Symbolic Execution and Software Testing
Part I

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 $ \sim 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

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

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

model checking

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

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

model checking

property

always(φ or ψ)

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]
symbolic pathfinder (spf)

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
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.
Code that swaps 2 integers

```java
int x, y;

if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
}

if (x > y)
    assert false;
```

Concrete Execution Path

1. `x = 1, y = 0`
2. `1 > 0 ? true`
3. `x = 1 + 0 = 1`
4. `y = 1 - 0 = 1`
5. `x = 1 - 1 = 0`
6. `0 > 1 ? false`
**Code that swaps 2 integers**

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

**Symbolic Execution Tree**

```
[PC: true] x = X, y = Y

[PC: true] X > Y ?
  false
  [PC: X ≤ Y] END
  true
  [PC: X > Y] x = X + Y

[PC: X > Y] y = X + Y - Y = X

[PC: X > Y] x = X + Y - X = Y

[PC: X > Y] Y > X ?
  false
  [PC: X > Y ∧ Y ≤ X] END
  true
  [PC: X > Y ∧ Y > X] END False!
```

*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
int x, y;

if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x > y)
        assert false;
}
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 explicity

takes advantage of jpf’s optimizations!
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
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
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
Concrete execution of IADD byte-code:

```java
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:

```java
public class IADD extends Instruction { ... 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);
        }
    }
}
```
Concrete execution of IFGE byte-code:

```java
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:

```java
public class IFGE extends Instruction {
    ....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
        } return next;
    }
}
```
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
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

\[ x + 1 \rightarrow Math.sin \rightarrow \sin(x + 1) \]

symbolic expression w/ un-interpreted function handled directly by solver (Choco)
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)
public class Node {
  int elem;
  Node next;

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

consider executing
next = t.next;

E0 -> E1 -> null

E0 -> E1 -> ?

E0 -> E1 -> next

E0 -> E1 -> 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
    }
}
class TreeNode {
    int elem;
    TreeNode left;
    TreeNode right;

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

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] != CONST_-1 &&
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]
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
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

JPF the swiss army knife of Java™ verification

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)
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
<table>
<thead>
<tr>
<th>outline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Part 1</strong></td>
</tr>
<tr>
<td>▶ introduction: symbolic execution</td>
</tr>
<tr>
<td>▶ symbolic pathfinder: symbolic execution for Java bytecode</td>
</tr>
<tr>
<td>▶ input data structures</td>
</tr>
<tr>
<td>▶ multi-threading</td>
</tr>
<tr>
<td><strong>Part 2</strong></td>
</tr>
<tr>
<td>▶ dynamic techniques</td>
</tr>
<tr>
<td>▶ the DART algorithm</td>
</tr>
<tr>
<td>▶ concolic execution</td>
</tr>
<tr>
<td><strong>Part 3</strong></td>
</tr>
<tr>
<td>▶ challenges</td>
</tr>
<tr>
<td>▶ solving complex constraints</td>
</tr>
<tr>
<td>▶ parallel and compositional techniques</td>
</tr>
<tr>
<td>▶ abstraction</td>
</tr>
<tr>
<td>▶ symbolic execution with mixed concrete-symbolic solving</td>
</tr>
<tr>
<td><strong>Part 4</strong></td>
</tr>
<tr>
<td>▶ applications</td>
</tr>
<tr>
<td>▶ current and future work</td>
</tr>
</tbody>
</table>
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
**Code that swaps 2 integers**

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

**Symbolic Execution Tree**

- **Path Condition**
  - If `x > y`, then `x = x + y`, `y = x - y`, `x = x - y`.
  - If `x > y` and `y ≤ X`, then `false`.
  - If `x > y` and `Y > X`, then `false`.
  - If `x > Y ∧ Y ≤ X`, then `false`.
  - If `x > Y ∧ Y > X`, then `false`.

