Symbolic Execution and Software Testing Part 1

Corina Păsăreanu
CMU Silicon Valley/ NASA Ames Research Center

NATO International Summer School 2012,
Marktoberdorf, Germany

outline

Part 1

- introduction: symbolic execution
- symbolic pathfinder: symbolic execution for Java bytecode
- input data structures
- multi-threading

Part 2

- dynamic techniques
- the DART algorithm
- concolic execution

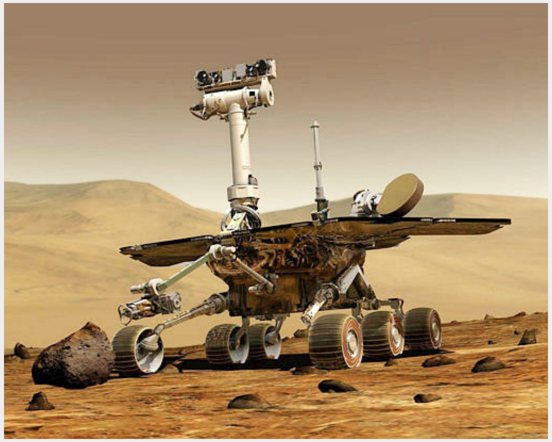
Part 3

- challenges
- solving complex constraints
- parallel and compositional techniques
- abstraction
- symbolic execution with mixed concrete-symbolic solving

Part 4

- applications
- current and future work

software is everywhere



errors are expensive …

annual cost of software errors to US economy is $ ~60B [NIST'02]

approaches to finding errors

model checking

automatic, exhaustive

scalability issues

static analysis

automatic, scalable, exhaustive

reported errors may be spurious

testing

reported errors are real

may miss errors

well accepted technique; state of practice

our approach

combine model checking and symbolic execution for test case generation

testing vs model checking

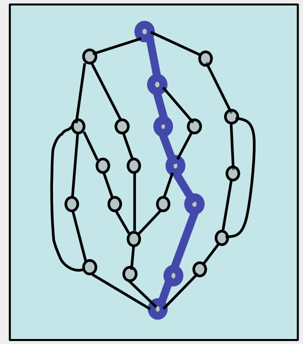
program / model

```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 …
 tail=(tail+1)%size;
 return buffer[tail];
}
```

test oracle

testing / simulation



OK

error

program / model

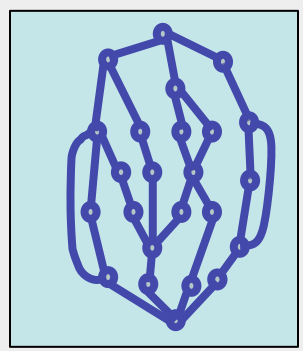
```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 …
 tail=(tail+1)%size;
 return buffer[tail];
}
```

property

always(ϕ **or** ψ)

model checking

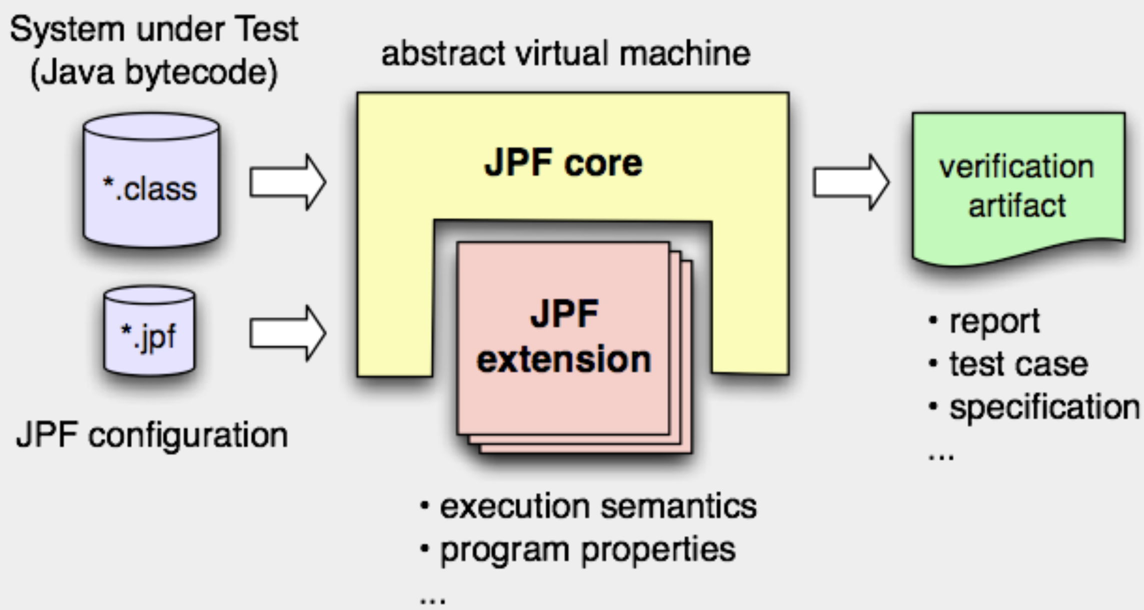


OK

error trace

```
Line 5: …
Line 12: …
…
Line 41:…
Line 47:…
```

java pathfinder (jpf)



extensible virtual machine framework for java bytecode verification
workbench to implement all kinds of verification tools

typical use cases:

software model checking (detection of deadlocks, races, assert errors)

test case generation (symbolic execution) ... and many more

java pathfinder (jpf)

scalability

- on-the-fly partial order reduction
- configurable search strategies
- user definable heuristics, choice generators

awards

- NASA 2003, IBM 2007, FLC 2009

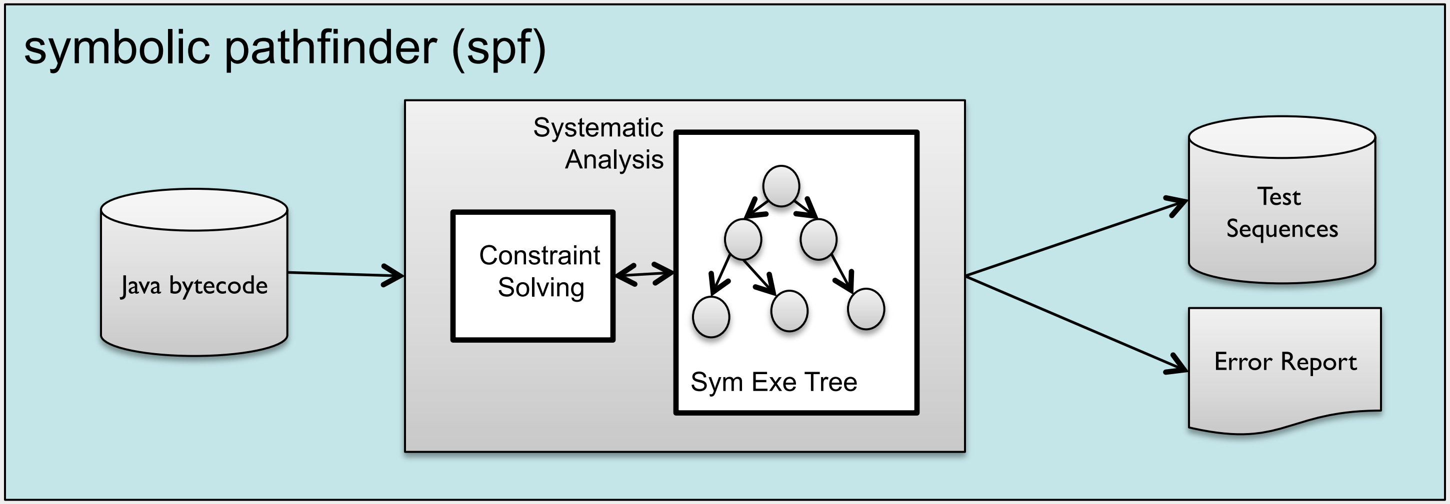
open sourced

- http://babelfish.arc.nasa.gov/trac/jpf

largest application

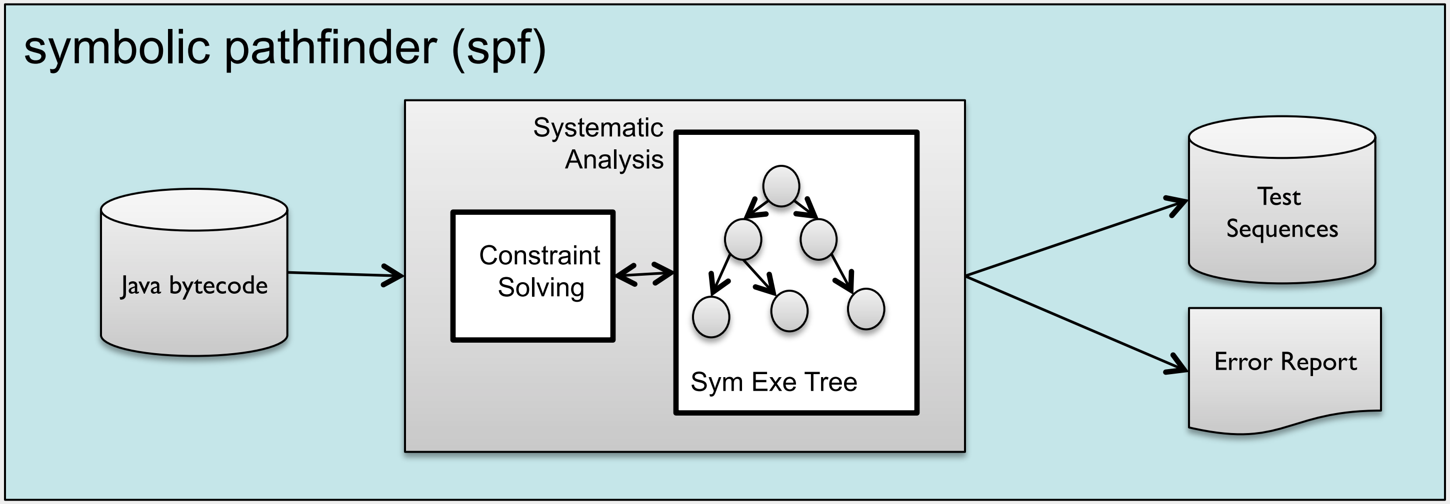
- Fujitsu (one million lines of code)

symbolic pathfinder (spf)



combines symbolic execution, model checking and constraint solving
applies to executable models and code
handles dynamic data structures, loops, recursion, multi-threading; arrays and strings
java pathfinder extension project [TACAS'03, ISSTA'08, ASE'10]

users



academia

uiuc.edu, unl.edu, utexas.edu, byu.edu, umn.edu, Stellenbosch Za, Waterloo Ca, Charles University Prague Cz, …

industry (Fujitsu)

NASA (Ames, Langley)

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 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

symbolic execution

received renewed interest in recent years … due to

- algorithmic advances
- increased availability of computational power and decision procedures

applications

- test-case generation, error detection, …

tools, many open-source

- UIUC: CUTE, jCUTE, Stanford: EXE, KLEE, UC Berkeley: CREST, BitBlaze
- Microsoft's Pex, SAGE, YOGI, PREfix
- NASA's Symbolic (Java) Pathfinder
- IBM's Apollo, Parasoft's testing tools etc.

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

↓

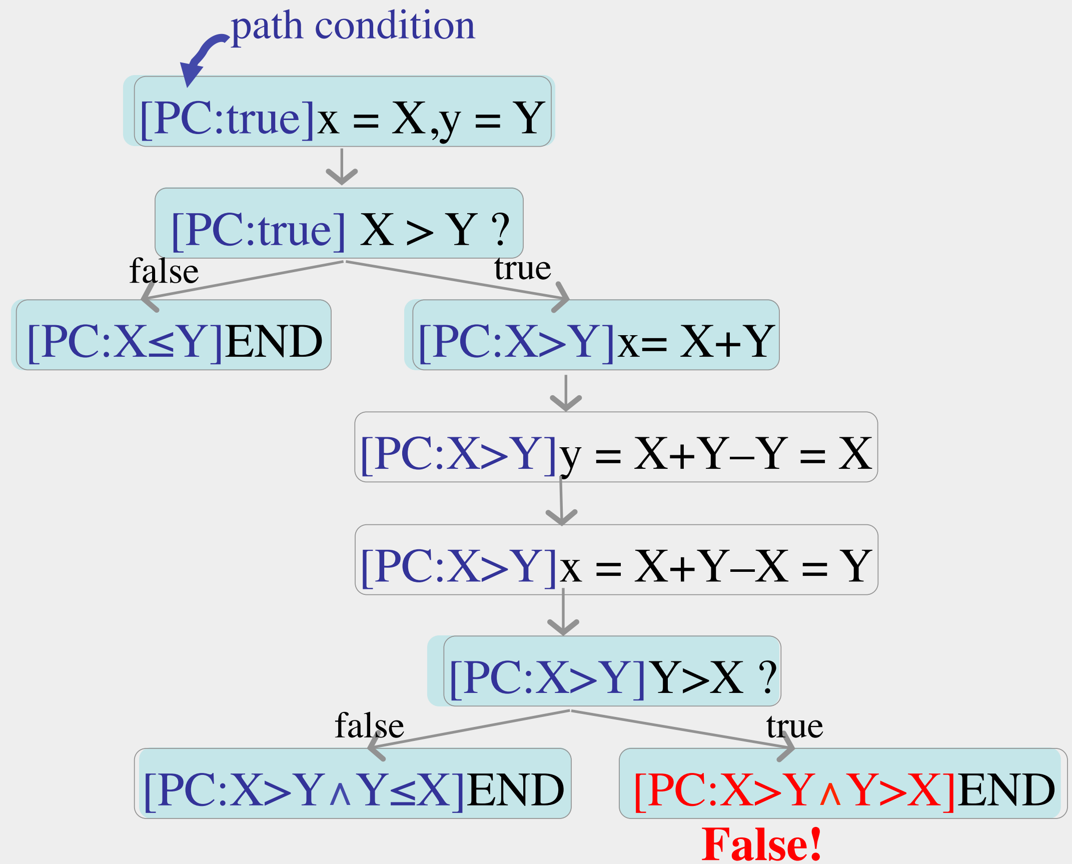
0 > 1 ? false

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 PCs: obtain test inputs

testing coverage

- statement and branch coverage
- state and transition coverage
- path coverage (default)
- MC/DC (modified condition/decision coverage)
- predicate coverage

coverage

```
int x, y;

if (x > y) {

  x = x + y;   ⇐

  y = x – y;   ⇐

  x = x – y;   ⇐

  if (x > y)

    assert false;   ⇐

}
```

statement coverage

symbolic pathfinder (spf)

java pathfinder (jpf) used for systematic exploration

- symbolic execution tree
- multi-threading
- property checking
- backtracking – when PC un-satisfiable
- different search strategies (depth-first, breadth-first)

lazy initialization for input data structures [TACAS' 03]

- non-determinism handles aliasing in input data structures
- different heap configurations explored explicitely

takes advantage of jpf's optimizations!

symbolic pathfinder (spf)

- no state matching performed
  - some abstract state matching
- symbolic search space may be infinite due to loops, recursion
  - we put a limit on the search depth

implementation

non-standard interpreter of byte-codes

- replaces concrete execution semantics of byte-codes with symbolic execution
- enables jpf-core to perform systematic symbolic analysis

attributes

- symbolic information stored in attributes associated with the program data
- propagated dynamically during symbolic execution

implementation

choice generators

- handle non-deterministic choices in branching conditions

listeners

- collect and print results: path conditions, test vectors or test sequences
- influence the search

native peers

- model native libraries
- e.g. capture `Math` library calls and send them to the constraint solver

mixed concrete-symbolic solving

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);
   }
 }
}
```

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;
   } } }
```

decision procedures

used to check path conditions

- if path condition is un-satisfiable, backtrack
- solutions of satisfiable constraints used as test inputs

