Symbolic Execution and Program Testing

JAMES C. KING
IBM THOMAS J. WASTON RESEARCH CENTER
PRESENTED BY: MENG WU
History of Symbolic Execution


• James C. King. Symbolic execution and program testing. CACM, 19(7):385–394, 1976. (most cited)


Symbolic execution and program testing

Full Text: PDF

Author: James C. King IBM Thomas J. Watson Research Center, Yorktown Heights, NY

Published in:
- Magazine: Communications of the ACM CACM Homepage archive
  Volume 19 Issue 7, July 1976
  Pages 385-394
  ACM New York, NY, USA
  table of contents doi>10.1145/360248.360252

1976 Article

Bibliometrics
- Downloads (6 Weeks): 64
- Downloads (12 Months): 415
- Downloads (cumulative): 6,715
- Citation Count: 528
Problems in Program Testing

Predicates

Program

Assertion

Req1: enumerate all possible input values

Req2: explore all feasible paths
Problems in Program Testing

Predicates

Program

Assertion

Req1: enumerate all possible input values

Req2: explore all feasible paths
Problems in Program Testing

Predicates

Program

Assertion

Req1: enumerate all possible input values
Symbolic Execution

Req2: explore all feasible paths
BMC or Abstraction
Main Ideas

- Generalize testing by using unknown symbolic variables in evaluation
- Update a symbolic state formula after each statement
- Check the path constraints/conditions
Main Ideas

Insights:

- ‘Execute’ programs with symbols: track symbolic state rather than concrete input
- ‘Execute’ many program paths simultaneously: when execution path diverges, fork and add constraints on symbolic values
- When ‘execute’ one path, we actually simulate many test runs, since we are considering all the inputs that can exercise the same path
Example

Consider the program below, which reads in a value and fails if the input is 6.

```
y = read()
y = 2 \times y
if (y == 12)
    fail()
print("OK")
```

- Manual test creation: build test with input 6
- Auto-Test?
  - y is 32-bit integer
  - How many test inputs for full coverage? $2^{32}$
Example

Consider the program below, which reads in a value and fails if the input is 6.

```plaintext
y = read()
y = 2 * y
if (y == 12)
    fail()
    print("OK")
```

- **y** is symbolic: \( y = s \)
- \( y = 2 \times s \) // still symbolic
- Fork execution, add constraints to each path
- **true** path constraint: \( 2s == 12 \)
- Need constraint solver
That`s all you need to know!
• Definition: execution state
  • Line number
  • values of variables (symbolic/concrete): \(x = s_1\), \(y = s_2 + 3s_4\)
  • Path Condition (PC): conjunction of constraints (boolean formulas) over symbols:
    \(s_1 > 0 \land \alpha_1 + 2s_2 > 0 \land \neg(s_3 > 0)\)
More Details

• Execute assignment: evaluate RHS symbolically, assign to LHS as part of the state.

• Execute IF (r) / then / else: fork
  • then: PC ← PC ∧ r
  • else: PC ← PC ∧ ¬r

• Termination: solve constraint
```c
1 int y;
2 int z;
3 ...
4 int foo(int x) {
5   if (x > 0) {
6     y = y + x;
7   } else {
8     y = y - x;
9   }
10  if (x > 0) {
11    z = z - y;
12  } else {
13    z = z + y
14  }
15 }
```
Execution tree properties

- For each satisfiable leaf exists a concrete input for which the real program will reach same leaf ⇒ can generate test

**Comutativity**

- PC's associated with any two *satisfiable* leaves are distinct ⇒ code coverage.
Applications

1. \texttt{int a = \alpha, b = \beta, c = \gamma;}
2.       \texttt{// symbolic}
3. \texttt{int x = 0, y = 0, z = 0;}
4. \texttt{if (a) {}
5.     \texttt{x = -2;}
6. }\texttt{}
7. \texttt{if (b < 5) {}
8.     \texttt{if (!a && c) { y = 1; }}
9. \texttt{z = 2;}
10. }\texttt{}
11. \texttt{assert(x+y+z!=3)\hfill \text{path condition}}
Detecting Infeasible Paths

1. int a = α, b = β, c = γ;
2. // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.    x = -2;
6. }
7. if (b < 5) {
8.    if (!a && c) { y = 1; }
9.    z = 2;
10.}
11. assert(x+y+z!=3)

Suppose we require $\alpha = \beta$

Infeasible!
Test Input Generation

1. int a = α, b = β, c = γ;
2.     // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.     x = -2;
6. }
7. if (b < 5) {
8.     if (!a && c) { y = 1; }
9.     z = 2;
10.}
11. assert(x+y+z!=3)