**False!**

**Solve PCs: obtain test inputs**

*example: symbolic execution*
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]
classic symbolic execution is a static technique

dynamic techniques

– collect symbolic constraints during concrete executions
– DART = Directed Automated Random Testing
– Concolic (Concrete Symbolic) testing
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
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)
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
int x, y;

if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x > y)
        assert false;
}

create symbolic variables x, y

Solve: !(x \leq y)
Solution: x=1, y=0

x = 0, y = 0
int x, y;

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

Concrete Execution

Symbolic Execution

equal x, y

create symbolic variables x, y

Path Constraint

directed search
```c
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`
```c
t
```
```c
int x, y;

if (x > y) {
  x = x + y;
  y = x - y;
  x = x - y;
  if (x > y)
    assert false;
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
<th>Path Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = 1, y = 1</code></td>
<td>create symbolic variables x, y</td>
<td><code>x &gt; y</code></td>
</tr>
<tr>
<td><code>x = x+y</code></td>
<td></td>
<td><code>x = x+y</code></td>
</tr>
<tr>
<td><code>y = x</code></td>
<td></td>
<td><code>y = x</code></td>
</tr>
</tbody>
</table>

**Directed search**
int x, y;

if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x > y)
        assert false;
}

directed search

Concrete Execution

Symbolic Execution

Path Constraint

create symbolic variables x, y

x > y

y = x

x = y

x = 0, y = 1
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
Path Constraint: x > y

Solve: x > y AND !(y ≤ x)
Impossible: DONE!

y = x
x = y
y ≤ x
another example

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

using concrete values
the power of DART

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
<th>Path Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void test(int x, int y) {</strong></td>
<td><strong>x = 3, y = 7</strong></td>
<td><strong>create symbolic variables x, y</strong></td>
</tr>
<tr>
<td>int z = x<em>x</em>x;</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>if (y==z)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>assert false;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>}</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
the power of DART

```java
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

x = 3, y = 7
void test(int x, int y) {
    int z = x*x*x;
    if (y==z)
        assert false;
}

x = 3, y = 7
z = 27

create symbolic variables x, y
z = x*x*x

y != x*x*x

Solve: !(y!=x*x*x)
Non-linear -- not possible to solve!
the power of DART

```c
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)

DART solution: use concrete value of z
the power of DART

```c
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=27)`

DART solution: use concrete value of `z`
```c
void test(int x, int y) {
    int z = x*x*x;
    if (y==z)
        assert false;
}
```

Concrete Execution:
- \(x = 3, y = 27\)
- \(z = 27\)

Symbolic Execution:
- create symbolic variables \(x, y\)
- \(z = x^3\)

Path Constraint:
- \(Y == x^3\)

Error discovered!
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 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 (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 is a Visual Studio 2010 Power Tool

– 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
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))
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX file systems</td>
<td>ext2, ext3, JFS</td>
</tr>
<tr>
<td>UNIX utilities</td>
<td>Coreutils, Busybox, Minix</td>
</tr>
<tr>
<td>MINIX device drivers</td>
<td>pci, lance, sb16</td>
</tr>
<tr>
<td>Library code</td>
<td>PCRE, uClibc, Pintos</td>
</tr>
<tr>
<td>Packet filters</td>
<td>FreeBSD BPF, Linux BPF</td>
</tr>
<tr>
<td>Networking servers</td>
<td>udhcppd, Bonjour, Avahi, WsMp3</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>HiStar kernel</td>
</tr>
<tr>
<td>Computer vision code</td>
<td>OpenCV</td>
</tr>
</tbody>
</table>
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.

http://klee.llvm.org

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

example code

```c
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
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 1\textsuperscript{st} assertion
• possibly runs forever, without hitting 2\textsuperscript{nd} assertion

calculate invariants
use loop invariant
• i+n=Sym\textsubscript{n}

loop summary (last iteration)
• Pre\textsubscript{loop}=(Sym\textsubscript{n}>0)
• Post\textsubscript{loop} = (n=1 & i+n=Sym\textsubscript{n})

simplified from [Godefroid&Luchau ISSTA’11]
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 \( \text{Pre}_{\text{loop}} = (\text{Sym}_n > 0) \)
• update symbolic state with \( \text{Post}_{\text{loop}} = (n=1 \& i+n=\text{Sym}_n) \), i.e. \( i=\text{Sym}_n-1 \)

results in PC
• \( \text{Sym}_n > 0 \& \text{Sym}_n-1!=200 \& \text{Sym}_n! =100 \)

running DART on \( n=0 \)
• will generate 4 tests to hit both assertions
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
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
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:

\[ E_1 > E_2 \land E_2 > E_3 \land E_2 \leq E_4 \land E_1 > E_4 \]

new state:

\[ E_1 > E_2 \land E_2 > E_3 \land E_2 < E_4 \land E_1 > E_4 \]

normalized using existential quantifier elimination

set of concrete states represented by stored state

set of concrete states represented by new state
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**