SMT solvers

- Satisfiability Modulo Theories
- given a formula in first-order logic, with associated background theories, is the formula satisfiable?

see also:

- SMTLIB -- repository for SMT formulas (common format) and tools
- SMTCOMP – annual competition of SMT solvers

decision procedures

- spf uses
  - SMT solvers: Yices, CVC3
  - solvers for complex constraints: Choco, Coral
  - string solvers: Hampi, IASolver …
- generic interface
  - easy to extend with new constraint solvers and decision procedures
- new interface [Visser et al FSE'12]

mathematical functions

model-level interpretation

`$x + 1` → *Math.sin* → `sin($x + 1)`

symbolic expression
w/ un-interpreted function handled
directly by solver (Choco)

input data structures

challenge

lazy initialization [TACAS' 03, SPIN' 05]

non-determinism handles aliasing

- jpf explores different heap configurations explicitly

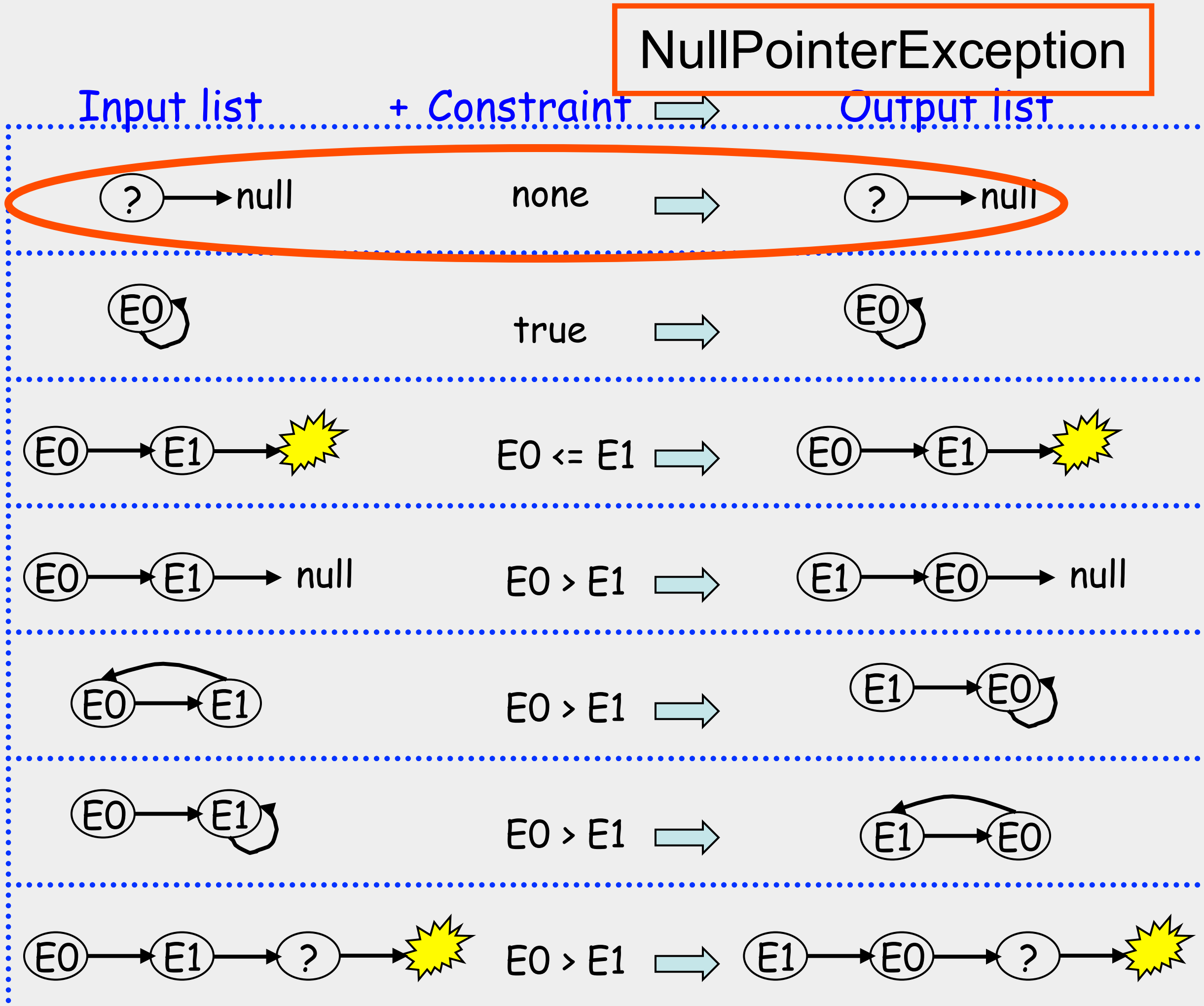
implementation

- GETFIELD, GETSTATIC bytecode instructions modified
- listener prints input heap constraints and method effects (outputs)

example

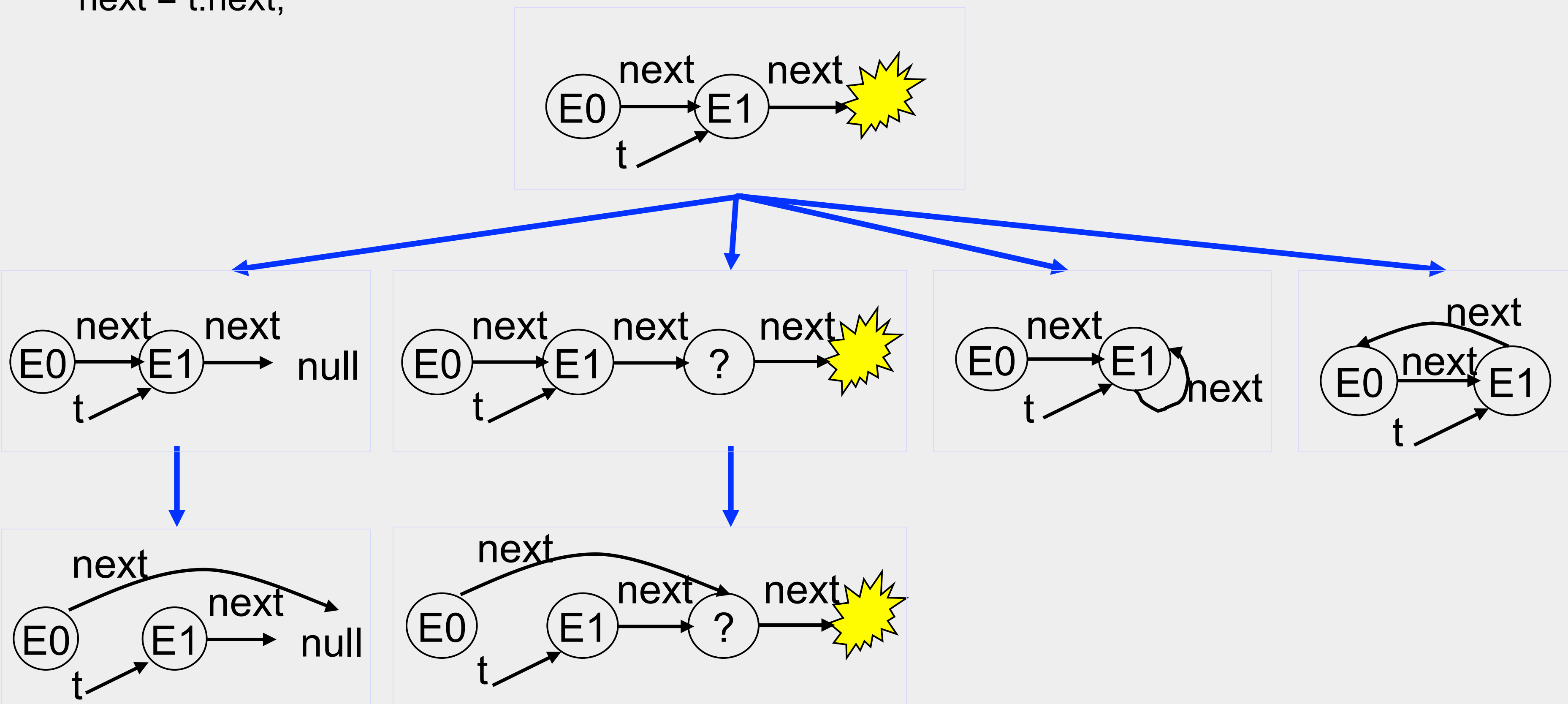
```
class Node {
    int elem;
    Node next;

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



lazy initialization

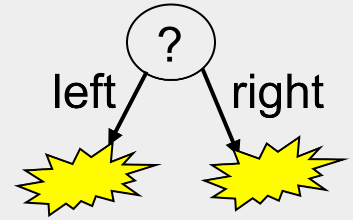
consider executing
next = t.next;



lazy initialization

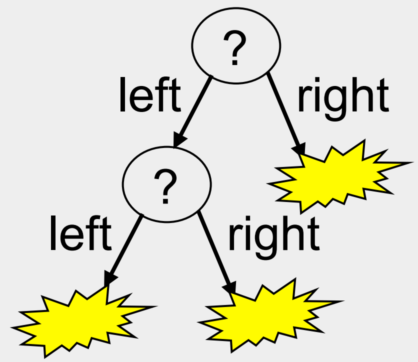
```
if (f is uninitialized) {
    if(f is reference field of type T) {
        non-deterministically initialize f to
        • null
        • a new object of type T (with un-initialized fields)
        • a previously initialized object of type T
    }
    if (f is numeric or string field) {
        initialize f to a new symbolic value
    }
}
```

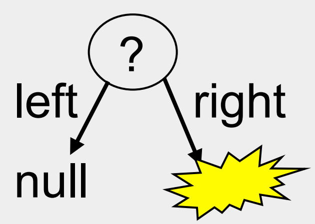
lazy initialization and garbage collection

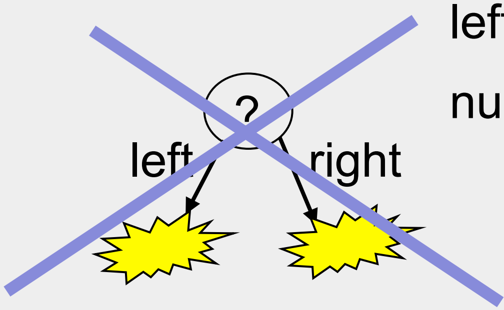



```
class TreeNode {
    int elem;
    TreeNode left;
    TreeNode right;

