Test Selection for Result Inspection via Mining Predicate Rules

Wujie Zheng, Michael R. Lyu
Computer Science and Engineering
Chinese University of Hong Kong
{wjzheng,lyu}@cse.cuhk.edu.hk

Tao Xie
Department of Computer Science
North Carolina State University
xie@csc.ncsu.edu

Abstract

It is labor-intensive to manually verify the outputs of a large set of tests that are not equipped with test oracles. Test selection helps to reduce this cost by selecting a small subset of tests that are likely to reveal faults. A promising approach is to dynamically mine operational models as potential test oracles and then select tests that violate them. Existing work mines operational models from verified passing tests based on dynamic invariant detection. In this paper, we propose to mine common operational models, which are not always true in all observed traces, from a set of unverified tests based on mining predicate rules. Specifically, we collect values of simple predicates at runtime and then generate and evaluate predicate rules as potential operational models after running all the tests. We then select tests that violate the mined predicate rules for result inspection. Preliminary results on the Siemens suite and the grep program show the effectiveness of our approach.

1. Introduction

Testing involves three main steps: generating a set of test inputs, executing those inputs on the program under test, and then checking whether the test executions reveal faults. Among these steps, test-input generation and test-result inspection require intensive human labor. Recently, there have been various practical approaches on automatic test-input generation [10, 12, 13]. However, test-result inspection still remains a largely manual task (unless a priori specification is available, which is uncommon). Sometimes developers can use certain test oracles, such as assertions, user-defined contracts [10], and memory monitoring tools. But these oracles either require a large amount of manual effort to construct, or are limited in checking specific kinds of faults.

Test selection helps to reduce this cost by selecting a small subset of tests that are likely to reveal faults. A promising approach is to dynamically mine operational models as potential test oracles and then select tests that violate them. Existing approaches such as DIDUCE [4], Jov [14], and Eclat [9] mine operational models from verified passing tests based on dynamic invariant detection. For example, Jov and Eclat mine operational models using Daikon [3] from a set of manually written passing unit tests whose results are verified with manually written assertions. Due to nontrivial effort for writing the assertions, the number of these existing passing unit tests is often limited. It is well known that operational models mined from a limited number of data samples could be noisy and thus many model violations could be false positives. DIDUCE mines models from normal execution of long-running applications and relaxes the models gradually. At the beginning of a program run, many presumed operational models may be violated and a violation that reveals a fault can be overwhelmed by the noise.

In this paper, we propose to mine common operational models, which are not always true in all observed traces, from a (potentially large) set of unverified tests. Our approach does not require an existing set of verified tests and can avoid the noise caused by a small number of data samples. As a common operational model is not always true over the whole set of tests, the type of Daikon inference techniques does not work anymore. Alternatively, we may generate and collect all the potential models at runtime and evaluate them after running all the tests. However, such an approach can incur high runtime overhead if Daikon-like operational models, which are in a large number, are used.

