Concolic Testing: An Application of Solvers

Vijay Ganesh
Affiliation: University of Waterloo
A Foundation for Software Engineering
Logic Abstractions of Computation

Bob Floyd (1967)
Tony Hoare (1968,70)
Amir Pnueli (1977)
Ed Clarke (1982)
...
A Foundation for Software Engineering

Logic Abstractions of Computation

Formal Methods
Program Analysis
Logics (Boolean, ...)
Automatic Testing
Program Synthesis

Bob Floyd (1967)
Tony Hoare (1968, 70)
Amir Pnueli (1977)
Ed Clarke (1982)
...
Software Engineering & SAT/SMT Solvers
An Indispensable Tactic for Any Strategy

SE Goal: Reliable/Secure Software

Formal Methods
Program Analysis
Automatic Testing
Program Synthesis

Vijay Ganesh
Software Engineering & SAT/SMT Solvers
An Indispensable Tactic for Any Strategy

Formal Methods
Program Analysis

Automatic Testing
Program Synthesis

SAT/SMT Solvers
Software Engineering using Solvers

Engineering, Usability, Novelty

Program Reasoning Tool

Program Specification

Logic Formulas

SAT/SMT Solver

Program is correct? or Generate Counterexamples (test cases)
SAT/SMT Solver Research Story
A 1000x Improvement: Democratization of Logic

- Solver-based programming languages
- Compiler optimizations using solvers
- Solver-based debuggers
- Solver-based type systems
- Solver-based concurrency bug finding
- Solver-based synthesis
- Bio & Optimization

- Concolic Testing
- Program Analysis
- Equivalence Checking
- Auto Configuration

- Bounded MC
- Program Analysis
- AI

1,000,000 Constraints
100,000 Constraints
10,000 Constraints
1,000 Constraints

The SAT/SMT Problem

- Rich logics (Modular arithmetic, Arrays, Strings,...)
- NP-complete, PSPACE-complete,...
- Practical, scalable, usable, automatic
- Enable novel software reliability approaches

Logic

Formula

(q ∨ p ∨ ¬r)
(q ∨ ¬p ∨ r)
...

Solver

SAT

UNSAT
Dynamic Systematic Testing

Some History
Dynamic Systematic Testing

Some History

Symbolic execution for testing first proposed by Lori Clarke (1975)
ACM SIGSOFT Outstanding Researcher Award 2012
Some History

Symbolic execution for testing first proposed by Lori Clarke (1975)
ACM SIGSOFT Outstanding Researcher Award 2012

Follow up work by J.C. King (1976)
Dynamic Systematic Testing

Some History

☑ Symbolic execution for testing first proposed by Lori Clarke (1975)
   ACM SIGSOFT Outstanding Researcher Award 2012

☑ Follow up work by J.C. King (1976)

☑ Rediscovered/modified in the context of powerful solvers, analysis and appropriate concretizations by two independent groups
Dynamic Systematic Testing

Some History

- Symbolic execution for testing first proposed by Lori Clarke (1975) and recognized with the ACM SIGSOFT Outstanding Researcher Award 2012.

- Follow up work by J.C. King (1976).

- Rediscovered/modified in the context of powerful solvers, analysis and appropriate concretizations by two independent groups led by Patrice Godefroid and Koushik Sen (2005).
Symbolic execution for testing first proposed by Lori Clarke (1975)  
ACM SIGSOFT Outstanding Researcher Award 2012

Follow up work by J.C. King (1976)

Rediscovered/modified in the context of powerful solvers, analysis and appropriate concretizations by two independent groups


Dawson Engler et al. (2005)
Dynamic Systematic Testing
Some History

- Symbolic execution for testing first proposed by Lori Clarke (1975)
  ACM SIGSOFT Outstanding Researcher Award 2012

- Follow up work by J.C. King (1976)

- Rediscovered/modified in the context of powerful solvers, analysis and appropriate concretizations by two independent groups
  - Dawson Engler et al. (2005)

- Many follow up works by George Candea, Dawn Song, David Molnar, researchers in the audience,...
Dynamic Systematic Testing
Some History

✓ Symbolic execution for testing first proposed by Lori Clarke (1975)
   ACM SIGSOFT Outstanding Researcher Award 2012

✓ Follow up work by J.C. King (1976)

✓ Rediscovered/modified in the context of powerful solvers, analysis and
  appropriate concretizations by two independent groups


✓ Dawson Engler et al. (2005)

✓ Many follow up works by George Candea, Dawn Song, David Molnar,
  researchers in the audience,...