- Diagram showing transitions between states $V_0$, $V_1$, $V_2$, and $V_3$.
- PC: $V_0 \leq v \land V_1 \leq v$

**unmatched!**

- Diagram showing an unmatched state transition.
- PC: $V_0 \leq v \land V_1 \leq v \land V_2 \leq v$

**abstracted symbolic states**

- Diagram showing the abstracted states $1$, $2$, and $3$.
- $E_1 = V_0 \land E_2 = V_1 \land E_3 = V_2$
- PC: $V_0 \leq v \land V_1 \leq v$

**UI**

- Diagram showing the transition to state $3$.
- $E_1 = V_0 \land (E_2 = V_1 \lor E_2 = V_2) \land E_3 = V_3$
- PC: $V_0 \leq v \land V_1 \leq v \land V_2 \leq v$
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:

\[
\phi(g) = \bigvee \phi(w), \text{ with } \phi(w) = \text{pre}(w) \land \text{post}(w)
\]

\( g \)'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]
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

$$\phi (\text{is\_positive}) = (x > 0 \land ret = 1) \lor (x \leq 0 \land ret = 0)$$

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

$$[(s[0] > 0 \land ret_0 = 1) \lor (s[0] \leq 0 \land ret_0 = 0)] \land$$
$$[(s[1] > 0 \land ret_1 = 1) \lor (s[1] \leq 0 \land ret_1 = 0)] \land \ldots \land$$
$$[(s[N-1] > 0 \land ret_{N-1} = 1) \lor (s[N-1] \leq 0 \land ret_{N-1} = 0)] \land$$
$$(ret_0 + ret_1 + \ldots + ret_{N-1} = 3)$$
handling complex mathematical constraints

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

\[
\sqrt{\left(\left(\left(\left(1.0 \cdot \left(\left(1.0 \cdot \left(1.0 - \cos(c_1 \cdot x_5)\right)\right)\right)\right)\right) - \left(c_2/c_2\right) \cdot \left(1.0 - \cos(c_1 \cdot x_5)\right)\right)\right)^2} > 999.0 \land (c_1 \cdot x_5) > 0.0 \land x_3 > 0.0 \land x_6 > 0.0 \land c_1 = 0.017\ldots \land c_2 = 68443.0 \land e_1 = \left(\frac{\sqrt{x_2^2}}{\tan(c_1 \cdot x_3)}\right) / c_2 \land e_2 = \frac{\sqrt{x_6^2}}{\tan(c_1 \cdot x_3)}
\]
target application of solver: programs that
- use floating-point arithmetic
- call math functions

input:  \(\sqrt{\text{pow}((x1 + (e1 * \cos(x4)) - \ldots \right)

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.
another method for constraint solving

input: $\sqrt{\text{pow}(((x1 + (e1 \times (\cos(x4) - \ldots \text{\_interval})}

output: $\{x1=[99.9\ldots, 100.0], x2=[99.9\ldots, 100.0], \ldots\}, \ldots$

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)
publicly available applications from the aerospace domain

<table>
<thead>
<tr>
<th>Subject</th>
<th># constraints</th>
<th># conjuncts</th>
<th># functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apollo Autopilot</td>
<td>800</td>
<td>39</td>
<td>3</td>
</tr>
<tr>
<td>Collision Detection (CDx)</td>
<td>800</td>
<td>63</td>
<td>6</td>
</tr>
<tr>
<td>Conflict Probe</td>
<td>33</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Turn Logic</td>
<td>329</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>

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

[Website]
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
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]
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
assume hash(x) = 10 * x:
PC: X > 3 ∧ Y > 10 ∧ Y = hash(X)

\[ \begin{align*}
\text{simplePC} \\
\text{complexPC}
\end{align*} \]

solve simplePC
use solution X = 4 to compute h(4) = 40
simplify complexPC: Y = 40
solve again
simplified PC: X > 3 ∧ Y > 10 ∧ Y = 40 satisfiable!
```c
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;
}
```

