2014 IEEE 25th International Symposium on Software Reliability Engineering

An Empirical Study on the Scalability of Selective Mutation Testing

Jie Zhang, Muyao Zhu, Dan Hao, Lu Zhang
Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
Institute of Software, School of EECS, Peking University, China
{zhangjie12,zhumy12,haod,zhanglu}@sei.pku.edu.cn

***Abstract*—Software testing plays an important role in ensuring software quality by running a program with test suites. Mutation testing is designed to evaluate whether a test suite is adequate in detecting faults. Due to the expensive cost of mutation testing, selective mutation testing was proposed to select a subset of mutants whose effectiveness is similar to the whole set of generated mutants. Although selective mutation testing has been widely investigated in recent years, many people still doubt whether it can suit well for large programs. To study the scalability of selective mutation testing, we systematically explore how the program size impacts selective mutation testing through four projects (including 12 versions all together). Based on the empirical study, for programs smaller than 16 KLOC, selective mutation testing has surprisingly good scalability. In particular, for a program whose number of lines of executable code is E, the number of mutants used in selective mutation testing is proportional to E^c, where c is a constant whose value is between 0.05 and 0.25.**

I. Introduction

Software testing plays an important role in ensuring software quality. By running the software under test with test suites, testers discover faults when they observe unexpected behavior. A well-designed test suite may detect much more faults than a poorly-designed one. To evaluate a test suite's ability of detecting faults, mutation testing [8], [12] is proposed. In mutation testing, based on the original program a large number of buggy programs (the so-called *mutants*) are generated, and then executed by using every test case from the test suite. The more mutants a test suite kills[1], the more adequate the test suite is.

Although mutation testing is useful in measuring the adequacy of a test suite, it is usually very expensive to execute the test suite against all mutants. For example, in the work of Zhang et al. [42], a small program whose number of lines of code is only 513 actually has 23,847 mutants. It is costly to run test cases on such a large number of mutants. To handle this problem, Mathur [24] proposed selective mutation testing, which uses a subset of mutants to achieve the similar effectiveness as traditional mutation testing. In other words, if a test suite adequate[2] to a subset of mutants is also adequate to the whole set of mutants, this subset is regarded as sharing the same effectiveness as the whole set of mutants in evaluating test suites. That is, this subset is sufficient in representing the whole set. To measure to what extent a subset of mutants represents the whole set of mutants, we define the sufficiency score for any subset of mutants. The larger sufficiency score a subset has, the more representative the subset is.

To find a subset of mutants that are sufficient in representing the whole set of mutants, selective mutation testing [30], [2], [26] has been widely studied. However, existing research is usually evaluated using only small programs [4], [32], [42]. Therefore, many researchers doubt whether selective mutation testing can be applied to large programs [42], [18]. That is, mutation testing, especially selective mutation testing, may have the scalability concern in practice.

In this paper, we explored the scalability of selective mutation testing by performing an empirical study on 12 versions of 4 projects. For each subject, we first generated its whole set of mutants and produced a set of non-equivalent mutants by removing the mutants whose behavior is the same as the original program's. Then we constructed a subset of mutants by randomly selecting mutants from the set of non-equivalent mutants so that the subset of mutants achieves some specified sufficiency score (i.e., 90%-99%). To learn the scalability of selective mutation testing, we analyzed how the program size and the total number of non-equivalent mutants affect the effectiveness of selective mutation testing based on the subsets of selected mutants. Through the empirical study, selective mutation testing has surprisingly good scalability for programs up to 16 KLOC. In particular, as either the program size or the total number of non-equivalent mutants increases, the number of selected mutants increases slowly, while the proportion of selected mutants decreases. Quantitatively, for any program whose number of lines of executable code is E, its proportion of selected mutants used in selective mutation testing is proportional to E^b where b is a constant whose value is between -0.95 and -0.75, and the number of mutants in the subset is proportional to E^c where c is a constant whose value is between 0.05 and 0.25.

Our paper makes the following contributions:

- An empirical study on exploring how the program size and the number of non-equivalent mutants impact the proportion and the number of mutants used in selective mutation testing. To our knowledge, this is the first

[1] A test case kills a mutant when it causes the behavior of the mutant to differ from the original program. A test suite kills a mutant when at least one test case kills the mutant.

[2] That is, the test suite can kill all mutants in this subset.

1071-9458/14 $31.00 © 2014 IEEE
DOI 10.1109/ISSRE.2014.27

empirical study on systematically studying the scalability of selective mutation testing.
- Empirical evidence that selective mutation testing has good scalability.

The rest of this paper is organized as follows: Section II introduces the background and motivation. Section III introduces the details of the empirical design. Section IV presents the results and analysis of the empirical study. Section V discusses some related issues. Section VI discusses the related work. Section VII concludes.

II. Background and Motivation

In this section, we present some background for mutation testing and selective mutation testing in Section II-A and Section II-B, and the motivation for our work in Section II-C.

A. Mutation Testing

Mutation testing was firstly proposed to evaluate whether a test suite is adequate in detecting program faults [12], [8]. In mutation testing, a large number of mutants are generated by changing the original program's code according to some basic rules (i.e., mutation operators[3]). Then, a test suite T is executed against each mutant as well as the original program. A test case t in T is said to kill a mutant m when $O(m,t)$ (i.e., the output of the mutant m with t being the input) is different from $O(p,t)$ (i.e., the output of the original program p with t being the input). A test suite is said to kill a mutant when at least one test case in the test suite kills the mutant. Note that there may be some mutants which are functionally equal to the original program although the source code has been changed. That is, no test suites can kill such mutants as their outputs are always the same as the original program's. These mutants are called *equivalent mutants*, which are considered of no help in evaluating any test suite. All the rest of mutants except for equivalent mutants are called *non-equivalent mutants*. For any program P and its test suite T, the adequacy of T (denoted as $S(T)$) can be evaluated by *mutation score*, which is defined based on the set of non-equivalent mutants generated for P (denoted as M_{ne}) and the set of mutants killed by T (denoted by $M_k(T)$) as follows:

$$S(T) = \frac{\#M_k(T)}{\#M_{ne}} \quad (1)$$

In this formula, $\#M_k(T)$ represents the number of mutants killed by T, and $\#M_{ne}$ represents the number of non-equivalent mutants. $S(T)$ is between 0 and 1. The closer $S(T)$ is to 1, the more adequate T is. That is, the more mutants a test suite can kill, the more adequate the test suite is in detecting faults.

