MaTRIX
Maintenance-Oriented Test Requirements Identifier and Examiner

Mary Jean Harrold†
Taweesup (Term) Apiwattanapong,† Raúl Santelices,†
Pavan Kumar Chittimalli,‡ Alessandro Orso†

†College of Computing, Georgia Institute of Technology
‡Tata Research Development & Design Centre, TCS Limited

Supported by Tata Consultancy Services (TCS) Limited and by NSF
Regression Testing

Program P

P’ Version of P

T
Regression Testing

Program P

P' Version of P

T-T'

T'

T

T'

T'
Regression Testing

How well do T, T’, T,” or any test suites exercise P’ with respect to changes?

Is there suitable guidance for creating new test cases that target the modified behavior?
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5) {
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
}
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5) {
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
}
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5){
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0){
            print(x);
        } else {
            print(10/x);
        }
    }
}
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5) {
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
    ...
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5) {
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
}

Motivating Example
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5) {
            x = 5/(x-5);
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
    ...
}
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5){
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0){
            print(x);
        } else {
            print(10/x);
        }
    }
    ...
}
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5){
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0){
            print(x);
        } else {
            print(10/x);
        }
    }
}
Motivating Example

public class E {
    void simple (int i) {
        int x = i;
        if (x > 5){
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0){
            print(x);
        } else {
            print(10/x);
        }
    }
}

Tests satisfy test requirements for criteria but don’t reveal fault in s3
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5){
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0){
            print(x);
        } else {
            print(10/x);
        }
    }
}

Motivating Example

Criteria require
• Execution of the change and entities affected by change

But don’t require
• Infection of the state after change
• Propagation of state to output where it can be observed

Tests satisfy test requirements for criteria but don’t reveal fault in s3
Computation of Testing Requirements

Our technique adds these requirements to the criteria

Criteria require
• Execution of the change and entities affected by change

But don’t require
• Infection of the state after change
• Propagation of state to output where it can be observed
public class E {
    void simple (int i) {
        int x = i;
        if (x >= 5) {
            x = 5/(x-5);
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
}

<table>
<thead>
<tr>
<th>PC</th>
<th>SS(x)</th>
<th>PC'</th>
<th>SS'(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>i₀</td>
<td>true</td>
<td>i₀</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>(i₀&gt;5)</td>
<td>5/(i₀-5)</td>
<td>(i₀&gt;=5)</td>
<td>5/(i₀-5)</td>
</tr>
<tr>
<td>(i₀&lt;=5)</td>
<td>i₀-1</td>
<td>(i₀&lt;5)</td>
<td>i₀-1</td>
</tr>
<tr>
<td>(i₀&gt;5)</td>
<td>5/(i₀-5)-1</td>
<td>(i₀&gt;=5)</td>
<td>5/(i₀-5)-1</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>(i₀==0)</td>
<td>0</td>
<td>(i₀==0)</td>
<td>0</td>
</tr>
<tr>
<td>(i₀&lt;=5) ∧</td>
<td>i₀-1</td>
<td>(i₀&lt;5) ∧</td>
<td>i₀-1</td>
</tr>
<tr>
<td>(i₀!=0)</td>
<td></td>
<td>(i₀!=0)</td>
<td></td>
</tr>
<tr>
<td>(i₀&gt;5) ∧</td>
<td>5/(i₀-5)-1</td>
<td>(i₀&gt;=5) ∧</td>
<td>5/(i₀-5)-1</td>
</tr>
<tr>
<td>(i₀!=0)</td>
<td></td>
<td>(i₀!=0)</td>
<td></td>
</tr>
</tbody>
</table>

PC—path condition  SS—symbolic state
Computation of Testing Requirements

Conditions for propagation of infected states:

1. The execution in P’ reaches $s_i'$ and the execution in P does not reach $s_i$; or

2. The execution in P’ reaches $s_i'$ and the execution in P reaches $s_i$; however, $s_i'$ and $s_i$ have different symbolic states.

```java
public class E {
    void simple (int i) {
        int x = i;
        if (x > 5) {
            x = 5/(x-5);
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
}
```
## Computation of Testing Requirements

**public class E**

```java
type void simple (type int i)

    type int x = i;

    if (x > 5)
    
    x = (5/(x-5));

    x = x - 1;

    if (x == 0)
    
    print(x);

    else

    print(10/x);

}
```

**But (as we discussed yesterday)**

- symbolic execution is expensive
- won’t scale to large programs
- can’t be applied for entire paths
- etc.

**Our technique has two ways to improve efficiency**

<table>
<thead>
<tr>
<th>PC—path condition</th>
<th>SS—symbolic state</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS(x)</td>
<td>SS’(x)</td>
</tr>
<tr>
<td>i₀</td>
<td>i₀</td>
</tr>
<tr>
<td>”</td>
<td>”</td>
</tr>
<tr>
<td>5/(i₀-5)</td>
<td>5/(i₀-5)-1</td>
</tr>
<tr>
<td>i₀⁻¹</td>
<td>i₀⁻¹</td>
</tr>
<tr>
<td>”</td>
<td>”</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(i₀⁻¹)</td>
<td>(i₀⁻¹)</td>
</tr>
<tr>
<td>5/(i₀-5)⁻¹</td>
<td>5/(i₀-5)-1</td>
</tr>
</tbody>
</table>

