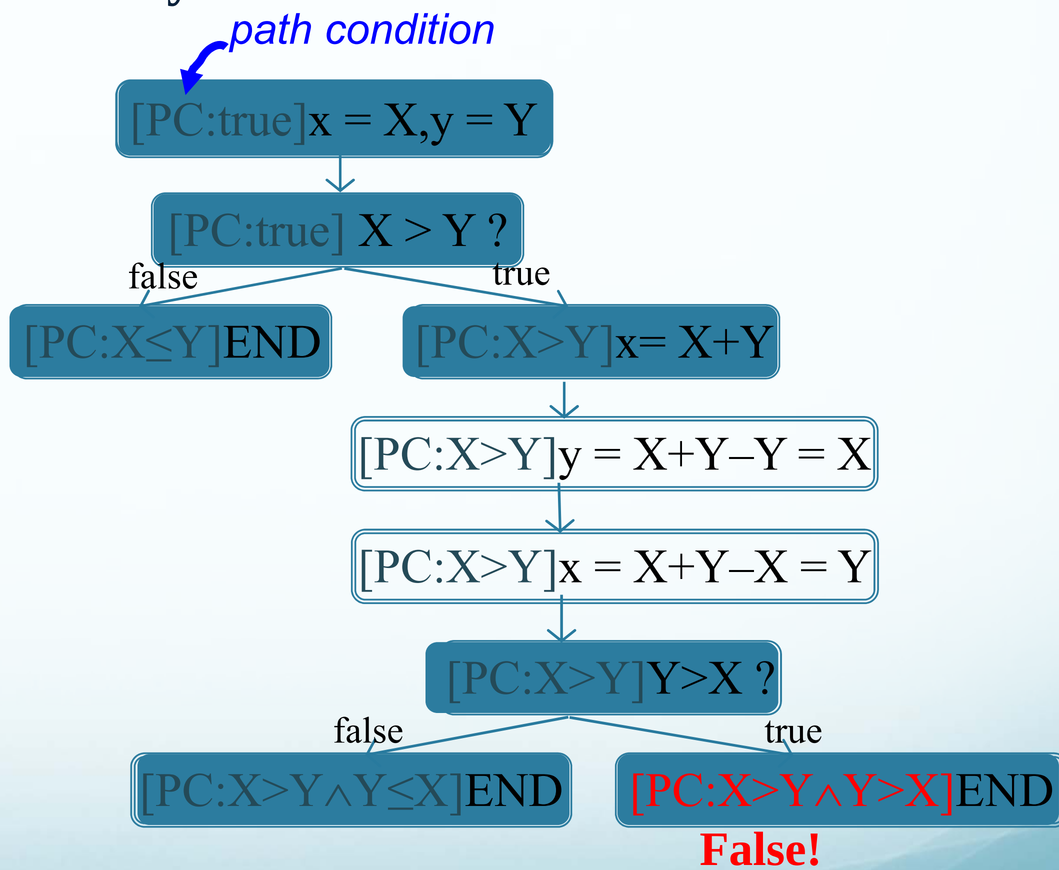
0 > 1 ? false

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 path conditions → test inputs

Questions

- What about loops?
- What about overflow?

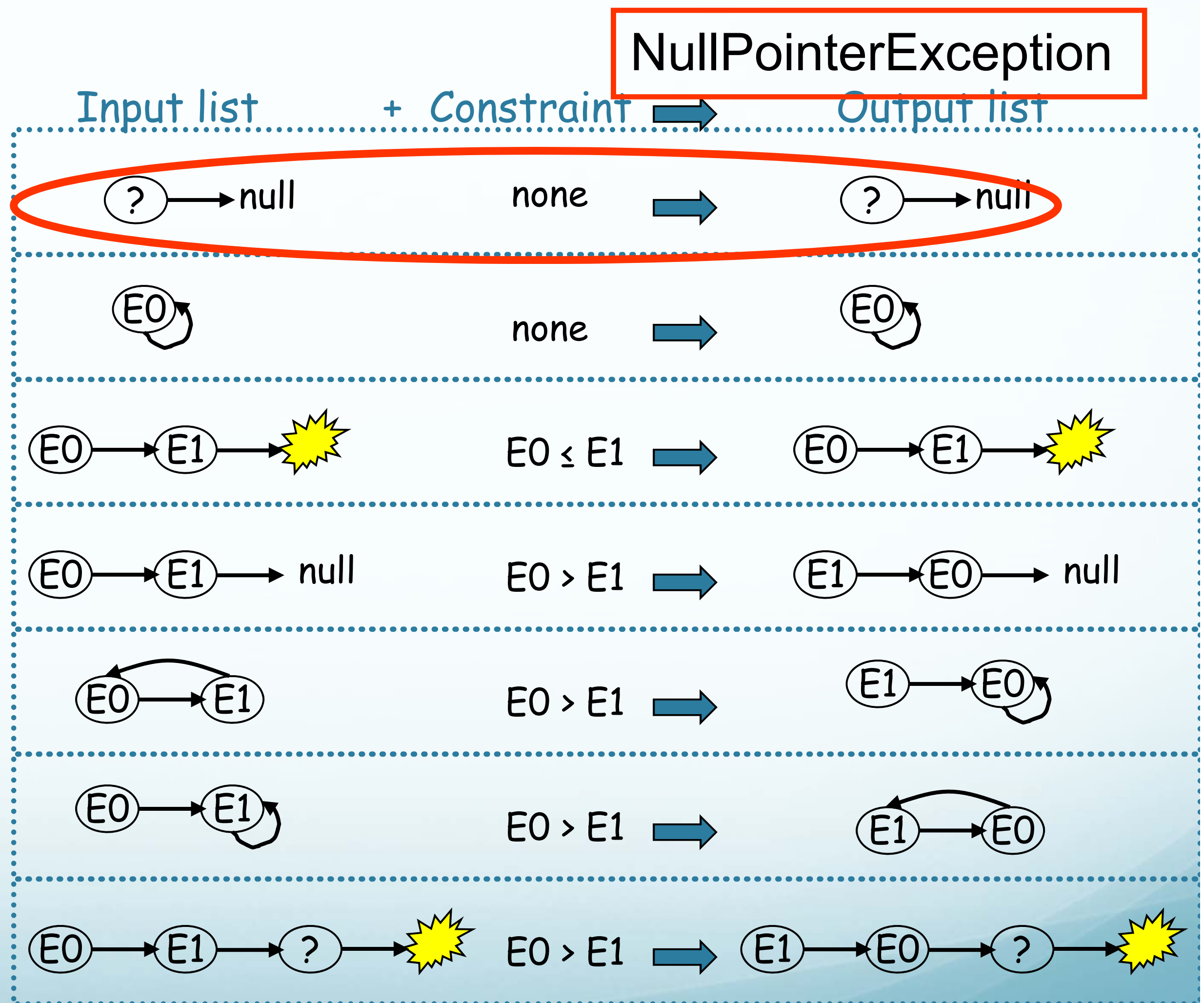
- What about multi-threading?
- What about data structures?

