DART: Directed Automated Random Testing

CUTE: Concolic Unit Testing Engine

Slide Source: Koushik Sen from Berkeley
Verification and Testing

- We would like to prove programs correct
Verification and Testing

- We would like to prove programs correct
- World prefers to test and validate
Verification and Testing

- We would like to prove programs correct
- World prefers to test and validate
  - Evidence: 50%-80% software development cost goes into testing only
  - Why?
We would like to prove programs correct

World prefers to test and validate
- Evidence: 50%-80% software development cost goes into testing only
- Why?

One Reason:
Testing can expose Unforeseen Behaviors
Goals of Testing

- Generate test inputs
- Execute program on generated test inputs
- Catch assertion violations
- Problem: how to ensure that all reachable statements are executed
- Solution:
  - Explore all feasible execution paths
Execution of Programs

- All Possible Execution Paths
  - Binary tree
    - Computation tree
  - Internal node $\rightarrow$ conditional statement execution
  - Edge $\rightarrow$ execution of a sequence of non-conditional statements
  - Each path in the tree represents an equivalence class of inputs

Diagram:
- Binary tree with conditional and non-conditional statements.
Concolic Testing

- Combine random testing (concrete execution) and symbolic testing (symbolic execution)

\[
\text{Concrete} + \text{Symbolic} = \text{Concolic}
\]
Running 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;
        }
    }
}
```
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

```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>
<tr>
<td>path condition</td>
<td></td>
</tr>
</tbody>
</table>

Example:
- **Concrete state**: $x = 22, y = 7$
- **Symbolic state**: $x = x_0, y = y_0$
Concolic Testing Approach

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

Concrete state:
- `x = 22`, `y = 7`, `z = 14`

Symbolic state:
- `x = x_0`, `y = y_0`, `z = 2*y_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;
        }
    } else {
        // Symbolic State
        x = x0, y = y0, z = 2*y0
    }
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 22, y = 7, z = 14</td>
<td>x = x0, y = y0, z = 2*y0</td>
</tr>
</tbody>
</table>

- **Concrete State**: `x = 22, y = 7, z = 14`  
- **Symbolic State**: `x = x0, y = y0, z = 2*y0`
- **Path Condition**: `2*y0 != x0`
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>x = 22, y = 7, z = 14</td>
<td>x = x₀, y = y₀, z = 2*y₀</td>
<td>2*y₀ != x₀</td>
</tr>
</tbody>
</table>

Symbolic Execution

Solve: 2*y₀ == x₀
Solution: x₀ = 2, y₀ = 1

Concrete state

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 22, y = 7, z = 14</td>
<td>x = x₀, y = y₀, z = 2*y₀</td>
<td>2*y₀ != x₀</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</td>
<td>x = x₀, y = y₀</td>
<td></td>
</tr>
</tbody>
</table>
Concolic Testing Approach

```cpp
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 = 2, y = 1, z = 2</td>
<td>x = x₀, y = y₀, z = 2⁺y₀</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;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Concrete State</strong></td>
<td><strong>Symbolic State</strong></td>
<td><strong>2*y0 == x0</strong></td>
</tr>
<tr>
<td>x = 2, y = 1, z = 2</td>
<td>x = x0, y = y0, z = 2*y0</td>
<td>2*y0 == x0</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>
<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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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;
        }
    }
}
```

Concrete Execution

- `x = 2, y = 1, z = 2`
- `x = x₀, y = y₀, z = 2*y₀`

Symbolic Execution

- `x₀ ≤ y₀+10`
- `2*y₀ == x₀`

Path Condition

- `2*y₀ == x₀`
- `x₀ ≥ y₀ + 10`

Solve: $(2*y₀ == x₀) - (x₀ > y₀ + 10)$

Solution: $x₀ = 30, y₀ = 15$
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>State</th>
<th>Symbolic Execution</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 30, y = 15</td>
<td>x = x₀, y = y₀</td>
<td>path condition</td>
</tr>
</tbody>
</table>

Symbolic Execution

Concrete state

 Symbolic state

<table>
<thead>
<tr>
<th>State</th>
<th>Symbolic Execution</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 30, y = 15</td>
<td>x = x₀, y = y₀</td>
<td>path condition</td>
</tr>
</tbody>
</table>
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: 2*y₀ == x₀

Program Error: x₀ > y₀+10
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
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
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
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
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
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
Directed Search: Summary

- Dynamic test generation to **direct** executions along alternative program paths
  - collect symbolic constraints at branch points (whenever possible)
  - negate one constraint at a branch point to take other branch (say \( b \))
  - call constraint solver with new path constraint to generate new test inputs
  - next execution driven by these new test inputs to take alternative branch \( b \)
  - check with dynamic instrumentation that branch \( b \) is indeed taken
- Repeat this process until all execution paths are covered
  - May never terminate!
- Significantly improves code coverage vs. pure random testing
DART for C: Implementation Details

3 possible outcomes:
- Error found
- Complete coverage
- Run forever...

Constraint solver(s) (e.g., lp_solve.so)
DART: Success Stories

- Tested a C implementation of a security protocol (Needham-Schroeder) with a known attack
  - About 400 lines of C code; experiments on a Linux 800Mz P-III machine
  - DART takes less than 2 seconds (664 runs) to discover a (partial) attack, with an unconstrained (possibilistic) intruder model
  - DART takes 18 minutes (328,459 runs) to discover a (full) attack, with a realistic (Dolev-Yao) intruder model
  - DART found a new bug in this C implementation of Lowe’s fix to the NS protocol (after 22 minutes of search; bug confirmed by the code’s author)

- In contrast, a systematic state-space search of this program composed with a concurrent nondeterministic intruder model using VeriSoft (a sw model checker) does not find the attack
Limitations

- Path Space of a Large Program is Huge
  - Path Explosion Problem
Limitations

- Path Space of a Large Program is Huge
  - Path Explosion Problem

Entire Computation Tree

Explored by Concolic Testing
Limitations

- Path Space of a Large Program is Huge
  - Path Explosion Problem

How bad is this in Practice?

For a simple parser (PL/0) concolic testing failed to generate a valid program within 24 hours.

For the vi editor, CUTE only tested 3 character long inputs.