Calibrated Mutation Testing

Jaechang Nam
The Hong Kong University of Science and Technology
Kowloon, Honk Kong
jcnam@cse.ust.hk

David Schuler · Andreas Zeller
Saarland University, Saarbrücken, Germany
{ds, zeller}@cs.uni-saarland.de

***Abstract*—During mutation testing, artificial defects are inserted into a program, in order to measure the quality of a test suite and to provide means for improvement. These defects are generated using predefined mutation operators—inspired by faults that programmers tend to make. As the type of faults varies between different programmers and projects, mutation testing might be improved by *learning from past defects*—Does a sample of mutations similar to past defects help to develop better tests than a randomly chosen sample of mutations? In this paper, we present the first approach that uses *software repository mining* techniques to *calibrate mutation testing* to the defect history of a project. Furthermore, we provide an implementation and evaluation of calibrated mutation testing for the JAXEN project. However, first results indicate that calibrated mutation testing cannot outperform random selection strategies.**

***Index Terms*—mutation testing, version control, bug database**

I. Introduction

One goal of software testing is to find defects as early as possible, e.g. by using automated test suites. But how do we know that a test suite is good at finding defects? *Mutation testing* aims to answer this question by seeding artificial defects (mutations) into a program, and then checking whether the test suite detects them. But how representative mutations are for real faults? Andrews et al. [2] showed in their study that mutations are better representatives for real faults than faults generated by hand. However, different types of faults are made in different projects [9]. Therefore, mutation testing might be improved by learning from the defect history. As future defects are often similar to past defects, mutations that are similar to past defects can help to develop tests that also detect future bugs. This motivated us to develop *calibrated mutation testing*, a technique that produces mutations according to characteristics of defects that were made in the past. To this end, we mine the repository of a project for past fixes. These fixes represent attempts to cure a defect, and thus provide information about past defects. By producing mutation sets that share properties with these past fixes we *calibrate* mutation testing to the *defect history* of a project (Figure 1 summarizes our approach).

In this paper, we make the following contributions:

- We show how to mine and categorize fixes from different sources.
- We demonstrate how to calibrate mutation testing from past fixes.

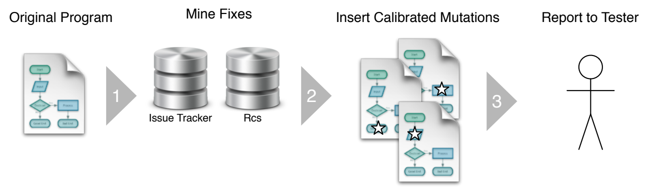

Fig. 1. The process of calibrated mutation testing. First fixes are mined from different sources. Then the mutations are (Step 2) calibrated to characteristics of the fixes and (Step 3) presented to the tester.

- We provide an experimental evaluation of our approach for the JAXEN [1] project.

The paper is organized as follows. First, we describe how to mine and classify fixes for a project (Section II). Using data from the collected fixes, we give details on how to calibrate mutation testing to the defect history of a project (Section III). In our evaluation (Section IV), we compare the defect detection capability of prioritizations that are based on different selection schemes. Then, we discuss the threats to validity (Section V), and the related work (Section VI), and close with conclusion and consequences (Section VII).

II. Classifying Past Fixes

Calibrated mutation testing aims to produce mutations similar to past defects, as made during the development of a project. To this end, we need to obtain information about these defects. We do this by learning from fixes, as they represent attempts to cure a defect, and thus describe a defect. In the following sections, we describe how to collect and classify past fixes for a software project.

A. Mining Fix Histories

Defects are constantly fixed during the development of a project. These fixes, however, are documented in different ways. Thus, we mine fixes for a project from different sources:

- *Revision Control System (RCS)*: Revision Control System such as cvs, svn, and git provide commit messages, and these messages may contain the keyword ‘fix’ to represent if revisions are for fixing defects or features. We use the Kenyon framework [4] for collecting such commits automatically. Then, we manually validate if they are really fix revisions.
- *Issue Tracker connected with RCS logs*: Several commit logs may contain the issue index numbers of an issue