Generalized Symbolic Execution [TACAS'03]

- Handles dynamically allocated data structures and multi-threading
- Key elements:
  - Lazy initialization for input data structures
  - Standard model checker (Java PathFinder) for multi-threading
- Model Checker
  - Analyzes thread inter-leavings
  - Leverages optimizations
    - Symmetry and partial order reductions, abstraction etc.
  - Generates and explores the symbolic execution tree
  - Explores different heap configurations explicitly -- non-determinism handles aliasing

Example Analysis

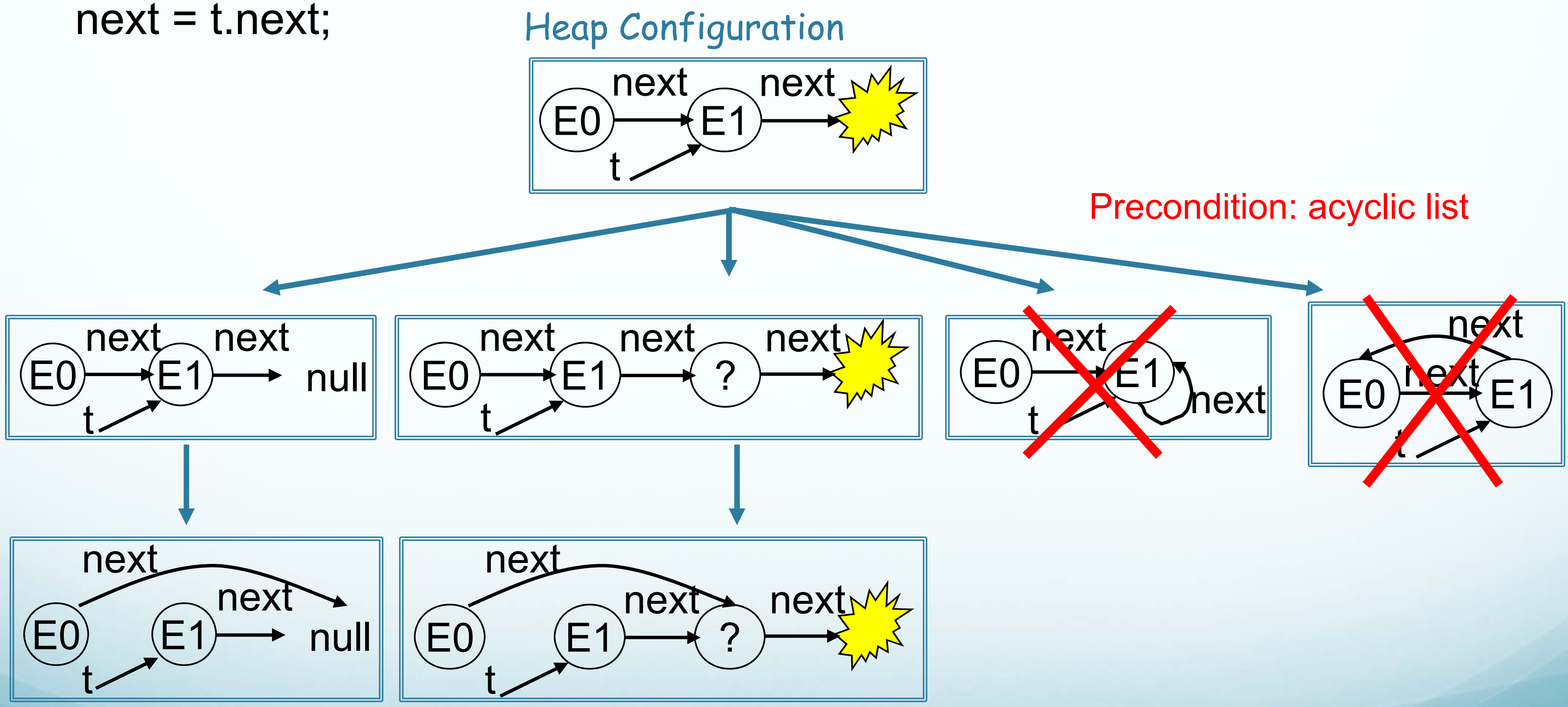
```
class Node {
    int elem;
    Node next;

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



Lazy Initialization (illustration)

consider executing

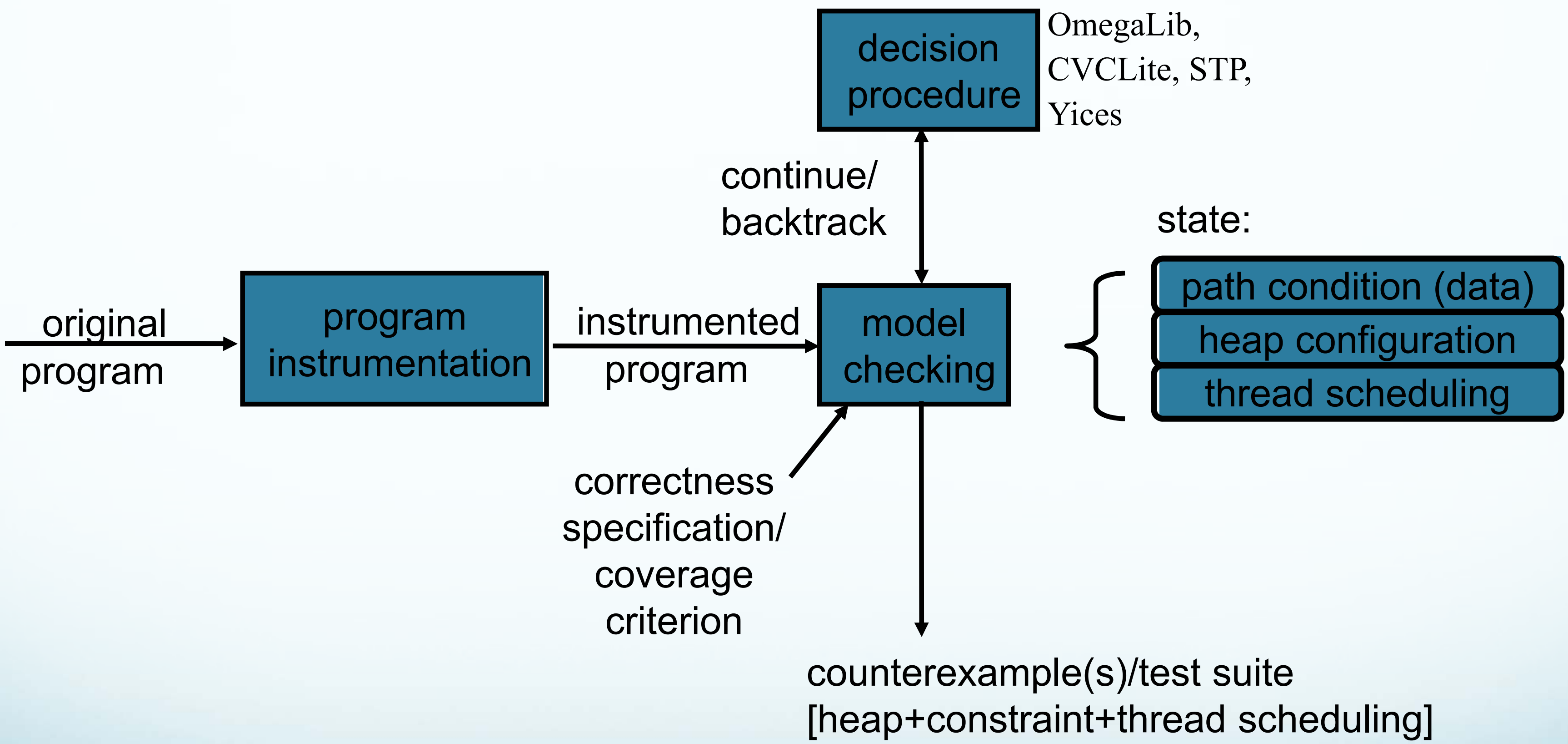
next = t.next;



Implementation

- Symbolic execution of Java programs
- Code instrumentation
  - Programs instrumented to enable JPF to perform symbolic execution
  - Replace concrete operations with calls to methods that implement symbolic operations
  - General: could use/leverage any model checker
- Decision procedures used to check satisfiability of path conditions
  - Omega library for integer linear constraints
  - CVCLite, STP (Stanford), Yices (SRI)

Implementation via Instrumentation



Symbolic PathFinder

- No longer uses code instrumentation
- Implements a non-standard interpreter of byte-codes
  - Enables JPF to perform symbolic analysis
  - Replaces standard byte-code execution with non-standard symbolic execution
- During execution checks for assert violations, run-time errors, etc.
- Symbolic information:
  - Stored in attributes associated with the program data
  - Propagated dynamically during symbolic execution
- Choice generators and listeners:
  - Non-deterministic choices handle branching conditions
  - Listeners print results: path conditions, test vectors/sequences
- Native peers model native libraries:
  - Capture `Math` calls and send them to the constraint solver
- Generic interface for multiple decision procedures
  - Choco, IASolver, CVC3, Yices, HAMPI, CORAL [NFM11], etc.

Example: IADD

Concrete execution of IADD byte-code:

```
public class IADD extends
   Instruction { …

 public Instruction execute(…
   ThreadInfo th){

   int v1 = th.pop();

   int v2 = th.pop();

   th.push(v1+v2,…);

   return getNext(th);

 }

}
```

Symbolic execution of IADD byte-code:

```
public class IADD extends
   ….bytecode.IADD { …
 public Instruction execute(…
   ThreadInfo th){
   Expression sym_v1 = ….getOperandAttr(0);
   Expression sym_v2 = ….getOperandAttr(1);
   if (sym_v1 == null && sym_v2 == null)
     // both values are concrete
     return super.execute(… th);
   else {
     int v1 = th.pop();
     int v2 = th.pop();
     th.push(0,…); // don't care
     …
     ….setOperandAttr(Expression._plus(
    sym_v1,sym_v2));
     return getNext(th);
   }
 }
}
```

Handling Branching Conditions

- Involves:
  - Creation of a non-deterministic choice in JPF's search
  - Path condition associated with each choice
  - Add condition (or its negation) to the corresponding path condition
  - Check satisfiability (with *Choco, IASolver, CVC3 etc.*)
  - If un-satisfiable, instruct JPF to backtrack
- Created new choice generator

```
public class PCChoiceGenerator
    extends IntIntervalGenerator {
    PathCondition[] PC;
    …
}
```

Example: IFGE

Concrete execution of IFGE byte-code:

```
public class IFGE extends
    Instruction { …

 public Instruction execute(…
    ThreadInfo th){

    cond = (th.pop() >=0);

    if (cond)

     next = getTarget();

    else

     next = getNext(th);

    return next;

 }

}
```

Symbolic execution of IFGE byte-code:

```
public class IFGE extends
   ….bytecode.IFGE { …
 public Instruction execute(…
   ThreadInfo th){
   Expression sym_v = ….getOperandAttr();
   if (sym_v == null)
     // the condition is concrete
     return super.execute(… th);
   else {
     PCChoiceGen cg = new PCChoiceGen(2);…
     cond = cg.getNextChoice()==0?false:true;
     if (cond) {
        pc._add_GE(sym_v,0);
        next = getTarget();
     }
     else {
        pc._add_LT(sym_v,0);
        next = getNext(th);
     }
     if (!pc.satisfiable()) … // JPF backtrack
     else cg.setPC(pc);
     return next;
   } } }
```

Complex mathematical constraints

- Model-level interpretation of calls to math functions

`$x + 1` ⟶ *Math.sin* ⟶ `sin($x + 1)`

Symbolic expression
(un-interpreted function)
denoting the result value of the call

CORAL solver [NFM'11]

- Target applications:
  - Symbolic execution of programs that manipulate floating-point variables
  - Use floating-point arithmetic
  - Call specific math functions (from java.lang.Math)

Common in software from NASA

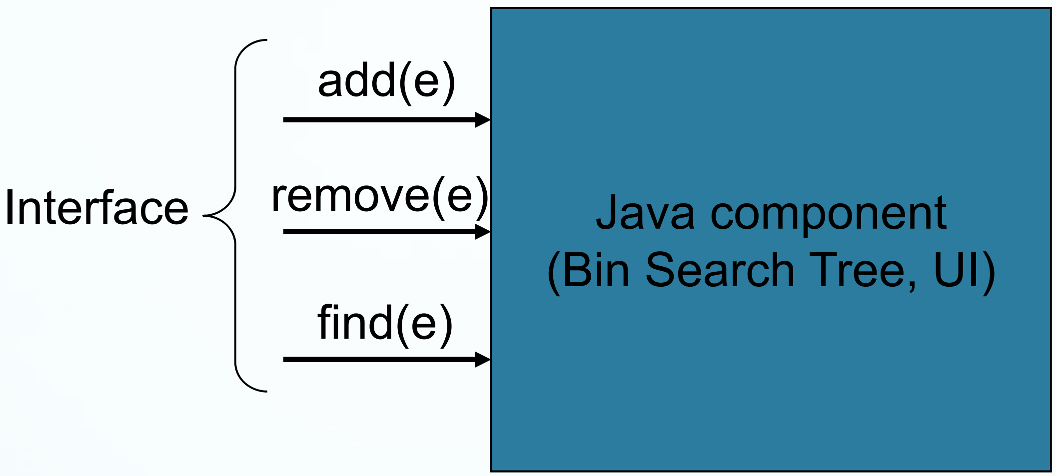
sqrt(exp(x+z))) < pow(z, x) $\wedge$ x>0 $\wedge$ y>1 $\wedge$ z>1 $\wedge$ y<x+2 $\wedge$ w=x+2 $\Rightarrow$ **Coral Solver** $\Rightarrow$ {x=4.31, y=6.08, z=9.51, w=6.31}

- Meta-heuristic solver
  - Distance-based fitness function
- Particle swarm optimization (PSO)
  - Search simulates movements in a group of animals
  - Used opt4j library (see opt4j.sourceforge.net)

CORAL solvers

- Meta-heuristic solver
  - Distance-based fitness function
- Optimizations
  - Identification of dependent variables
  - Inference of variable domains
  - Efficient evaluation of constraints
- Particle swarm optimization (PSO)
  - Similar to GA, but with fixed-sized population
  - Search simulates movements in a group of animals
    - Implemented very efficiently (matrix operations)
    - Parameters to calibrate local and global influence
  - Used opt4j library (see opt4j.sourceforge.net)

Applications: Test Input and Sequence Generation



Generated test sequence:

```
BinTree t = new
BinTree();
 t.add(1);
 t.add(2);
 t.remove(1);
```

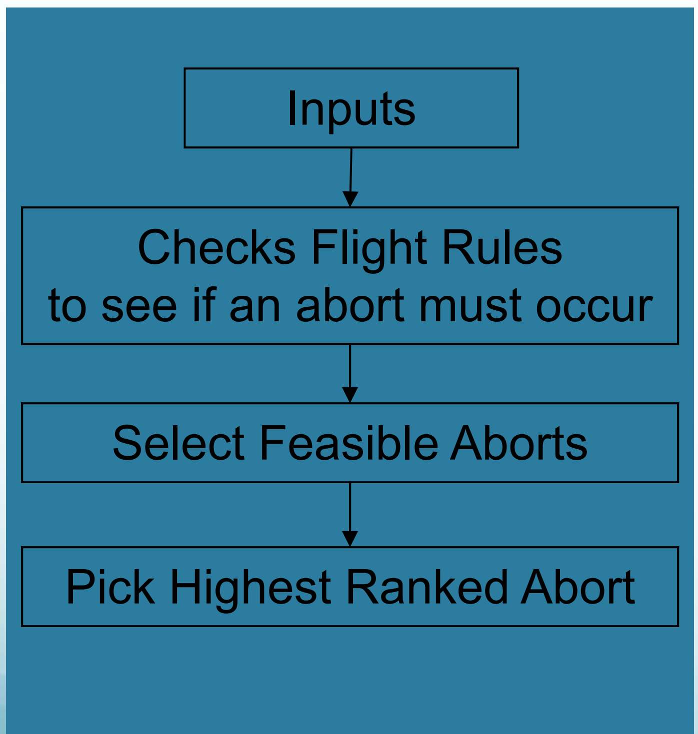
- SymbolicSequenceListener generates JUnit tests
- JUnit tests can be run directly by the developers
- Measure coverage (e.g. MC/DC)
- Support for abstract state matching

[ISSTA'04, ISSTA'06]

Application: Onboard Abort Executive (OAE)

Prototype for CEV ascent abort handling being developed by JSC GN&C

OAE Structure



Results

- Baseline
  - Manual testing: time consuming (~1 week)
  - Guided random testing could not cover all aborts
- Symbolic PathFinder
  - Generates tests to cover all aborts and flight rules
  - Total execution time is < 1 min
  - Test cases: 151 (some combinations infeasible)
  - Errors: 1 (flight rules broken but no abort picked)
  - Found major bug in new version of OAE
  - Flight Rules: 27 / 27 covered
  - Aborts: 7 / 7 covered
  - Size of input data: 27 values per test case
- Integration with End-to-end Simulation
  - Input data constrained by physical laws

    Example: inertial velocity can not be 24000 ft/s when the geodetic altitude is 0 ft
  - Need to encode these constraints explicitly

[ISSTA'08]

Generated Test Cases and Constraints

Test cases:

```
// Covers Rule: FR A_2_A_2_B_1: Low Pressure Oxidizer Turbo pump speed limit exceeded
// Output: Abort:IBB
CaseNum  1;
CaseLine in.stage_speed=3621.0;
CaseTime 57.0-102.0;

// Covers Rule: FR A_2_A_2_A: Fuel injector pressure limit exceeded
// Output: Abort:IBB
CaseNum  3;
CaseLine in.stage_pres=4301.0;
CaseTime 57.0-102.0;
…
```

Constraints:

```
//Rule: FR A_2_A_1_A: stage1 engine chamber pressure limit exceeded Abort:IA
PC (~60 constraints):
in.geod_alt(9000) < 120000 && in.geod_alt(9000) < 38000 && in.geod_alt(9000) < 10000 &&
in.pres_rate(-2) >= -2 && in.pres_rate(-2) >= -15 &&
in.roll_rate(40) <= 50 && in.yaw_rate(31) <= 41 && in.pitch_rate(70) <= 100 && …
```

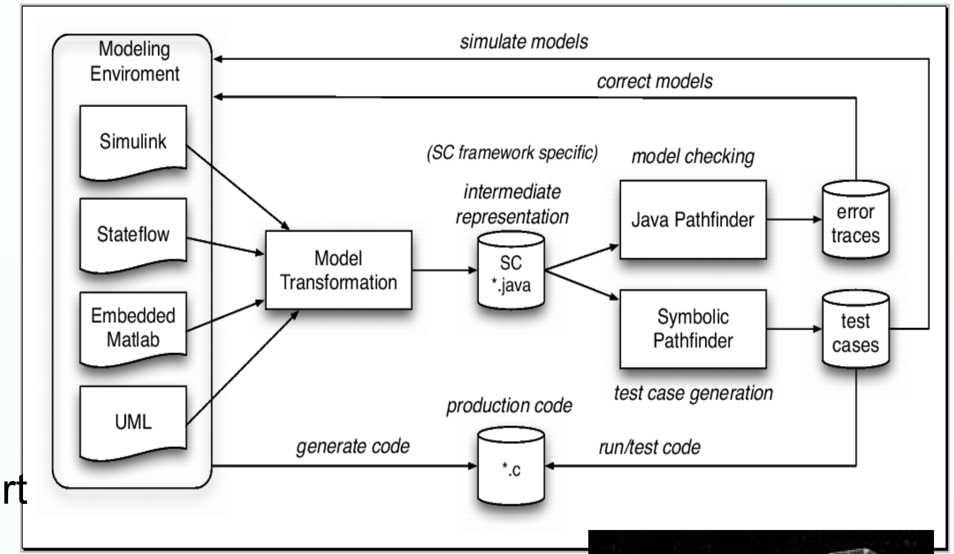
Test-Sequence Generation for Multiple Statechart Models

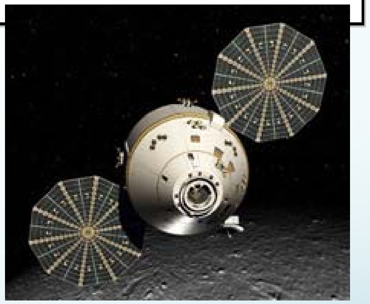
Polyglot Framework [ISSTA'11]

- Analysis for UML, Stateflow and Rhapsody interactive models
- Automated test sequence generation
- High degree of coverage
  - state, transition, path
- Pluggable semantics
- Study discrepancies between multiple statechart formalisms

Demonstrations:

- Orion's Pad Abort-1
- Ares-Orion communication
- JPL's MER Arbiter
- Apollo lunar autopilot





Orion orbits the moon (Image Credit: Lockheed Martin).

Shown: Polyglot Framework for model-based analysis and test case-generation; test cases used to test the generated code and to discover discrepancies between models and code.

Dynamic Techniques

- Classic symbolic execution is a static technique
- Dynamic techniques
  - Collect symbolic constraints during concrete executions
  - DART = Directed Automated Random Testing
  - Concolic (Concrete Symbolic) testing

P. Godefroid

DART = Directed Automated Random Testing

- Dynamic test generation
  - Run the program starting with some random inputs
  - Gather symbolic constraints on inputs at conditional statements
  - Use a constraint solver to generate new test inputs
  - Repeat the process until a specific program path or statement is reached (classic dynamic test generation [Korel90])
  - Or repeat the process to attempt to cover ALL feasible program paths (DART [Godefroid et al PLDI'05])
- Detect crashes, assert violations, runtime errors etc.

P. Godefroid

Directed Search

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 0, y = 0	create symbolic variables x, y	

Directed Search

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

	Concrete Execution	Symbolic Execution	Path Constraint
		create symbolic variables x, y	
			$x \leq y$
}	$x = 0, y = 0$		

Solve: $!(x \leq y)$

Solution: $x=1, y=0$

Directed Search

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 1, y = 0	create symbolic variables x, y	

Directed Search

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

	Concrete Execution	Symbolic Execution	Path Constraint
		create symbolic variables x, y	
			x > y
←	x = 1, y = 0		

Directed Search

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

	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	

Directed Search

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables x, y	
		x > y
x = 1, y = 1	x = x+y y = x	

Directed Search

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables x, y	
		x > y
	y = x	
x = 0, y = 1	x = y	

Directed Search

Concrete Execution	Symbolic Execution	Path Constraint
	create symbolic variables x, y	
		x > y
	y = x	
	x = y	
x = 0, y = 1		y ≤ x

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Solve: x> y AND !(y≤x)

Impossible: DONE!

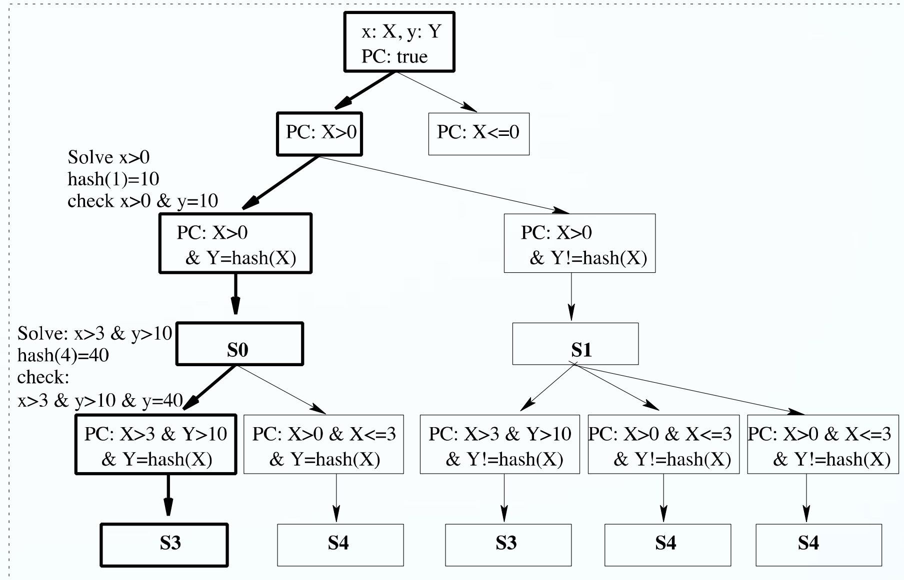
Dynamic Test Generation

- Very popular
- Implemented and extended in many interesting ways
- Many tools
  - PEX, SAGE, CUTE, jCUTE, CREST, SPLAT, etc
- Many applications
  - Bug finding, security, web and database applications, etc.
- EXE (Stanford Univ. [Cadar et al TISSEC 2008])
  - Related dynamic approach to symbolic execution

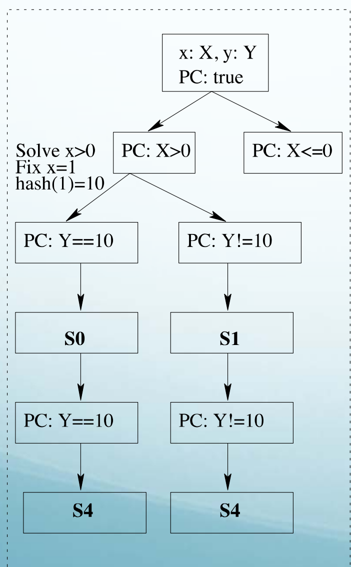
A Comparison [ISSTA'11]

