Symbolic Execution and Program Testing

JAMES C.KING

IBM THOMAS J.WASTON RESEARCH CENTER

PRESENTED BY: MENG WU

History of Symbolic Execution

- Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. In ICRS, pages 234– 245, 1975.
- James C. King. Symbolic execution and program testing. CACM, 19(7):385–394, 1976. (most cited)
- Leon J. Osterweil and Lloyd D. Fosdick. Program testing techniques using simulated execution. In ANSS, pages 171–177, 1976.
- William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. IEEE Transactions on Software Engineering, 3(4):266–278, 1977.

About the Paper

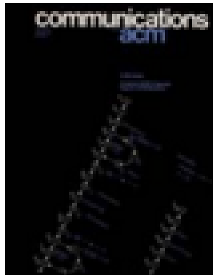
Symbolic execution and program testing

Full Text: PDF

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

1976 Article

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

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 constrains/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

Example [edit]

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

```
y = read()
y = 2 * 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

Example [edit]

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

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

y is symbolic: y = s

y = 2 * s // still symbolic

Fork execution, add constraints to each path

***true* path constraint:**

2*s==12

Need constraint solver

That`s all you need to know!

More Details

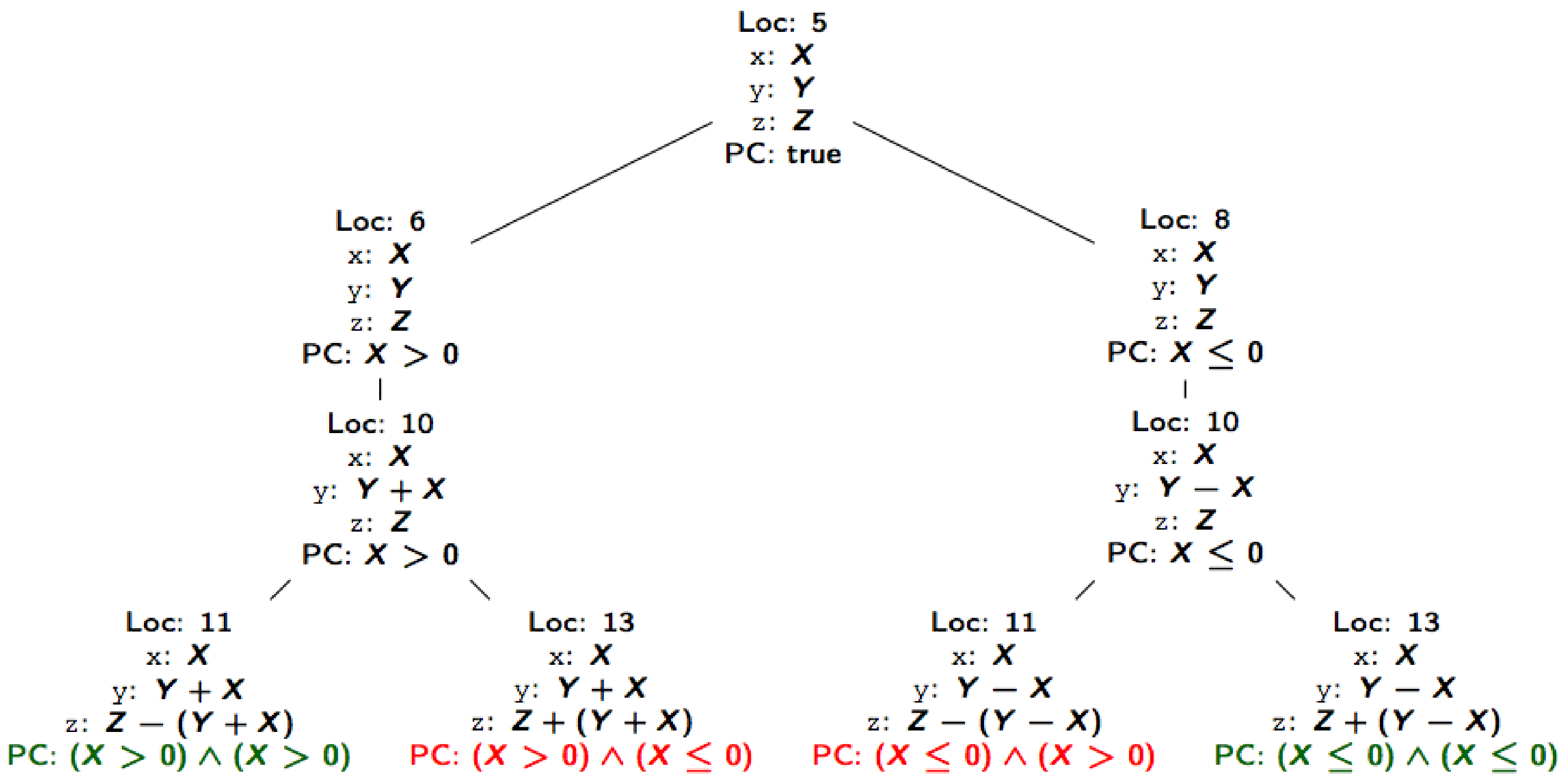
- Definition: execution state
  - Line number
  - values of variables (symbolic/concrete): $x=s_1$, $y=s_2+3*s_4$
  - Path Condition (PC): conjunction of constraints (boolean formulas) over symbols:
    $s_1>0 \land \alpha_1+2*s_2>0 \land \neg(s_3>0)$

More Details

- Execute assignment: evaluate RHS symbolically, assign to LHS as part of the the state.
- Execute IF (r) / then / else: fork
    - then: $PC \longleftarrow PC \wedge r$
    - else: $PC \longleftarrow PC \wedge \neg r$
- Termination: solve constraint

Execution tree

