Test Selection for Result Inspection via Mining Predicate Rules

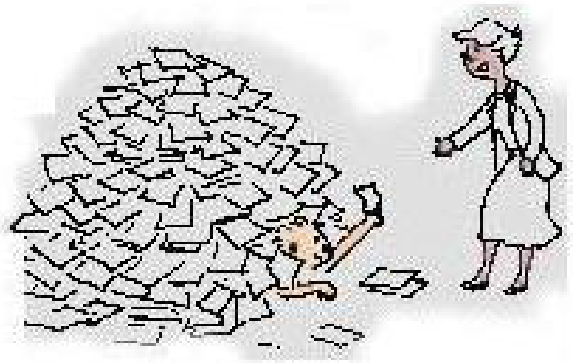
Wujie Zheng, Michael R. Lyu
The Chinese University of Hong Kong

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)

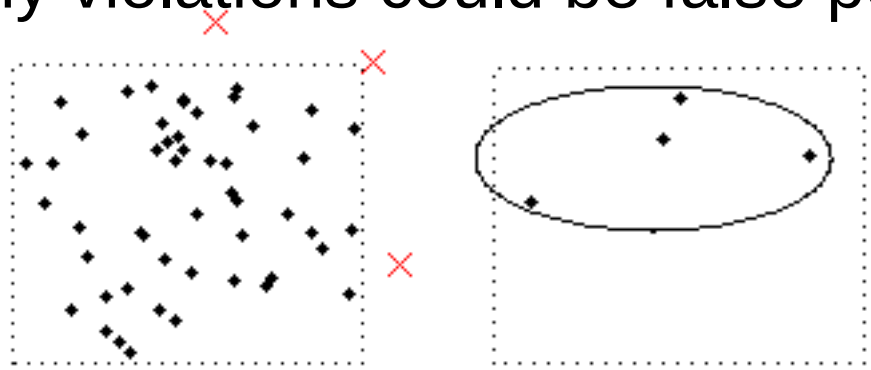
$i, s := 0, 0;$
do $i \neq n \rightarrow$
$\quad i, s := i + 1, s + b[i]$
od

Precondition: $n \geq 0$
Postcondition: $s = (\sum j : 0 \leq j < n : b[j])$
Loop invariant: $0 \leq i \leq n$ and $s = (\sum j : 0 \leq j < i : b[j])$

- Select tests that violate the existing invariants (Jov, Eclat, DIDUCE)

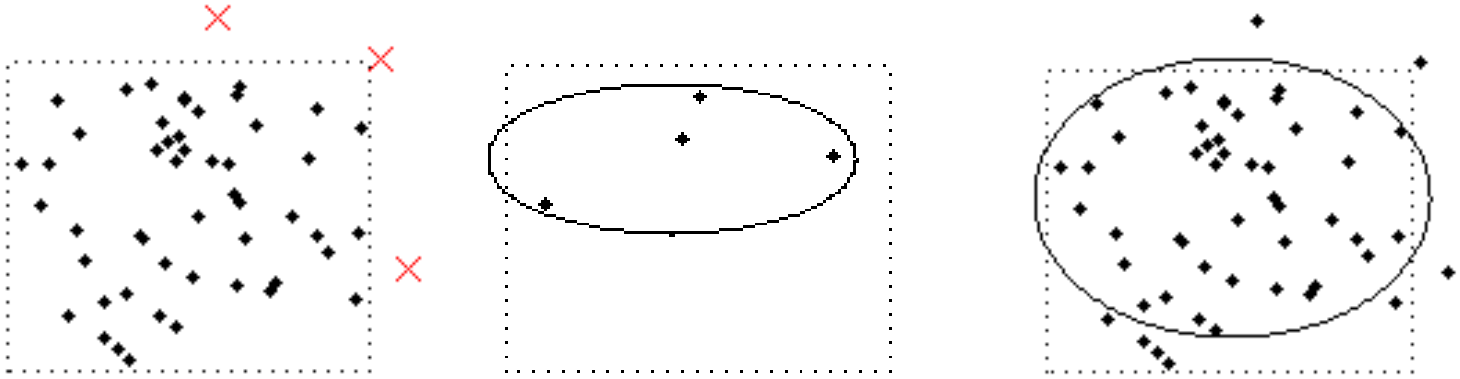
Previous Work: Mining Operational Models from Passing Tests

- 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.



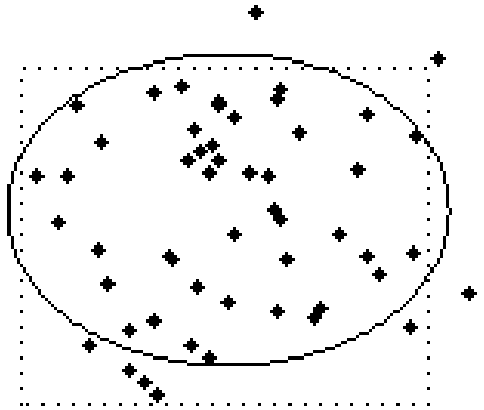
Our Approach: Mining Operational Models from Unverified Tests

- 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

```
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**

Their violations cannot predict the failures accurately

- **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.

Our Approach: Mining Operational Models from Unverified Tests

- The preliminary algorithm
  - Collect values of simple predicates at runtime
  - Mine predicate rules
    - x=>y, where x and y are simple predicates
    - For each predicate y, select rule x=>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 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

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!