```
void test(int x, int y) {
  if (x > 0) {
   if (y == hash(x))//hash(x)=10*x
     S0;
   else
     S1;
   if (x > 3 && y > 10)
   //if (y > 10)
     S3;
   else
     S4;
  }
}
```

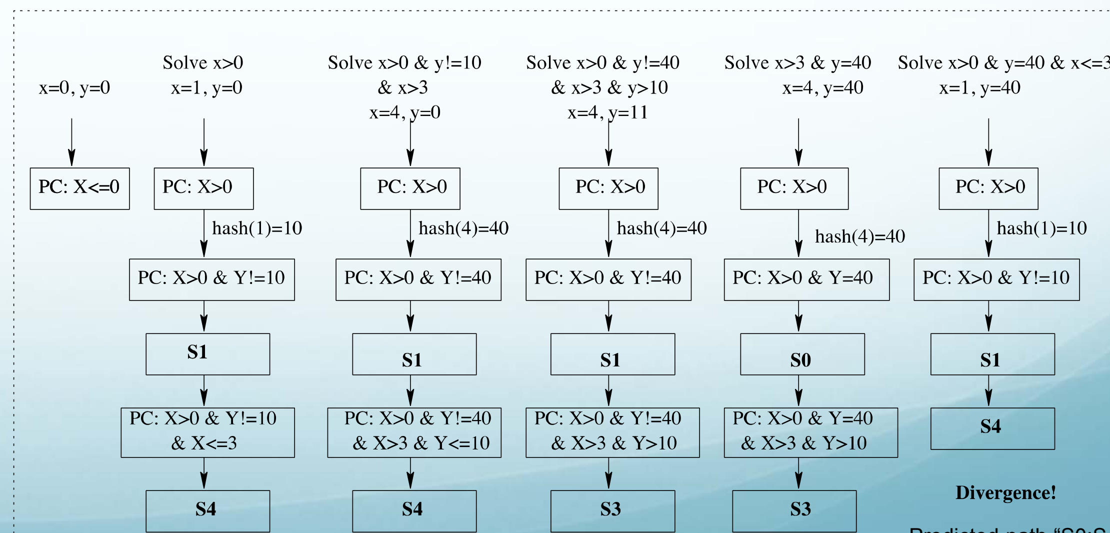
Example



"Classic" sym exe w/ mixed concrete-symbolic solving: all paths covered



EXE results: stmt "S3" not covered



Predicted path "S0;S4"
!= path taken "S1;S4"

DART results: path "S0;S4" not covered

Mixed Concrete-Symbolic Solving [ISSTA'11]

- Use un-interpreted functions for external library calls
- Split path condition PC into:
  - simplePC – solvable constraints
  - complexPC – non-linear constraints with un-interpreted functions
- Solve simplePC
  - Use obtained solutions to simplify complexPC
  - Check the result again for satisfiability

Example (assume hash(x) = 10 *x):

PC: $\underbrace{X>3 \wedge Y>10}_{\text{simplePC}} \wedge \underbrace{Y=hash(X)}_{\text{complexPC}}$

Solve simplePC; use solution X=4 to compute h(4)=40

Simplify complexPC: Y=40

Solve again: $X>3 \wedge Y>10 \wedge Y=40$ Satisfiable!

Challenge

Path explosion

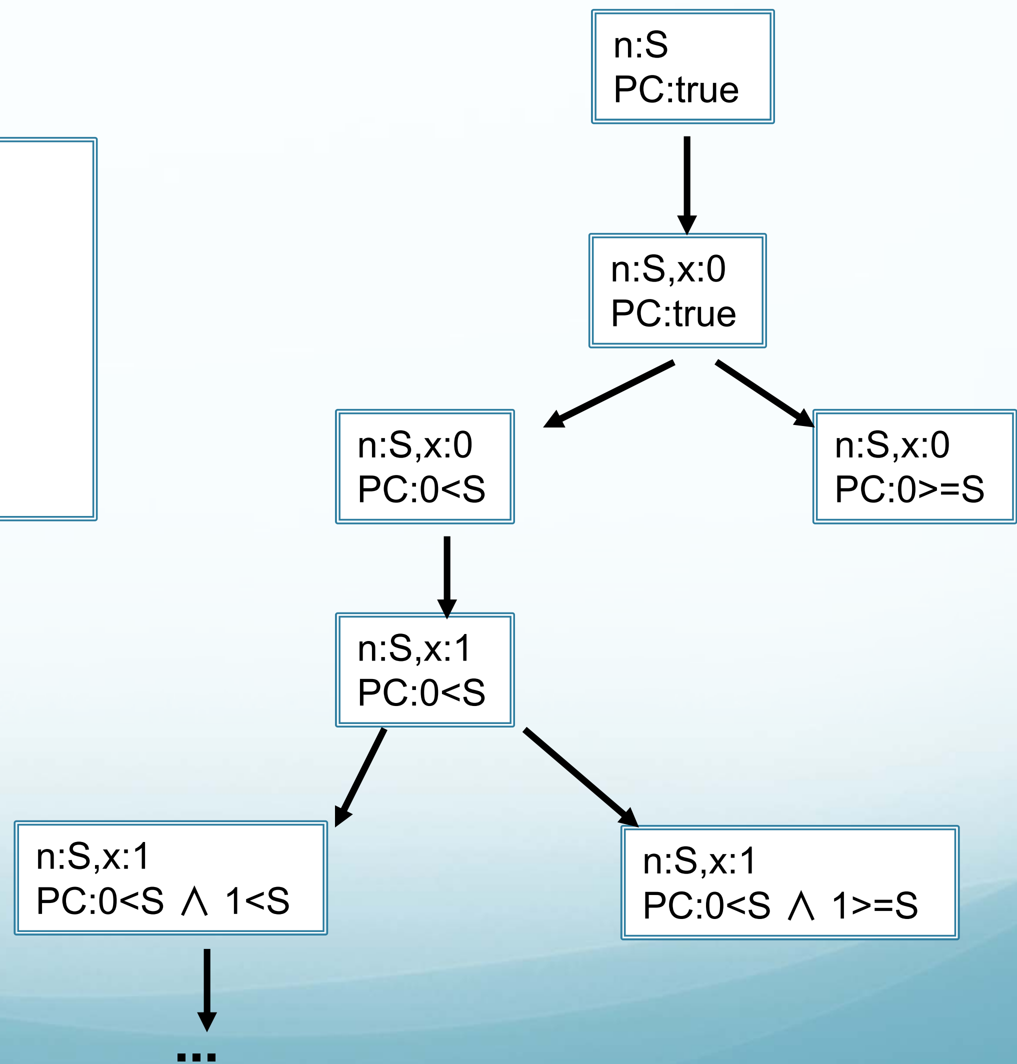
Symbolic execution of a program may result in a very large, possibly infinite number of paths

Problem: loops and recursion

Example Code

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

Infinite symbolic execution tree



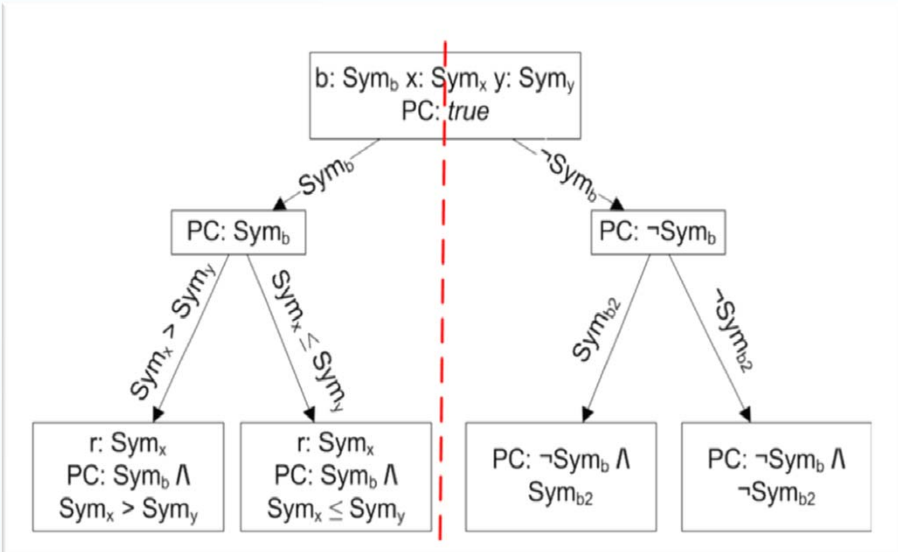
Solutions

- Dealing with loops and recursion
  - Put bound on search depth or on number of PCs
  - Stop search when desired coverage achieved
  - Loop abstraction [Saxena et al ISSTA'09] [Godefroid ISSTA'11]
- Solutions addressing path explosion
  - Parallel Symbolic Execution
  - Abstract State Matching
  - Compositional DART = SMART

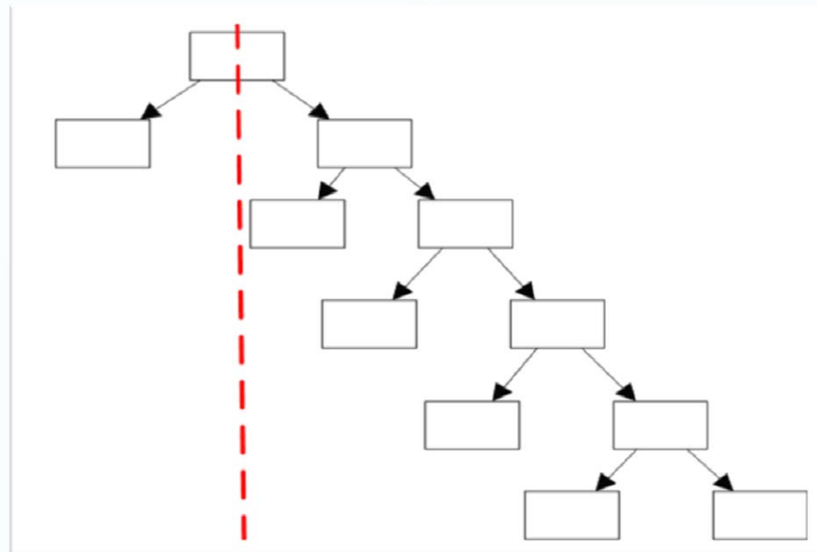
Parallel Symbolic Execution

- Path explosion
  - Increases exponentially with the number of inputs specified as symbolic
  - Very expensive in terms of time (weeks, months)
- Solution
  - Speed-up symbolic execution using parallel or distributed techniques
- Symbolic execution is amenable to parallelization
  - No sharing between sub-trees

Balancing partitions



Nicely Balanced – linear speedup



Poorly Balanced – no speedup

- Solutions
  - Simple static partitioning [ISSTA'10]
  - Dynamic partitioning [Andrew King's Masters Thesis at KSU, Cloud9 at EPFL, Fujitsu]

Simple Static Partitioning

- Static partitioning of tree with light dynamic load balancing
  - Flexible, little communication overhead
- Constraint-based partitioning
  - Constraints used as initial pre-conditions
  - Constraints are disjoint and complete
- Approach
  - Shallow symbolic execution => produces large number of constraints
  - Constraints selection – according to frequency of variables
  - Combinatorial partition creation
- Intuition
  - Commonly used variables likely to partition state space in useful ways
- Results
  - maximum analysis time speedup of 90x observed using 128 workers and a maximum test generation time speedup of 70x observed using 64 workers.

Abstract State Matching

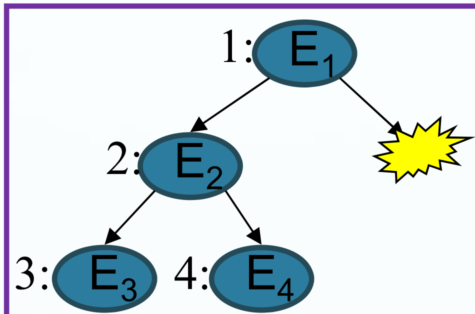
- State matching – subsumption checking [SPIN' 06, J. STTT 2008]