[3] In mutation testing, mutation operators are used to define some operations like statement deletion or replacing some boolean operators by other boolean operators.

B. Selective Mutation Testing

As a program may have a large number of mutants, it is usually very expensive to execute a test suite on all generated mutants in mutation testing, even for small programs. To reduce such cost, selective mutation testing was proposed to use a subset of mutants as the representative of the whole set guaranteeing that the subset has similar ability of evaluating test suites as the whole set [24].

Basically, existing work in selective mutation testing can be classified into operator-based mutation selection and random mutation selection. The former first chooses a subset of mutation operators and then generates all mutants using these selected operators, whereas the latter randomly selects mutants from the whole set of mutants generated by all mutation operators [42]. Existing work on selective mutation testing mainly focuses on presenting various approaches to selecting representative mutants.

As this paper investigates how the program size impacts the number of mutants needed in selective mutation testing, we used random mutation selection based on the following reasons. First, Zhang et al. [42] showed that operator-based mutant selection is not superior to random mutant selection, which means that these two techniques can be regarded as equal in effectiveness. Second, in random mutant selection each mutant is selected with equal probability whereas in operator-based mutant selection mutants are selected based on mutation operators so that the selection process used in this paper tends to have no bias.

C. Motivation

Most prior work on selective mutation testing is evaluated on small programs. For example, Offutt et al. [27] used 10 Fortran programs with totally 206 statements. Barbosa et al. [4] conducted their experiments with 27 C programs with totally 619 statements. Namin et al. [32] used seven C programs with totally 2188 statements. That is, existing work on selective mutation testing tends to use small programs in the evaluation because researchers suspect that it may be much more costly to run a large program than a small program in mutation testing, as a large program may have a large number of mutants and finally produce a large subset of representative mutants.

Although some work on selective mutation testing is evaluated on some larger programs, none of these work systematically studied whether selective mutation testing has the scalability concern. In particular, Gligoric et al. [11] used several large programs with more than 50,000 lines of code. However, they chose only concurrent programs as they explored selective mutation for concurrent mutation operators. Zhang et al. [40] used 11 real-world projects whose number of lines of code is from 2681 to 36910, but they did not systematically study the scalability problem in selective mutation testing.

To apply mutation testing in practice, it is important to learn whether selective mutation testing has the scalability problem. Real-world programs are usually much larger than the programs used in the prior evaluation. Therefore, it is

necessary to learn how the cost of selective mutation testing changes as the program size increases so as to estimate whether it is possible to apply selective mutation testing in practice. If the number of mutants used in selective mutation testing does not increase quickly as the program size increases, the cost of selective mutation testing for a large program may be acceptable and thus selective mutation testing can be applied to practice. If the number of mutants used in selective mutation testing increases quickly as the program size increases, the cost of selective mutation testing for a large program tends to be unacceptable and thus selective mutation testing is not applicable in practice at all. In one word, it is necessary to learn whether selective mutation testing has the scalability issue.

III. Empirical Setup

In this section, we describe our empirical setup in details. In particular, we present the research questions in Section III-A, the subjects in Section III-B, the measurement in Section III-C, the process of our empirical study in Section III-D and the threats to validity in Section III-E.

A. Research Questions

This empirical study aims to investigate the scalability of selective mutation testing. That is, the research question of the empirical study is: how good the scalability of selective mutation testing is.

As selective mutation testing aims to select a subset of mutants from the set of generated mutants, the number and the proportion of selected mutants tend to be dependent on the total number of non-equivalent mutants, which is intuitively related to the program size. Therefore, in the empirical study we conducted the following three studies to answer the research question on scalability.

- In the first study, we explored how the number of non-equivalent mutants impacts the number and the proportion of selected mutants used in selective mutation testing.
- In the second study, we explored how the program size impacts the number and the proportion of selected mutants used in selective mutation testing.
- In the third study, we explored the quantitative relation between the program size and the number of selected mutants used in selective mutation testing, as well as the quantitative relation between the program size and the proportion of selected mutants used in selective mutation testing.

B. Subjects

To investigate the scalability of selective mutation testing, we collected a series of programs to incorporate more than one project for each program-size range. In particular, we collected more than 30 versions of four multi-version projects: *JGraphT*[4], *Time&Money*[5], *Barbecue*[6], and *XML Security*[7]. However, some versions cannot compile or execute normally in our environment, and some versions cannot be processed correctly by Javalanche 0.4[8]. At last, 12 versions of the four projects above are chosen in our study.

Table I
Basic Information of Subjects

Subject	Version	Size	Test Case	Mutants	
				Total	Non-equivalent
Time&Money	0.2	1059	104	1050	761
	0.3	1403	146	1413	1124
Barbecue	1.5.0-alpha	4689	51	16318	563
JGraphT	0.5.0	2393	48	1122	692
	0.5.1	3248	63	1644	1039
	0.5.2	3609	57	1666	1051
	0.5.3	3670	57	1663	1032
	0.6.0	4098	68	2095	1344
	0.7.1	8829	84	3610	2360
	0.7.3	10141	102	4576	2942
XML Security	1.2.1	16218	97	8404	2687
	1.3.0	15158	95	7899	2731

We chose multi-version projects as such projects usually have several programs whose sizes increase gradually. Following some of the previous work [25], each version of a project is viewed as a subject in the empirical study. This is reasonable, because the code base of one project typically changes significantly during its evolution. Take *JGraphT* as an example: in its 7 versions, the lines of code change enormously from 2,393 LOC to 10,141 LOC. New code produces new mutants, thus, different versions of one project could demonstrate much diversity themselves.

In the empirical study, each subject has been suited with a test suite, which is collected during development. Following the same procedures of previous studies on mutation testing [32], [42], [40], in this empirical study, the mutants that cannot be killed by any test case from the given test suite are taken as equivalent mutants[9]. By removing these equivalent mutants, we got the set of non-equivalent mutants, which can be killed by the given test suite.