To mine common operational models efficiently, we propose an approach based on mining predicate rules. Specifically, we collect values of simple predicates at runtime and then generate and evaluate predicate rules as potential operational models after running all the tests. We use the predicate schemes of Cooperative Bug Isolation (CBI) tools [6], which have been used in statistical debugging. A predicate rule is an implication relationship between predicates. Before describing the procedure of our approach, we first explain the motivation of mining predicate rules as potential operational models with an example.

```
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

Figure 1 shows an example program, the associated predicates, some tests, and their corresponding true predicates. Assume that there is an "off-by-one" fault in Line 4. The program would fail if $x > 0 \wedge y \geq x$. The program is trivial but it illustrates a case where a failure is caused by a fault under some triggering conditions. In passing tests, the program should satisfy a precondition $x \leq 0 \vee y < x$, which is a real operational model. We observe that a failure is not likely to be predicted by the violation of a single predicate. Each of $P1 \sim P4$ is not a necessary condition to satisfy the real operational model, and their violations may not indicate failures. On the other hand, the predicate rule $P1 \Rightarrow P4$ corresponds to a precondition $x \leq 0 \vee y \leq x$, which is similar to and weaker than the real operational model. Since $P1 \Rightarrow P4$ is 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. This example shows that predicate rules, which model relationships between predicates, may be good approximations of real operational models (not necessarily the same as the real models) to be used in test selection.

The procedure of our approach is as follows. We collect two values of a predicate y: true and false, which are respectively denoted as y and $!y$ for simplicity. For each predicate y, we mine the rules $X \Rightarrow y$ and $X \Rightarrow !y$, where X is a conjunction of other predicates. These rules model the implication relationships between predicates. A rule $X \Rightarrow y$ is evaluated by its confidence, which is defined as the ratio between the number of tests that satisfy $X \wedge y$ and the number of tests that satisfy X. When a rule's confidence is not equal to 1, the higher the rule's confidence is, the more suspicious its violations are in indicating a failure. We then select tests that violate the mined predicate rules for result inspection.

We have conducted a set of experiments on the Siemens suite and the *grep* program in the SIR repository [1]. Preliminary results show that our approach is effective in selecting a small subset of tests that have high fault-detection capability.

2. Related Work

The implementation of our approach is based on the Cooperative Bug Isolation (CBI) tools [6]. CBI uses lightweight instrumentation to collect feedback reports that contain truth values of predicates (simple Boolean expressions at various program points) in executions. There exist several statistical-debugging approaches that use CBI to collect many such feedback reports and then find predicates that are predictive of observed failures such as crashing. Moreover, Nainar et al. [7] showed that the conjunctions of two predicates may be predictive of observed failures and can be used to help fault localization. Correspondingly, the predicates may follow some rules in the passing tests. This study inspires our work to mine predicate rules as potential operational models to select tests. Different from previous work on statistical debugging, our approach is based on unsupervised learning because all tests are unverified or unlabeled.

There exist a number of approaches for test selection based on mining operational models. Xie and Notkin [14] developed an operational violation approach called Jov for unit-test selection. They mined operational models using Daikon [3] from a set of manually written passing unit tests and selected automatically generated test inputs that violated the operational models. Pacheco and Ernst [9] developed a similar tool named Eclat, which further distinguishes illegal and fault-revealing inputs with some strategies. Hangal and Lam [4] developed DIDUCE that extracts operational models dynamically from long-running program executions. These existing approaches mine operational models from verified passing tests, whereas our approach mines common operational models from a set of unverified tests.

There also exist some approaches for test selection based on mining algebraic models or clustering. Xie and Notkin [15] developed an approach for automatically identifying special and common unit tests based on algebraic models. Their approach selects a test as a special test if the test exercises a certain program behavior that is not exhibited by most other tests. Although our approach shares a similar rationale with their approach, our approach mines operational models instead of algebraic models, which are applicable only in object-oriented unit testing. Dickinson et al. [2] used clustering analysis to partition executions based on structural profiles, and employed sampling techniques to select executions from clusters for observations. Our approach not only selects tests but also mines potential operational models that may guide result inspection.

There has been a lot of work on regression test selection [8] and test case prioritization [11]. Although these techniques also select a small number of tests that are likely to reveal (regression) faults, their objectives are quite different from ours. These techniques aim at reducing the time of running test cases, while our work aims at reducing the effort of result inspection when test oracles are not available.

3. Approach

This section presents the proposed test selection approach. We first mine predicate rules as potential operational models and then select tests that violate them for result inspection.

3.1. Mining Predicate Rules

We use CBI tools [6] to instrument the programs under test and collect values of predicates. In particular, we use two schemes:

- branches: At each conditional (branch), two predicates are tracked, indicating whether the true or false branches were ever taken.
- returns: At each scalar-returning function call site, three predicates are tracked: whether the returned value is < 0, $== 0$, or > 0.

For each predicate y, we mine the rules $X \Rightarrow y$ and $X \Rightarrow !y$, where X is a conjunction of other predicates. These rules model implication relationships between predicates. A program that is not of poor quality should pass most of the tests. Therefore, the predicate rules mined from a set of unverified tests may be similar to the real models in passing tests. To reduce complexity, our current implementation mines only the rules $x \Rightarrow y$ and $x \Rightarrow !y$, where x is a single predicate. For each predicate y and any other predicate x, we generate the rules $x \Rightarrow y$ and $x \Rightarrow !y$ for evaluation. We plan to use advanced data mining techniques such as association rule mining to mine more general rules in our future work.