  - Obtained through DFS traversal of "rooted" heap configurations
    - Roots are program variables pointing to the heap
    - Unique labeling for "matched" nodes
    - Check logical implication between numeric constraints
  - Not enough to ensure termination
- Abstraction
  - Store abstract versions of explored symbolic states
  - Use subsumption checking to determine if an abstract state is re-visited
  - Decide if the search should continue or backtrack

Abstract State Matching

- Enables analysis of under-approximation of program behavior
- Preserves errors to safety properties -- useful for testing
- Automated support for two abstractions (inspired by shape analysis [TVLA]
  - Singly linked lists
  - Arrays
- No refinement!
- See [Albarghouthi et al. CAV10] for symbolic execution with automatic abstraction-refinement

State Matching: Subsumption Checking

Stored state:

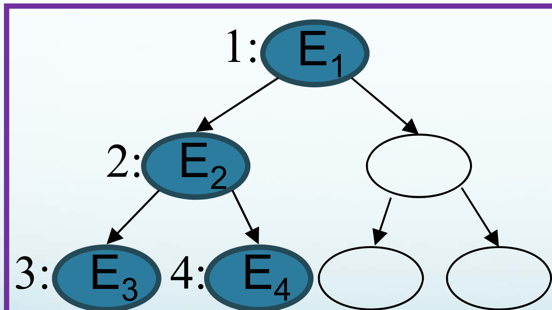


$E_1 > E_2 \wedge$
$E_2 > E_3 \wedge$
$E_2 \le E_4 \wedge$
$E_1 > E_4$

Set of concrete states represented by stored state

$\supseteq$ $\Uparrow$ $\supseteq$

New state:



$E_1 > E_2 \wedge$
$E_2 > E_3 \wedge$
$E_2 < E_4 \wedge$
$E_1 > E_4$

Set of concrete states represented by new state

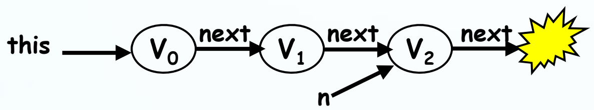
Normalized using existential quantifier elimination

Abstractions for Lists and Arrays

- Shape abstraction for singly linked lists
  - Summarize contiguous list elements not pointed to by program variables into summary nodes
  - Valuation of a summary node
    - Union of valuations of summarized nodes
  - Subsumption checking between abstracted states
    - Same algorithm as subsumption checking for symbolic states
    - Treat summary node as an "ordinary" node
- Abstraction for arrays
  - Represent array as a singly linked list
  - Abstraction similar to shape abstraction for linked lists

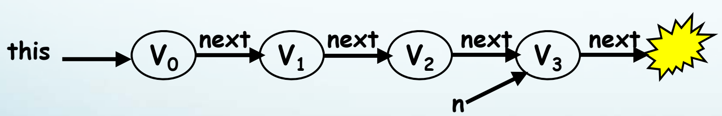
Abstraction for Lists

Symbolic states



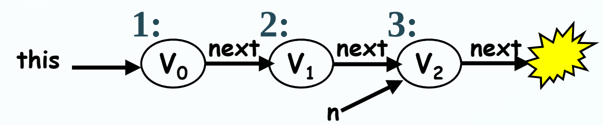
PC: $V_0 \leq v \wedge V_1 \leq v$

Unmatched!



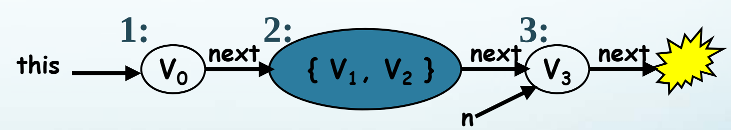
PC: $V_0 \leq v \wedge V_1 \leq v \wedge V_2 \leq v$

Abstracted symbolic states



$E_1 = V_0 \wedge E_2 = V_1 \wedge E_3 = V_2$

PC: $V_0 \leq v \wedge V_1 \leq v$



$E_1 = V_0 \wedge (E_2 = V_1 \vee E_2 = V_2) \wedge E_3 = V_3$

PC: $V_0 \leq v \wedge V_1 \leq v \wedge V_2 \leq v$

Compositional DART [POPL'07]

- Idea: compositional dynamic test generation
  - use summaries of individual functions like in inter-procedural static analysis
  - if f calls g, analyze g separately, summarize the results, and use g's summary when analyzing f
  - A summary $\varphi(g)$ is a disjunction of path constraints expressed in terms of input pre-conditions and output post-conditions:

    $\varphi(g) = \vee\varphi(w)$, with $\varphi(w) = pre(w) \wedge post(w)$

- g's outputs are treated as symbolic inputs to calling function f
- SMART: Top-down strategy to compute summaries on a demand-driven basis from concrete calling contexts
- Same path coverage as DART but can be exponentially faster!
- Follow-up work
  - Anand et al. [TACAS'08], Godefroid et al. [POPL'10]

Example