```
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

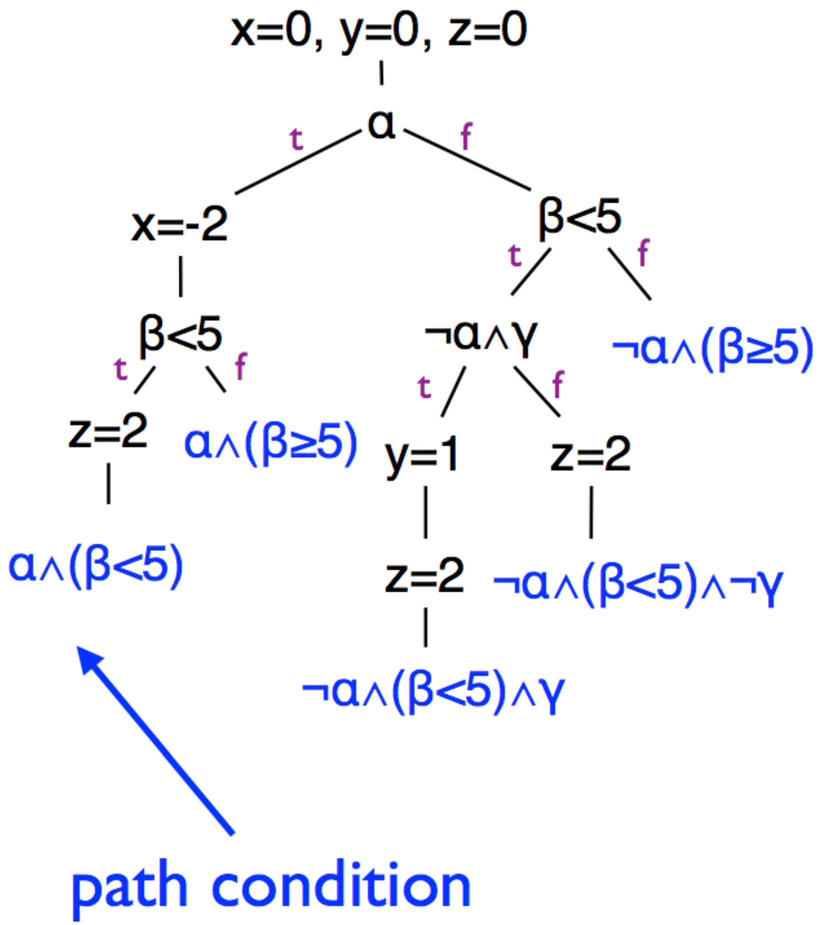
- For each satisfiable leaf exists a concrete input for which the real program will reach same leaf ⇒ can generate test