    void GCIssue() {
        if(left !=null)
          left = null;
        if(right!=null)
          right=null;
    }
}
```







garbage collection

Solution:
No garbage collection for objects created with lazy initialization!

test generation for input data structures

generated constraints with lazy initialization

PCconstraint # = 1

input[320].elem > input[320].next[247].elem

heap PCconstraint # = 6

input[320].next[247].next[248] != input[320] &&

input[320].next[247].next[248] != input[320].next[247] &&

input[320].next[247].next[248] != CONST_-1 &&

input[320].next[247] != input[320] &&

input[320].next[247] != CONST_-1 &&

input[320] != CONST_-1

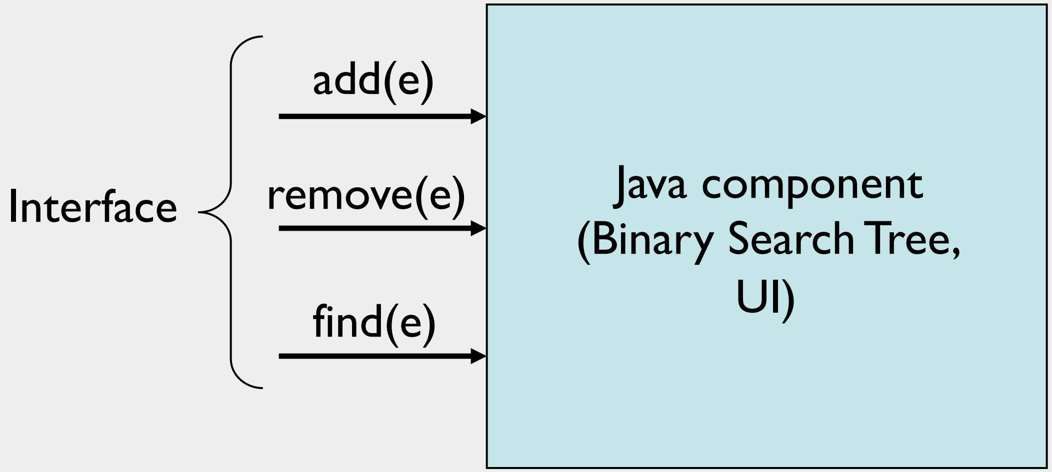
use Korat to solve them/generate test inputs

- a tool for constraint-based generation of structurally complex test inputs for Java programs.

http://korat.sourceforge.net/

test sequence generation [ISSTA'04,ISSTA'06]

test sequence generation



Generated test sequence:

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

SymbolicSequenceListener generates JUnit tests:

- method sequences (up to user-specified depth)
- method parameters

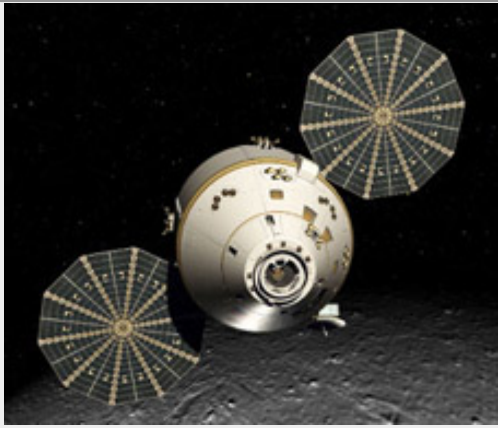
JUnit tests can be run directly by the developers

measure coverage

support for abstract state matching

extract specifications

applications



Orion orbits the moon
(Image Credit: Lockheed Martin)

NASA control software [ISSTA'08]

- manual testing: time consuming (~1 week)
- guided random testing could not obtain full coverage
- spf generated ~200 tests to obtain full coverage in <1min
- found major bug in new version

Polyglot [ISSTA'11, NFM'12]

- analysis and test case generation for UML, Stateflow and Rhapsody models
- pluggable semantics for different statechart formalisms
- analyzed MER Arbiter, Ares-Orion communication

Tactical Separation Assisted Flight Environment (T-SAFE) [NFM'11, ICST'12]

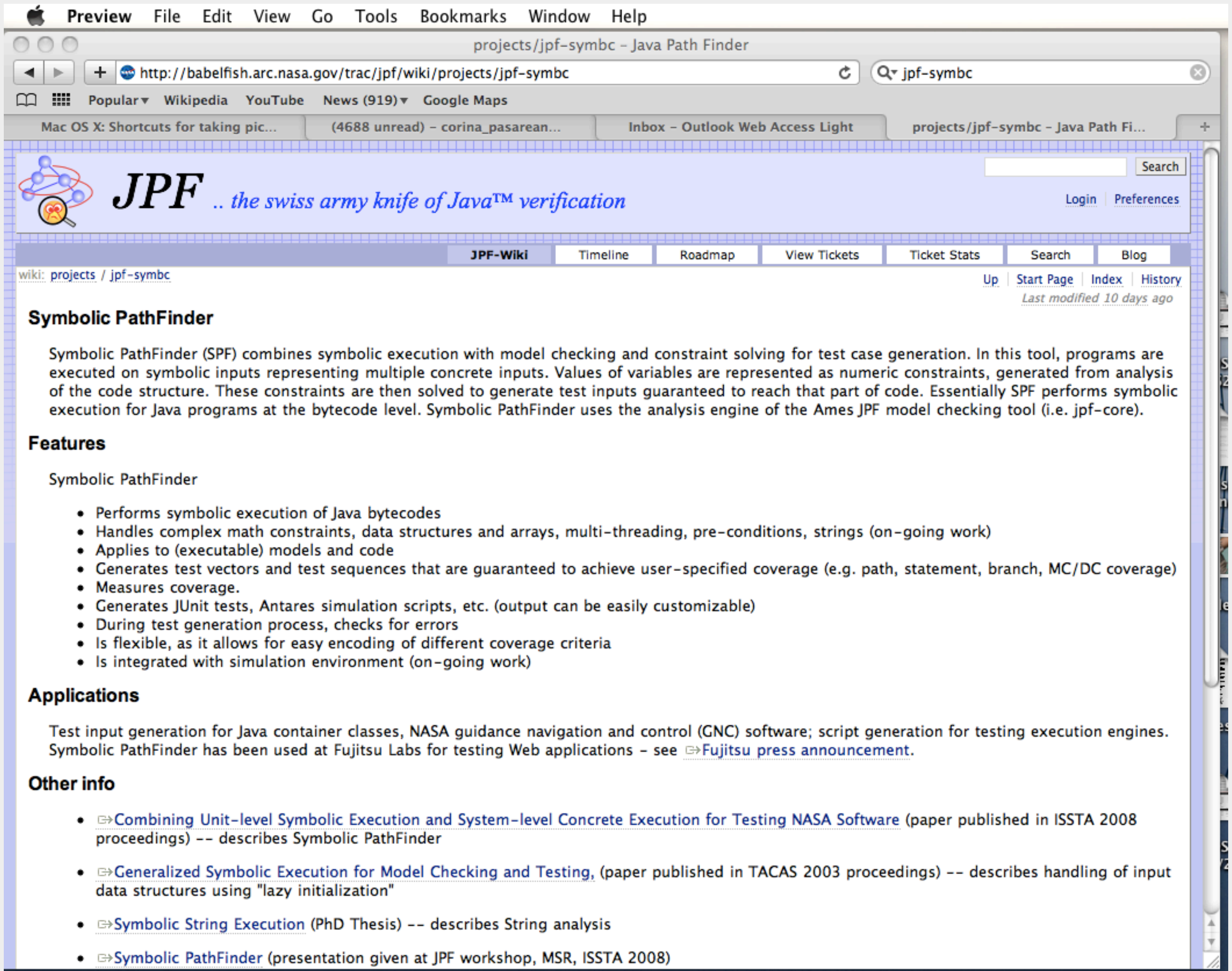
- integration with CORAL for solving complex mathematical constraints

test case generation for Android apps …

symbolic pathfinder

available from jpf distribution

http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc



Preview File Edit View Go Tools Bookmarks Window Help

projects/jpf-symbc – Java Path Finder

http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc

jpf-symbc

Popular Wikipedia YouTube News (919) Google Maps

Mac OS X: Shortcuts for taking pic... | (4688 unread) – corina_pasarean... | Inbox – Outlook Web Access Light | projects/jpf-symbc – Java Path Fi...

JPF .. *the swiss army knife of Java™ verification*

Search

Login | Preferences

JPF-Wiki | Timeline | Roadmap | View Tickets | Ticket Stats | Search | Blog

wiki: projects / jpf-symbc

Up | Start Page | Index | History

Last modified 10 days ago

Symbolic PathFinder

Symbolic PathFinder (SPF) combines symbolic execution with model checking and constraint solving for test case generation. In this tool, programs are executed on symbolic inputs representing multiple concrete inputs. Values of variables are represented as numeric constraints, generated from analysis of the code structure. These constraints are then solved to generate test inputs guaranteed to reach that part of code. Essentially SPF performs symbolic execution for Java programs at the bytecode level. Symbolic PathFinder uses the analysis engine of the Ames JPF model checking tool (i.e. jpf-core).

Features

Symbolic PathFinder

- Performs symbolic execution of Java bytecodes
- Handles complex math constraints, data structures and arrays, multi-threading, pre-conditions, strings (on-going work)
- Applies to (executable) models and code
- Generates test vectors and test sequences that are guaranteed to achieve user-specified coverage (e.g. path, statement, branch, MC/DC coverage)
- Measures coverage.
- Generates JUnit tests, Antares simulation scripts, etc. (output can be easily customizable)
- During test generation process, checks for errors
- Is flexible, as it allows for easy encoding of different coverage criteria
- Is integrated with simulation environment (on-going work)

Applications

Test input generation for Java container classes, NASA guidance navigation and control (GNC) software; script generation for testing execution engines. Symbolic PathFinder has been used at Fujitsu Labs for testing Web applications – see Fujitsu press announcement.

Other info

- Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software (paper published in ISSTA 2008 proceedings) -- describes Symbolic PathFinder
- Generalized Symbolic Execution for Model Checking and Testing, (paper published in TACAS 2003 proceedings) -- describes handling of input data structures using "lazy initialization"
- Symbolic String Execution (PhD Thesis) -- describes String analysis
- Symbolic PathFinder (presentation given at JPF workshop, MSR, ISSTA 2008)

how to run spf

go to: http://babelfish.arc.nasa.gov/trac/jpf/

download: jpf-core and jpf-symbc

set up the site properties

examples in jpf-symbc

    src/examples/summerschool

how to run them (in eclipse):

    select a .jpf configuration file

    run with run-JPF-symbc

Symbolic Execution and Software Testing Part 2

Corina Păsăreanu
CMU Silicon Valley/ NASA Ames Research Center

NATO International Summer School 2012,
Marktoberdorf, Germany

outline

Part 1

- introduction: symbolic execution
- symbolic pathfinder: symbolic execution for Java bytecode
- input data structures
- multi-threading

Part 2

- dynamic techniques
- the DART algorithm
- concolic execution

Part 3

- challenges
- solving complex constraints
- parallel and compositional techniques
- abstraction
- symbolic execution with mixed concrete-symbolic solving

Part 4

- applications
- current and future work

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 path condition

- condition on inputs – for the execution to follow that path
- check path condition satisfiability – explore only feasible paths

symbolic state

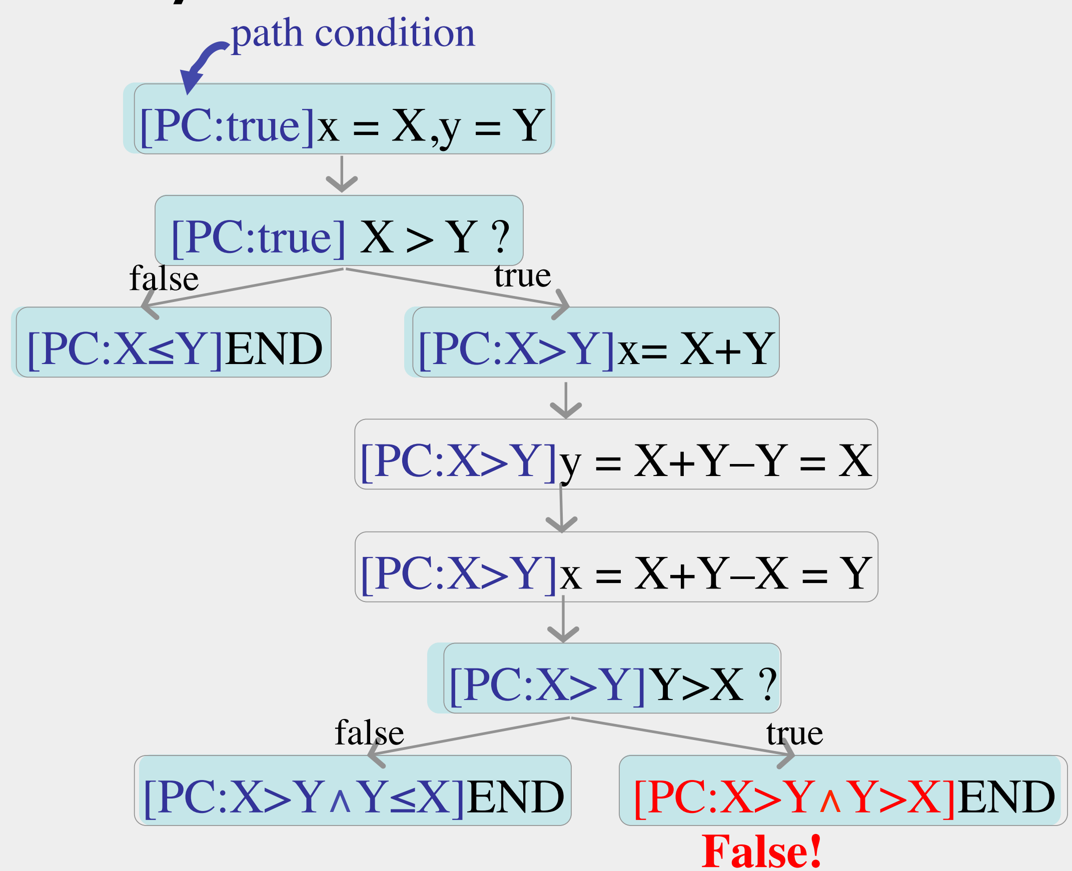
- symbolic values/expressions for variables
- path condition
- program counter

example: symbolic execution

Code that swaps 2 integers

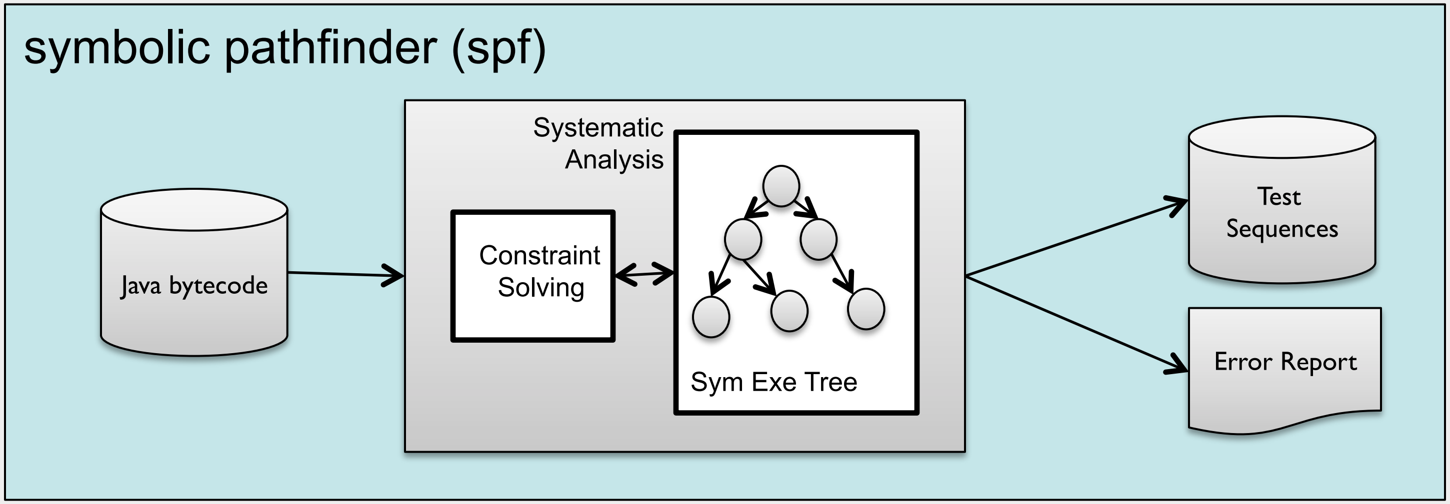
```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Symbolic Execution Tree



Solve PCs: obtain test inputs

symbolic pathfinder (spf)



combines symbolic execution, model checking and constraint solving

applies to executable models and code

handles dynamic data structures, loops, recursion, multi-threading; arrays and strings

java pathfinder extension project [TACAS'03, ISSTA'08, ASE'10]

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

dynamic techniques

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.

thanks P. Godefroid

DART: Directed Automated Random Testing

1. Automated extraction of program interface from source code
2. Generation of test driver for random testing through the interface
3. Dynamic test generation to direct executions along alternative program paths
   - Together: (1)+(2)+(3) = DART
   - DART can detect program crashes and assertion violations.
   - Any program that compiles can be run and tested this way:

     No need to write any test driver or harness code!
   - (Pre- and post-conditions can be added to generated test-driver)

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

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

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

Solve: !(x≤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

x = 1, y = 0

Symbolic Execution

create symbolic variables x, y

Path Constraint

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	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 = x+y	
x = 1, y = 1	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

x = 0, y = 1

Symbolic Execution

create symbolic variables x, y

y = x

x = y

Path Constraint

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	
		$x > y$
	$y = x$	
	$x = y$	
$x = 0, y = 1$		$y \leq x$

Solve: $x > y$ AND $!(y \leq x)$

Impossible: DONE!

another example

```
void test(int x, int y) {
  int z = x*x*x;
  if (y==z)
    assert false;
}
```

using concrete values

the power of DART

Concrete Execution	Symbolic Execution	Path Constraint
x = 3, y = 7	create symbolic variables x, y	

```
void test(int x, int y) {   <--
  int z = x*x*x;
  if (y==z)
    assert false;
}
```

the power of DART

```
void test(int x, int y) {
  int z = x*x*x;
  if (y==z)
    assert false;
}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 3, y = 7	create symbolic variables x, y	
z = 27	z = x*x*x	

the power of DART

```
void test(int x, int y) {
  int z = x*x*x;
  if (y==z)
    assert false;
}
```

Concrete Execution

x = 3, y = 7

z = 27

Symbolic Execution

create symbolic variables x, y

z = x*x*x

Path Constraint

y != x*x*x

Solve: !(y!=x*x*x)

Non-linear -- not possible to solve!

the power of DART

```
void test(int x, int y) {
  int z = x*x*x;
  if (y==z)
    assert false;
}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 3, y = 7	create symbolic variables x, y	
z = 27	z = x*x*x	
		y != x*x*x

Solve: !(y!=x*x*x)

DART solution: use concrete value of z

the power of DART

```
void test(int x, int y) {
  int z = x*x*x;
  if (y==z)
    assert false;
}
```

Concrete Execution	Symbolic Execution	Path Constraint
x = 3, y = 7	create symbolic variables x, y	
z = 27	z = x*x*x	
		y != x*x*x

Solve: !(y!=27)

DART solution: use concrete value of z

the power of DART

```
void test(int x, int y) {
  int z = x*x*x;
  if (y==z)
    assert false;
}
```

	Concrete Execution	Symbolic Execution	Path Constraint
	x = 3, y = 27	create symbolic variables x, y	
	z = 27	z = x*x*x	
			Y == x*x*x
←	Error discovered!		

DART: Directed Automated Random Testing

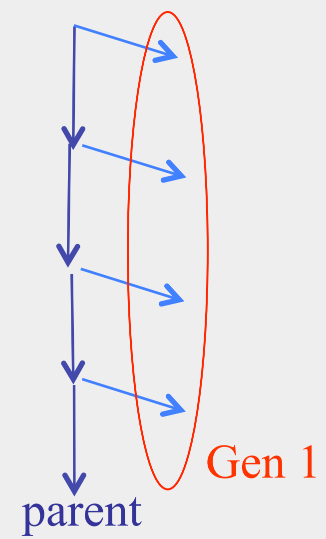
- very popular
- easy to implement
- implemented and extended in many interesting ways
- many tools
  - PEX, SAGE, CUTE, jCUTE, CREST, SPLAT, etc
- many applications
  - bug finding, security, web and database applications, etc.