```
int is_positive(int x) {
  if (x>0) return 1;
  return 0;
}
#define N 100
void top (int s[N]) {// N inputs
 int i, cnt=0;
 for (i=0;i<N; i++)
   cnt=cnt+is_positive(s[i]);
 if (cnt == 3) error(); // (*)
 return;
}
```

Program P = {top, is_positive} has 2^N feasible paths

DART will perform 2^N runs

SMART will perform only 4 runs

- 2 to compute summary
  $\varphi(\text{is\_positive}) = (x>0 \wedge ret=1) \vee (x \leq 0 \wedge ret=0)$
- 2 to execute both branches of (*) by solving:
  $[(s[0]>0 \wedge ret_0=1) \vee (s[0] \leq 0 \wedge ret_0=0)] \wedge$
  $[(s[1]>0 \wedge ret_1=1) \vee (s[1] \leq 0 \wedge ret_1=0)] \wedge \cdots \wedge$
  $[(s[N-1]>0 \wedge ret_{N-1}=1) \vee (s[N-1] \leq 0 \wedge ret_{N-1}=0)] \wedge$
  $(ret_0+ret_1+ \cdots + ret_{N-1}=3)$

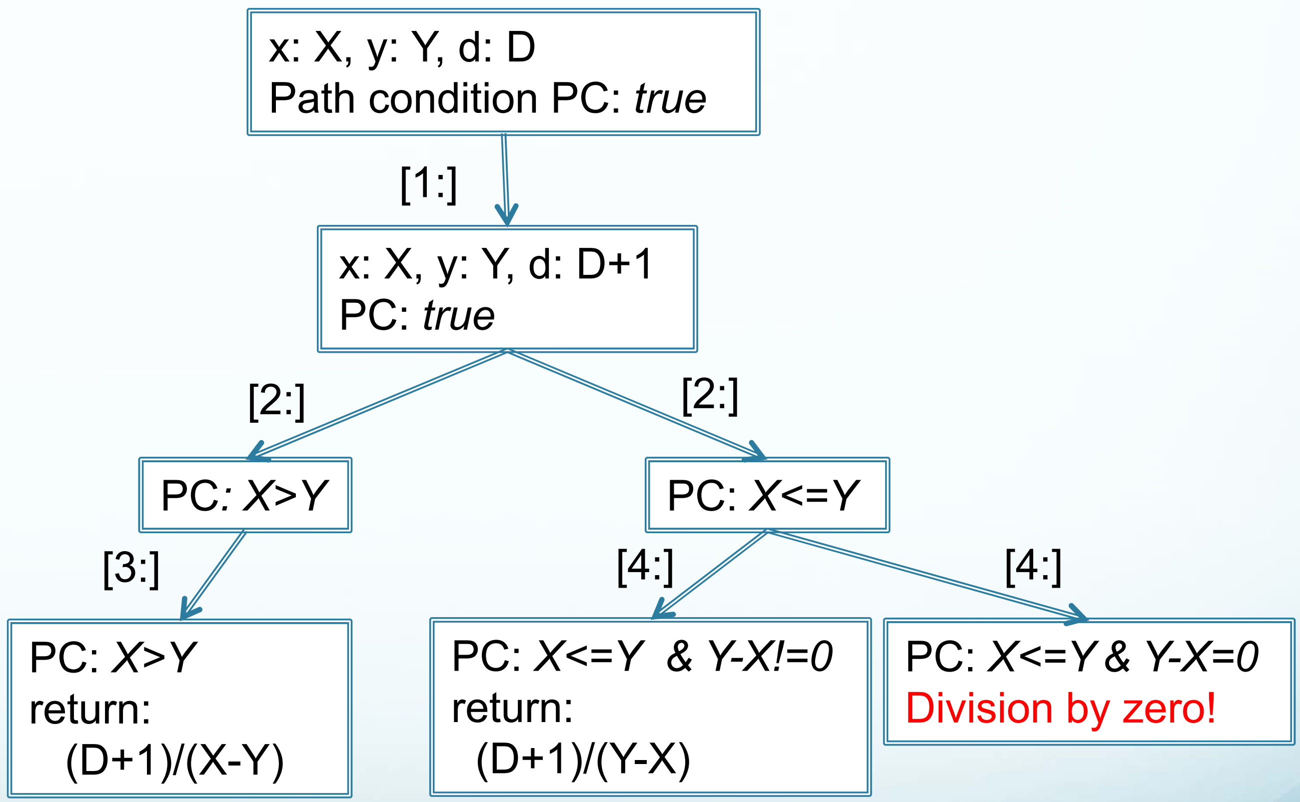
P. Godefroid

Applications – An Example

Method m:

```
1: d=d+1;
2:  if (x > y)
3:    return d / (x-y);
     else
4:    return d / (y-x);
```

Symbolic execution tree:



Solve path conditions → test inputs

Auto-generated JUnit Tests

```
@Test public void t1() {
    m(1, 0, 1);
}
@Test public void t2() {
    m(0, 1, 1);
}
@Test public void t3() {
    m(1, 1, 1);
}
```

Pass ✔

Pass ✔

Fail ✗ PC: $X<=Y$ & $Y\text{-}X=0 \Leftrightarrow X=Y$

Achieves full path coverage

Program Repair and Synthesis

- Add JML pre-condition:
  - @Requires(“x!=y)
- Add argument check in m:
  - If(x==y) throw new IllegalArgumentException(“requires: x!=y”)
- Add expected clause to test t3:

```
@Test(expected=ArithmeticException.class)
 public void t3() {
   m(1, 1, 1);
 }
```

Will fix the error or produce more useful output

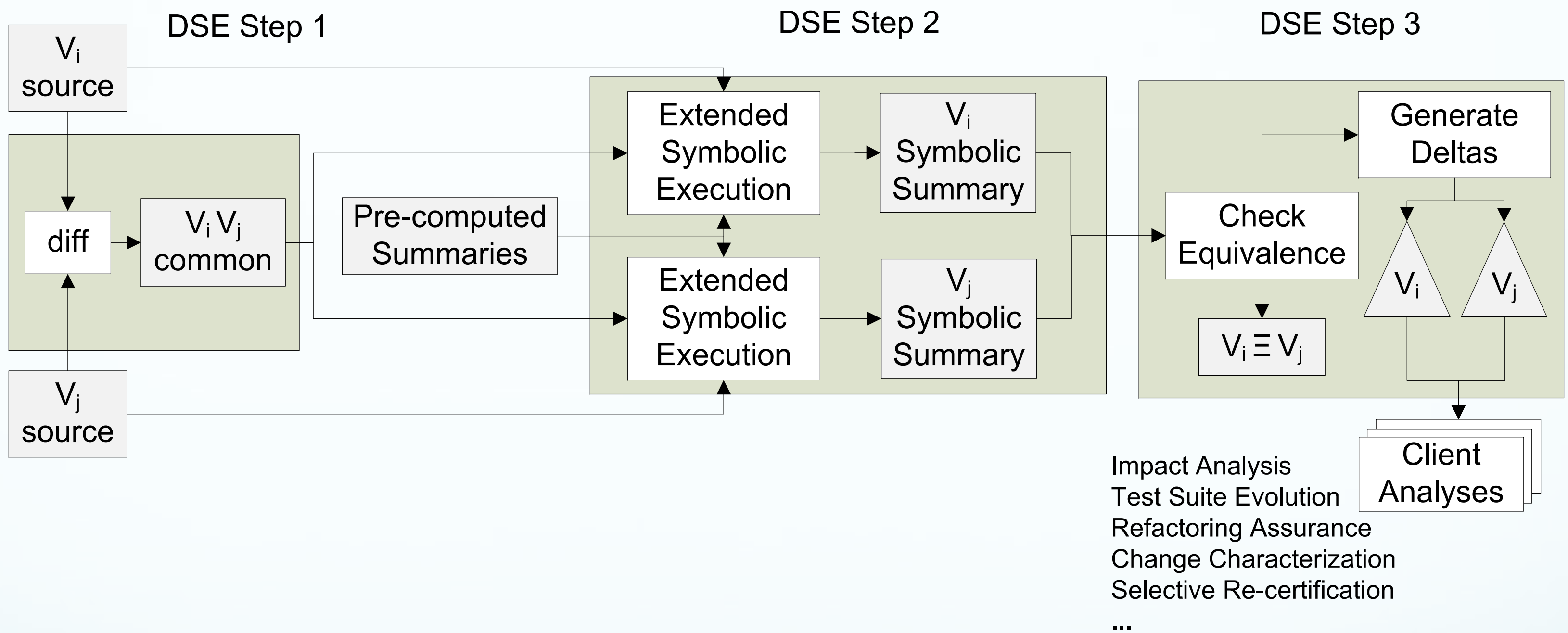
One can do more sophisticated program repairs.

See [ICSE’11 “Angelic Debugging”]

Invariant Generation

- Pre-condition:
  - “x!=y”
- Post-condition:
  - “\result==((x>y) ? (d+1)/(x-y) : (d+1)/(y-x))”
- Use inductive and machine learning techniques to generate loop invariants
- See DySy [Csallner et al ICSE’08], also [SPIN’04]

Differential Symbolic Execution



- Computes logical difference between two program versions
- [FSE08, Person et al PLDI11]

Applications

- Automated test-input generation
  - test vectors and test sequences
- Error detection, Invariant generation
- Program and data structure repair
- Security
- Robustness and stress testing
- Regression testing etc.

Challenges

- Scalability
  - Compositional techniques [Godefroid, POPL'07]
  - Pruning redundant paths [Boonstoppel et al, TACAS'08]
  - Heuristic search [Brunim & Sen, ASE'08] [Majumdar & Se, ICSE'07]
  - Parallel techniques [Siddiqui & Khurshid, ICSTE'10] [Staats & Pasareanu, ISSTA'10]
  - Incremental techniques [Person et al, PLDI'11]
- Complex non-linear mathematical constraints
  - Un-decidable or hard to solve
  - Heuristic solving [Lakhotia et al., ICTSS'10][Souza et al, NFM'11]
- Testing web applications and security problems
  - String constraints [Bjorner et al, 2009] …
  - Mixed numeric and string constraints [ISSTA'11] [Fujitsu]
- Not covered:
  - Symbolic execution for formal verification [Coen-Porisini et al, ESEC/FSE'01], [Dillon, ACM TOPLAS'90], [Harrison & Kemmerer'88]
  - …

Symbolic Execution and Software Testing

- King [Comm. ACM 1976] , Clarke [IEEE TSE 1976]
- Received renewed interest in recent years
  - Algorithmic advances
  - Increased availability of computational power and decision procedures
- Tools, many open-source
  - NASA's Symbolic (Java) Pathfinder
    http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
  - UIUC's CUTE and jCUTE
    http://osl.cs.uiuc.edu/~ksen/cute
  - Stanford's KLEE
    http://klee.llvm.org/
  - UC Berkeley's CREST and BitBlaze
    http://code.google.com/p/crest
  - Microsoft's Pex, SAGE, YOGI, PREfix
    http://research.microsoft.com/en-us/projects/pex/
    http://research.microsoft.com/en-us/projects/yogi
  - IBM's Apollo, Parasoft's testing tools etc.
- Bibliography on symbolic execution (Saswat Anand): http://sites.google.com/site/symexbib/
- See ICSE'11 Impact Project talk on Thursday

Thank you!

References

Java PathFinder http://babelfish.arc.nasa.gov/trac/jpf/

[TACAS03]Sarfraz Khurshid, Corina S. Pasareanu, Willem Visser: Generalized Symbolic Execution for Model Checking and Testing. TACAS 2003: 553-56

- [CAV10] Aws Albarghouthi, Arie Gurfinkel, Ou Wei, Marsha Chechik: Abstract Analysis of Symbolic Executions. CAV 2010: 495-510
- [FSE08] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, Corina S. Pasareanu: Differential symbolic execution. SIGSOFT FSE 2008: 226-237
- [PLDI11] **Directed Incremental Symbolic Execution** Suzette Person (NASA Langley Reseach Center), Guowei Yang (The Universiity of Texas at Austin), Neha Rungta (NASA Ames Research Center), and Sarfraz Khurshid (The University of Texas at Austin)
- [ICSE08] C. Csallner, N. Tillmann, Y. Smaragdakis: DySy: dynamic symbolic execution for invariant inference. ICSE 2008: 281-290
- [SPIN04] Corina S. PasareanWillem Visser: Verification of Java Programs Using Symbolic Execution and Invariant Generation. SPIN 2004: 164-181
- [ICSE11] Angelic Debugging S**atish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik** *IBM Research, USA; UC Berkeley, USA*
- ISTTA 2011 – loop abstractions
- Give example of infinite lopps
- Prateek Saxena, Pongsin Poosankam, Stephen McCamant, Dawn Song: Loop-extended symbolic execution on binary programs. ISSTA 2009: 225-236
- Rupak Majumdar, Indranil Saha: Symbolic Robustness Analysis. IEEE Real-Time Systems Symposium 2009: 355-363
- Kiran Lakhotia, Nikolai Tillmann, Mark Harman, Jonathan de Halleux: FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution. ICTSS 2010: 142-157