The basic information of the 12 subjects is presented in Table I. Column "Size" lists the number of lines of executable code of each subject by removing blank statements and comments. Column "Test Case" lists the number of test cases in the test suite. Column "Total" lists the total number of generated mutants, and column "Non-equivalent" lists the total number of non-equivalent mutants. From the table, for the 12 subjects, the program sizes range from 1,059 to 16,218.

C. Measurement

In selective mutation testing, to measure how good a subset of mutants represents the whole set of mutants (i.e., how sufficient the subset is in evaluating test suites), we adopted the same metric as most researchers used [35], [36], [42], which is to use the mutation score of a test suite that is adequate for a subset of mutants. In particular, for a subset of mutants

[4] http://sourceforge.net/projects/jgrapht/
[5] http://sourceforge.net/projects/timeandmoney/
[6] http://sourceforge.net/projects/barbecue/
[7] http://www.aleksey.com/xmlsec/

[8] We used Javalanche 0.4 as a mutation tool in our empirical study, which is available at http://www.javalanche.org/.
[9] Conceptually, equivalent mutants refer to the mutants whose behavior is always the same as the original program.

(denoted as M_s), we constructed an adequate test suite T_s to kill all the mutants in M_s and used the mutation score of this test suite on the whole set of non-equivalent mutants (denoted as M_{ne}) to measure to what extent the subset M_s represents the whole set M_{ne} on evaluating T_s. That is, the mutation score of a test suite T_s, which is constructed to be adequate to M_s, is defined by the following formula. In the formula, $\#M_k(T_s)$ represents the number of mutants killed by T_s among the whole set of generated mutants, and $\#M_{ne}$ represents the total number of non-equivalent mutants in the whole set.

$$E(M_s, T_s) = \frac{\#M_k(T_s)}{\#M_{ne}} \tag{2}$$

To learn the sufficiency of a subset of mutants in representing the whole set of mutants on evaluating **any test suite**, we define the sufficiency score for any subset of mutants. In particular, for a subset of mutants M_s, its sufficiency score denoted as $E(M_s)$ is defined as follows, where n is the total number of test suites (i.e., T_{s_1}, T_{s_2}, ..., T_{s_n}) used to measure the sufficiency of M_s.

$$E(M_s) = \frac{\sum_{i=1}^{n} E(M_s, T_{s\_i})}{n} \tag{3}$$

From this formula, for a subset of mutants M_s, we use the average mutation score of many adequate test suites to measure the sufficiency of M_s so as to alleviate the bias of test suites. As the value of $E(M_s, T_{s\_i})$ is between 0 and 1, the value of $E(M_s)$ is also between 0 and 1. Moreover, a subset M_s with larger $E(M_s)$ tends to be more sufficient in representing the whole set of mutants. Ideally, when $E(M_s)$ is equal to 1, the corresponding subset is as sufficient as the whole set in evaluating a test suite.

D. Experimental Procedure

To study the scalability of selective mutation testing, we performed an empirical study on all the subjects based on the following procedure.

For each subject, we first constructed subsets with the smallest number of mutants that achieve some specified sufficiency score based on the following steps.

- **Step 1: Get non-equivalent mutants.** First, we used all mutation operators of Javalanche to generate the whole set of mutants. Then we ran all test cases on these generated mutants and got a set of mutants that can be killed by these test cases. Following the same procedures as previous work [32], [42], [40], in this empirical study, we took the mutants that cannot be killed by any test case as equivalent mutants. That is, we regarded the mutants that can be killed by some test cases as the non-equivalent mutants. The set of non-equivalent mutants is denoted as M_{ne}.
- **Step 2: Construct a subset of mutants.** In this step, we randomly selected mutants from the set of non-equivalent mutants to produce the smallest subset of mutants that achieves some specified sufficiency score, which is set to be 90%, 91%, . . ., 98%, and 99%. In particular, for every specified sufficiency score, we first randomly selected one mutant from the set of non-equivalent mutants and added it to the subset of selected mutants M_s. Then we constructed 20 test suites (denoted as T_{s_i} where $i = 1, 2, \ldots, 20$) each of which is constructed as follows: including one test case at a time until all the mutants in the subset are killed. We ran the set of non-equivalent mutants M_{ne} by using these test suites respectively, and recorded the set of killed mutants $M_k(T_{s\_i})$ for each test suite T_{s_i}. Based on Formula 2 and Formula 3, we calculated the sufficiency score of M_s. If the sufficiency score $E(M_s)$ is smaller than the specified sufficiency score, we enlarged M_s by randomly adding one mutant at a time, and repeated the preceding process. Otherwise, we completed constructing a subset of mutants that achieves the specified sufficiency score.
- **Step 3: Get the average size of subsets.** To avoid biased results due to random selection, we repeated the second step for 20 times and thus produced 20 subsets of mutants each of which achieves the specified sufficiency score. Then we computed the average number of mutants in these subsets. This average number is used to represent how many mutants are needed in selective mutation testing to achieve some specified sufficiency score.
- **Step 4: Get the average proportion of subsets.** As selective mutation testing aims to reduce the cost of mutation testing by removing some mutants, it is necessary to learn the proportion of the mutants in the selected subset to the whole set of non-equivalent mutants. Therefore, we further divided the average number of mutants in subsets by the total number of non-equivalent mutants and got the average proportion of subsets.

Based on the data collected following the preceding process, we conducted three studies as follows. In the first study, we analyzed how the number of non-equivalent mutants impacts the number and the proportion of selected mutants in the subset, which is used in selective mutation testing. In the second study, we analyzed how the program size impacts the number and the proportion of selected mutants in the subset, which is used in selective mutation testing. In the third study, we quantitatively analyzed the relation between the program size and the number of mutants in the subset, as well as the relation between the program size and the proportion of mutants in the subset.

E. Threats to Validity

In this section we discuss some threats to validity in the empirical study. The threat to internal validity lies in the implementation of the empirical study. To reduce the possible faults in implementation, the authors reviewed the code of the empirical study carefully. The threat to external validity mainly lies in subjects and the mutation tool. Although we chose 12 versions of four real-world Java programs from different domains, these subjects may be not representative of other projects, especially in other programming languages.