EXE (Stanford Univ. [Cadar et al TISSEC 2008])

- related dynamic approach to symbolic execution

white-box fuzzing [NDSS'08]

- white-box Fuzzing = "DART meets Fuzz"
  - Black-box Fuzzing = randomly "fuzz"(modify) a well-formed input; simple but effective
- apply DART to large applications (not unit)
  - Binary level
  - Thousands of inputs, millions of instructions
- start with a well-formed input (not random)
- combine with a generational search (not DFS)
  - negate 1-by-1 each constraint in a path constraint
  - generate many children for each parent run
  - challenge all the layers of the application sooner
  - leverage expensive symbolic execution
- search spaces are huge, the search is partial…
- yet effective at finding bugs !



SAGE

SAGE found many new security bugs in Windows applications

Cost of each Microsoft Security Bulletin: $Millions

Cost due to worms (Slammer, CodeRed, Blaster, etc.):$Billions

apps: image processors, media players, file decoders,…

many bugs triaged as "security critical, severity 1, priority 1" (would trigger Microsoft security bulletin if known outside MS)

bugs missed by black-box fuzzers or static analysis

used daily in various Microsoft groups

Thanks P. Godefroid

CUTE, jCUTE, CREST, PEX

CUTE (for C) and jCUTE (for Java)

- extends DART to handle multi-threading programs with dynamic data structures
- pointer constraints and dynamic partial order reduction

CREST is a new extensible open source tool that performs dynamic testing for C

PEX is Microsoft's dynamic testing tool for .NET code

many, many other tools …

PEX

- Pex is a Visual Studio 2010 Power Tool
  - http://msdn.microsoft.com/en-us/vstudio/bb980963.aspx
  - Power Tools are a set of enhancements, tools and command-line utilities
- used by several groups within Microsoft
- externally, available under academic and commercial licenses
- Pex in the browser
  - http://pexforfun.com

Thanks N.Tillmann

EXE and KLEE

symbolic execution tools for C

- perform mixed symbolic/concrete execution
- model memory with bit-level accuracy
- systems code often treats memory as un-typed bytes and observes a single memory location in multiple ways

employ various constraint-solver optimizations, in addition to those implemented in the STP solver:

- irrelevant constraint elimination, cex caching, etc.

use search heuristics to get high-coverage

can interact with the external environment (KLEE)

EXE and KLEE

targeted at low-level systems code.
found bugs (including security vulnerabilities)

UNIX file systems	ext2, ext3, JFS
UNIX utilities	Coreutils, Busybox, Minix
MINIX device drivers	pci, lance, sb16
Library code	PCRE, uClibc, Pintos
Packet filters	FreeBSD BPF, Linux BPF
Networking servers	udhcpd, Bonjour, Avahi, WsMp3
Operating Systems	HiStar kernel
Computer vision code	OpenCV

KLEE

http://klee.llvm.org

open-sourced in June 2009

extended by several research groups

- wireless sensor networks
- schedule memoization in multithreaded code
- automated debugging
- online gaming
- exploit generation, etc.

Thanks C. Cadar

Symbolic Execution and Software Testing Part 3

Corina Păsăreanu
CMU Silicon Valley/ NASA Ames Research Center

NATO International Summer School 2012,
Marktoberdorf, Germany

outline

Part 1

- introduction: symbolic execution
- symbolic pathfinder: symbolic execution for Java bytecode
- input data structures
- multi-threading

Part 2

- dynamic techniques
- the DART algorithm
- concolic execution

Part 3

- challenges
- solving complex constraints
- parallel and compositional techniques
- abstraction
- symbolic execution with mixed concrete-symbolic solving

Part 4

- applications
- current and future work


challenges
path explosion
complex constraints
handling native calls

path explosion

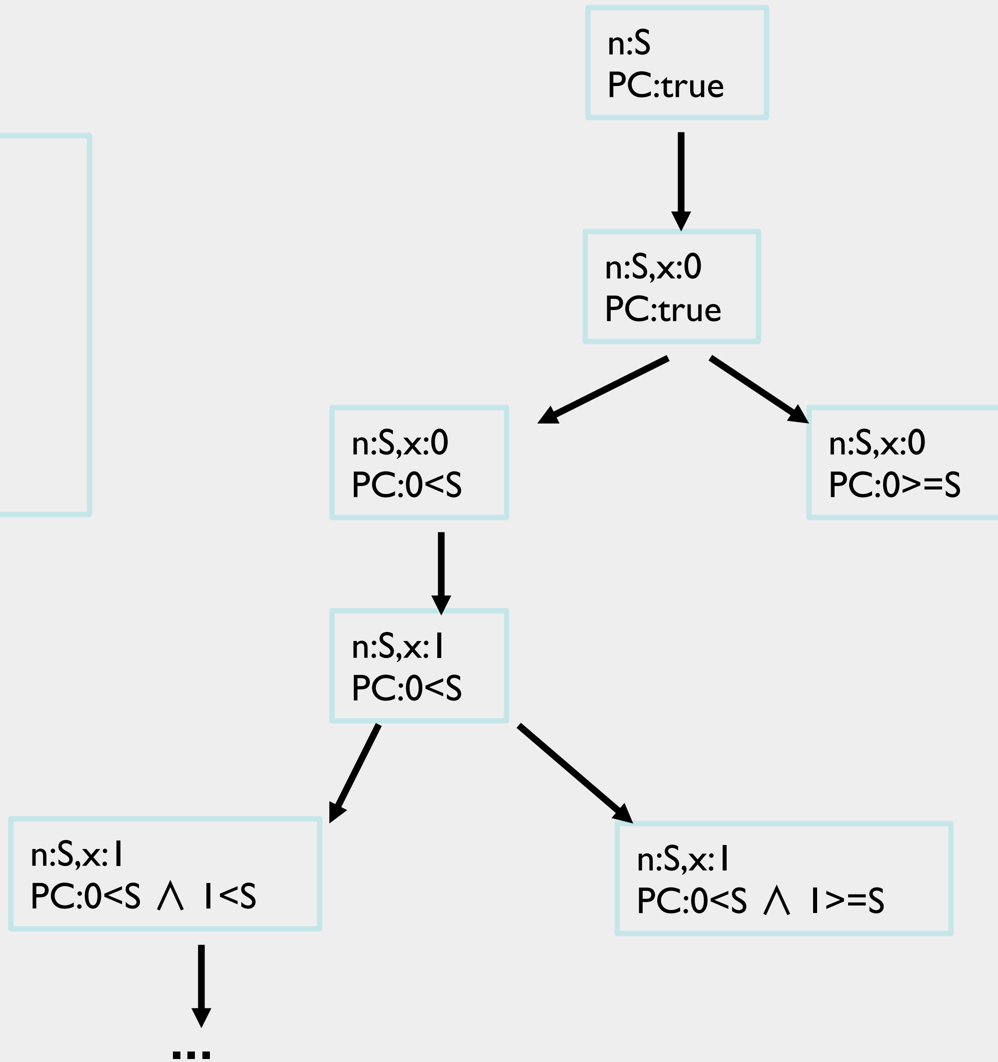
symbolic execution of a program may result in a very large, possibly infinite number of paths

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] [Strejček and Trtík ISSTA'12]

addressing path explosion

- parallel symbolic execution
- abstract state matching
- compositional DART = SMART

loop summaries

```
void test(int n) {
 int i=0;
 while(n>0) {
   if(i==200) assert false;
   i=i+1;
   n=n-1;
 }
 if (i==100) assert false;
}
```

symbolic execution

- generates 201 tests to hit 1st assertion
- possibly runs forever, without hitting 2nd assertion

use loop invariant

- $i+n=Sym_n$

loop summary (last iteration)

- $Pre_{loop}=(Sym_n>0)$
- $Post_{loop}= (n=1 \ \& \ i+n=Sym_n)$

simplified from [Godefroid&Luchaup ISSTA'11]

loop summaries

```
void test(int n) {
 int i=0;
 while(n>0) {
   if(i==200) assert false;
   i=i+1;
   n=n-1;
 }
 if (i==100) assert false;
}
```

on last loop iteration

- update PC with $Pre_{loop}=(Sym_n>0)$
- update symbolic state with $Post_{loop}= (n=1 \ \& \ i+n=Sym_n)$, i.e. i=Sym_n-1

results in PC

- $Sym_n>0 \ \& \ Sym_n-1!=200 \ \& \ Sym_n!=100$

running DART on n=0

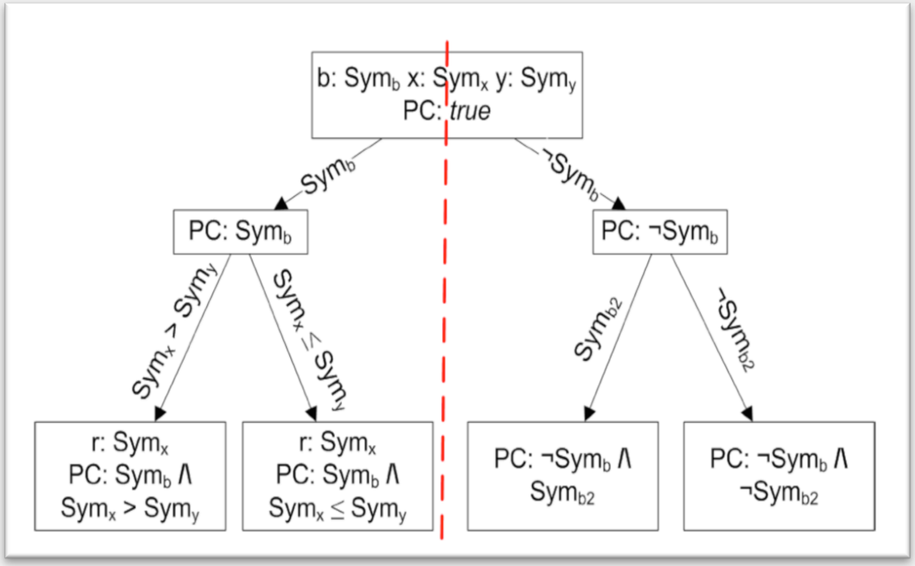
- will generate 4 tests to hit both assertions

parallel symbolic execution

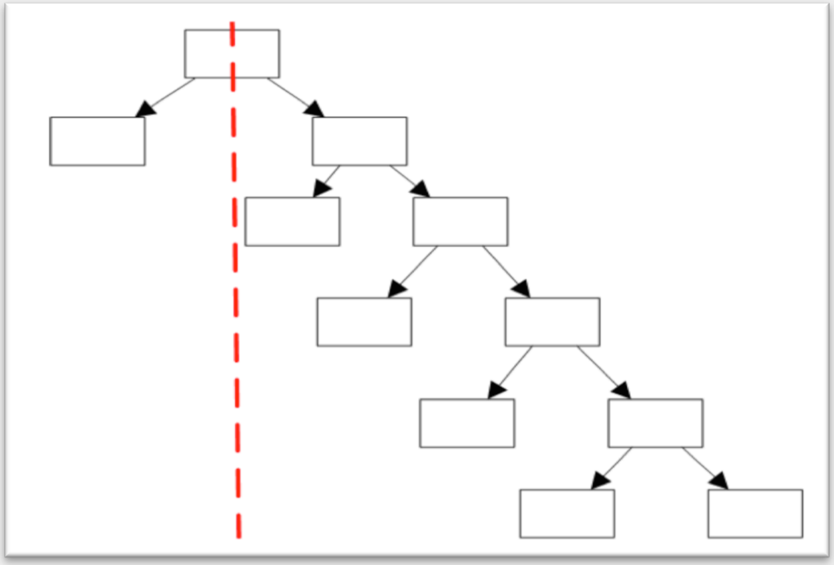
symbolic execution very amenable to parallelization

- no sharing between sub-trees

balancing partitions



nicely balanced – linear speedup



poorly balanced – no speedup

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

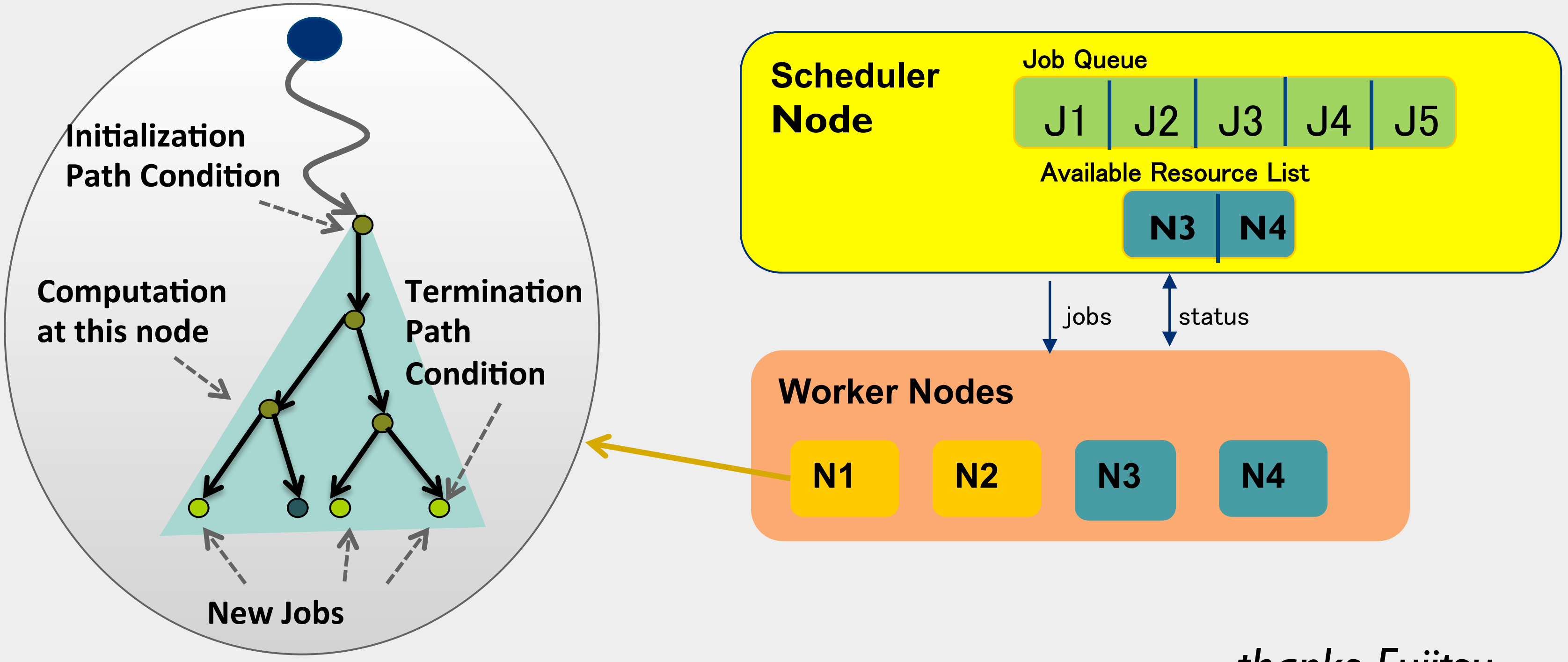
- commonly used variables likely to partition state space in useful ways

close to linear speed-up when using 128 workers

fujitsu applications

distributed symbolic execution over cloud

- adaptive dynamic partitioning
- heuristics to partition jobs on the fly based on system resources and job characteristics and history
- close to linear speed-up is possible in > 90% of the cases



thanks Fujitsu

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])