**PC—path condition**

**SS—symbolic state**

**TAIC PART**

**August 2006**
Computation of Testing Requirements

public class E {
    void simple (int i) {
        int x = i;
        if (x >= 5) {
            x = (5/(x-5));
        } else {
            print(x);
        }
    }
}

1. Perform **partial symbolic execution (PSE)** beginning immediately before the change
   - computes conditions in terms of variables immediately before change
   - avoids symbolic execution from beginning of program to change
public class E {
    void simple(int i) {
        int x = i;
        if (x >= 5) {
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
}
### Computation of Testing Requirements

```java
public class E {
    void simple (int i) {
        int x = i;
        if (x >= 5) {
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
}
```

<table>
<thead>
<tr>
<th>PC</th>
<th>SS(x)</th>
<th>PC'</th>
<th>SS'(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>true</td>
<td>$x_0$</td>
<td>true</td>
<td>$x_0$</td>
</tr>
<tr>
<td>$(x_0&gt;5)$</td>
<td>$5/(x_0-5)$</td>
<td>$(x_0&gt;=5)$</td>
<td>$5/(x_0-5)$</td>
</tr>
</tbody>
</table>

1. Perform **partial symbolic execution (PSE)** beginning immediately before the change
   - computes conditions in terms of variables immediately before change
   - avoids symbolic execution from beginning of program to change

**Don’t need to solve conditions—can still monitor for their satisfaction**
## Computation of Testing Requirements

```java
public class E {
    void simple (int i) {
        int x = i;
        if (x >= 5){
            x = (5/(x-5));
        } else {
            x = x - 1;
        }
        if (x == 0){
            print(x);
        } else {
            print(10/x);
        }
    }
}
```

<table>
<thead>
<tr>
<th>PC</th>
<th>SS(x)</th>
<th>PC'</th>
<th>SS'(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>true</td>
<td>x₀</td>
<td>true</td>
<td>x₀</td>
</tr>
<tr>
<td>(x₀&gt;5)</td>
<td>5/(x₀-5)</td>
<td>(x₀&gt;=5)</td>
<td>5/(x₀-5)</td>
</tr>
</tbody>
</table>

2. Perform PSE for some specified distance (user selected) instead of to output statements

- computes conditions on states at intermediate points (i.e., distances)
- bounds depth, avoids symbolic execution to outputs
public class E {
    void simple (int i) {
        int x = i;
        if (x >= 5) {
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
}

Computation of Testing Requirements

<table>
<thead>
<tr>
<th>Distance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0—after change</td>
<td>after change</td>
</tr>
<tr>
<td>1—after 1 dependence</td>
<td>after 1 dependence</td>
</tr>
<tr>
<td>2—after 2 dependences</td>
<td>after 2 dependences</td>
</tr>
<tr>
<td>3—after 3 dependences</td>
<td>after 3 dependences</td>
</tr>
</tbody>
</table>

And so on until output
public class E {
    void simple (int i) {
        int x = i;
        if (x >= 5) {
            x = (5/(x-5));
        } else {
            x = x - 1;
        }
    }
}

2. Perform PSE for some specified distance (user selected) instead of to output statements

- computes conditions on states at intermediate points (i.e., distances)
- bounds depth, avoids symbolic execution to outputs

Greater distances improve confidence in propagation to output
Computation of Testing Requirements

public class E {
    void simple (int i) {
        int x = i;
        if (x >= 5) {
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
}

Distance 1

PC'(s3) and (not PC(s3))
→ (x₀ >= 5) and (not (x₀ > 5))
→ (x₀ >= 5) and (x₀ <= 5)
→ (x₀ == 5)
public class E {
    void simple (int i) {
        int x = i;
        if (x >= 5) {
            x = (5/(x-5));
        }
        x = x - 1;
        if (x == 0) {
            print(x);
        } else {
            print(10/x);
        }
    }
    ...
}

Use of Testing Requirements

1. Instrument program so that probe checks for condition before change (e.g., after s1)
2. Assist developer in satisfying criterion and improving confidence in testing
3. Generate test if condition can be satisfied (future work)
Empirical Study: Setup

Goal:
To compare the effectiveness of our changed-based criteria with statement and all-uses coverage criteria (based on changes)

Implementation: uses differencing, Java Pathfinder, instrumenter, data-/control-dependence analysis, etc.

Subjects: Tcas (4 versions) and Schedule (3 versions) (each version has one fault)

Method:
• Randomly generate 50 test suites per criterion.
• Record the number of test suites that produce different outputs.
Effectiveness Study: Results
Percentage of test suites revealing different behaviors over 50 test suites that satisfy each criterion.

Effectiveness Study: Results
Percentage of test suites revealing different behaviors over 50 test suites that satisfy each criterion.
Conclusions

New technique

- Identifies (creates), examines (monitors) test requirements related to change(s)
- Uses symbolic execution but gains efficiency
  - partial symbolic execution so avoids performing symbolic execution from beginning of program
  - partial symbolic execution to specified distances from change so bounds depth of symbolic execution
- Size of symbolic execution tree related to change instead of size of program
- Empirical evaluation show promise of approach
Current and Future Work

Current

• Completing infrastructure
• Performing experiments—additional subjects, more complex changes, scalability, limitations

Future

• Expand technique to handle multiple changes, changes involving multiple statements
• Use conditions for automatic test-case generation
Questions?