Symbolic Execution

Testing, Quality Assurance, and Maintenance
Winter 2017

Prof. Arie Gurfinkel

based on slides by Prof. Johannes Kinder and others
Symbolic Execution

Automatically explore program paths
- Execute program on “symbolic” input values
- “Fork” execution at each branch
- Record branching conditions

Constraint solver
- Decides path feasibility
- Generates test cases for paths and bugs
History

Int. Conference on Reliable Software 1975

James C. King:
A new approach to program testing

Robert S. Boyer, Bernard Elspas, Karl N. Levitt:
SELECT—a formal system for testing and debugging programs
by symbolic execution

Recent work on proving the correctness of programs
by formal analysis [5] shows great promise and appears
to be the ultimate technique for producing reliable pro-
grams. However, the practical accomplishments in this
area fall short of a tool for routine use. Fundamental
problems in reducing the theory to practice are not
likely to be solved in the immediate future.
History (2)

SAT / SMT solvers lead to boom in 2000s

- Constraint solving becomes a commodity
- Makes classic algorithms viable in practice

Conceptual breakthroughs (Dynamic Symbolic Execution)

- Patrice Godefroid, Nils Klarlund, Koushik Sen: *DART: directed automated random testing*. PLDI 2005
int Max(int a, int b, int c, int d) {
    return Max(Max(a, b), Max(c, d));
}

int Max(int x, int y) {
    if (x <= y) return y;
    else return x;
}
Checking Path Feasibility

```prolog
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)
(declare-fun d () Int)
(assert (< 0 a))
(assert (< 0 b))
(assert (< 0 c))
(assert (< 0 d))
(assert (< a b))
(assert (> b c))
(assert (> c d))
(check-sat)
(get-model)
```

at
model
(define-fun b () Int
  3)
(define-fun a () Int
  1)
(define-fun c () Int
  2)
(define-fun d () Int
  1)
```c
int proc(int x) {
    int r = 0
    if (x > 8) {
        r = x - 7
    }
    if (x < 5) {
        r = x - 2
    }
    return r
}
```

Satisfying assignments:
- X = 9
- X = 4
- X = 7

Test cases:
- proc(9)
- proc(4)
- proc(7)
Symbolic Execution

Analysis of programs by tracking symbolic rather than actual values

- a form of Static Analysis

Symbolic reasoning is used to reason about all the inputs that take the same path through a program

Builds constraints that characterize

- conditions for executing paths
- effects of the execution on program state
Symbolic Execution

Uses symbolic values for input variables.

Builds constraints that characterize the conditions under which execution paths can be taken.

Collects **symbolic path conditions**

- a path condition for a path P is a formula PC such that PC is satisfiable if and only if P is executable

Uses theorem prover (**constraint solver**) to check if a path condition is satisfiable and the path can be taken.
Symbolic State

A **symbolic state** is a pair $S = (\text{Env}, \text{PC})$, where

- $\text{Env} : L \rightarrow E$ is a mapping, called an **environment**, from program variables to symbolic expressions (i.e., FOL terms)
- $\text{PC}$ is a FOL formula called a **path condition**

A concrete state $M : L \rightarrow Z$ satisfies a symbolic state $S = (\text{Env}, \text{PC})$ iff

$$M \models (\text{Env}, \text{PC}) \text{ iff } \left( \bigwedge_{v \in L} M(v) = \text{Env}(v) \right) \land \text{PC is SAT}$$

Program semantics are extended to symbolic states

- each program statement updates symbolic variables and
- extends the path condition to reflect its operational semantics
Example: Symbolic State Satisfiability

\[ Env = \begin{cases} 
  x & \mapsto X \\
  y & \mapsto Y 
\end{cases} \quad PC = X > 5 \land Y < 3 \]

\[
[x \mapsto 10, y \mapsto 1] \models \neg S \quad [x \mapsto 1, y \mapsto 10] \models \neg S
\]

\[ Env = \begin{cases} 
  x & \mapsto X + Y \\
  y & \mapsto Y - X 
\end{cases} \quad PC = 2 \times X - Y > 0 \]

\[
[x \mapsto 10, y \mapsto 1] \models \neg S \quad [x \mapsto 1, y \mapsto 10] \models \neg S
\]
Symbolic Evaluation/Execution

Symbolic execution creates a functional representation of a path in a Control Flow Graph of a program

For a path $P_i$

- $D[P_i]$ is the domain for path $P_i$
  - the inputs that force the program to take path $P_i$

- $C[P_i]$ is the computation for path $P_i$
  - the result of executing the path
Functional Representation of an Executable Component

$P : X \rightarrow Y$

$P$ is composed of partial functions corresponding to the executable paths

$P = \{P_1, \ldots, P_r\}$

$P_i : X_i \rightarrow Y$
Functional Representation of an Executable Component

$X_i$ is the domain of path $P_i$

Denoted $D[P_i]$ 

$X = D[P_1] \cup \ldots \cup D[P_r] = D[P]$

$D[P_i] \cap D[P_j] = \emptyset, \ i \neq j$
Exercise: Find a Violation

```c
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);
```

\[ a = \alpha, \ b = \beta, \ c = \gamma \]
\[ x=0, \ y=0, \ z=0 \]
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);
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);

a = α, b = β, c = γ
x=0, y=0, z=0
x=-2
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);
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);
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);
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);

a = α, b = β, c = γ

path condition
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);

a = α, b = β, c = γ
x=0, y=0, z=0

path condition
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);

a = α, b = β, c = γ

path condition
```c
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);
```
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);
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);
```c
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);
```
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);
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);

\[ a = \alpha, \ b = \beta, \ c = \gamma \]
\[ x=0, \ y=0, \ z=0 \]

\[
\text{path condition}
\]

a = false, b = 2, c = true -> x = 0, y = 1, z = 2 : Assert(0+1+2 != 3)
Finding Bugs

Symbolic execution enumerates paths

- Runs into bugs that trigger whenever path executes
- Assertions, buffer overflows, division by zero, etc., require specific conditions

Error conditions

- Treat assertions as conditions
- Creates explicit error paths

```c
assert x != NULL
if (x == NULL)
    abort();
```
Finding Bugs

Instrument program with properties
  • Translate any safety property to reachability

Division by zero
\[ y = \frac{100}{x} \quad \Rightarrow \quad \text{assert } x \neq 0 \quad y = \frac{100}{x} \]

Buffer overflows
\[ a[x] = 10 \quad \Rightarrow \quad \text{assert } x \geq 0 \quad \&\& \quad x < \text{len}(a) \]

Implementation is usually implicit
Many problems remain

Code that is hard to analyze

Path explosion

• Complex control flow
• Loops
• Procedures

Environment (what are the inputs to the program under test?)

• pointers, data structures, …
• files, data bases, …
• threads, thread schedules, …
• sockets, …