Research Topics in Software Quality

Automated Program Testing using Static and Dynamic Symbolic Execution
Acknowledgements

These notes draw from:
- Talk by Koushik Sen on “Dynamic Symbolic Execution”
Methods for reducing and identifying faults

Safer Programming Languages
(e.g., automatic memory management)

Type Systems

Static Program Analysis
(e.g., compute at compile-time all possible values of a variable that should never be null)

Model Checking
(e.g., prove reachability of automaton state)

Program Verification

Dynamic Program Analysis
(e.g., log run-time values of variable during an execution)

Run-time Monitoring

Testing

Model-driven Engineering
(e.g., generate embedded software from state diagram)
Generating Tests

Given a program, can we automatically generate a set of inputs that, upon execution, will achieve a coverage target within a particular execution budget?

Test for uncaught run-time exceptions and violations of developer-specified assertions in the source code.
Static Symbolic Execution

Evaluate the program under test on symbols that substitute for user inputs.

At each decision point, fork evaluation and accumulate condition==true or condition==false in path condition.

For test to reach particular program point, obtain program input by solving path condition for concrete values for symbols.

The large-scale production of reliable programs is one of the fundamental requirements for applying computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements even when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a program is expected to handle is presented to the program. If the program is judged to produce correct results for the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

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 programs. 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.

Program testing and program proving can be considered as extreme alternatives. While testing, a programmer can be assured that sample test runs work correctly by carefully checking the results. The correct execution for inputs not in the sample is still in doubt. Alternatively, in program proving the programmer formally proves that the program meets its specification for all executions without being required to execute the program at all. To do this he gives a precise specification of the correct program behavior and then follows a formal proof procedure to show that the program and the specification are consistent. The confidence in this method hinges on the care and accuracy employed in both the creation of the specification and in the construction of the proof steps, as well as on the attention to machine-dependent issues such as overflow, rounding, etc.

This paper describes a practical approach between these two extremes. From one simple view, it is an enhanced testing technique. Instead of executing a program on a set of sample inputs, a program is "symbolically" executed for a set of classes of inputs. That is, each symbolic execution result may be equivalent to a large number of normal test cases. These results can be checked against the programmer's expectations for correctness either formally or informally.

The class of inputs characterized by each symbolic execution is determined by the dependence of the program's control flow on its inputs. If the control flow of the program is completely independent of the input variables, a single symbolic execution will suffice to check all possible executions of the program. If the control flow of the program is dependent on the inputs, one must resort to a case analysis. Often the set of input
Symbolic Store

Map from variable to symbolic expression over input symbols.

\[ \sigma_s \in SymStore : Var \rightarrow Sym \]

Expressions evaluate to symbolic expression rather than value.

\[ \sigma_s : x \mapsto x_0, y \mapsto y_0 \]

\[ z = x + y \]

\[ \sigma_{s'} : x \mapsto x_0, y \mapsto y_0, z \mapsto x_0 + y_0 \]

Path Condition

Records history of branches taken at decision points.

\[ PC_O : true \]

Symbolic execution forked at conditionals, chosen branch condition conjoined to PC.

3. ...

4. if(z > y + 1)
   4a. { ... }
   4b. else { ... }

5. ...

\[ PC_{4a} : true \land \ldots \land (z_0 > y_0 + 1) \]

\[ PC_{4b} : true \land \ldots \land \neg(z_0 > y_0 + 1) \]

Corresponds to path through so-called computation tree.
Example static symbolic execution

```c
1 int twice(int v) {
2     return 2*v;
3 }
4
5 void testme(int x, int y) {
6     z = twice(y);
7     if(z==x) {
8         if(x>y+10) {
9             ERROR;
10        }
11     }
12 }
13
14 int main() {
15     x=sym_input();
16     y=sym_input();
17     testme(x,y);
18     return 0;
19 }
```

\[
\sigma_s : \{ x \mapsto x_0, \quad y \mapsto y_0 \}
\]

\[
PC : \text{true}
\]
Example static symbolic execution

```c
int twice(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = twice(y);
    if(z==x) {
        if(x>y+10) {
            ERROR;
        }
    }
}

int main() {
    x=sym_input();
    y=sym_input();
    testme(x,y);
    return 0;
}
```

assignments update symbolic store with symbolic expression

\[ \sigma_s : \{ x \mapsto x_o, \quad y \mapsto y_0, \quad z \mapsto 2 \ast y_0 \} \]

\[ PC : true \]
Example static symbolic execution

1   int twice(int v) {
2       return 2*v;
3   }
4   void testme(int x, int y) {
5       z = twice(y);
6       if(z==x) {
7           if(x>y+10) {
8               ERROR;
9           }
10          }
11     }
12   int main() {
13       x=sym_input();
14       y=sym_input();
15       testme(x,y);
16       return 0;
17   }

\[
\begin{align*}
\sigma_s : & \{ x \mapsto x_0, \\
y \mapsto y_0, \\
z \mapsto 2 \times y_0 \}
\end{align*}
\]