There may be a large number of predicate rules. We are interested only in the rules that are likely to be true oracles and are violated by some tests. To evaluate the likelihood of a rule to be a true oracle, we use the concept of confidence. The confidence of $X \Rightarrow y$ is defined as the ratio between the number of tests that satisfy $X \wedge y$ and the number of tests that satisfy X. The confidence of $X \Rightarrow !y$ is defined as the ratio between the number of tests that satisfy $X \wedge !y$ and the number of tests that satisfy X. We do not consider the absolute frequency of a rule since a failure may be reflected by the violation of a non-frequent rule. If a rule's confidence is 1, it can be omitted as there is no violation of this rule. We then select a subset of rules that have high confidences. More specifically, for each predicate y, we select the most confident rule $X \Rightarrow y$ and the most confident rule $X \Rightarrow !y$. Another possible way is to select the rules whose confidences are higher than a preset threshold, whose value may be application-dependent.

3.2. Test Selection

Given a set of predicate rules, selecting all the tests that violate any of them may result in a large subset of the tests. Instead, we select only a small subset of the tests that violate all the predicate rules at least once. We select tests in a way that the most confident rules are violated by the selected tests first. Initially, the set of selected tests is empty. We sort the selected predicate rules in the descending order of confidence. From the top to bottom, if a rule is not violated by any of the previously selected tests, we select the first test that violates the rule. Finally, in a greedy way all the selected rules can be violated by the selected tests. We also rank the selected tests in the order of selection.

4. Preliminary Results

We have implemented the proposed approach and applied it to select tests in the Siemens suite [5] and the *grep* program [1]. We next describe the experimental subjects and the preliminary results.

4.1. Subjects

The first subject is the well known Siemens suite [5]. The Siemens suite contains 130 faulty versions of 7 programs: print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. The programs range in size from 170 to 540 lines and the numbers of tests prepared by previous researchers range from 1,052 to 5,542. Each faulty version has one manually injected fault. The 130 faulty versions simulate a wide spectrum of realistic faults.

The second subject is the *grep* program, which is a unix utility to search a file for a pattern. The source code of version 2.4.1 is downloaded from the Subject Infrastructure Repository [1]. It includes 13,358 lines of C code. The downloaded software package also contains a suite of 470 test cases and 12 faults. Only 3 faults can be detected by some test case in the suite in our environment. The three versions with the faults are then used in our experiments.

4.2. Results

For test selection, we measure the number of selected tests and their ability to reveal faults. A set of tests is said to reveal a fault if the faulty program fails on one or more of the tests. We want to reveal as many faults as possible with a small number of selected tests. Table 1 shows the results of our approach compared with the original test set. We also use random sampling to select the same number of tests. The random sampling is run 5 times and the average result is presented as the baseline.

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

We observe that our approach can select a small subset of tests that are likely to reveal faults. On average, only 1.53% (45/2945) of the original tests are needed to be checked. Despite small sized, the selected tests can still reveal 74.6% (97/130) of the faults, while the results of random sampling technique can reveal only 45.4% (59/130) of the faults. We also observe that for all the seven programs, our approach is consistently better than the random sampling approach.

We next describe the experimental results on the three versions of the *grep* program. The three versions 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. In addition, for each version, there is at least one failing test ranked in top 20. We also randomly select 20 tests for each version. In the 5 times of random selection, the selected tests never reveal the faults of the first two versions but always reveal the faults of the third version. Overall, our approach is effective in selecting tests that have high fault-detection capability.

5. Conclusions and Future Work

We have proposed a novel approach for test selection without a priori specification. We mine common operational models, which are not always true in all observed traces, from a set of unverified tests based on mining predicate rules. Specifically, we collect values of simple predicates at runtime and then generate and evaluate predicate rules as potential operational models after running all the tests. We then select tests that violate the mined predicate rules for result inspection. Preliminary results on the Siemens suite and the *grep* program show that our approach is effective in selecting tests that have high fault-detection capability.

We plan to pursue several future directions for our new approach. First, we plan to combine our approach with automatic test generation tools. Our current experiments are based on existing test suites that are well prepared. It is valuable to investigate how our approach can work on automatically generated test sets. In addition, we plan to explore how the mined operational models can help guide test generation. Second, we plan to study the characteristics of mined common operational models and compare them with invariants mined by Daikon. It is also interesting to investigate differences between common operational models mined from faulty versions and those mined from correct versions. Third, we currently mine only rules containing two predicates. We plan to explore mining more general rules containing several predicates.

Acknowledgments

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4158/08E), and partially supported by US ARO grant W911NF-08-1-0443 and NSF grant CCF-0725190.

References

[1] `http://sir.unl.edu/php/index.php`.

[2] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *ESEC / SIGSOFT FSE*, pages 246–255, 2001.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[4] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, pages 291–301, 2002.

[5] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pages 191–200, 1994.

[6] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[7] P. A. Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *ISSTA*, pages 5–15, 2007.

[8] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *SIGSOFT FSE*, pages 241–251, 2004.

[9] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, 2005.

[10] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007.

[11] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Software Eng.*, 27(10):929–948, 2001.

[12] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[13] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, pages 97–107, 2004.

[14] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *ASE*, pages 40–48, 2003.

[15] T. Xie and D. Notkin. Automatically identifying special and common unit tests for object-oriented programs. In *ISSRE*, pages 277–287, 2005.