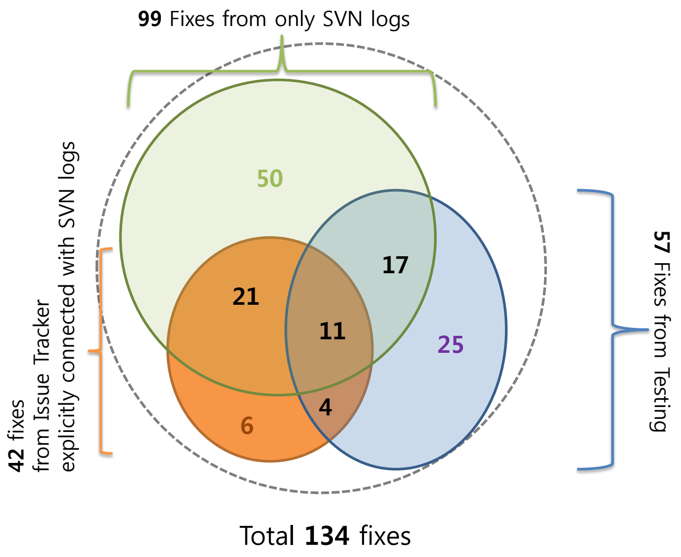
99 Fixes from only SVN logs
50
17
21
11
25
4
6
42 fixes from Issue Tracker explicitly connected with SVN logs
57 Fixes from Testing
Total 134 fixes

Fig. 2. Collected fixes for the JAXEN project.

tracking system. We can collect the revisions containing a these numbers automatically with Kenyon, and then manually check whether they are really fix revisions corresponding to a certain issue.

- *Testing*: We can compare the test results for two adjacent revisions to obtain fixes. If a test case fails on the former revision and passes in the latter one, this is a fix revision.

B. Subject Project

A project has to fulfill several criteria, to be suitable for calibrated mutation testing. It has to have a version history that lasts long enough, such that we can learn from past fixes. There has to be an issue tracker from which we can extract bugs that were fixed. Furthermore, it has to come with a JUnit test suite, such that we can test several revisions automatically and apply mutation testing on it.

The JAXEN XPath engine fulfills these criteria. It has a version history of 1319 revisions that span a duration of about 7 years. It is of medium size (12,438 lines of code), and comes with a JUnit test suite that consists of 690 test cases.

Figure 2 shows the fixes that we collected for JAXEN, with the techniques described in the previous section. In total we got 134 fixes. Out of these, 99 fixes were extracted from the version archive (svn), 42 from the issue tracker, and 57 from comparing the test results of subsequent revisions. Some of the fixes extracted from different sources are overlapping, which is shown by a Venn-Diagramm. For example, 21 of the fixes that were extracted from the version archive were also extracted from the issue tracker.

C. Fix Categorization

The collected fixes were categorized according to the fix pattern taxonomy proposed by Pan et al. [16]. The taxonomy categorizes fixes based on syntactical factors and their source code context. For example, the 'IF-APC' pattern describes fixes that add a precondition check via an if statement. In total Pan et al. propose 27 fix patterns. However, some fixes could not be categorized by the taxonomy, so we added 12 new fix patterns.

TABLE I
FIX PATTERN PROPERTIES AND THEIR VALUES.

Properties	**Values**
Flow	control(C), data(D)
Fix change	add(A), remove(R), modify(M)
Syntactic	if(IF), method call(MC), operation(OP), switch(SW), exception handling(EX), loop(LP), variable declaration(DC), casting(CT)

For each fix pattern, we also determined several properties that describe its effects. The *flow* property describes the impact of the fix on control and data flow. The *change* property indicates if new code was added, or existing code was changed or deleted. The *syntactic* property describes which type of statements were involved in the fix (e.g. if statements or method calls). Table I shows all values for the different properties. Note that a fix pattern can also be associated with multiple values from each category.

Furthermore, we also determined the location for each fix—that is the package, class, and method that were affected by the fix. Thereby, we can identify areas of the program that are more vulnerable than others.

By using these patterns and properties, each fix can be categorized. Although the fixes were categorized manually, each of the categorization steps could be automated.

III. Calibrated Mutation Testing

The idea of calibrated mutation testing is to adapt mutations to the defects that occurred during the development of the project. We obtain information about the defects by mining past fixes. Using this data, we define mutation operators that mimic defects that were fixed (Section III-A), and devise mutation selection schemes based on the properties of previous fixes (Section III-B).

A. Mutation Operators

To generate mutations similar to actual defects, we use the fix patterns that are based on actual fixes. For a fix pattern we derive a mutation operator that reverses the changes described by the pattern. Thereby, we introduce defects similar to fixes that are represented by this pattern.