PC : true \land 2 \times y_0 = x_0

fork execution at conditional, record chosen branch in each path condition
# Example static symbolic execution

```c
int twice(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = twice(y);
    if(z==x) {
        if(x>y+10) {
            ERROR;
        }
    }
}

int main() {
    x=sym_input();
    y=sym_input();
    testme(x,y);
    return 0;
}
```

---

**Found a symbolic state that corresponds to an error**

- **Along true branch:**
  - $\sigma_s : \{ x \mapsto x_o, \ y \mapsto y_0, \ z \mapsto 2*y_0 \}$
  - $PC : true \land 2*y_0 = x_o \land x_o > y_0 + 10$

- **Along false branch:**
  - $\sigma_s : \{ x \mapsto x_o, \ y \mapsto y_0, \ z \mapsto 2*y_0 \}$
  - $PC : true \land 2*y_0 = x_o \land x_o \leq y_0 + 10$
Solving path conditions

Each PC determines to a specific path through the program's execution tree.

Such a path corresponds to multiple possible concrete program runs: those on the input values satisfying the PC.

Off-the-shelf constraint solvers can return a variable assignment (out of many possible) that satisfies the PC, or determine that the PC is not satisfiable.

\[
\begin{align*}
&x_0 = 0, \\
&y_0 = 1,
\end{align*}
\]

\[\sigma_s : \{ x \mapsto x_o, \\
y \mapsto y_0, \\
z \mapsto 2 \cdot y_0 \} \]

\[PC : true \land 2 \cdot y_0 \neq x_o\]

\[
\begin{align*}
&x_0 = 2, \\
&y_0 = 1,
\end{align*}
\]

\[\sigma_s : \{ x \mapsto x_o, \\
y \mapsto y_0, \\
z \mapsto 2 \cdot y_0 \} \]

\[PC : true \land 2 \cdot y_0 = x_o \land x_o \leq y_0 + 10\]

\[
\begin{align*}
&x_0 = 30, \\
&y_0 = 15,
\end{align*}
\]

\[\sigma_s : \{ x \mapsto x_o, \\
y \mapsto y_0, \\
z \mapsto 2 \cdot y_0 \} \]

\[PC' : true \land 2 \cdot y_0 = x_o \land x_o > y_0 + 10\]
Symbolic Execution in a Nutshell

symbolic program state
- expression: current expression being evaluated (or instruction counter)
- store: map from variable to expression over symbolic inputs
- path condition: logic formula (constraints) over symbolic expressions
  - correspond to branches taken at decision points (e.g., conditionals)

symbolic evaluation
- assignment: update store
- expression: evaluates to symbolic expression rather than a value
- decision points:
  - “fork” evaluation along each branch
  - expand path condition with new constraint for chosen branch

solve path condition in interesting program states
- at end of program, upon violation of assertion, or uncaught run-time error
  - if satisfiable, constraint solver assigns values to symbolic inputs
  - concrete execution with those values will end up in same program state

suffices to duplicate symbolic program state, and add its successors to a work list (i.e., a depth/breadth/priority-first search)
Exhaustive Path Exploration: Infeasible Paths

branches for which the PC cannot be satisfied, need not be explored
no variable assignment will ever lead the program’s execution along such a path
can significantly reduce search space

e.g., path 1, 2, 6, 7

1 if(x > 10) {
2   …
3 } else {
4   …
5 }
6 if(x <= 10) {
7   …
8 } else {
9   …
10 }
Exhaustive Path Exploration: Infinite Feasible Paths

infinite number of feasible execution paths possible when termination condition is symbolic

```
void testme_inf() {
    int sum = 0;
    int N = sym_input();
    while(N>0) {
        sum = sum + N;
        N = sym_input();
    }
}
```

in practice: limit the search of symbolic execution e.g., limit loop iterations explored

limiting search also work around for the problem of path explosion 3 conds, 8 paths

#paths is usually exponential in #conditionals e.g., stop after timeout, limit number of explored paths, limit exploration depth
Directed, Limited Path Exploration: Heuristics

BUT limiting search may cause defects to be missed (i.e., sse is incomplete!)
critical to explore most relevant paths first, within fixed cost budget
depth/breadth-first search unaware of system under test
≠ random input generation (which has infinitely low probability to reach deep program parts!)

random search strategies
a) pick next state to explore at random from work list
b) randomly restart search from the beginning if no defects have been found for a while
c) randomly select one of two feasible paths at branches

coverage-guided search strategies
optimize #statements or #branches covered
count hits and prioritize lowest scores

generational search strategy
search generation n:
  explore a single path, collect PC
search generation n+1:
  negate one condition of previous PC
to steer search to unexplored program part