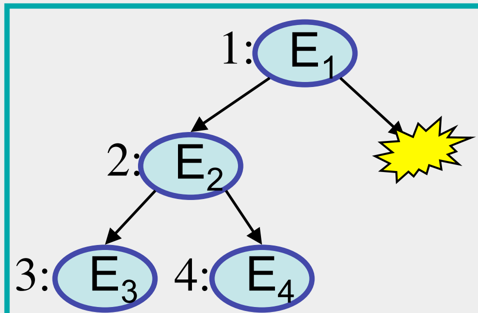
- singly linked lists
- arrays

no refinement!

see [Albarghouthi et al. CAV10] for symbolic execution with automatic abstraction-refinement

state matching with subsumption checking

stored state:

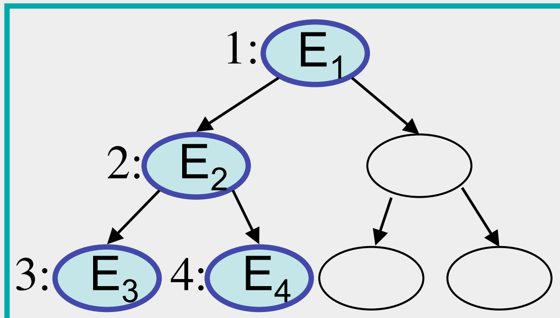


$E_1 > E_2 \wedge$
$E_2 > E_3 \wedge$
$E_2 \leq E_4 \wedge$
$E_1 > E_4$

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

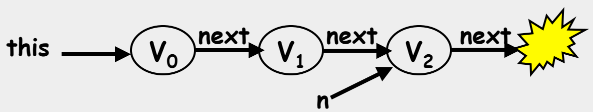
- summarize contiguous list elements not pointed to by program variables into summary nodes
- valuation of a summary node: union of valuations of summarized nodes
- subsumption checking between abstracted states
  - same algorithm as subsumption checking for symbolic states
  - treat summary node as an “ordinary” node

abstraction for arrays

- represent array as a singly linked list
- abstraction similar to shape abstraction for linked lists

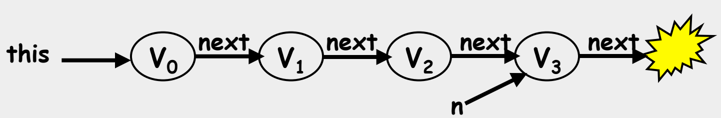
abstraction for lists

symbolic states



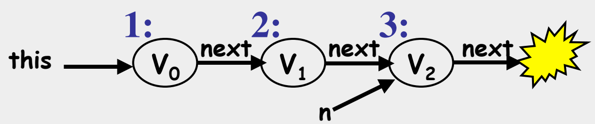
PC: $V_0 \leq v \wedge V_1 \leq v$

unmatched!



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

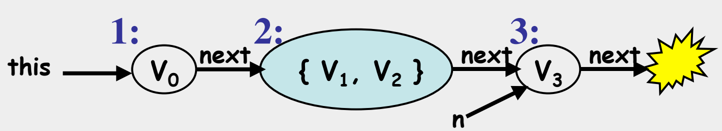
abstracted symbolic states



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

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

$\sqsubseteq$



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

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

compositional DART [POPL'07]

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 $\phi(g)$ is a disjunction of path constraints expressed in terms of input pre-conditions and output post-conditions:

  $\phi(g) = \bigvee \phi(w)$, with $\phi(w) = pre(w) \wedge post(w)$

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

ϕ (is_positive) = $(x>0 \land ret=1) \lor (x \le 0 \land ret=0)$

2 to execute both branches of (*) by solving:

$[(s[0]>0 \land ret_0=1) \lor (s[0] \le 0 \land ret_0=0)] \land$
$[(s[1]>0 \land ret_1=1) \lor (s[1] \le 0 \land ret_1=0)] \land \ldots \land$
$[(s[N-1]>0 \land ret_{N-1}=1) \lor (s[N-1] \le 0 \land ret_{N-1}=0)] \land$
$(ret_0+ret_1+ \ldots + ret_{N-1}=3)$

P. Godefroid

handling complex mathematical constraints

example constraint generated for a module from TSAFE (Tactical Separation Assisted Flight Environment)

```
sqrt(pow(((x1 + (e1 * (cos(x4) – cos((x4 + (((1.0 * (((c1 * x5) * (e2/c2))/x6)) * x2)/e1))))))
– (((e2/c2)) * (1.0 – cos((c1 * x5))))),2.0)) > 999.0  & (c1 * x5) > 0.0 &
x3 > 0.0 & x6 > 0.0 & c1 = 0.017… &
c2 = 68443.0 & e1 = ((pow(x2,2.0)/tan((c1*x3)))/c2) &
e2 = pow(x6,2.0)/tan(c1*x3)
```

coral solver

target application of solver: programs that

- use floating-point arithmetic
- call math functions

TSAFE example

input: sqrt(pow(((x1 + (e1 * (cos(x4) – …

output: {x1=100.0, x2=98.48…, x3=3.08…E-11, …}

approach: combine meta-heuristic search and interval solving [NFM'11, ICST'12]

meta-heuristic search

explores candidate solutions

- start with random solutions
- refine candidate set based on *fitness function*
- inherently incomplete

local search

- uses one single candidate solution
- e.g., Alternating Variable Method (AVM), hill climbing, simulated annealing, etc.

global search

- uses several candidate solutions
- e.g., Particle Swarm Optimization (PSO), genetic algorithms, etc.

interval solving

another method for constraint solving

input: sqrt(pow(((x1 + (e1 * (cos(x4) – …

output: {x1=[99.9…, 100.0], x2=[99.9…, 100.0], …}, …

interval

intervals may not contain solutions!

our approach: combine techniques

meta-heuristic search

+ good for finding exact solutions in large search spaces
- may get lost in local maxima

interval solving

+ good for computing parts of solution space
- does not compute solutions

seed meta-heuristic search
with inputs drawn from intervals
(intuition: better initial states)

evaluation

publicly available applications from the aerospace domain

Subject	# constraints	# conjuncts	# functions
Apollo Autopilot	800	39	3
Collision Detection (CDx)	800	63	6
Conflict Probe	33	7	5
Turn Logic	329	3	20

TSAFE units

evaluated CORAL configurations

meta-heuristic search alone

- AVM
- PSO (previously found it better than GA)

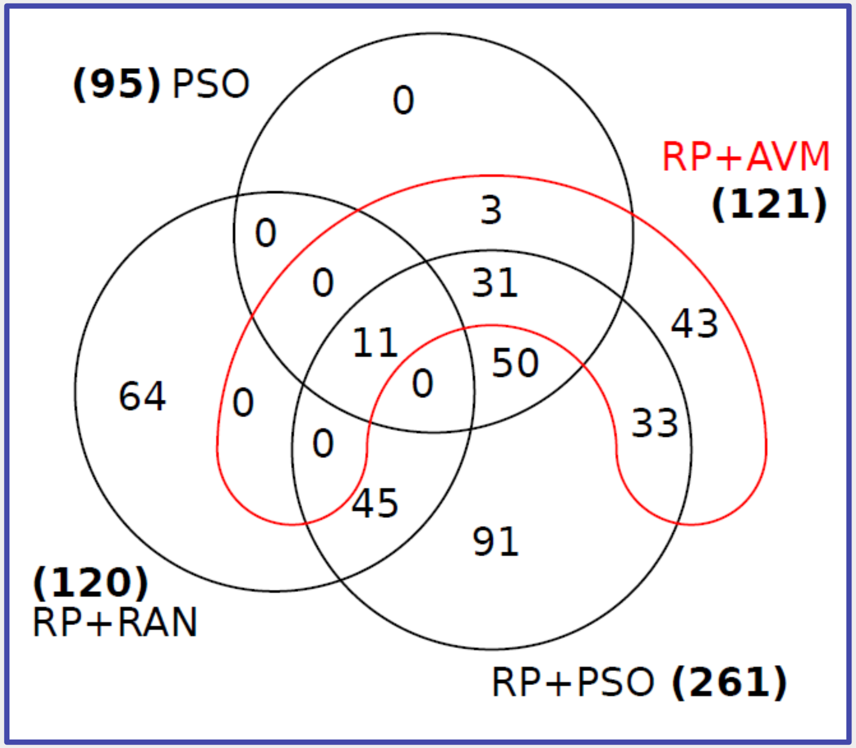
interval solving w/ RealPaver (RP) alone

- RP+RAN (choose random values from interval)

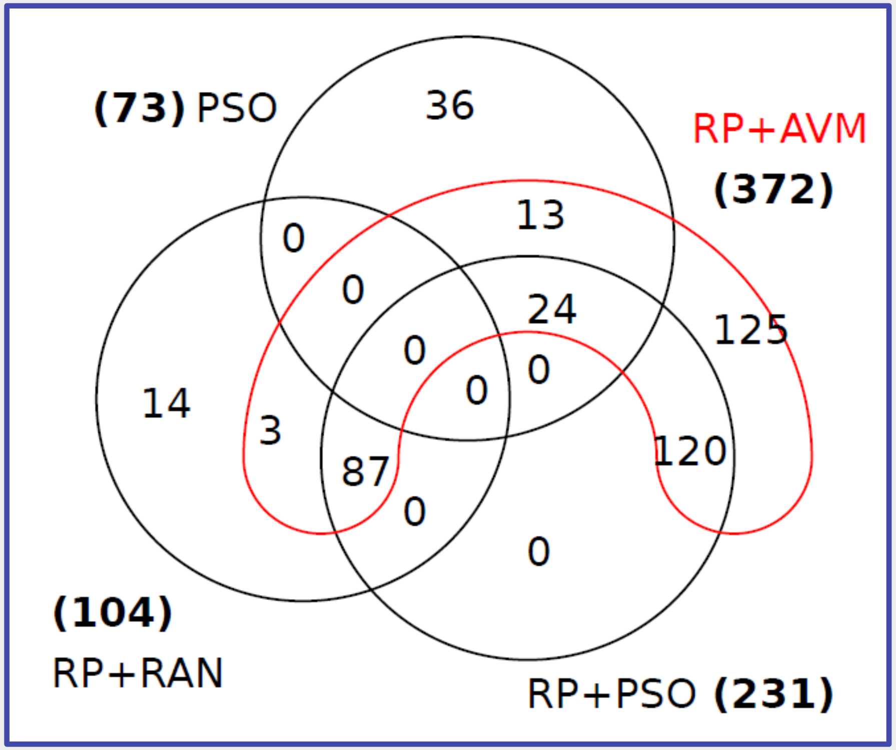
combinations of IS with global and local search

- RP+AVM – optimistic vs RV reported intervals
- RP+PSO – not so optimistic

results for Apollo and CDx



Apollo



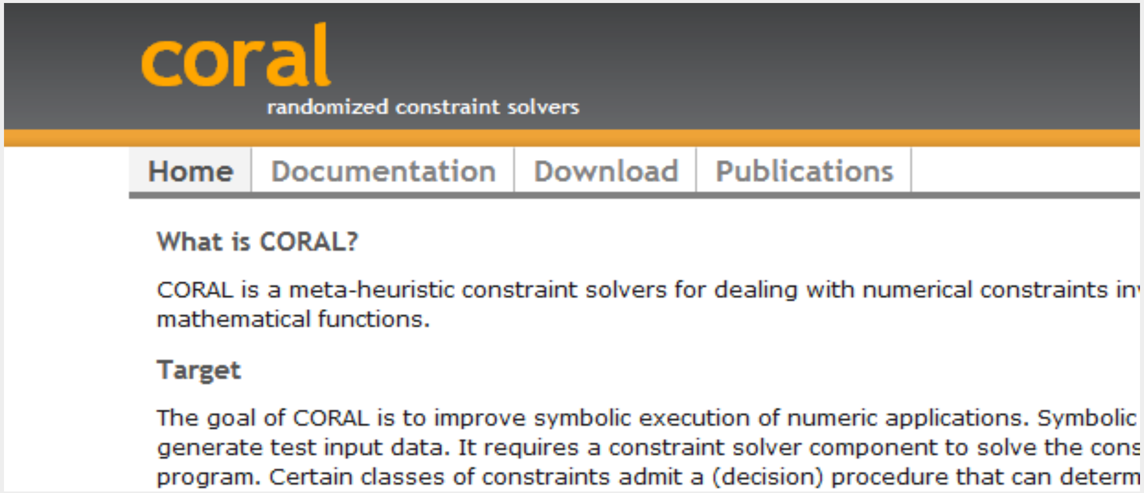
CDx

conclusions for CORAL

combination solved more constraints than meta-heuristic search or interval solving alone

- both global and local search help interval solving
- complementary: should be run together in parallel

http://pan.cin.ufpe.br/coral



handling native code

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

S0, S1, S3, S4 =
statements we wish to cover

hash is native or can not be handled by decision procedure

handling native code

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

hash is native or can not be handled by decision procedure

S0, S1, S3, S4 = statements we wish to cover

symbolic execution can not handle it!

solution:
provide "model" for *hash*
or
mixed concrete-symbolic solving [ISSTA'11]

mixed concrete-symbolic solving

use function symbols for external library calls

split path condition PC into:

- simplePC – solvable constraints
- complexPC – non-linear constraints with function symbols

solve simplePC

- use obtained solutions to simplify complexPC

check the result again for satisfiability

similar to DART

mixed concrete-symbolic solving

assume hash(x) = 10 *x:

PC: $X>3 \wedge Y>10 \wedge Y=hash(X)$

simplePC complexPC

solve simplePC

use solution X=4 to compute h(4)=40

simplify complexPC: Y=40

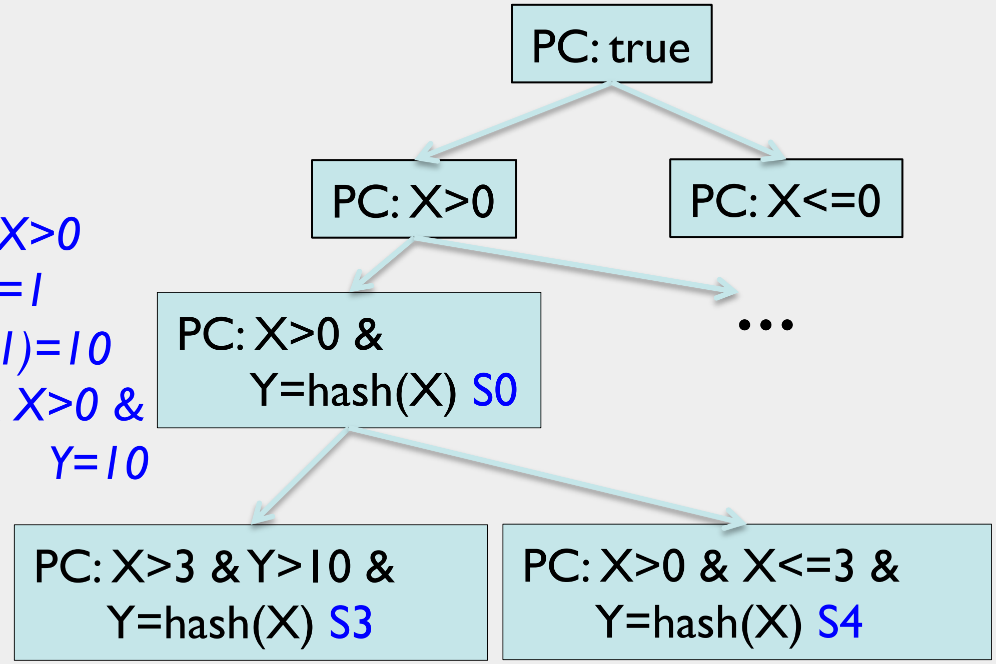
solve again

simplified PC: $X>3 \wedge Y>10 \wedge Y=40$ satisfiable!