For the JAXEN project, we examined the 10 most frequent fix patterns, and checked which mutation operators reverse this pattern. The results are summarized in Table II. The first column gives the fix pattern ordered by their frequency (column 5). Columns 2-4 show the properties associated with each fix pattern. The mutation operator that corresponds to a fix pattern is given in the last column. For 8 out of these 10 patterns, we found a corresponding mutation operator. For our experiments we used JAVALANCHE [17, 18], a mutation testing framework developed with focus on automation and efficiency. One of the mutation operators that corresponds to

TABLE II
FIX CLASSIFICATION RESULTS FOR JAXEN. 10 MOST FREQUENT FIX PATTERNS.

Pattern Name	Properties			Frequency	Mutation Operator
	Flow	Change	Syntactic		
Change of Assignment Expression (AS-CE)	D	M	OP	18	replace assignments
Addition of an Else Branch (IF-ABR)	CD	A	IF	13	skip else
Change of if condition expression (IF-CC)	C	M	IF	13	negate jump in if
Addition of Precondition Check (IF-APC)	C	A	IF	9	remove check
Method call with different actual parameter values (MC-DAP)	D	M	MC	9	replace method arguments
Addition of a Method Declaration (MD-ADD)	D	A	DC	9	-
Addition of Precondition Check with jump (IF-APCJ)	C	A	IF	8	skip if
Removal of an Else Branch (IF-RBR)	CD	R	IF	7	always else
Modify exception message (EX-MOD)	N	M	EX	6	-
Addition of Operations in an Operation Sequence (SQ-AMO)	D	A	MC	6	remove method calls

a pattern was already implemented in JAVALANCHE, and for the seven others we had to implement new mutation operators.

For example, the corresponding mutation operator for the 'IF-APC' pattern (see Section II-C) removes checks. Checks are if conditions without an else part. Thereby, code that was previously guarded by the check gets executed regardless of the check result.

B. Mutation selection schemes

As we do not want to apply all possible mutations exhaustively, we propose different *selection schemes* that aim to represent past defects. To this end, a selection scheme takes properties of past fixes into account and selects mutations according to these properties. Therefore, we also mapped the characteristics described in Section II-C to mutations, and derived different selection schemes:

- *Pattern-based scheme:* Past fixes can be described by fix patterns. Many of these fix patterns can be related to mutation operators as described in Section III-A. By using this relation, we can select a set of mutations that reflects the distribution of fix patterns among the past fixes.
- *Property-based schemes:* A fix pattern is characterized by different properties (flow, change, and syntactic properties). We calculate the distribution of these properties among all fixes. Then, we select a set of mutations that manipulate these properties such that the distribution of properties among the fixes is reflected—e.g. if more fixes are associated with a property, also more mutations are selected to manipulate this property. In this way, we get three different mutation selection schemes, as there are three different properties.
- *Location-based schemes*: Source code locations where defects were fixed in the past may be more vulnerable than other locations. Thus, we collect the locations of the fixes and count their occurrences. Then, we select mutations according to the distribution of fix locations. By considering three different granularity levels (package, class, and method level), we obtain three different location-based schemes.
- *Random scheme*: A scheme that randomly selects mutations from all possible mutations serves as a benchmark.

From the different selection schemes, we obtain different sets of mutations, which are calibrated to different aspects of the defect history of the project. Note that the mutation selection schemes are not deterministic. When a selection scheme is applied multiple times, different mutations may be chosen, and only the distribution of the underlying characteristic stays the same.

IV. EVALUATION

For the evaluation of our approach, we were interested if mutation schemes that are based on properties of previous fixes help to develop better tests than schemes that are based on a random selection of mutations. But how do we define better or good tests in this context? Although there are many different opinions on what makes a good test, for our evaluation, we consider a test to be good when it *detects bugs*.

Therefore, in an ideal setting, we would first apply each mutation selection scheme to a project, and develop tests that cover the selected mutations. Then, we would check how many bugs these tests detect. Unfortunately, the first step is very hard, as it would involve tremendous human effort to write tests for all mutations, and the second step is impossible, as we do not know all bugs that are in a project. Thus, we propose an alternative evaluation setting that is based on the version history of a project.

A. Evaluation Setting

For a project we check out every revision from the revision control system. Then, we compile each revision using the build scripts of this revision. If the revision can successfully be compiled, we also run its unit tests and record the results of the individual tests. With this approach, we obtain a matrix that depicts which test passes or fails on which revision. Using this data, we consider a test to detect a bug, if the test fails on a revision and passes on a later revision.