+ combinations of the above
Limitations of Static Symbolic Execution: Solving SMT formulas

SMT (Satisfaction Modulo Theories) formulas
- traditional logic formula in conjunctive normal form (k-SAT)
  - e.g., \((a \lor \neg b \lor c) \land (\neg b \lor \neg x \lor y) \land (b \lor b \lor x)\)
- + theory-specific operators
  - theory of linear, real arithmetic: \(x \leq 5, y \neq z\)
  - theory of uninterpreted functions: \(f(x) = f(y) \land f(f(x)) \neq f(f(y))\)
  - theory of bit vectors, arrays, lists, strings:
    - \(s.equals(q) \land s.startswith("uvw")\)
    - \(\land q.endswith("xyz") \land s.length() < a \land (a+b) < 6 \land b > 0\)
    - ....

Size of PC grows with exploration depth
- SMT solver calls might quickly exhaust execution budget:
  - determining feasibility of paths
  - determining truth of developer-specified assertions
  - generating input satisfying a path condition

3-SAT first known NP-complete problem!
SMT-LIB: shared interface to multiple SMT solvers

http://www.smtlib.org/

(declare-const a Bool)
(declare-const b Bool)
(define-fun demorgan () Bool
   (= (and a b) (not (or (not a) (not b)))))
(assert (not demorgan))
(check-sat)

ask z3

unsat

http://rise4fun.com/z3/tutorial/guide

Online Z3 solver + tutorial by Microsoft
Limitations of Static Symbolic Execution: Solving SMT formulas

symbolic execution engine should use domain knowledge to optimize successive calls to solver

eliminate irrelevant variables from formula
  e.g., when checking satisfiability of branch condition, only include variables it depends on transitively (careful with aliasing)

cache solutions to formulas

\[(x + y < 10) \land (x > 5) \iff \{x = 6, y = 3\}\]

cached solution can be reused for weaker formulas, without calling the solver
\[(x + y < 10)\]

cached solutions can be tested on stronger formulas, call solver if solution is not satisfied
\[(x + y < 10) \land (x > 5) \land (y \geq 0)\]
Limitations of Static Symbolic Execution:
Edges of the Application Code

```c
int obscure(int x, int y) {
    if (x == hash(y))
        return -1;
    return 0;
}
```

might encounter constraints that SMT solver cannot handle

probably encounter library, operating system, remote procedure calls

- attempt to symbolically execute callee,
  but code might not be available (e.g., 3rd-party service) or very complex (e.g., libc)
- symbolically execute simpler code (e.g., standard C library for embedded systems), older JRE
  but might not be compatible with the system under test
- symbolically execute a model of such code (e.g., model symbolic state returned by each
  procedure for a given symbolic input state), but defining a truthful model is prohibitively
  expensive (e.g., model a file system!)
Dynamic Symbolic Execution

Also known as Concolic = Concrete + Symbolic execution.

Program runs as usual, but has been modified (i.e., instrumented with additional instructions) to maintain symbolic state in the background.

Does require concrete values for initial input (e.g., chosen randomly).

Generational search: explore one path in one run, path condition of previous run is used to generate variant of previous inputs for next run.

When constraint in PC cannot be solved, replace symbolic expression by the observed concrete run-time value!
Example Dynamic Symbolic Execution

```c
int double(int v) {
    return 2 * v;
}

void testme(int x, int y) {
    z = double(y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```

- **Concrete Store**
  - `x = 22, y = 7`

- **Symbolic Store**
  - `x = x_0, y = y_0`

- **Path Condition**
Example Dynamic Symbolic Execution

```c
int double(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = double(y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```

- **Concrete Store**: $x = 22$, $y = 7$, $z = 14$
- **Symbolic Store**: $x = x_0$, $y = y_0$, $z = 2y_0$
- **Path Condition**: $x_0 > y_0 + 10$
Example Dynamic Symbolic Execution

int double(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = double(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

// concrete store
x = 22, y = 7, z = 14

// symbolic store
x = x₀, y = y₀, z = 2*y₀

// path condition
2*y₀ != x₀
Example Dynamic Symbolic Execution

```c
int double(int v) {
    return 2*v;
}

void testme(int x, int y) {
    int z = double(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }

    // Concrete store
    x = 22, y = 7, z = 14

    // Path condition
    2*y0 != x0

    // Solve: 2*y0 == x0
    // Solution: x0 = 2, y0 = 1
}
```
Example Dynamic Symbolic Execution

```c
int double(int v) {
    return 2*v;
}

void testme(int x, int y) {
    x = double(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```
Example Dynamic Symbolic Execution

```c
int double(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = double(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