symbolic execution

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

native int hash(x) {
  return x*10;
}
```



solve X>0
get X=1
hash(1)=10
check X>0 & Y=10

solve X>3 & Y>10
get X=4
hash(4)=40
check X>3 & Y>10 & Y=40

potential for unsoundness

```
test (int x, int y) {
  if (x>=0 && x>y && y == x*x)
    assert false;
  else
    ...;
}
```

not reachable

PC: X>=0 & X > Y & Y = X*X S0

simplePC X>=0 & X>Y

complexPC Y = X*X

X=0, Y=-1

Y=0*0=0

simplified PC X>=0 & X>Y & Y=0

is sat which implies assert is reachable!

Must add constraints on the solutions back into simplified PC

X>=0 & X>Y & Y=0 & X=0

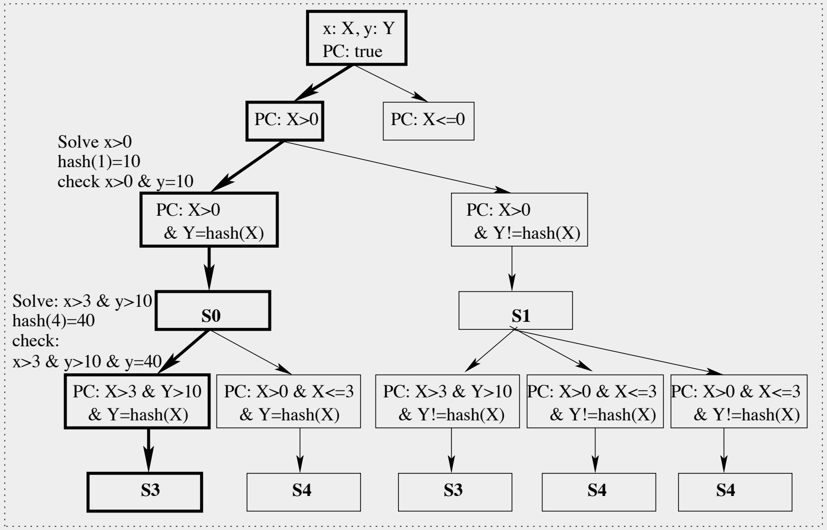
not sat!

DART/Concolic will diverge instead

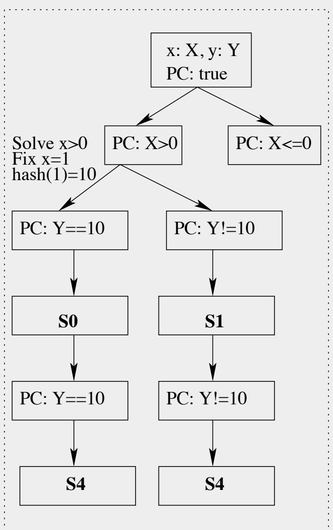
example

```
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;
  }
}
```

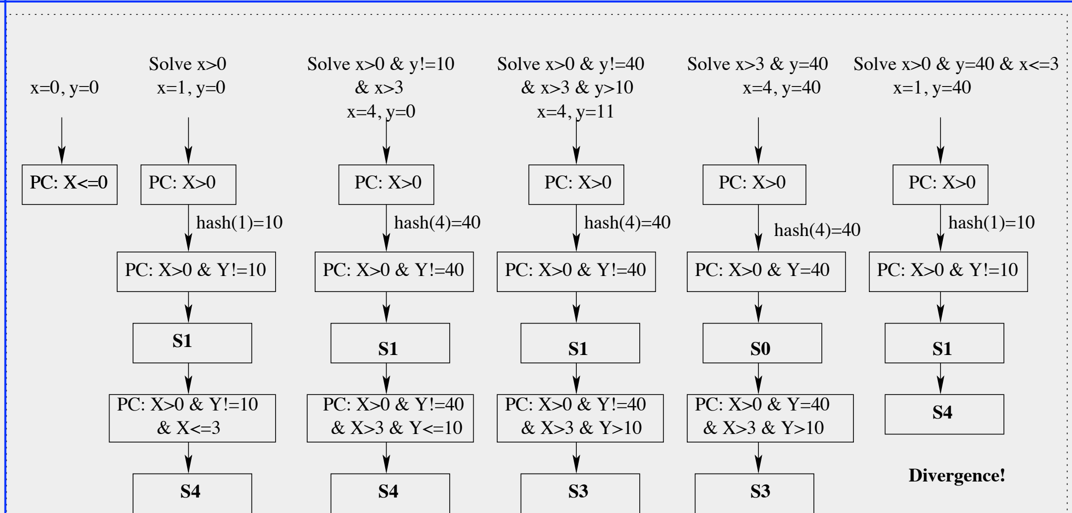
Example



Mixed concrete-symbolic solving: all paths covered



EXE results: stmt “S3” not covered



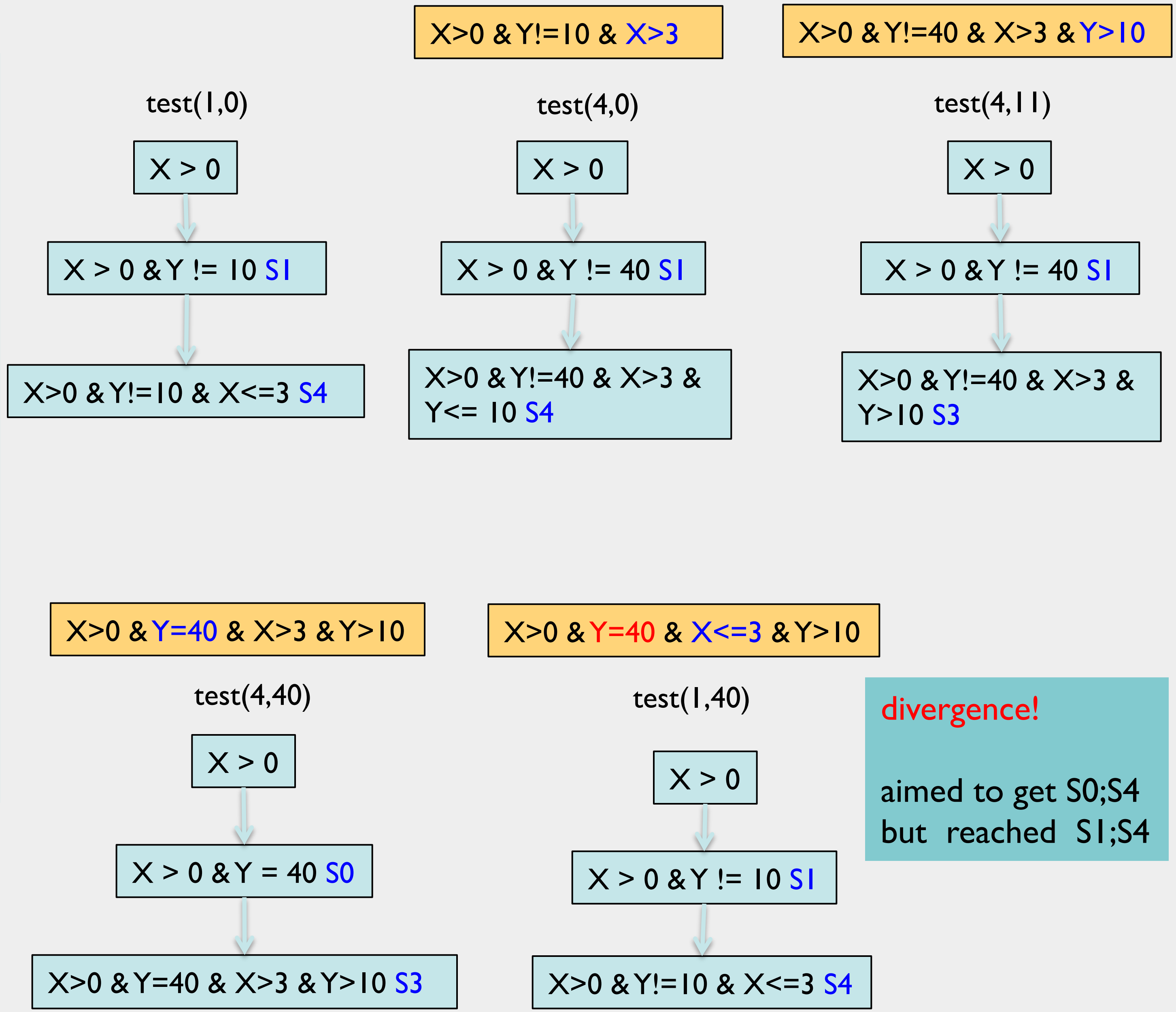
DART results: path “S0;S4” not covered

Predicted path “S0;S4”
!= path taken “S1;S4”

running DART

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

native int hash(x) {
  return x*10;
}
```



mixed concrete-symbolic solving vs dart

both techniques incomplete

incomparable in power (see paper)

mixed concrete-symbolic solving can handle only "pure", side-effect free functions

- DART does not have the limitation; will likely diverge

see also "higher order test generation" – P. Godefroid [PLDI'11]

- uses combination of validity checking and un-interpreted functions
- generates tests from validity proofs
- implementation challenge

solving string constraints

testing web applications – challenge

handling complex constraints involving strings and numerics

String s, q;

integer a, b;

s.equals(q) && s.startswith(“uvw”) && q.endswith(“xyz”) &&

s.length()<a && (a+b)<6 && b>0

unsatisfiable!

solving string constraints

solution – string solver

- maintain separate constraint set for Integer/Boolean and Real
- maintain separate constraint set for string variables – represented as FSMs or regular expressions
- pass learned constraints from one domain to another and iterate to fixed point or time out

string solver – incorporated in SPF (thanks to Willem Visser … still work in progress)

independent solution provided by Fujitsu

Symbolic Execution and Software Testing Part 4

Corina Păsăreanu
CMU Silicon Valley/ NASA Ames Research Center

NATO International Summer School 2012,
Marktoberdorf, Germany

outline

Part 1

- introduction: symbolic execution
- symbolic pathfinder: symbolic execution for Java bytecode
- input data structures
- multi-threading

Part 2

- dynamic techniques
- the DART algorithm
- concolic execution

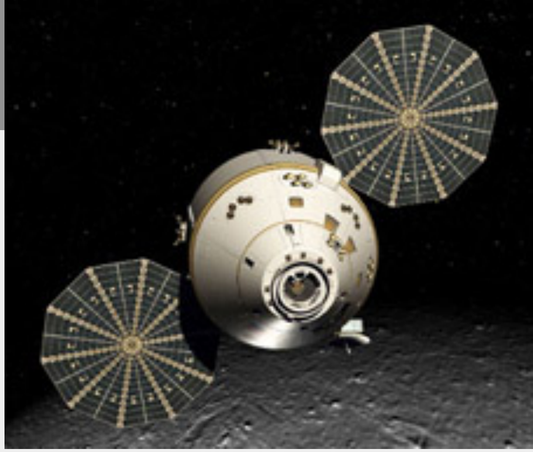
Part 3

- challenges
- solving complex constraints
- parallel and compositional techniques
- abstraction
- symbolic execution with mixed concrete-symbolic solving

Part 4

- applications
- current and future work

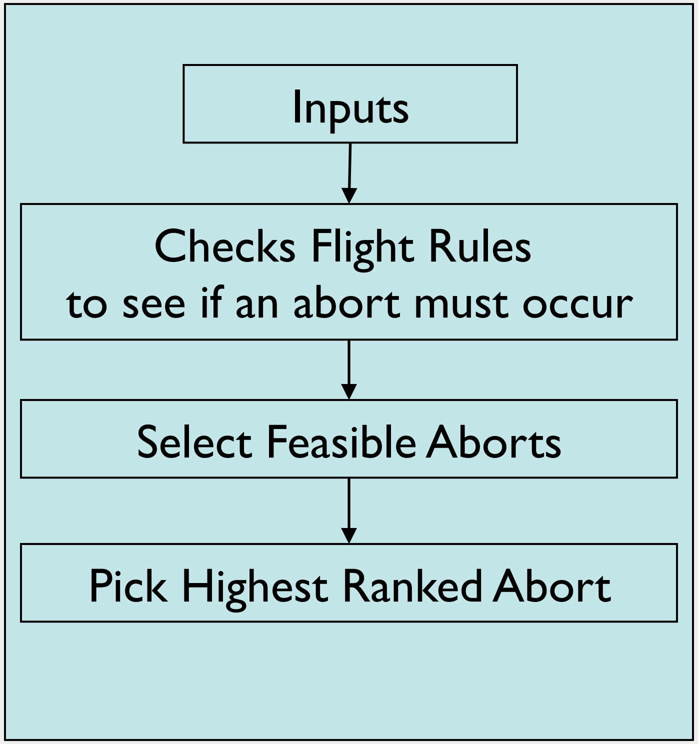
testing the Onboard Abort Executive (OAE)



Orion orbits the moon
(Image Credit: Lockheed Martin)

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

[ISSTA'08]

generated test cases and constraints

test cases:

// Covers Rule: FR A_2_A_2_B_1: Low Pressure Oxodizer Turbopump 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 && …

polyglot

large programs such as NASA Exploration

- build multiple systems that interact via safety-critical protocols
- designed with different Statechart variants
- a unified verification framework needed