To reduce this threat, in the future, we will conduct more experiments using more projects of different sizes (i.e., especially large projects) and in other programming languages like C/C++. Mutation tool directly affects the number and quality of mutants. In our empirical study, we used Javalanche as the mutation tool, but Javalanche may not be representative of other mutation tools. To reduce this threat, we will use other mutation tools to generate mutants in the future.

IV. Results and Analysis

In this section, we present the results and analysis for the three studies in Sections IV-A, IV-B, and IV-C. Finally, we summarize the findings in Section IV-D.

A. Study I

Figure 1 presents the proportion of selected mutants as the total number of non-equivalent mutants increases for some specified sufficiency score (i.e., from 90% to 99%) , where the horizonal axis represents the total number of non-equivalent mutants and the vertical axis represents the proportion of selected mutants. From this figure, when the total number of non-equivalent mutants increases from 500 to 3,000, the proportion of selected mutants in selective mutation testing decreases dramatically from 9% to 1%. That is, although a large program may generate a large number of non-equivalent mutants, only a very small proportion of them are needed in selective mutation testing.

Moreover, for some given set of non-equivalent mutants, when its sufficiency score increases, the proportion of selected mutants increases as well. However, the proportion is close when the sufficiency score is from 90% to 98%.

Furthermore, we draw Figure 2 to show the number of selected mutants as the total number of non-equivalent mutants increases for some specified sufficiency score, whose vertical axis represents the number of selected mutants used in selective mutation testing. From this figure, when the total number of non-equivalent mutants increases from 500 to 3,000, the number of selected mutants usually increases as well except when the number of non-equivalent mutants is around 2,600. We suspect the reason for this exception to lie in the structures of target programs (i.e., the two programs of *XML-Security*). That is, due to the structure of these programs, the generated mutants may be effective and thus few mutants are sufficient to represent the whole set of non-equivalent mutants in selective mutation testing.

B. Study II

Table II presents the number and proportion of selected mutants for programs of different sizes with some specified sufficiency score. As this study aims to explore the relation between the program size and the number as well as the proportion of mutants used in selective mutation testing, we ordered all the subjects by their size and removed their names for ease of observation. Figures 3 and 4 present the proportion of selected mutants and the number of selected mutants as the program size increases, respectively.

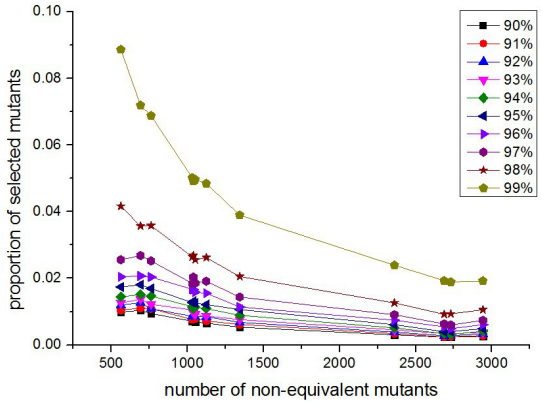

Figure 1. Between the number of non-equivalent mutants and the proportion of selected mutants

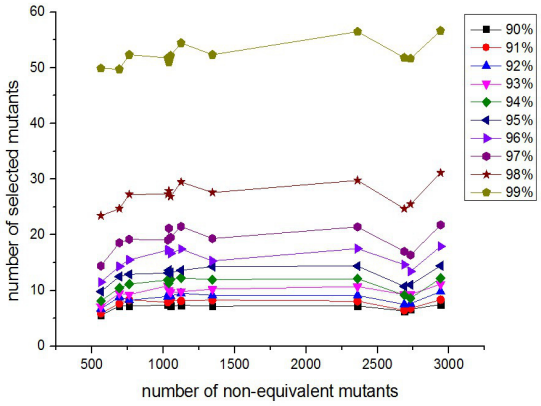

Figure 2. Between the number of non-equivalent mutants and the number of selected mutants