For evaluating our approach we pick a specific revision. Then, we compute all fixes as described in Section II up to this revision. By learning from these fixes, we create mutation sets according to different selection schemes. For these mutation sets, we check by which tests each mutation is detected. Then, we prioritize the tests in a way that they are sorted by the number of additional mutations that they detect. In this way,

TABLE III
AVERAGE NUMBER OF DEFECTS DETECTED BY PRIORITIZATIONS CREATED ACCORDING TO DIFFERENT MUTATION SELECTION SCHEMES FOR REVISION 1229 OF JAXEN. VALUES THAT ARE BETTER THAN THE VALUES FOR THE RANDOM SCHEME ARE UNDERLINED AND STATISTICALLY SIGNIFICANT VALUES ARE SHOWN IN BOLD FACE.

Top x	**Fix**		**Location**			**Property**		**Random**
percent	**Pattern**	**Package**	**Class**	**Method**	**Flow**	**Change**	**Syntactic**	
10	**0.39**	0.75	0.53	0.68	0.49	0.72	0.56	0.58
20	0.85	**1.45**	**0.67**	0.87	0.94	1.31	1.13	1
30	**1.39**	**2.61**	**1.3**	**2.54**	1.78	2.19	1.67	1.95
40	**2.28**	3.68	**2.07**	**3.93**	2.79	3.11	**2.62**	3.24
50	**3.02**	4.47	**2.8**	**4.78**	**3.47**	3.9	**3.47**	4.14
60	**3.82**	5.13	**3.84**	**5.59**	4.21	4.44	4.19	4.59
70	**4.64**	5.63	5.84	**6.58**	4.85	4.94	4.86	5.32
80	**5.58**	6.29	6.27	6.62	**5.65**	6.06	5.76	6.4
90	8.95	8.63	8.66	**9.31**	8.46	8.63	8.81	8.85
100	10	10	10	10	10	10	10	10

we obtain a test prioritization for each scheme. To assess the quality of a prioritization, we check how many future bugs are detected by the tests in the top x percent of the prioritization. Again, the number of future bugs a test detects is derived from the previously produced matrix.

B. Evaluation Results

We applied our approach on two revisions of JAXEN (revisions 1229 and 931). We randomly chose these two among the revisions that had enough (≥ 5) tests that detect defects in future revisions. For each revision, we applied the different selection schemes, and produced 100 different mutation sets for each scheme. Then, we checked the average failure detection ratios of the prioritizations that were produced for these sets.

Table III shows the results for applying the different selection schemes on revision 1229 of JAXEN. The columns give the selection scheme the prioritization is based on, and the rows give the percentage of tests considered. A value in the table depicts how many future faults are found on average by tests in the top-x percent of a prioritization. Values that are better than the values for the random scheme are underlined, and statistically significant values are shown in bold face.

For the detected defects, especially for the random prioritization, one would expect a linear distribution, i.e. for the top x percent $x/10$ bugs are detected. However all schemes perform worse. This is due to the setup of the evaluation. First the mutations are chosen, an then the tests are prioritized according to these mutations. For the random scheme also the mutations, which are used for the prioritization, are randomly chosen and *not the tests*. When a test that detects a bug only detects few or no mutations it is ranked low in all prioritization schemes, and therefore, like in this case the prioritization schemes perform worse than a linear distribution, which would be achieved by randomly choosing tests.

The results for the prioritization based on the fix pattern scheme (column 2) gives constantly worse values than the ones for the random scheme (last column) for the top 10-80 percent, and 7 out of them are significantly worse. Two of the location based schemes, the package and method based scheme (column 3 and 5), produce most times better results than the random scheme. For the scheme that is based on the package location (column 3) however, this is only statistically significant for the top 20 and 30 percent of the tests cases. The scheme that is based on the method location of previous fixes (column 5) has statistically significant better values for the top 30 to 70 percent. For the scheme that is based on class location (column 4) this trend does not seem to hold as it produces values that are worse than the ones for the random scheme. This effect seems counterintuitive and might be to nature of the tests. Tests that perform well on mutations in a package seem to be better in defect detection than tests that perform well on mutations in defect prone classes. More specific tests that perform well on mutations in defect prone methods are again better in defect detection. The test prioritizations generated for the property based techniques (column 6-8), on the other hand, perform worse than the prioritization based on a random selection of mutations, except for the change property based scheme, which has better values for top 10-30 percent. However, these values are not statistically significant.