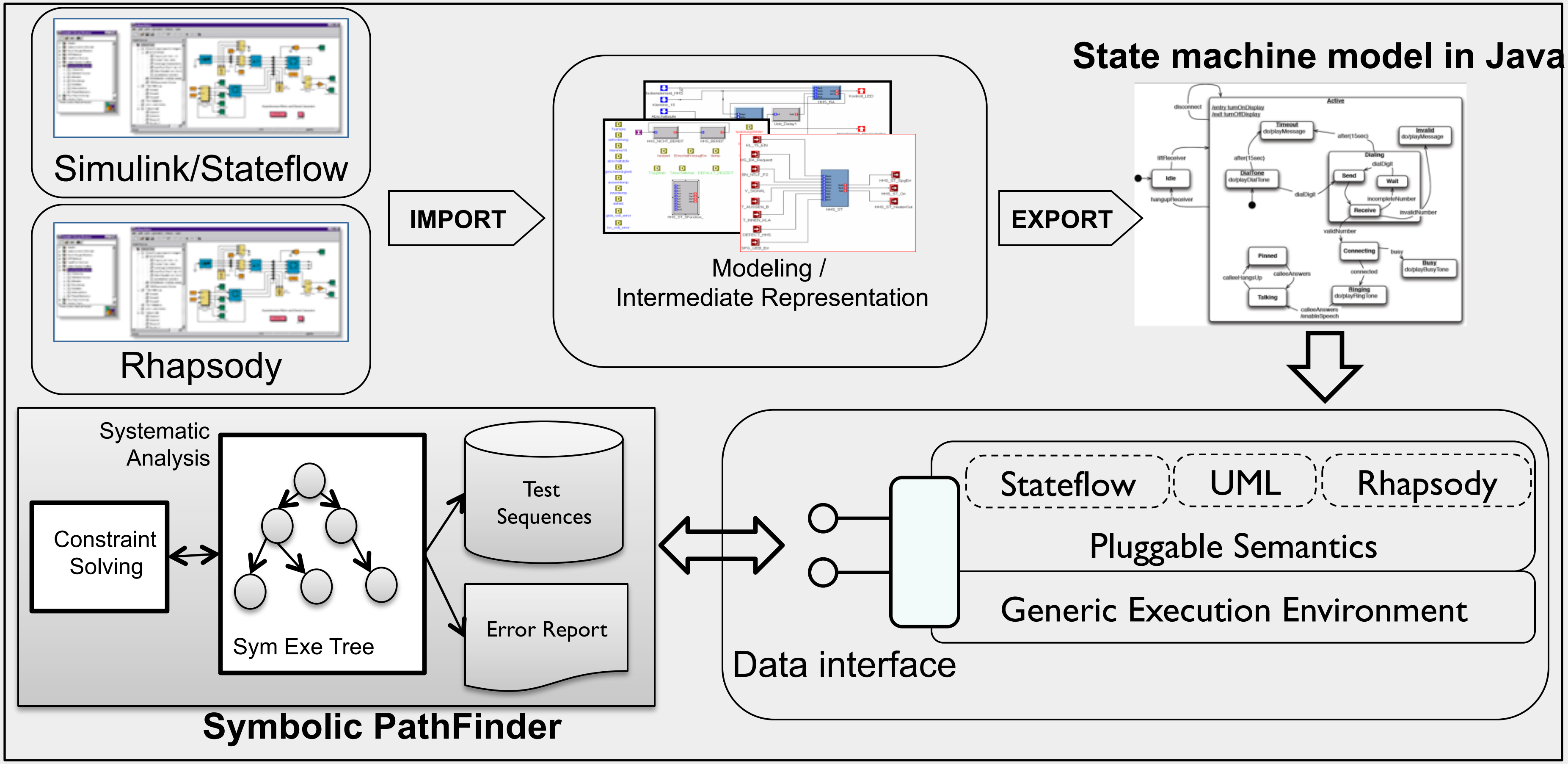
polyglot

- modeling and analysis for multiple Statechart formalisms
- captures interactions between components
- formal semantics that captures the variants of Statecharts
- applied to JPL's MER arbiter, Ares-Orion communication

collaboration w/ Vanderbilt University and University of Minnesota

[ISSTA'11,NFM'12]

polyglot

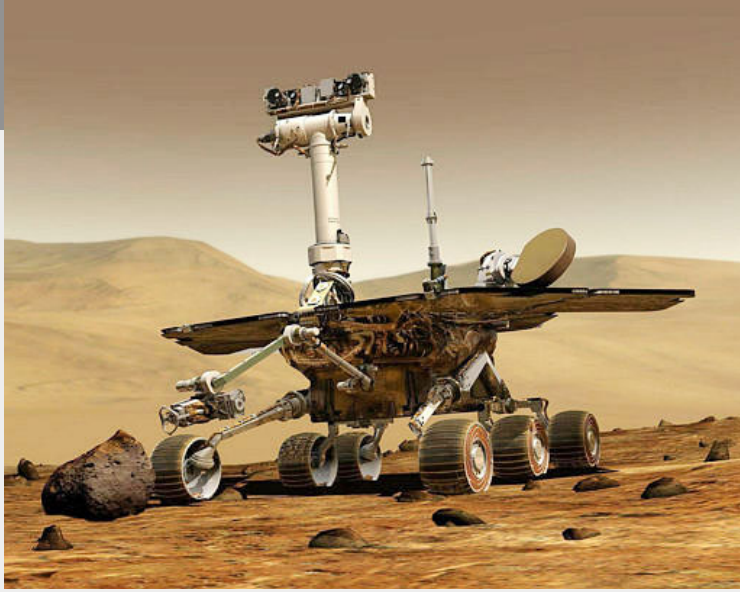


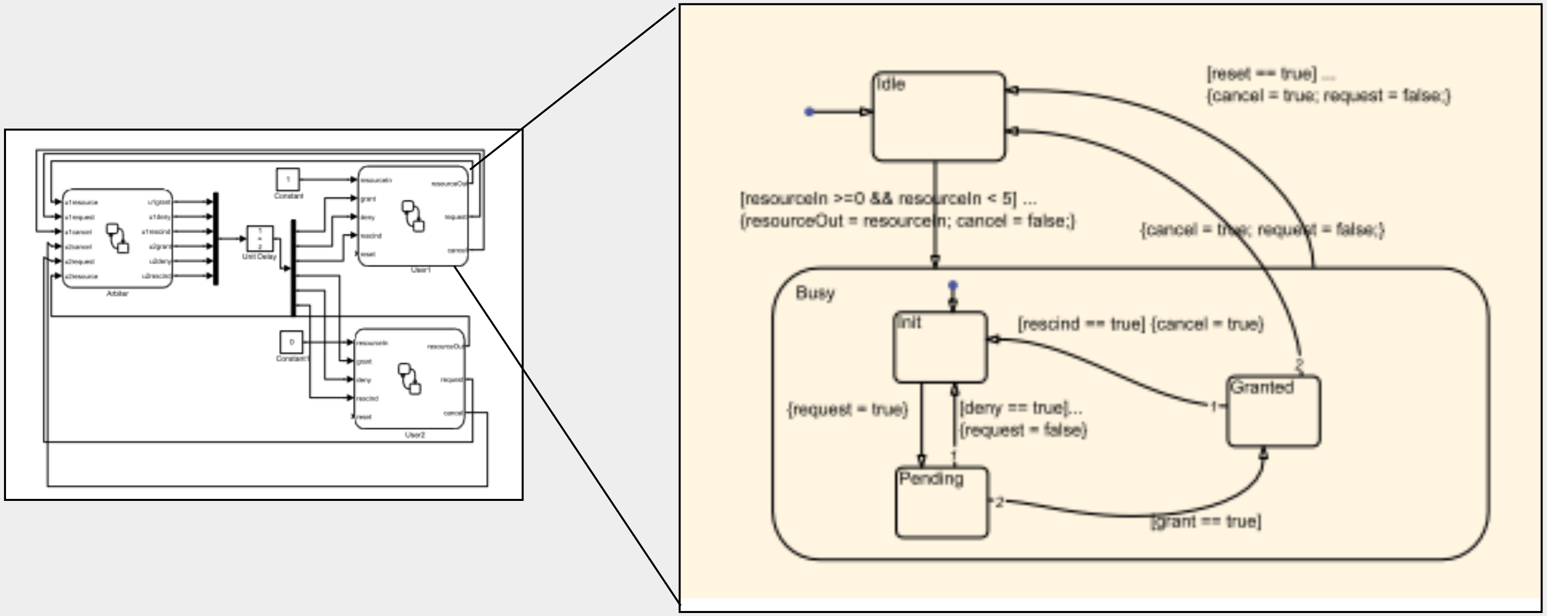
Simulink/Stateflow
Rhapsody
IMPORT
Modeling /
Intermediate Representation
EXPORT
State machine model in Java
Systematic
Analysis
Constraint
Solving
Sym Exe Tree
Test
Sequences
Error Report
Symbolic PathFinder
Stateflow
UML
Rhapsody
Pluggable Semantics
Generic Execution Environment
Data interface

example

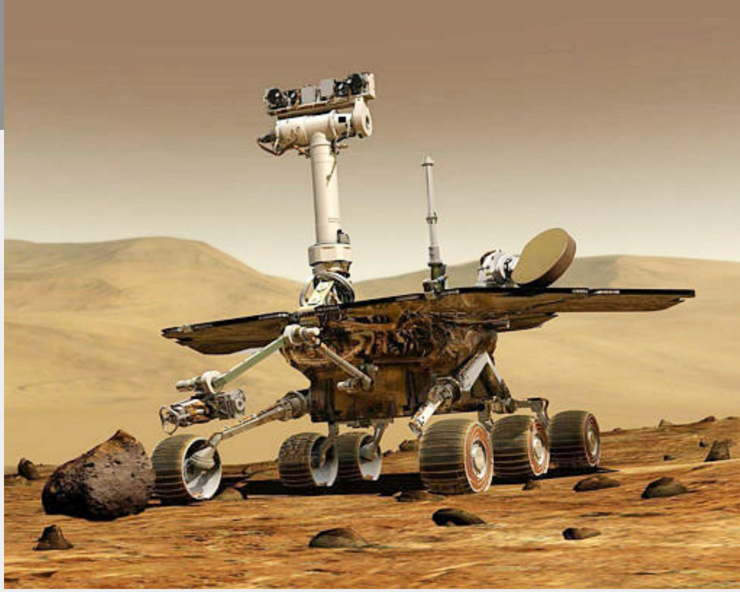
simplified model of the arbiter module
Mars Exploration Rover

- 3 Statecharts: 1 server, 2 clients
- Server grants/denies/rescinds resources





example (cont'd)



server contains: 33 pseudo-states (junctions), 15 atomic states, 2 orthogonal states and 58 transitions; 108 total elements

each user has 2 pseudostates, 4 atomic states, 1 compound state and 9 transitions; 16 total elements

Table 2: Experimental results

Semantics, Seq. size	Total # Test Cases	Property	Memory, Time
U1 Stateflow, 4	125	true	20 M, 43 s
U1 Stateflow, 5	412	true	22 M, 2 m 04 s
U1 Stateflow, 6	1343	true	24 M, 6 m 46 s
U1 UML, 4	57	false	21 MB, 21 s
U1 UML, 5	155	false	21 MB, 53 s
U1 UML, 6	579	false	23 MB, 2 m 50 s
U1 Rhapsody, 4	57	false	21 MB, 21 s
U1 Rhapsody, 5	155	false	21 MB, 55 s
U1 Rhapsody, 6	579	false	23 MB, 2 m 45 s

test generation for ttethernet protocol

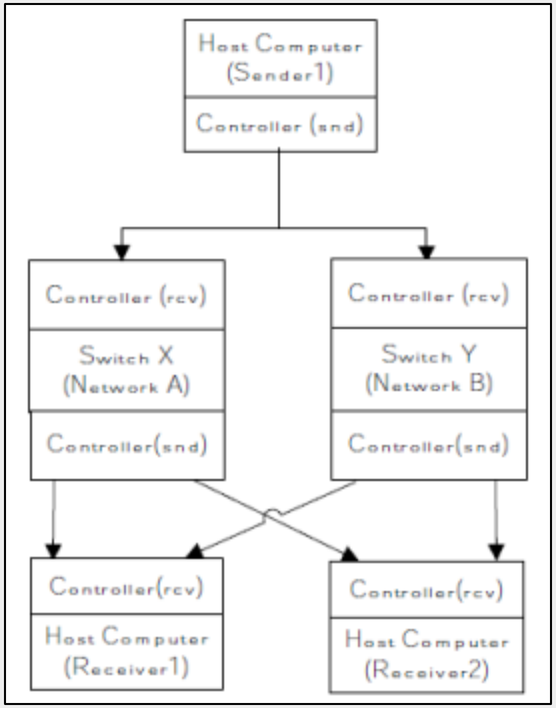
fault tolerant version of Ethernet protocol used by NASA in space networks
assure reliable network communications.

developed PVS model of basic version of the TTEthernet protocol

framework for translating models into Java multi-threaded code

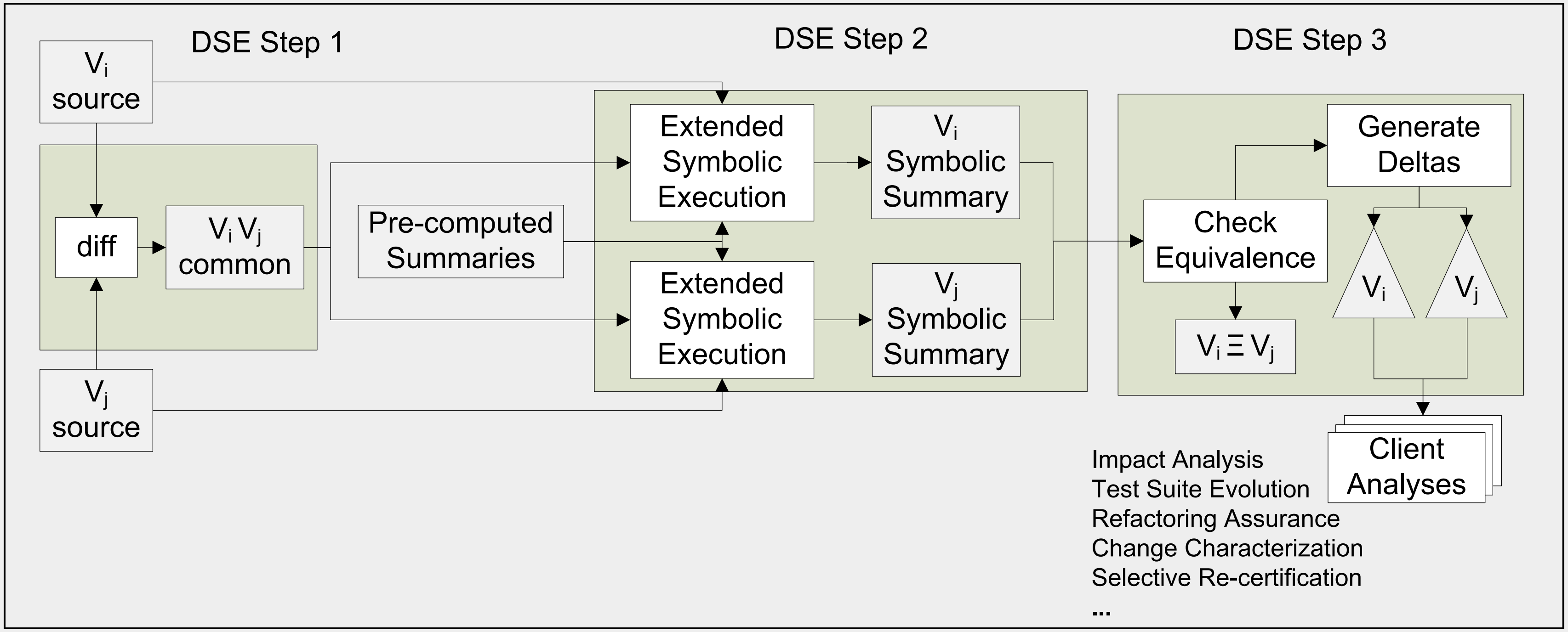
SPF analysis
- filtering of test cases to satisfy the various fault hypothesis
- verification of fault-tolerant properties
- demonstrated test case generation for TTEthernet's Single Fault Hypothesis

[w/ NASA Langley]