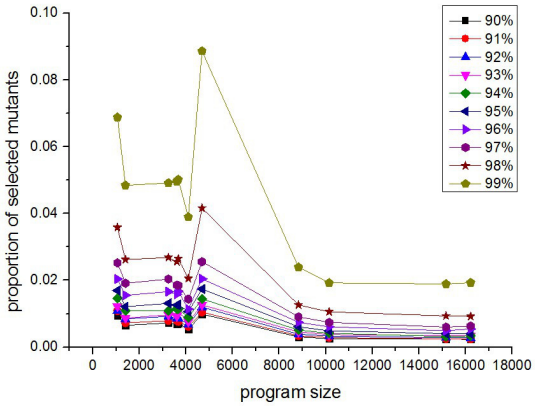

Figure 3. Between the program size and the proportion of selected mutants

Table II
RESULTS

Selected Mutants	Subject Size	Specified Sufficiency Score									
		90%	91%	92%	93%	94%	95%	96%	97%	98%	99%
Number of Selected Mutants	1059	7.18	8.3	8.26	9.26	11.16	12.90	15.5	19.18	27.24	52.34
	1403	7.34	8.12	9.50	9.78	12.24	13.62	17.44	21.48	29.46	54.44
	2393	7.18	7.64	8.82	9.48	10.46	12.50	14.32	18.56	24.68	49.74
	3248	7.52	8.20	9.54	10.04	11.30	13.64	17.26	21.18	27.86	51.04
	3609	7.20	8.32	9.00	9.68	12.00	13.22	16.66	19.50	26.86	52.16
	3670	7.38	7.88	8.92	10.84	11.88	13.14	17.26	19.06	27.28	51.76
	4098	7.16	8.28	9.14	10.22	11.94	14.28	15.30	19.32	27.58	52.34
	4689	5.56	5.88	6.72	7.10	8.12	9.80	11.50	14.40	23.42	49.88
	8829	7.22	8.08	9.10	10.70	12.10	14.38	17.52	21.42	29.76	56.46
	10141	7.48	8.34	9.80	11.12	12.18	14.42	17.92	21.76	31.12	56.64
	15158	6.62	6.80	7.64	9.30	8.60	11.02	13.42	16.36	25.50	51.64
	16218	6.30	6.44	7.50	9.28	9.16	10.74	14.64	17.00	24.68	51.82
Proportion of Selected Mutants	1059	0.94%	1.09%	1.09%	1.22%	1.47%	1.70%	2.04%	2.52%	3.58%	6.88%
	1403	0.65%	0.72%	0.85%	0.87%	1.09%	1.21%	1.55%	1.91%	2.62%	4.84%
	2393	1.04%	1.10%	1.27%	1.37%	1.51%	1.81%	2.07%	2.68%	3.57%	7.19%
	3248	0.72%	0.79%	0.92%	0.97%	1.09%	1.31%	1.66%	2.04%	2.68%	4.91%
	3609	0.69%	0.79%	0.86%	0.92%	1.14%	1.26%	1.59%	1.86%	2.56%	4.96%
	3670	0.72%	0.76%	0.86%	1.05%	1.15%	1.27%	1.67%	1.85%	2.64%	5.02%
	4098	0.53%	0.62%	0.68%	0.76%	0.89%	1.06%	1.14%	1.44%	2.05%	3.89%
	4689	0.99%	1.04%	1.19%	1.26%	1.44%	1.74%	2.04%	2.56%	4.16%	8.86%
	8829	0.31%	0.34%	0.39%	0.45%	0.51%	0.61%	0.74%	0.91%	1.26%	2.39%
	10141	0.25%	0.28%	0.33%	0.38%	0.41%	0.49%	0.61%	0.74%	1.06%	1.93%
	15158	0.24%	0.25%	0.28%	0.34%	0.31%	0.40%	0.49%	0.60%	0.93%	1.89%
	16218	0.23%	0.24%	0.28%	0.35%	0.34%	0.40%	0.54%	0.63%	0.92%	1.93%

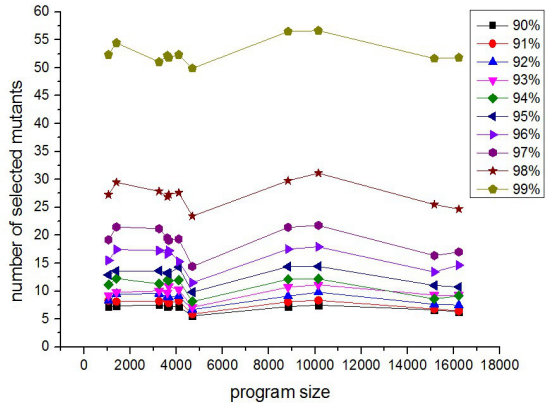

Figure 4. Between the program size and the number of selected mutants

From Table II and Figure 3, the proportion of selected mutants changes dramatically (including increases and decreases) when the program size increases from 1,000 to 10,000. However, when the program size increases from 10,000, the proportion of selected mutants decreases from 2% to 1%. When the program size increases from 10,000, the proportion of selected mutants becomes steady (i.e., with 0.23% being the smallest proportion). That is, for large programs, the proportion of selected mutants needed in selective mutation testing tends to be very small. Therefore, to achieve a high sufficiency score (i.e., from 90% to 99%), developers only need to select a small proportion (i.e., smaller than 1%) of mutants for a large program. Moreover, as the program size increases, such proportion tends to decrease. This observation indicates that selective mutation testing has good scalability and may be applied to large programs with low cost.

When the sufficiency score increases from 90% to 98%, the proportion of selected mutants for any program increases slightly. Moreover, from Figure 3, the topper curve (i.e., the proportion results for a large sufficient score) always changes faster than the lower one (i.e., the proportion results for a small sufficient score). This observation indicates that the proportion of selected mutants decreases fast for a large sufficient score.

From Table II and Figure 4, the number of selected mutants does not increase dramatically when the program size increases. That is, even for large programs, a small number of mutants can well representative the whole set of mutants. This observation also indicates that selective mutation testing has good scalability.

Note that both for Figure 3 and Figure 4, when the program size increases from 1,000 to 10,000, the proportion or the number of selected mutants changes dramatically. This observation is as expected, because the proportion and the number of selected mutants tends to be dependent on the total number of non-equivalent mutants. While in our study the number of non-equivalent mutants varies dramatically for small programs due to the following possible reasons. First, in mutation testing, mutants are generated by applying mutation operators to target programs under test. Due to the characteristics of mutation testing, some types of statements (e.g., conditional statements) are likely to generate more mutants than other types of statements (e.g., simple assignment statements). The total number of mutants generated for small programs may vary dramatically because some small programs may contain many statements of the former types whereas some programs may not. However, as large programs contain a large number of statements, these programs are more likely to contain statements of the former types as well as statements of the latter types. As the total number of mutants may vary

dramatically for small programs, the number of non-equivalent mutants may vary dramatically as well. Second, the original test suite of several subjects may be of lower quality than the test suites for other subjects, causing fewer non-equivalent mutants. Take *Barbecue* as an example, the test suite kills only 563 out of 16318 mutants. The ratio (i.e., 563/16318 = 3.45%) is much lower than the ratios for other subjects (all higher than 31.90%).

C. Study III

From Figure 1, we suspect that there is close non-linear association between the proportion of selected mutants and the total number of non-equivalent mutants. To verify our hypothesis, we conducted curve fitting for Figure 1 using OriginPro 9[10]. Curve fitting [21] is a process of constructing a curve or mathematical function that best fits a set of data so as to summarize the relationship between variables. Based on our observation on this figure, in curve fitting we tried three non-linear functions: exponential functions, logarithm functions, and power functions. We finally found out that one power function, $Y = a * X^b$, fits Figure 1 best, which is presented in Table III. In Table III, the R^2 value denotes how good the function fits the set of data (i.e., experimental results in this study), which is between 0 and 1. The larger it is, the better the function fits the data. From the table, for function $Y = a * X^b$, the values of R^2 are very close to 1 for each specified sufficient score. Therefore, the function $Y = a * X^b$ can be used to represent the association between the program size and the proportion of selected mutants.