Comutativity

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

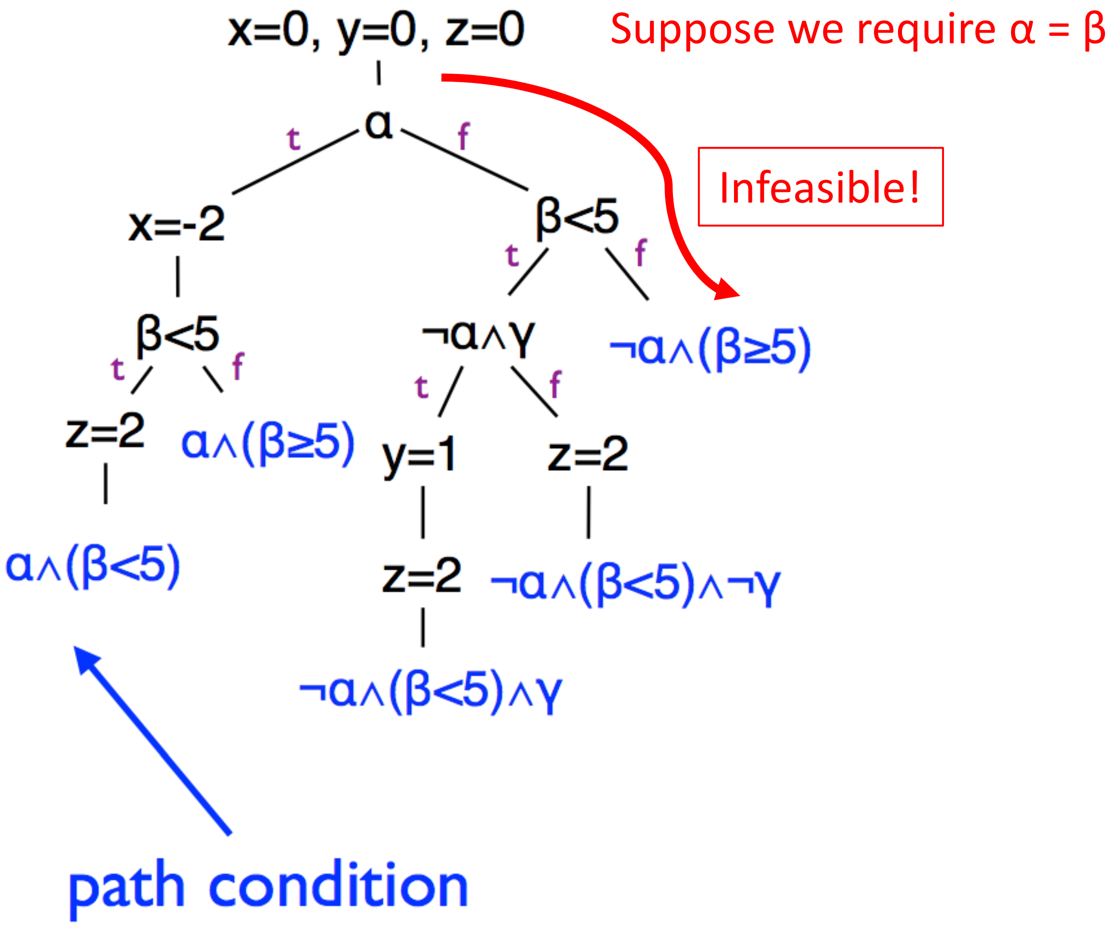
Applications

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



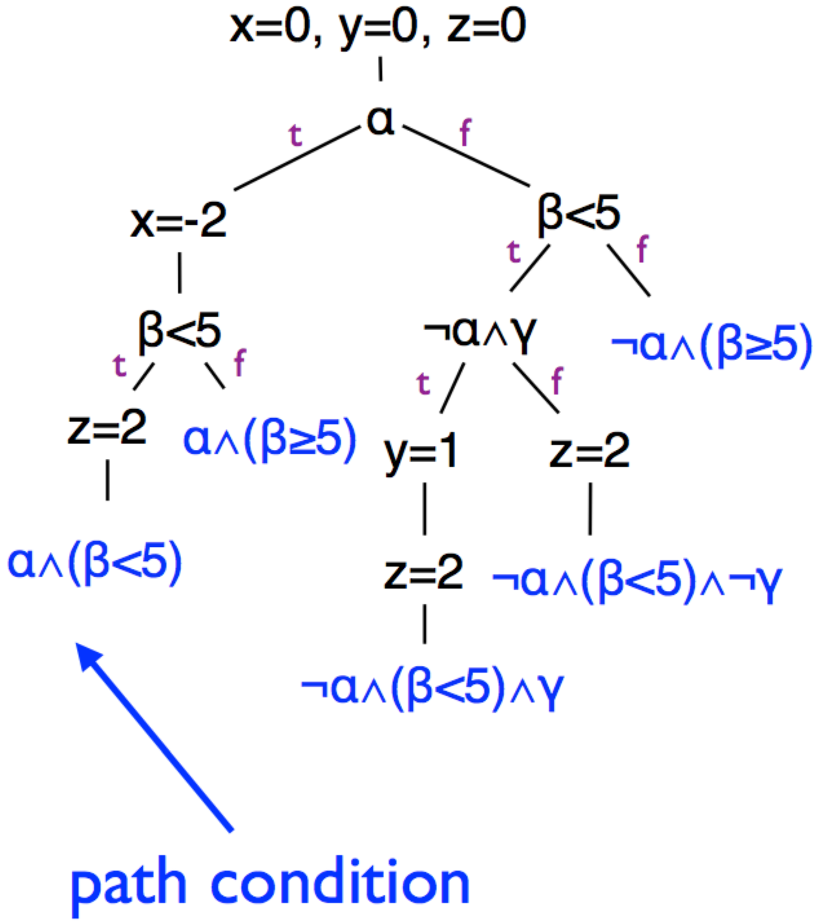
Detecting Infeasible Paths

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Test Input Generation

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



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

Path 1: $\alpha = 1, \beta = 1$
Path 2: $\alpha = 1, \beta = 6$
Path 3 ...

Bug Finding

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

i_{input} **i_{input} = -1 Trigger the bug**

True branch:

$2* i_{input}$ ^2 + $2* i_{input}$ < 1

i = - $2* i_{input}$ ^2 - $2* i_{input}$

i == 0

False Branch: **always safe**

$2* i_{input}$ ^2 + $2* i_{input}$ >= 1

i = $2* i_{input}$ ^2 + $2* i_{input}$

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 [edit]

Tool	It can analyze Arch/Lang	url	Can anybody use it/ Open source/ Downloadable
KLEE	LLVM	http://klee.github.io/	yes
FuzzBALL	VineIL/native	http://bitblaze.cs.berkeley.edu/fuzzball.html	yes