**symbolic execution**

- **PC: true**
- **PC: X>0**
- **PC: X<=0**
- **PC: X>0 & Y=hash(X) S0**
- **PC: X>0 & X<=3 & Y=hash(X) S4**
- **PC: X>3 & Y>10 & Y=hash(X) S3**
- **solve X>0**
  - get X=1
  - hash(1)=10
  - check X>0 & Y=hash(X) S0
- **solve X>3 & Y>10**
  - get X=4
  - hash(4)=40
  - check X>3 & Y>10 & Y=hash(X) S3
- **solve X>0**
  - get X=1
  - hash(1)=10
  - check X>0 & Y=hash(X) S0
potential for unsoundness

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

PC: \( x \geq 0 \) & \( x > y \) & \( y = x^2 \)  S0

simplePC \( x \geq 0 \) & \( x > y \)  \( x = 0, y = -1 \)

complexPC \( y = x^2 \)  \( y = 0 * 0 = 0 \)

simplified PC \( x \geq 0 \) & \( x > y \) & \( y = 0 \)

is sat which implies assert is reachable!

Must add constraints on the solutions back into simplified PC

\( x \geq 0 \) & \( x > y \) & \( y = 0 \) & \( x = 0 \)

not sat!

DART/Concolic will diverge instead
void test(int x, int y) {
    if (x > 0) {
        if (y == hash(x)) //hash(x)=10*x
            S0;
        else
            S1;
    } else
        S4;
}

Example

Mixed concrete-symbolic solving: all paths covered

EXE results: stmt "S3" not covered

DART results: path "S0;S4" not covered
running DART

```java
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;
}
```

divergence!
aimed to get S0;S4 but reached S1;S4
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
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

<table>
<thead>
<tr>
<th>Part 1</th>
<th>Part 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>introduction: symbolic execution</td>
<td>dynamic techniques</td>
</tr>
<tr>
<td>symbolic pathfinder: symbolic</td>
<td>the DART algorithm</td>
</tr>
<tr>
<td>execution for Java bytecode</td>
<td>concolic execution</td>
</tr>
<tr>
<td>input data structures</td>
<td></td>
</tr>
<tr>
<td>multi-threading</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part 3</th>
<th>Part 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>challenges</td>
<td>applications</td>
</tr>
<tr>
<td>solving complex constraints</td>
<td>current and future work</td>
</tr>
<tr>
<td>parallel and compositional</td>
<td></td>
</tr>
<tr>
<td>techniques</td>
<td></td>
</tr>
<tr>
<td>abstraction</td>
<td></td>
</tr>
<tr>
<td>symbolic execution with mixed</td>
<td></td>
</tr>
<tr>
<td>concrete-symbolic solving</td>
<td></td>
</tr>
</tbody>
</table>
testing the Onboard Abort Executive (OAE)

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

OAE Structure

- **Inputs**
- **Checks Flight Rules** to see if an abort must occur
- **Select Feasible Aborts**
- **Pick Highest Ranked Abort**

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]
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 && …
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]
**Rhapsody**

**IMPORT**

Simulink/Stateflow

Rhapsody

State machine model in Java

**EXPORT**

Symbolic PathFinder

Modeling / Intermediate Representation

Stateflow

UML

Rhapsody

Systematic Analysis

Constraint Solving

Sym Exe Tree

Test Sequences

Error Report

Pluggable Semantics

Generic Execution Environment

Data interface
simplified model of the arbiter module
Mars Exploration Rover

- 3 Statecharts: 1 server, 2 clients
- Server grants/denies/rescinds resources
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>
<thead>
<tr>
<th>Semantics, Seq. size</th>
<th>Total # Test Cases</th>
<th>Property</th>
<th>Memory, Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1 Stateflow, 4</td>
<td>125</td>
<td>true</td>
<td>20 M, 43 s</td>
</tr>
<tr>
<td>U1 Stateflow, 5</td>
<td>412</td>
<td>true</td>
<td>22 M, 2 m 04 s</td>
</tr>
<tr>
<td>U1 Stateflow, 6</td>
<td>1343</td>
<td>true</td>
<td>24 M, 6 m 46 s</td>
</tr>
<tr>
<td>U1 UML, 4</td>
<td>57</td>
<td>false</td>
<td>21 MB, 21 s</td>
</tr>
<tr>
<td>U1 UML, 5</td>
<td>155</td>
<td>false</td>
<td>21 MB, 53 s</td>
</tr>
<tr>
<td>U1 UML, 6</td>
<td>579</td>
<td>false</td>
<td>23 MB, 5 m 40 s</td>
</tr>
<tr>
<td>U1 Rhapsody, 4</td>
<td>57</td>
<td>false</td>
<td>21 MB, 21 s</td>
</tr>
<tr>
<td>U1 Rhapsody, 5</td>
<td>155</td>
<td>false</td>
<td>21 MB, 55 s</td>
</tr>
<tr>
<td>U1 Rhapsody, 6</td>
<td>579</td>
<td>false</td>
<td>23 MB, 2 m 45 s</td>
</tr>
</tbody>
</table>
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]
computes logical difference between two program versions
uses loop and method summaries
[Person et al. FSE’08, Person et al PLDI’11]
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]
```c
int compute (int curr, int thresh, int step) {
    int delta = 0;
    if(curr<thresh) {
        delta = thresh-curr;
        if((curr+step)<thresh)
            return -delta;
        else
            return 0;
    } else {
        int counter=0;
        while(curr>=thresh) {
            curr=curr-step;
            counter++;
        }
        return counter;
    }
}
```