For each specified sufficient score, we present the values of parameters a and b in its best-fit function $Y = a * X^b$ by Table IV. When the sufficiency score increases from 90% to 98%, the values of a increase slowly, whereas when the sufficiency score increases from 98% to 99%, the values of a increase dramatically. This observation is consistent with our observation in the first two studies. That is, the proportion of selected mutants does not increase dramatically when the sufficiency score increases from 90% to 98%, but increases dramatically when the score increases from 98% to 99%. We suspect the reason to be that the sufficiency score 99% is very close to 100% and thus it is hard and costly to achieve such a high score.

When the sufficiency score increases from 90% to 99%, the values of b remain stable with slight changes. In particular, the values of b are between -0.7628 and -0.9390. That is, exponent b in the function $Y = a * X^b$ can be viewed as unchangeable for any specified sufficient score no smaller than 90%. Therefore, for any program whose number of non-equivalent mutants is X, the proportion of the selected mutants used in selective mutation testing (denoted as Y) is proportional to X^b, where b is a constant between -0.95 and -0.75.

As people are usually concerned about how many mutants are needed in selective mutation testing for a specified program size, we further quantitatively investigated the relation between the program size and the number of selected mutants as well as the relation between the program size and the proportion of selected mutants based on the former analysis.

Intuitively, the number of non-equivalent mutants is associated with the program size. Based on the empirical results, we found that the number of non-equivalent mutants is generally proportional to program size. Specifically, for any program whose number of lines of executable code is E, the number of non-equivalent mutants X is roughly about $0.3E$[11].

Based on the quantitative relation among the number of non-equivalent mutants, the program size, and the proportion of mutants used in selective mutation testing, we may induce the quantitative relation between the program size and the proportion of the selected mutants as follows. As $Y = a * X^b$ where b is a constant between -0.95 and -0.75 and $X \approx 0.3E$, then we got $Y = a * X^b \approx a * (0.3E)^b = a * (0.3^b) * E^b$. That is, for a program whose number of lines of executable code is E, the proportion of mutants used in selective mutation is proportional to E^b where b is a constant between -0.95 and -0.75.

If we use Z to represent the number of selected mutants, we got $Z/X = Y \approx a * (0.3^b) * E^b$. Therefore, $Z \approx a * (0.3^b) * E^b * 0.3E = a * (0.3^{b+1}) * E^{b+1}$. For ease of expression, we use c to denote $b + 1$. As b is a constant between -0.95 and -0.75, c is a constant whose value is between 0.05 and 0.25. Therefore, for a program whose number of lines of executable code is E, the number of mutants used in selective mutation testing is proportional to E^c, where constant c is between 0.05 and 0.25.

D. Summary

From our three studies, we found that selective mutation testing has surprisingly good scalability for programs up to 16 KLOC. In particular, we got the following findings all together.

- As the number of non-equivalent mutants increases, the number of selected mutants used in selective mutation testing increases slowly, while the proportion of selected mutants used in selective mutation testing decreases.
- As the program size increases, the number of selected mutants used in selective mutation testing increases slowly, while the proportion of the selected mutants used in selective mutation testing decreases.
- For any program whose number of lines of executable code is E, the number of selected mutants used in selective mutation testing is proportional to E^c where c is a constant between 0.05 and 0.25, and the proportion of selected mutants is proportional to E^b where b is a constant between -0.95 and -0.75.

[10] OriginPro is a scientific graphing and data analysis tool developed by OriginLab Corporation, which is available at http://www.originlab.com/index.aspx?go=Products/OriginPro

[11] Note that 0.3 is not a precise value proper for any program, but a roughly estimated value detected from all the 12 subjects.

Table III
THE BASIC INFORMATION OF THE BEST-FIT FUNCTIONS

Category	**Best-fit Function**	R^2									
		90%	91%	92%	93%	94%	95%	96%	97%	98%	99%
power	$Y = a * X^b$	0.9470	0.9227	0.9362	0.9286	0.9109	0.9359	0.9134	0.9265	0.9808	0.9973

Table IV
THE VALUES OF a AND b FOR EVERY FITTED CURVE OF $Y = a * X^b$

Function	Parameters	90%	91%	92%	93%	94%	95%	96%	97%	98%	99%
$Y = a * X^b$	a	2.4430	2.3287	2.4443	1.9283	2.5214	3.2951	2.9768	4.6559	9.9790	34.0318
	b	-0.8484	-0.8277	-0.8186	-0.7704	-0.7893	-0.8057	-0.7628	-0.7984	-0.8580	-0.9390

V. DISCUSSION

In this section, we discuss some issues in our empirical study.

First, the surprisingly good scalability of selective mutation testing makes it possible to apply selective mutation testing in practice. From the empirical study, for any program whose number of lines of executable code is E (i.e., E is smaller than 16 KLOC in our study), its number of mutants used in selective mutation testing is proportional to E^c where c is a constant whose value is usually between 0.05 and 0.25. That is, the number of mutants used in selective mutation testing increases slowly as the program size increases. Thus, the execution time of these mutants increases slightly. Furthermore, to speedup the execution of these selected mutants, we may use more than one processors to run these mutants simultaneously.

Second, the mutants are selected with the same probability in our empirical study similar to previous studies [37], [4], [42] by ignoring their possibly different cost. As shown by previous study [30], the cost of mutation testing mainly lies in the need to compile and execute a large number of mutants against each test case. As mutants are generated by different mutation operators, they are different from each other in the compiling and executing time. Therefore, when selecting mutants for mutation testing, it is important to consider the cost of mutants besides their sufficiency. In our future work, we will further explore the scalability problem of selective mutation testing by considering the cost of mutants.

Third, the quantitative relations among the number and the proportion of selected mutants, the program size, and the number of non-equivalent mutants may be used to construct prediction models for selective mutation testing. From this empirical study, we found there exists quantitative relations among the number of selected mutants, the proportion of selected mutants, the program size, and the number of non-equivalent mutants. Based on such quantitative relations, we can predict how many and what proportion of mutants are needed for any program to achieve a specified sufficient score. That is, the quantitative relations facilitate the practical usage of selective mutation testing.