JPF	java	http://babelfish.arc.nasa.gov/trac/jpf	yes
jCUTE	java	https://github.com/osl/jcute	yes
janala2	java	https://github.com/ksen007/janala2	yes
KeY	java	http://www.key-project.org/	yes
S2E	llvm/qemu/x86	http://dslab.epfl.ch/proj/s2e	yes
Pathgrind[4]	native 32bit valgrind based	https://github.com/codelion/pathgrind	yes
Mayhem	binary	http://forallsecure.com/mayhem.html	no
Otter	C	https://bitbucket.org/khooyp/otter/overview	yes
SymDroid	Dalvik bytecode	http://www.cs.umd.edu/~jfoster/papers/symdroid.pdf	no
Rubyx	Ruby	http://www.cs.umd.edu/~avik/papers/ssarorwa.pdf	no
Pex	.NET Framework	http://research.microsoft.com/en-us/projects/pex/	no
Jalangi	JavaScript	https://github.com/SRA-SiliconValley/jalangi	yes
Kite	llvm	http://www.cs.ubc.ca/labs/isd/Projects/Kite/	yes
pysymemu	amd64/native	https://github.com/feliam/pysymemu/	yes
Triton	x86-64	http://triton.quarkslab.com	yes
angr	libVEX based	http://angr.io/	yes

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-liner Constrains, 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