<u>Shown</u>: Minimal configuration for testing agreement in TTEthernet

differential symbolic execution – NASA Langley



computes logical difference between two program versions

uses loop and method summaries

[Person et al. FSE'08, Person et al PLDI'11]

memoized symbolic execution

stores symbolic execution tree for re-use

uses trie data structure

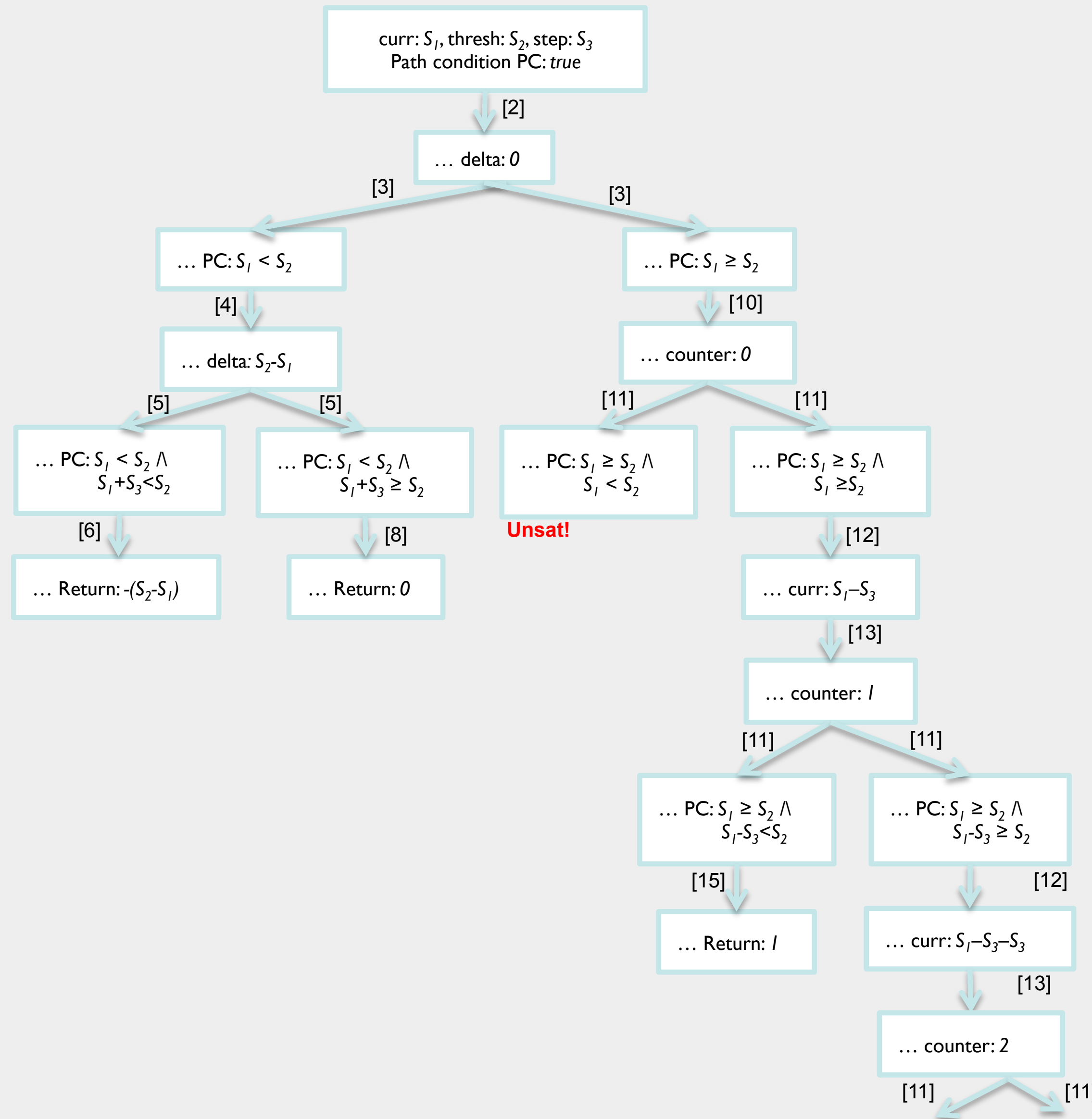
- stores only the choices in the tree
- maintained during successive symbolic execution runs

[ISSTA'12]

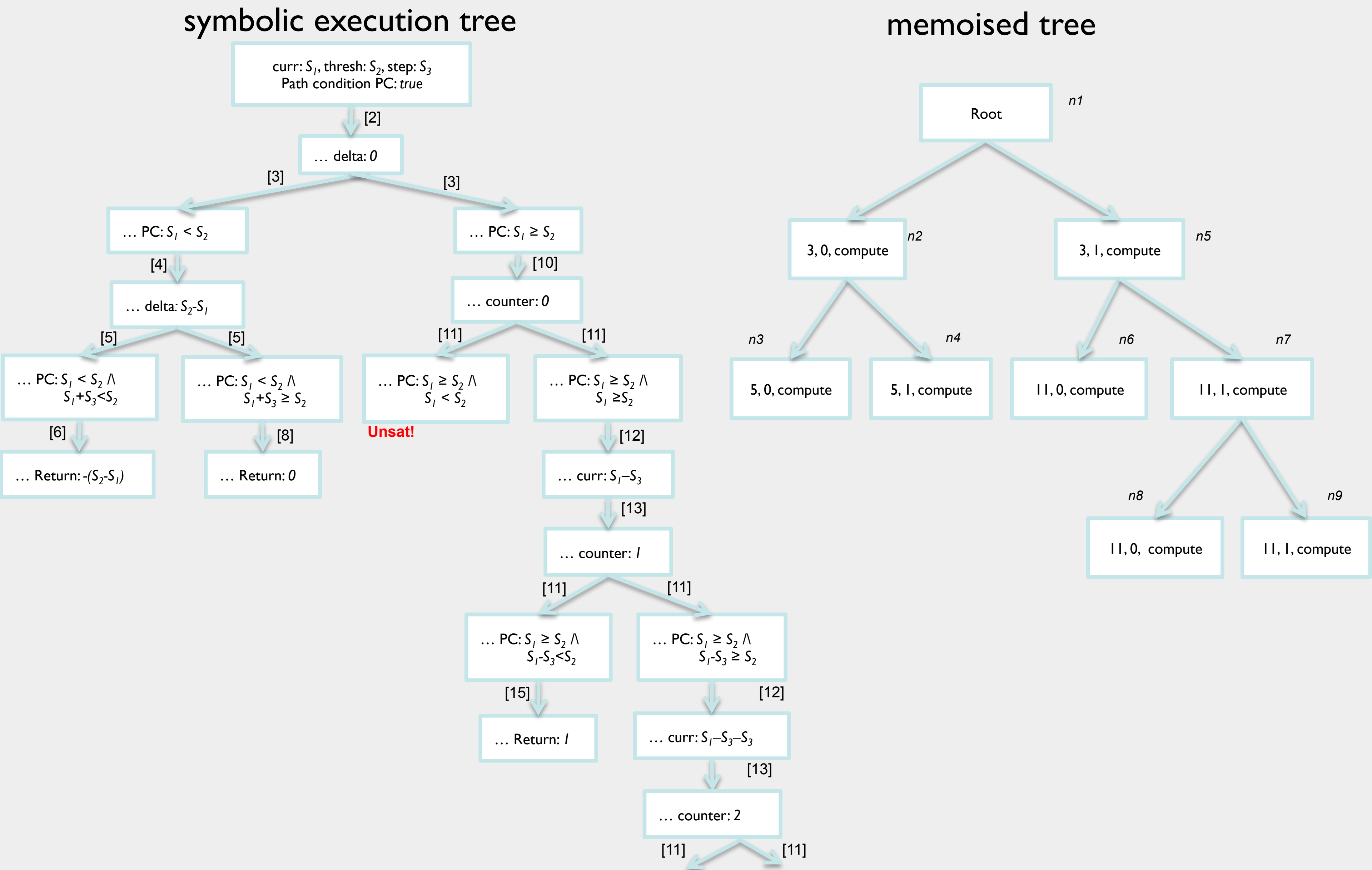
memoise – example

```
1  int compute (int curr,
                int thresh,
                int step) {
2    int delta = 0;
3    if(curr<thresh) {
4      delta = thresh-curr;
5      if((curr+step)<thresh)
6        return – delta;
7      else
8        return 0;
9    } else {
10     int counter=0;
11     while(curr>=thresh) {
12       curr=curr-step;
13       counter++;
14     }
15     return counter;
16   }
17 }
```

symbolic execution tree



memoise – example



applications

- iterative deepening
  - perform repeated symbolic execution with increasing depth
  - re-use results from smaller depths when exploring paths at larger depths
- regression analysis
  - analyze successive versions of a program
  - change impact analysis to identify nodes impacted by program change
  - re-execute only the paths impacted by the change
- heuristic guided symbolic execution
  - heuristic search of program paths, guided by the testing coverage achieved so far
  - iterative deepening – at each iteration discover paths that may lead to increased coverage
  - select only those paths in sub-sequent iterations

memoise

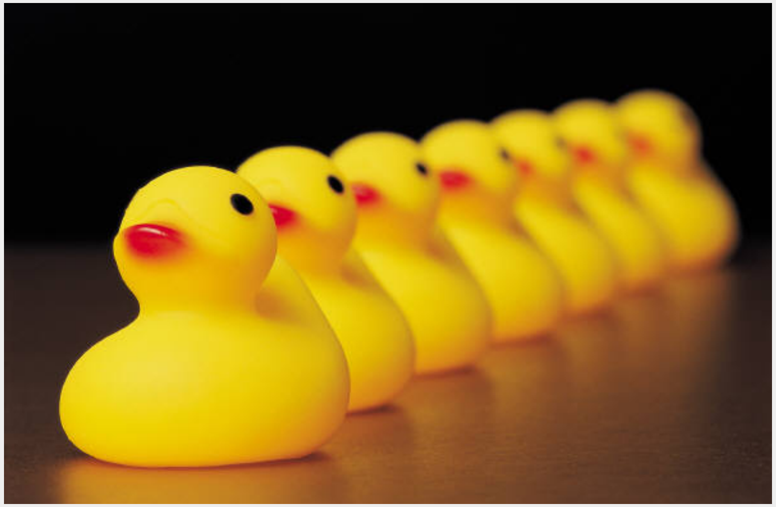
results – savings

- time (2x improvement)
- number of solver calls (up to 1000 less cals)
- number of states explored (1 order of magnitude improvement)

more applications

- continuous testing
- load balancing for parallel execution
- partial symbolic execution
- component certification

more enabled analyses



predictive testing [Majumdar & Sen ICSE'07]

- predicts errors from correct traces
- run an existing test suite
- perform a "concolic" execution along concrete tests
- check for assertion violations and other types of errors
- the assertions that hold along a concrete execution do not necessarily hold along the symbolic execution

robustness analysis [Majumdar & Saha RTSS'09]

- checks whether small perturbations in inputs cause only small changes in outputs
- based on symbolic execution and non-linear optimization
- computes maximum difference in program outputs over all program paths when a program input is perturbed
- generates a set of test vectors which demonstrate the worst-case deviations in outputs for small deviations in inputs

load testing [Zhang et al. ASE'11]

- validates whether system performance is acceptable under peak conditions
- symbolic execution used to compute values that induce load
- iterative-deepening approach favors program paths associated with a performance measure
- generated test suites induce program response times and memory consumption worse than compared alternatives

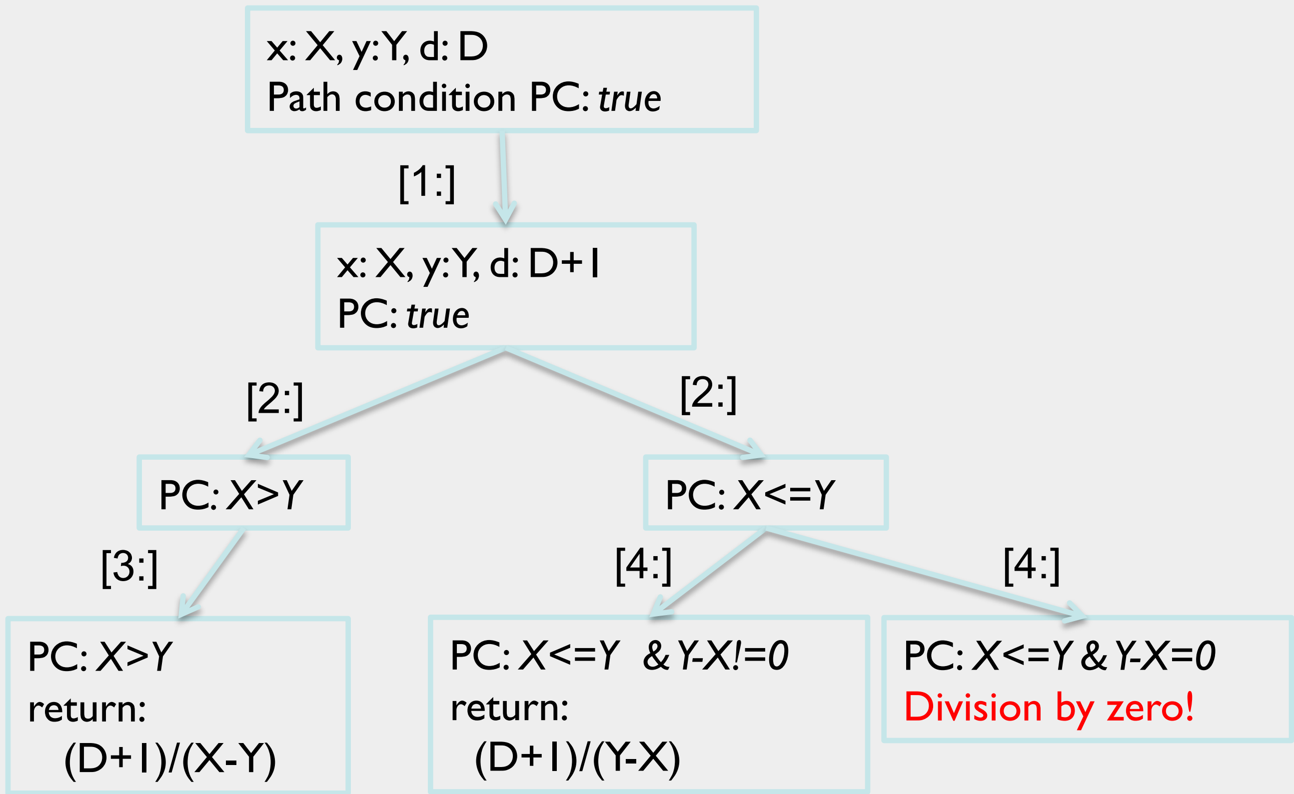
testing DB and GUI applications, security
many more …

a detailed example – "continuous" testing

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: $X<=Y$ & $Y-X=0 \Leftrightarrow X=Y$

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 e.g. [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]

proving properties of programs

Looping program:

```
X = init;
while (C(X))
    X = B(X);
assert P(X);
```

Program execution:

while …
true
while …
true
while …
true
…

*May be **infinite** …*

How to reason about infinite executions?

Find **loop invariant Inv**

Non-looping program:

```
X = init;
assert Inv(X);              Base Case
X = new symbolic values;
assume Inv(X);              ┐
if (C(X)) {                 │
    X = B(X);               ├ Induction Step
    assert Inv(X);          ┘
} else
    assert P(X);
```

*Has **finite** execution.*
Easy to reason about!

Problem:
*How do we come up with **Inv**?*
Requires great user ingenuity.
Many techniques that try to come up with Inv automatically.

symbolic execution and software testing

King [Comm. ACM 1976] , Clarke [IEEE TSE 1976]

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/Imperial 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
- Doron Peled's PET tool [CAV 2000]
- …

bibliography on symbolic execution (Saswat Anand):
http://sites.google.com/site/symexbib/

challenges

scalability

- 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]
- Compositional techniques [Godefroid, POPL'07]
- Incremental techniques [Person et al, PLDI'11]
- Loop abstraction [Saxena et al ISSTA'09] [Godefroid ISSTA'11] [Strejček and Trtík ISSTA'12]

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]
- Forward vs backward symbolic execution, precision issues …

current and future work for spf

- memoization [ISSTA'12 – Yang et al.]
  - saves symbolic execution tree for re-use
- probabilistic symbolic execution [ISSTA'12 – Dwyer et al.]
  - uses model counting for PCs to compute the probability of program statements
- new "green" constraint solver [FSE'12 – Visser et al.]
  - caches constraints for re-use

- reliability analysis (w/ A. Filieri and W. Visser)
  - computes probability of success or failure based on probabilistic usage profile
  - handles loops, multi-threading, data structures
- test case generation for Android apps
- program specialization
- multi-threading …

Thank you!