In order to check whether these results hold over the history of the project, we also checked an earlier revision. Table IV shows the corresponding results for revision 931. Note that for almost all prioritization schemes at 90 percent all the defects are detected. This is due to some tests that detect very few mutations but no defect. Therefore, they are always ranked very low by the different prioritizations. The prioritizations based on the fix pattern type performs better for top 10 and 20 percent but worse for the rest. The location based schemes, which performed best for revision 1229, do not perform well on this revision. Only three values are better than the ones for the random scheme, but none is statistically significant. Among the property based prioritization schemes, the scheme based on the flow property of fixes performs best. The top 10-30 percent perform better than the random scheme, but again not statistically significant. From these results, we can conclude that:

Random mutation selection schemes cannot be outperformed by selection schemes based on defect history.

TABLE IV
AVERAGE NUMBER OF DEFECTS DETECTED BY PRIORITIZATIONS CREATED ACCORDING TO DIFFERENT MUTATION SELECTION SCHEMES FOR REVISION 931 OF JAXEN. VALUES THAT ARE BETTER THAN THE VALUES FOR THE RANDOM SCHEME ARE UNDERLINED AND STATISTICALLY SIGNIFICANT VALUES ARE SHOWN IN BOLD FACE.

Top x	**Fix**		**Location**			**Property**		**Random**
percent	**Pattern**	**Package**	**Class**	**Method**	**Flow**	**Change**	**Syntactic**	
10	1.72	1.12	1.47	1.33	1.76	1.35	1.41	1.44
20	2.73	2.35	2.64	2.61	2.95	2.44	2.65	2.71
30	3.28	2.97	3.09	3.2	3.55	3.13	3.02	3.41
40	3.47	**3.35**	**3.25**	3.69	3.68	3.44	3.39	3.82
50	3.77	**3.65**	**3.53**	3.99	3.92	3.66	3.76	4.05
60	4.17	4.13	4.08	4.45	4.15	4.17	**3.93**	4.32
70	4.44	4.52	4.66	4.68	4.49	4.6	4.43	4.59
80	4.80	4.83	4.84	4.84	4.79	4.90	4.73	4.86
90	5	5	5	5	5	5	4.98	5
100	5	5	5	5	5	5	5	5

V. THREATS TO VALIDITY

Like any empirical study, this study is limited by threats to validity.

- *External Validity:* As the evaluation was only carried out on one project, we cannot claim that the results carry over to other projects. However, the results may serve as a starting point for further investigation and discussion.
- *Internal Validity:* Due to the randomness that goes into the selection schemes, they are affected by chance. To counter this threat, we produced 100 different mutation sets for each scheme, averaged the results, and compared the results to the average of 100 randomly chosen sets. Furthermore, we followed rigorous statistical procedures to evaluate the results.
- *Construct Validity:* We learned about past defects from fixes. Although we used 3 different methods to mine fixes, some of the fixes are missed—and thereby data about past defects. For our evaluation we had to make several approximations that may influence the results. The test cases to detect (kill) mutations were chosen from existing test cases rather than writing new ones. This creates a bias towards the type of tests, as some mutations are not detected, and tests that cover large parts of the program get preferred. In addition, we only evaluate against defects that were detected by tests that exist in an earlier revision. This gives only a few defects and creates a bias towards the type of defect.

VI. RELATED WORK

A. Mining software repository

Research studies on mining software repository have been conducted for various purposes such as defect detection and prediction, API usage recommendation, and developer social network analysis [5, 6, 8, 10, 16, 20]. That is because software repositories contain many software artifacts such as version control histories, issue reports, source codes at a certain point, email archives, etc.

Particularly, mining different kinds of patterns of software artifacts from the repositories was the first step to achieve those purposes. Livshits et al. [10] proposed DynaMine which is a tool finding 'common error patterns' from software revision histories. Zhong et al. [20] proposed MAPO which is a tool mining and recommending 'API usage patterns' by using source code repositories. While our approach collects fix patterns to generate artificial defects for mutation testing, they collected error patterns and used the patterns to discover possible anomalies in source code.

Pan et al. [16] extracted bug fix patterns from seven open source projects. In their study, they showed how similar and consistent bug fix patterns are across all projects. Based on their results, we analyzed fix patterns to generate artificial defects that we consider similar to real defects. To our knowledge, this is the first trial using past software histories to calibrate mutation testing.

B. Mutation testing

In one of the first publications on mutation testing DeMillo et al. [7] introduced the *coupling effect* that relates mutations to errors and is stated as follows:

Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.

