It is labor-intensive to manually verify outputs of a large set of tests not equipped with test oracles.

Test selection for result inspection helps to reduce the cost of test-result inspection by selecting a small subset of tests likely to reveal faults.

Previous Work: Mining operational models from passing tests

Mine operational models satisfied by all passing tests as test oracles, and then select violating tests for result inspection.

- Daikon: Mine rules over variables from passing tests
- Jov/Eclat: Select new tests violating operational models
- DIDUCE: Mine models of variables from normal execution of a long-running application

Limitations: The number of existing passing tests is often limited. Mined operational models could be noisy and thus many violations could be false positives.

Our Approach: Mining operational models from unverified tests

Mine common operational models, not always true in all observed traces, from a (potentially large) set of unverified tests based on mining predicate rules.

Rationale: A program not of poor quality should pass most of the tests, i.e., the real operational model should be satisfied by most of the tests. Mining common operational models from unverified tests can thus reduce the noise.

Challenges: (1) Cannot simply discard a potential common operational model whenever it is violated. (2) To collect the evaluations of all models for postmortem analysis could incur high runtime overhead (if Daikon-like operational models are used).

Solution:
- Collect values of simple predicates at runtime.
- Generate and evaluate predicate rules: implication relationships between predicates, as potential operational models after running all the tests.
  - Only mine rules x→y, where x and y are simple predicates
  - For each predicate y, select rule x→y with the highest confidence
  - Select a set of tests that violate all the mined predicate rules for result inspection.

Example Program

```
1 int testfun(int x, int y)
2 {
3     y = x + 0;
4     if (x == 0)
5         x = y + 1;
6     else
7         return y;
8 } return 0;
```

An Example Program

<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=1</td>
<td>0</td>
<td>1</td>
<td>P1, P4</td>
</tr>
<tr>
<td>2, x=0, y=1</td>
<td>1</td>
<td>0</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=0, y=0</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>

Tests and Predicate Profiles

Figure 1. An example program

Preliminary Results

- Subject 1: the Siemens suite
  - 130 faulty versions of 7 programs that range in size from 170 to 540 lines
  - On average, only 1.53% (45/2945) of the original tests are needed to be checked, which can still reveal 74.6% (97/130) of the faults, while random sampling can reveal only 45.4% (59/130) of the faults.

- Subject 2: the grep program
  - A unix utility to search a file for a pattern; 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 for these versions, which reveal all the 3 faults.
  - For each version, at least one failing test ranked in top 20.
  - Randomly select 20 tests for each version. In the 5 times of random selection, the selected tests do not reveal the faults of the first 2 versions but always reveal the faults of the 3rd version.

Table 1. Test selection in the Siemens suite

<table>
<thead>
<tr>
<th>Program</th>
<th>#Tests</th>
<th>#Passed</th>
<th>#Passed Tests</th>
<th>#Fails</th>
<th>#Fails Detected</th>
<th>#Tests</th>
<th>#Fails Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>praphon</td>
<td>4130</td>
<td>4089</td>
<td>340</td>
<td>86</td>
<td>3</td>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>nptmouth</td>
<td>4130</td>
<td>4089</td>
<td>340</td>
<td>86</td>
<td>3</td>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>nptmouth</td>
<td>4130</td>
<td>4089</td>
<td>340</td>
<td>86</td>
<td>3</td>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>nptmouth</td>
<td>4130</td>
<td>4089</td>
<td>340</td>
<td>86</td>
<td>3</td>
<td>41</td>
<td>3</td>
</tr>
</tbody>
</table>

Limitations: The number of existing passing tests is often limited. Mined operational models could be noisy and thus many violations could be false positives.

Previous Work: Mining operational models from passing tests

Daikon: Mine rules over variables from passing tests
Jov/Eclat: Select new tests violating operational models
DIDUCE: Mine models of variables from normal execution of a long-running application

Rationale: A program not of poor quality should pass most of the tests, i.e., the real operational model should be satisfied by most of the tests. Mining common operational models from unverified tests can thus reduce the noise.

Challenges: (1) Cannot simply discard a potential common operational model whenever it is violated. (2) To collect the evaluations of all models for postmortem analysis could incur high runtime overhead (if Daikon-like operational models are used).

Solution:
- Collect values of simple predicates at runtime.
- Generate and evaluate predicate rules: implication relationships between predicates, as potential operational models after running all the tests.
  - Only mine rules x→y, where x and y are simple predicates
  - For each predicate y, select rule x→y with the highest confidence
  - Select a set of tests that violate all the mined predicate rules for result inspection.

Example Program

```
1 int testfun(int x, int y)
2 {
3     y = x + 0;
4     if (x == 0)
5         x = y + 1;
6     else
7         return y;
8 } return 0;
```

An Example Program

<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=1</td>
<td>0</td>
<td>1</td>
<td>P1, P4</td>
</tr>
<tr>
<td>2, x=0, y=1</td>
<td>1</td>
<td>0</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=0, y=0</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>

Tests and Predicate Profiles

Figure 1. An example program

Preliminary Results

- Subject 1: the Siemens suite
  - 130 faulty versions of 7 programs that range in size from 170 to 540 lines
  - On average, only 1.53% (45/2945) of the original tests are needed to be checked, which can still reveal 74.6% (97/130) of the faults, while random sampling can reveal only 45.4% (59/130) of the faults.

- Subject 2: the grep program
  - A unix utility to search a file for a pattern; 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 for these versions, which reveal all the 3 faults.
  - For each version, at least one failing test ranked in top 20.
  - Randomly select 20 tests for each version. In the 5 times of random selection, the selected tests do not reveal the faults of the first 2 versions but always reveal the faults of the 3rd version.

Table 1. Test selection in the Siemens suite

<table>
<thead>
<tr>
<th>Program</th>
<th>#Tests</th>
<th>#Passed</th>
<th>#Passed Tests</th>
<th>#Fails</th>
<th>#Fails Detected</th>
<th>#Tests</th>
<th>#Fails Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>praphon</td>
<td>4130</td>
<td>4089</td>
<td>340</td>
<td>86</td>
<td>3</td>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>nptmouth</td>
<td>4130</td>
<td>4089</td>
<td>340</td>
<td>86</td>
<td>3</td>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>nptmouth</td>
<td>4130</td>
<td>4089</td>
<td>340</td>
<td>86</td>
<td>3</td>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>nptmouth</td>
<td>4130</td>
<td>4089</td>
<td>340</td>
<td>86</td>
<td>3</td>
<td>41</td>
<td>3</td>
</tr>
</tbody>
</table>

Limitations: The number of existing passing tests is often limited. Mined operational models could be noisy and thus many violations could be false positives.

References