Given Path Condition to Constrain Solver, it will produce test input for each path:

Path 1: α = 1, β = 1
Path 2: α = 1, β = 6
Path 3 ...

path condition
Bug Finding

```c
int foo(int i) {
    int j = 2*i;
    i = i++;
    i = i * j;
    if (i < 1)
        i = -i;
    i = j/i;
    return i;
}
```

\(i_{\text{input}} = -1\) Trigger the bug

**True branch:**
\[
2 \times i_{\text{input}} \mathbin{^2} + 2 \times i_{\text{input}} < 1
\]
\[
i = -2 \times i_{\text{input}} \mathbin{^2} - 2 \times i_{\text{input}}
\]
\[i == 0\]

**False branch: always safe**
\[
2 \times i_{\text{input}} \mathbin{^2} + 2 \times i_{\text{input}} \geq 1
\]
\[
i = 2 \times i_{\text{input}} \mathbin{^2} + 2 \times i_{\text{input}}
\]
\[i == 0\]
A Simple Symbolic Executor: EFFIGY

- Integer Value only
- IF, THEN, ELSE, DO, GO-TO, DO WHILE
- Basic Operators
- State Saving and Restore
- Completely User Guided Execution
# Modern Tools

## Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>It can analyze Arch/Lang</th>
<th>url</th>
<th>Can anybody use it?</th>
<th>Open source?</th>
<th>Downloadable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLEE</td>
<td>LLVM</td>
<td><a href="https://klee.github.io/">https://klee.github.io/</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FuzzBALL</td>
<td>VineaL/native</td>
<td><a href="http://bitblaze.cs.berkeley.edu/fuzzball.html">http://bitblaze.cs.berkeley.edu/fuzzball.html</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPF</td>
<td>java</td>
<td><a href="http://babelfish.arc.nasa.gov/trac/jpf">http://babelfish.arc.nasa.gov/trac/jpf</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jCUTE</td>
<td>java</td>
<td><a href="https://github.com/osl/jcute">https://github.com/osl/jcute</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>janaa2</td>
<td>java</td>
<td><a href="https://github.com/lksen007/janaa2">https://github.com/lksen007/janaa2</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeY</td>
<td>java</td>
<td><a href="http://www.key-project.org/">http://www.key-project.org/</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2E</td>
<td>llvm/qemu/x86</td>
<td><a href="http://dslab.epfl.ch/proj/s2e">http://dslab.epfl.ch/proj/s2e</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mayhem</td>
<td>binary</td>
<td><a href="http://foralisecure.com/mayhem.html">http://foralisecure.com/mayhem.html</a></td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Otter</td>
<td>C</td>
<td><a href="https://bitbucket.org/khoops/otter/overview">https://bitbucket.org/khoops/otter/overview</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SymDroid</td>
<td>Dalvik bytecode</td>
<td><a href="http://www.cs.umd.edu/~jfoster/papers/symdroid.pdf">http://www.cs.umd.edu/~jfoster/papers/symdroid.pdf</a></td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jalangi</td>
<td>JavaScript</td>
<td><a href="https://github.com/SRA-SiliconValley/jalangi">https://github.com/SRA-SiliconValley/jalangi</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kite</td>
<td>llvm</td>
<td><a href="http://www.cs.ubc.ca/labs/isl/Projects/Kite">http://www.cs.ubc.ca/labs/isl/Projects/Kite</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pysymenu</td>
<td>amd64/native</td>
<td><a href="https://github.com/feliam/pysymenu">https://github.com/feliam/pysymenu</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triton</td>
<td>x86-64</td>
<td><a href="http://triton.quarkslab.com">http://triton.quarkslab.com</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>angr</td>
<td>libVEX based</td>
<td><a href="http://angr.io">http://angr.io</a></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problems And Later Research

- Path Explosion (IF, Loops)
  - Search Strategy: Random Search, Coverage Guided Search
  - Concolic (concrete & symbolic) Testing
- Constrain Solving
  - Powerful SAT/SMT solver: Z3, STP, Yices
  - Non-linear Constraints, Float-point constrains, Quantifiers, Disjunction
- Memory Modeling
  - KLEE: Open source symbolic executor; Runs on top of LLVM
- Handling Concurrency
Thanks & Questions?

Reference:
Symbolic Execution for Software Testing: Three Decades Later”, CACM, Feb 2013, p 82-90

https://www.cs.umd.edu/class/fall2011/cmsc631/
http://www.seas.harvard.edu/courses/cs252/2011sp
https://en.wikipedia.org/wiki/Symbolic_execution