symbolic execution tree

- `curr`: $S_1$, `thresh`: $S_2$, `step`: $S_3$
- Path condition PC: true

1. **[2]** `delta: 0`
2. **[3]** `delta: $S_2 - S_1$`
3. **[3]** `delta: $S_2 - S_1$`
4. **[4]** `delta: $S_2 - S_1$`
5. **[5]** `delta: $S_2 - S_1$`
6. **[6]** `delta: $S_2 - S_1$`
7. **[7]** `delta: $S_2 - S_1$`
8. **[8]** `delta: $S_2 - S_1$`
9. **[9]** `delta: $S_2 - S_1$`
10. **[10]** `delta: $S_2 - S_1$`
11. **[11]** `delta: $S_2 - S_1$`
12. **[12]** `delta: $S_2 - S_1$`
13. **[13]** `delta: $S_2 - S_1$`
14. **[14]** `delta: $S_2 - S_1$`
15. **[15]** `delta: $S_2 - S_1$`
16. **[16]** `delta: $S_2 - S_1$`
17. **[17]** `delta: $S_2 - S_1$`

Unsat!
symbolic execution tree

memoised tree
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 subsequent iterations
results – savings
- time (2x improvement)
- number of solver calls (up to 1000 less calls)
- 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
check 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 …
Method m:
1: \( d = d + 1 \);
2: if \((x > y)\)  
3: \( \text{return } d / (x-y) \);  
else
4: \( \text{return } d / (y-x) \);

Symbolic execution tree:

\[ x: X, y: Y, d: D \]
Path condition PC: true

1: \[ x: X, y: Y, d: D+1 \]
Path condition PC: true

2: PC: \( X > Y \)
\( \text{return: } \frac{(D+1)}{(X-Y)} \)

2: PC: \( X \leq Y \)
\( \text{return: } \frac{(D+1)}{(Y-X)} \)

3: PC: \( X > Y \)

4: PC: \( X \leq Y \) 
& \( Y-X \neq 0 \)
\( \text{return: } \frac{(D+1)}{(Y-X)} \)

4: PC: \( X \leq Y \) 
& \( Y-X = 0 \)
Division by zero!

Solve path conditions → test inputs
@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 ⇔ X = Y

full path coverage
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”]
pre-condition:
“x!=y”

post-condition:
“\texttt{\texttt{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:**

\[ \text{X} = \text{init}; \]
\[ \text{while } (C(\text{X})) \]
\[ \quad \text{X} = B(\text{X}); \]
\[ \text{assert } P(\text{X}); \]

**Program execution:**

- while ...
- true
- while ...
- true
- while ...
- true
- ...

*May be* infinite ...

- How to reason about infinite executions?

**Find loop invariant Inv**

**Non-looping program:**

\[ \text{X} = \text{init}; \]
\[ \text{assert Inv(}X\text{)}; \]
\[ \text{X} = \text{new symbolic values}; \]
\[ \text{assume Inv(}X\text{)}; \]
\[ \text{if } (C(\text{X})) \{ \]
\[ \quad \text{X} = B(\text{X}); \]
\[ \quad \text{assert Inv(}X\text{)}; \]
\[ \} \text{ else} \]
\[ \quad \text{assert P(}X\text{)}; \]

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

**Base Case**

**Induction Step**

**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
- 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/
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!