- **Concrete Store**:
  - $x = 2$, $y = 1$
  - $z = 2$

- **Symbolic Store**:
  - $x = x_0$, $y = y_0$
  - $z = 2y_0$

- **Path Condition**
Example Dynamic Symbolic Execution

```c
int double(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = double(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

**Concrete Store:** $x = 2, y = 1, z = 2$

**Symbolic Store:** $x = x_0, y = y_0, z = 2*y_0$

**Path Condition:** $2*y_0 == x_0$
Example Dynamic Symbolic Execution

```c
int double(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = double(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Concrete Store</th>
<th>Symbolic Store</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 2, y = 1, z = 2</td>
<td>x = x₀, y = y₀, z = 2*y₀</td>
<td>2*y₀ == x₀, x₀ &lt;= y₀+10</td>
</tr>
</tbody>
</table>
Example Dynamic Symbolic Execution

```c
int double(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = double(y);
    if (z == x) {
        if (x > y+10) {
            // ERROR;
        }
    }
}
```

Solve: \(2y_0 = x_0\) and \(x_0 > y_0 + 10\)
Solution: \(x_0 = 30, y_0 = 15\)
Example Dynamic Symbolic Execution

```c
int double(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = double(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```
Example Dynamic Symbolic Execution

```c
int double(int v) {
    return 2*v;
}

void testme(int x, int y) {
    z = double(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

<table>
<thead>
<tr>
<th>x = 30, y = 15</th>
<th>x = x₀, y = y₀</th>
<th>2*𝑦₀ == 𝑥₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₀ &gt; y₀+10</td>
<td></td>
<td>x₀ &gt; y₀+10</td>
</tr>
</tbody>
</table>

Found error.
Example Concretization of Symbolic Expression

```c
int foo(int v) {
    return (v*v) % 50;
}

void testme(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete store:
- `x = 22, y = 7`

Symbolic store:
- `x = x_0, y = y_0`

Path condition:
Example Concretization of Symbolic Expression

```cpp
int foo(int v) {
    return (v*v) % 50;
}

void testme(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```

Solve: \((y_0*y_0) \% 50 == x_0\)
Don't know how to solve!
Stuck?

Concrete Store:
\begin{align*}
x &= 22, \
y &= 7, \
z &= 49
\end{align*}

Symbolic Store:
\begin{align*}
x &= x_0, \
y &= y_0, \
z &= (y_0*y_0) \% 50
\end{align*}

Path Condition:
\((y_0*y_0) \% 50 != x_0\)
Example Concretization of Symbolic Expression

```c
void testme(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Solve: `foo(y0) == x0`
Don’t know how to solve!
Stuck?

### Concrete Store

- `x = 22, y = 7, z = 49`

### Symbolic Store

- `x = x_0, y = y_0, z = foo(y_0)`
Example Concretization of Symbolic Expression

int foo(int v) {
    return (v*v) % 50;
}

void testme(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

Solve: \((y_0 \cdot y_0) \mod 50 = x_0\)
Don't know how to solve!
Not Stuck!
Use concrete state
Replace \(y_0\) by 7 (sound)

\(x = 22, y = 7, z = 49\)
\(x = x_0, y = y_0, z = (y_0 \cdot y_0) \mod 50\)
Example Concretization of Symbolic Expression

int foo(int v) {
    return (v*v) % 50;
}

void testme(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

Solve: 49 == x0
Solution: x0 = 49, y0 = 7

49 != x0

x = 22, y = 7, z = 48
x = x0, y = y0, z = 49
Example Concretization of Symbolic Expression

```c
int foo(int v) {
    return (v*v) % 50;
}

void testme(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Store:
- \(x = 49, y = 7\)

Symbolic Store:
- \(x = x_0, y = y_0\)

Path Condition:
- \(x_0, y_0\)
Example Concretization of Symbolic Expression

```c
int foo(int v) {
    return (v*v) % 50;
}

void testme(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```

Concrete store:
- \( x = 49, y = 7, z = 49 \)

Symbolic store:
- \( x = x_0, y = y_0, z = 49 \)

Path condition:
- \( 2y_0 = x_0 \)
- \( x_0 > y_0 + 10 \)

Found error.
Implementations

static symbolic execution:
- Symbolic PathFinder for JAVA (NASA effort!)

dynamic symbolic execution:
- Cute (research prototype), EXE, KLEE (open-source on top of LLVM) for C/C++
- jCute, Lime Concolic Tester for JVM platform (research prototypes)
- PEX for .NET platform (ran continuously by MS on cluster!)

![Graph showing coverage difference between KLEE and COREUTILS manual test suite](image)

**Figure 6:** Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests ($L_{man}$) from KLEE tests ($L_{klee}$) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE, which beats manual testing on all but 9 applications, often significantly.

```
paste -d\|\| abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\|\| abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

**Figure 7:** KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.