Fourth, the proportion of mutants used in selective mutation testing is more related to the number of non-equivalent mutants rather than the program size. Intuitively, the proportion of selected mutants is both related to the number of non-equivalent mutants and the program size. However, from Figure 1 and Figure 3, fewer fluctuant points exist in the former figure than the latter figure. That is, the curve fitting between the proportion and the number of non-equivalent mutants is better than that between the proportion and the program size. This observation is reasonable as the program size is approximately proportional to the number of non-equivalent mutants, whereas the number of selected mutants is directly related to the number of non-equivalent mutants. However, the relation between the proportion of mutants used in selective mutation testing and the program size is more useful when selective mutation testing is applied in practice.

Finally, the original test suites are more proper than randomly generated test cases. We decided to use the original test suites with two reasons. First, as we described in Section III.B, we follow previous work to use the original test suites in our experiments. Second, the original test suites are more real and objective than test suites we created ourselves. This may help reduce some possible bias. It is possible that the quality of some original test suites may not be very high. However, from our empirical study, very few test suites are of low quality, and do not affect our conclusion.

VI. RELATED WORK

In this section we further introduce the related work on reducing the cost of mutation testing in Section VI-A and the main applications of mutation testing in Section VI-B.

A. Cost Reduction Techniques

Mutation testing was first proposed by Demillo et al. [8] and Hamlet [12]. As it is usually very expensive to execute test suites on too many mutants, many researchers focus on presenting various techniques to reduce the cost of mutation testing. In particular, we divide these techniques into two categories: techniques on reducing the number of mutants in mutation testing (abbreviated as mutant reduction techniques) and techniques on reducing the execution time of mutants (abbreviated as time reduction techniques).

1) Mutant reduction techniques: To reduce the number of mutants in mutation testing, selective mutation testing is proposed by using a subset of mutants to represent the whole set of generated mutants. The techniques in selective mutation testing are typically classified into operator-based mutant selection and random mutant selection.

Operator-based mutant selection is to reduce the number of mutants by carefully choosing mutation operators before generation mutants. In particular, Mathur and Wong [37], [36] analyzed 22 mutation operators used by Mothra [5] and observed that several operators contribute to most of the generated mutants. Based on this, Offutt et al. [30] proposed *N-selective mutation*, which reduces the number of mutants by omitting the N most prevalent operators. Later, they [27] extended their previous approach and proposed to use five *sufficient mutation operators* to generate mutants that achieve almost the same effectiveness as the mutants generated by all mutation operators. Following their work, many researchers focused on detecting *sufficient mutation operators*. In particular, Barbosa et al. [4] presented 6 guidelines to determine sufficient mutation operators, with which they finally determined 10 sufficient mutation operators for C programs. In their approach, the sufficiency scores of the mutants generated by the 10 mutation operators are between 95.8% and 100%. Furthermore, Namin et al. [32] proposed to combine the execution information of a subset of mutants, and their approach identified 28 sufficient mutation operators in the evaluation. In specific field, for testing components, Jiang et al. [19], [20] used specially self-defined mutation operators instead of traditional mutation operators; for testing concurrent programs, Gligoric et al. [11] studied what mutation operators are sufficient for concurrent programs.

Random mutant selection is to randomly select mutants from the whole set of mutants, which are generated by all the mutation operators. After Acree et al. [1] proposed the first random mutant selection technique, Mathur and Wong [37], [36] empirically studied this technique by randomly selecting $x\%$ mutants, which are generated by the 22 mutation operators in Mothra [5].

Most of these work in selective mutation testing, including operator-based mutant selection and random mutant selection, focuses on presenting various techniques on selecting a subset of mutants to represent the whole set of mutants so as to reduce the cost in mutation testing. However, our work in this paper focused on the scalability issue of selective mutation testing.

Recently, Zhang et al. [42] conducted several empirical studies on comparing random mutant selection and operator-based mutant selection, and found that random mutant selection is as efficient as operator-based mutation selection. Furthermore, based on their empirical results we may infer that selective mutation testing may have good scalability, but their work did not systematically studied the scalability problem of selective mutation testing although this problem has important influence on the practical usage of selective mutation testing. Our work is the first empirical study systematically exploring the scalability problem of selective mutation testing.

2) Time Reduction Techniques: Another way of reducing the cost of mutation testing is to reduce the execution time of mutants.

To reduce the execution time of a mutant, Howden [17] proposed the concept of *weak mutation testing*, which divides a mutant into several components. Once a test suite causes a different behaviour of any component, the test suite kills the mutant. In this way, testers can know whether a mutant is killed without executing the whole mutant to the last line of code, and thus the execution time of a mutant is reduced. Later, Woodward and Halewood [38] proposed *firm mutation testing*, which is a compromise of weak mutation testing and traditional mutation testing. Offutt and Lee [28] conducted an empirical study on weak mutation testing and found that weak mutation testing can be better applied in unit testing of non-critical applications.

Another way of reducing time is to optimize the process of compiling and executing mutants. Emillo et al. [7] and Untch et al. [34] changed a compiler to make it able to compile all mutants at one time. Krauser et al. [22] and Offutt et al. [29] ran mutants in parallel to speed up mutation testing. Zhang et al. [43] proposed to prioritize and reduce tests in mutation testing so as to determine the set of killed and the set of non-killed mutants quickly. To facilitate mutation testing for evolving programs, Zhang et al. [44] proposed to speedup mutation testing for the current program by reusing the execution results of some mutants on the previous program.

These work aims to reduce the cost of mutation testing by reducing the execution time of mutants, but our work focuses on selecting a subset of mutants so as to reduce the cost of mutation testing.

B. Applications of Mutation Testing

The quality of a test suite can be measured by the mutation score in mutation testing. Based on this, many researchers used mutation testing to provide guidance in generating test cases. That is, various techniques [6], [39], [23], [14], [31], [45], [10], [15] have been proposed to generate test cases to achieve some mutation score.