✓ Beyond testing: Fault localization, repair, security,...
Dynamic Symbolic Testing
Symbolic/Concrete Execution + Solvers

Program Specification

Dynamic Symbolic Interpreter

Logic Formulas

SAT/SMT Solver

SAT/UNSAT

Program is correct?

or Generate Counterexamples (test cases)
int double (int v) {
    return 2*v;
}

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

The concolic testing slides are courtesy Koushik Sen
Concolic Testing: Example

```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;
        }
    }
}
```
Concolic Testing Approach

```
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 Execution</th>
<th>Symbolic Execution</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 22, y = 7</td>
<td>x = x₀, y = y₀</td>
<td></td>
</tr>
</tbody>
</table>
### Concolic Testing Approach

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>int double (int v) {</td>
<td></td>
</tr>
<tr>
<td>return 2*v;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>void testme (int x, int y) {</td>
<td>x = 22, y = 7,</td>
</tr>
<tr>
<td>z = double (y);</td>
<td>z = 14</td>
</tr>
<tr>
<td>if (z == x) {</td>
<td>x = x_0, y = y_0,</td>
</tr>
<tr>
<td>if (x &gt; y+10) {</td>
<td>z = 2*y_0</td>
</tr>
<tr>
<td>ERROR;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

- **Concrete State**:
  - \( x = 22, y = 7, z = 14 \)
  - \( x = x_0, y = y_0, z = 2*y_0 \)

- **Symbolic State**:
  - variables are symbolic
  - initial values: \( x = x_0, y = y_0 \)
  - \( z = 2*y_0 \)

- **Path Condition**: variables are symbolic
Concolic Testing Approach

```
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 Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>concrete state</td>
<td>symbolic state</td>
</tr>
<tr>
<td>path condition</td>
<td></td>
</tr>
</tbody>
</table>

For example:
- **Concrete**: `x = 22, y = 7, z = 14`
- **Symbolic**: `x = x_0, y = y_0, z = 2*y_0

2*y_0 != x_0
Concolic Testing Approach

```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 Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solve: 2*y₀ == x₀</td>
<td>Solution: x₀ = 2, y₀ = 1</td>
<td>2*y₀ != x₀</td>
</tr>
</tbody>
</table>

Solution: x₀ = 2, y₀ = 1

x = 22, y = 7, z = 14  

x = x₀, y = y₀, z = 2*y₀
Concolic Testing Approach

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>int double (int v) {</strong></td>
<td><strong>concrete state</strong></td>
<td><strong>2*y₀ == x₀</strong></td>
</tr>
<tr>
<td>return 2*v; }</td>
<td><strong>symbolic state</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>x = 2, y = 1,</strong></td>
<td></td>
</tr>
<tr>
<td><strong>void testme (int x, int y) {</strong></td>
<td><strong>path condition</strong></td>
<td></td>
</tr>
<tr>
<td>z = double (y);</td>
<td><strong>x = x₀, y = y₀,</strong></td>
<td></td>
</tr>
<tr>
<td>if (z == x) {</td>
<td><strong>z = 2</strong></td>
<td></td>
</tr>
<tr>
<td>if (x &gt; y+10) {</td>
<td><strong>z = 2*y₀</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ERROR;</strong></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Concolic Testing Approach

```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 Execution

Symbolic Execution

<table>
<thead>
<tr>
<th>Concrete State</th>
<th>Symbolic State</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 &gt; y₀ + 10</td>
</tr>
</tbody>
</table>
Concolic Testing Approach

```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 Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>concrete state</td>
<td>symbolic state</td>
</tr>
</tbody>
</table>

```
Solve: (2*y_0 == x_0) \textbf{AND} (x_0 > y_0 + 10)
```

Solution: \( x_0 = 30, y_0 = 15 \)

\[ 2*y_0 == x_0 \]
\[ x_0 \cdot y_0 + 10 \]

\( x = 2, y = 1, \)
\( z = 2 \)

\( x = x_0, y = y_0, \)
\( z = 2*y_0 \)
Concolic Testing Approach

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 Execution

Symbolic Execution

Concrete state

x = 30, y = 15

Symbolic state

x = x₀, y = y₀

Path condition
Concolic Testing Approach

```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 Execution

Symbolic Execution

- **Concrete State**
  - \( x = 30, \ y = 15 \)
- **Symbolic State**
  - \( x_0, \ y = y_0 \)
- **Path Condition**
  - \( 2y_0 == x_0 \)
  - \( x_0 > y_0 + 10 \)

Program Error
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- combine with valgrind to discover memory errors