Test Selection for Result Inspection via Mining Predicate Rules

Wujie Zheng wjzheng@cse.cuhk.edu.hk
Michael R. Lyu lyu@cse.cuhk.edu.hk
Tao Xie xie@csc.ncsu.edu

The Chinese University of Hong Kong
NC STATE UNIVERSITY

Problem

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

$i, s := 0, 0;$
do $i \neq n \rightarrow$
$\quad i, s := i + 1, s + b[i]$
od

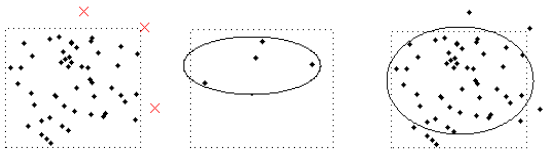
Precondition: n ≥ 0
Postcondition: s = (∑j : 0 ≤ j < n : b[j])
Loop invariant: 0 ≤ i ≤ n and s = (∑j : 0 ≤ j < i : b[j])

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.



Real operational model | Operational model learned from limited passing tests | Operational model learned from all the unverified tests

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 test(int x, int y)
2 {
3       if(x>0)
4             y = y-x;  // should be y=y-x+1;
5       if(y>0)                          P1: Line 3, x>0
6             return y;                  P2: Line 3, x<=0
7       else                             P3: Line 5, y>0
8             return 0;                  P4: Line 5, y<=0
9 }
```

An Example Program **Predicates**

Test input	Expected Output	Actual Output	Predicate Profiles
1. x=-1, y=0	0	0	P2, P4
2. x=0, y=1	1	1	P2, P3
3. x=1, y=0	0	0	P1, P4
4. x=1, y=1	1	0	P1, P4
5. x=1, y=2	2	1	P1, P3

Tests and Predicate Profiles

Figure 1. An example program

- **The real operational model**

The program would fail if $x > 0 \wedge y \geq x$

In passing tests, the program should satisfy a precondition $x \leq 0 \vee y < x$

- **The simple predicates**

We observe that a failure is not likely to be predicted by the violation of a single predicate.

- **The predicate rules**

$P1 \Rightarrow P4$ corresponds to a precondition $x \leq 0 \vee y \leq x$.

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.

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

Program	Original Test Set		Our approach		Random Sampling	
	#Tests	#Failed Tests (avg)	#Tests	#faulty versions detected	#Tests	#faulty versions detected
print_tokens	4130	69.1	41	6/7	41	2/7
print_tokens2	4115	223.7	47	10/10	47	6.2/10
replace	5542	105.8	76	26/31	76	13.8/31
schedule	2650	87.7	33	6/9	33	2/9
schedule2	2710	32.8	41	6/9	41	2.8/9
tcas	1608	38.5	38	26/41	38	15.6/41
tot_info	1052	82.6	23	17/23	23	16.2/23
all(avg)	2925	81.3	45	97/130	45	58.6/130

References

[1] http://sir.unl.edu/php/index.php
[2] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, 2001.
[3] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. *In ICSE*, pages 291–301, 2002.
[4] B. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.
[5] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In ECOOP, pages 504–527, 2005.
[6] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In ASE, pages 40–48, 2003.