Besides test case generation, mutation testing, especially mutants, are widely used to simulate faults in the evaluation of software testing techniques [25], [13], [41], [33]. Mutants can be used to simulate faults because previous work [3] presented an empirical evidence that mutants simulate real faults better than manually seeded faults. As mutants can be representative of real faults, Do et al. [9] revisited the effectiveness of existing test case prioritization techniques by conducting an empirical study on applying these techniques to programs whose faults are generated by mutants rather than manually seeded faults. Hou et al. [16] used mutation testing to simulate the misunderstanding between component users and providers. Recently, Zhang et al. [46] proposed to localize real faults by injecting mechanical faults, which are actually mutants.

These work either uses the mutant score based on the whole set of mutants to measure whether a test case is adequate so as to guide the process of test case generation, or uses mutants to simulate real faults so as to facilitate other tasks in software testing. Different from them, our work targets at the scalability of selective mutation testing, which is to select a sufficient subset of mutants.

VII. Conclusion

In this paper, we presented an empirical study on exploring the scalability of selective mutation testing, through four large projects, 12 versions all together. From the empirical study, for programs up to 16 KLOC, selective mutation testing has surprisingly good scalability. In particular, as either the program size or the total number of non-equivalent mutants increases, the number of selected mutants increases slowly, while the proportion of selected mutants decreases. Furthermore, for any program whose number of lines of executable code is E, the proportion of selected mutants used in selective mutation testing is proportional to E^b where b is a constant whose value is between -0.95 and -0.75, the number of selected mutants used in selective mutation testing is proportional to E^c where c is a constant whose value is between 0.05 and 0.25.

In future, we plan to present some prediction models for predicting the number and the proportion of mutants used in selective mutation testing for any program. We also plan to conduct similar experiments on larger projects and projects of other languages, such as C, C#, and Python by considering the cost issue of each mutant. Besides, we plan to conduct another pioneer empirical study on the scalability of operator-based selective mutation testing, though operator-based selective mutation testing is more complicated and time-consuming to measure and analyze its scalability.

VIII. Acknowledgments

This work is supported by the High-Tech Research and Development Program of China under Grant No.2013AA01A605, the National Basic Research Program of China (973) under Grant No. 2011CB302604, the Science Fund for Creative Research Groups of China under Grant No.61121063, and the National Natural Science Foundation of China under Grant Nos. 61272157, 61228203, 61432001.

References

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report, Georgia Institute of Technology, 1979.

[2] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Proc. GECCO*, pages 1338–1349, 2004.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. ICSE*, pages 402–411, 2005.

[4] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for c. *Software: Testing, Verification and Reliability*, 11(2):113–136, 2001.

[5] B. Choi, R. A. DeMillo, E. W. Krauser, R. Martin, A. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford. The mothra tool set (software testing). In *Proc. ICSS*, pages 275–284, 1989.

[6] R. DeMilli and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[7] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *Proc. COMPSAC*, pages 351–356, 1991.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[9] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[10] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[11] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *Proc. ISSTA*, pages 224–234, 2013.

[12] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, (4):279–290, 1977.

[13] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang. Is this a bug or an obsolete test? In *Proc. ECOOP*, pages 602–628. 2013.

[14] D. Hao, L. Zhang, M. Liu, H. Li, and J. Sun. Test-data generation guided by static defect detection. *Journal of Computer Science and Technology*, 24(2):284–293, 2009.

[15] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proc. FSE*, pages 212–222, 2011.

[16] S.-S. Hou, L. Zhang, T. Xie, H. Mei, and J. Sun. Applying interface-contract mutation in regression testing of component-based software. In *ICSM*, pages 174–183, 2007.

[17] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, (4):371–379, 1982.

[18] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[19] Y. Jiang, S.-S. Hou, J. Shan, L. Zhang, and B. Xie. Contract-based mutation for testing components. In *ICSM*, pages 483–492, 2005.

[20] Y. Jiang, S.-S. Hou, J. Shan, L. Zhang, and B. Xie. An approach to testing black-box components using contract-based mutation. *International Journal of Software Engineering and Knowledge Engineering*, 18(1):93–117, 2008.

[21] W. M. Kolb. *Curve fitting for programmable calculators*. Syntec, Incorporated, 1984.

[22] E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, 1991.

[23] M. Liu, Y.-F. Gao, J. Shan, J.-H. Liu, L. Zhang, and J. Sun. An approach to test data generation for killing multiple mutants. In *ICSM*, pages 113–122, 2006.

[24] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Proc. COMPSAC*, pages 604–605, 1991.

[25] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering*, 38(6):1258–1275, 2012.

[26] E. S. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software: Testing, Verification and Reliability*, 9(4):205–232, 1999.

[27] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[28] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.

[29] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar. Mutation testing of software using a mimd computer. In *Proc. ICPP*, 1992.

[30] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proc. ICSE*, pages 100–107, 1993.

[31] M. Papadakis, N. Malevris, and M. Kallia. Towards automating the generation of mutation tests. In *Proc. AST*, pages 111–118, 2010.

[32] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proc. ICSE*, pages 351–360, 2008.

[33] M. Staats, G. Gay, M. Whalen, and M. Heimdahl. On the danger of coverage directed test case generation. In *Proc. FASE*, pages 409–424. 2012.

[34] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *Proc. ISSTA*, number 3, pages 139–148, 1993.

[35] W. E. Wong. *On mutation and data flow*. PhD thesis, Purdue University, 1993.

[36] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

[37] W. E. Wong, A. P. Mathur, and J. C. Maldonado. Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In *Software Quality and Productivity*, pages 258–265. 1995.

[38] M. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proc. STVA*, pages 152–158, 1988.

[39] S. Xu and T. J. Frank. Forecasting the efficiency of test generation algorithms for combinational circuits. *Journal of Computer Science and Technology*, 15(4):326–337, 2000.

[40] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *Proc. ASE*, pages 92–102, 2013.

[41] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proc. ICSE*, pages 192–201, 2013.

[42] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proc. ICSE*, pages 435–444, 2010.

[43] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proc. ISSTA*, pages 235–245, 2013.

[44] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proc. ISSTA*, pages 331–341, 2012.

[45] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. d. Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *ICSM*, pages 1–10, 2010.

[46] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proc. OOPSLA*, pages 765–784, 2013.