Test Selection for Result Inspection via Mining Predicate Rules

Wujie Zheng, Michael R. Lyu  
The Chinese University of Hong Kong  

Tao Xie  
NC State University
Test Selection for Result Inspection

- Test result inspection
  - A main step in software testing, especially in automatic testing
  - Labor-intensive without test oracles

- Test selection for result inspection
  - Select a *small* subset of tests that are likely to *reveal faults*

*Hey! Check only these tests!*
Previous Work: Mining Operational Models from Passing Tests

- Mine invariants from passing tests (Daikon, DIDUCE)

```
i, s := 0, 0;
do i ≠ n →
   i, s := i + 1, s + b[i]
  od
```

Precondition: \( n \geq 0 \)
Postcondition: \( s = \left( \sum_{j=0}^{\min(j \leq n : b[j])} \right) \)
Loop invariant: \( 0 \leq i \leq n \) and \( s = \left( \sum_{j=0}^{\min(j < i : b[j])} \right) \)

- Select tests that violate the existing invariants (Jov, Eclat, DIDUCE)
Previous Work: Mining Operational Models from Passing Tests

- **Limitations**
  - The number of existing passing tests is often limited.
  - The mined operational models could be noisy and thus many violations could be false positives.
Our Approach: Mining Operational Models from Unverified Tests

- Existing passing tests -> unverified tests
- Dynamic invariants -> common operational models
Our Approach: Mining Operational Models from Unverified Tests

Why mining unverified tests can help?

- A program that is not of poor quality should pass most of the tests
- Common operational models mined from a large set of unverified tests could be good approximations of the real model
Our Approach: Mining Operational Models from Unverified Tests

- How to mine common operational models?
  - Cannot discard an operational model when it is violated
  - Collect the evaluations of all of them for postmortem analysis? May incur high runtime overhead
  - Our solution
    - Collect values of simple predicates at runtime (use CBI-tools)
    - Generate and evaluate predicate rules as potential operational models after running all the tests
      - A predicate rule is an implication relationship between predicates
Our Approach: Mining Operational Models from Unverified Tests

The real operational model
The program would fail if \( x > 0 \land y \geq x \)
In passing tests, the program should satisfy a precondition \( x \leq 0 \lor y < x \)

The simple predicates
Their violations cannot predict the failures accurately

The predicate rules
\( P_1 \Rightarrow P_4 \) corresponds to a precondition \( x \leq 0 \lor y \leq x \)
This is similar to and weaker than the real operational model. Its violation should also lead to the violation of the real operational model and indicate a failure, such as Test 5.

<table>
<thead>
<tr>
<th>Test input</th>
<th>Expected Output</th>
<th>Actual Output</th>
<th>Predicate Profiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. x=-1, y=0</td>
<td>0</td>
<td>0</td>
<td>P2, P4</td>
</tr>
<tr>
<td>2. x=0, y=1</td>
<td>1</td>
<td>1</td>
<td>P2, P3</td>
</tr>
<tr>
<td>3. x=1, y=0</td>
<td>0</td>
<td>0</td>
<td>P1, P4</td>
</tr>
<tr>
<td>4. x=1, y=1</td>
<td>1</td>
<td>0</td>
<td>P1, P4</td>
</tr>
<tr>
<td>5. x=1, y=2</td>
<td>2</td>
<td>1</td>
<td>P1, P3</td>
</tr>
</tbody>
</table>

Figure 1. An example program
Our Approach: Mining Operational Models from Unverified Tests

- The preliminary algorithm
  - Collect values of simple predicates at runtime
  - Mine predicate rules
    - $x \Rightarrow y$, where $x$ and $y$ are simple predicates
    - For each predicate $y$, select rule $x \Rightarrow y$ with the highest confidence
  - Select tests for result inspection
    - Sort the selected predicate rules in the descending order of confidence.
    - Select tests that violate the rules from the top to bottom
Preliminary Results

- **Subject 1: the Siemens suite**
  - 130 faulty versions of 7 programs that range in size from 170 to 540 lines
  - On average, 1.5% (45/2945) tests, detect 75% (97/130) faults
  - Random Sampling: 1.5% (45/2945) tests, 45% (59/130) faults

<table>
<thead>
<tr>
<th>Program</th>
<th>Original Test Set</th>
<th>Our approach</th>
<th>Random Sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Tests</td>
<td>#Failed Tests (avg)</td>
<td>#Tests</td>
</tr>
<tr>
<td>print_tokens</td>
<td>4130</td>
<td>69.1</td>
<td>41</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>4115</td>
<td>223.7</td>
<td>47</td>
</tr>
<tr>
<td>replace</td>
<td>5542</td>
<td>105.8</td>
<td>76</td>
</tr>
<tr>
<td>schedule</td>
<td>2650</td>
<td>87.7</td>
<td>33</td>
</tr>
<tr>
<td>schedule2</td>
<td>2710</td>
<td>32.8</td>
<td>41</td>
</tr>
<tr>
<td>teas</td>
<td>1608</td>
<td>38.5</td>
<td>38</td>
</tr>
<tr>
<td>tot_info</td>
<td>1052</td>
<td>82.6</td>
<td>23</td>
</tr>
<tr>
<td>all(avg)</td>
<td>2925</td>
<td>81.3</td>
<td>45</td>
</tr>
</tbody>
</table>
Preliminary Results

- Subject 2: the `grep` program
  - 13,358 lines of C code; 3 buggy versions that fail 3, 4, and 132 times running the 470 tests, respectively.
  - Our approach selects 82, 86, and 89 tests that reveal all the 3 faults.
  - For each version, there is at least one failing test ranked in top 20.
  - Randomly select 20 tests for 5 times: never reveal the first two faults but always reveal the third fault.
Future work

- Combine with automatic test generation tools

- Mine more general operational models
  - Incorporate non-binary information

- Study the characteristics of mined common operational models
  - Present them to the programmers
Thank you!