Offutt [13, 14] later provided evidence for the coupling effect with a study on higher order mutants. The results of our work can also be seen as a support for the coupling effect. They indicate that tests detecting randomly chosen mutations are as effective as tests detecting calibrated mutations.

Mutation selection was first proposed by Mathur [11], as a way to reduce the costs for mutation testing, by omitting the mutation operators that produce the most mutations. The goal of mutation selection is to produce a smaller set of mutations, such that tests that are sufficient (they detect all mutations) for the smaller set are also (almost) sufficient for all mutations.

In an empirical study on 10 Fortran programs Offutt et al. [15] showed that 5 out of 22 mutation operators are sufficient—test suites that detect all selected mutations also detect 99.5% of all mutations.

Barbosa et al. [3] did a similar study for C programs. They proposed 6 guidelines for selecting mutation operators. In an experiment on 27 programs, they determined 10 out of 39 operators producing a sufficient set with a precision of 99.6%.

A slightly different approach was taken by Namin et al. [12]. They tried to produce a smaller set of mutations that can be used to approximate the mutation score for all mutations. Using statistical methods, they came up with a linear model that generates less than 8 percent of all possible mutations, but accurately predicts the effectiveness of the test suite for the full set of mutations.

In a recent study, Zhang et al. [19] compared these 3 different operator-based selection techniques to random selection techniques. They showed that all techniques performed comparably well, and that random selection can outperform the best operator-based selection. This is similar to our results, were we found no significant difference between more sophisticated calibrated selection techniques and simple random techniques in terms of fault detection ratios for the corresponding test suites.

In contrast to our approach the selective approaches aim at minimizing the number of mutation and to approximate the results for all mutations, while our approach selects mutations calibrated to past defects in order to approximate future defects.

VII. Conclusion and Consequences

One application of mutation testing is to improve a test suite by analyzing not detected mutations, and developing new tests that detect them. In this paper we investigated whether calibrating mutations to the defect history improves this process, as future defects are often similar to past defects. Although some calibrated schemes showed better result than random selection for one revision, we found no scheme were this trend manifests for more revisions.

Our results can be seen in line with previous research by Zhang et al. [19]. They compared random selection schemes with different operator selection schemes, and in their evaluation random selection schemes were also not outperformed by more sophisticated schemes. Furthermore, our results might also be seen as support for the coupling effect—which states that tests that are sensitive enough to detect simple errors also detect complex errors. From our results, we can conclude that tests that detect randomly chosen mutations are as effective as tests that detect calibrated mutations.

Besides general improvements in terms of performance and scalability, our future work will focus on the following topics:

Different evaluation setup
: As our evaluation setting involves many compromises, it might be worthwhile to change the evaluation setup. For example automated test generators can be used to develop tests that detect mutations proposed by a selection scheme.

Broader evaluation
: Currently we only evaluated calibrated mutation testing for one project. It might be that this project does not suite our approach well. Therefore, we plan to automate all steps necessary for calibrated mutation testing and do a broader evaluation on more projects.

Acknowledgments. Jeremias Rößler, Rubin Wang, Dongsun Kim, and Gordon Fraser as well as the anonymous reviewers provided helpful feedback on earlier revisions of this paper.

References

[1] JAXEN `http://www.jaxen.org`.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proc. of the 27th International Conference on Software Engineering*, pages 402–411, 2005.

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

[4] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *ESEC/FSE-13: Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 177–186, 2005.

[5] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proc. of the 2006 international workshop on Mining software repositories*, MSR '06, pages 137–143, New York, NY, USA, 2006. ACM.

[6] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 24–35, New York, NY, USA, 2008. ACM.

[7] 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.

[8] S. Kim, E. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181 –196, 2008.

[9] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proc. of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, 2006.

[10] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *ESEC/FSE-13: Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 296–305, 2005.

[11] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *COMPSAC '91: Proc. of the fifteenth International Computer Software and Applications Conference*, pages 604–605, 1991.

[12] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ICSE '08: Proc. of the 30th international conference on Software engineering*, pages 351–360, 2008.

[13] A. J. Offutt. The coupling effect: fact or fiction. *ACM SIGSOFT Software Engineering Notes*, 14:131–140, 1989.

[14] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1:5–20, 1992.

[15] 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 (TOSEM)*, 5(2):99–118, 1996.

[16] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14:286–315, June 2009.

[17] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA'09: Proc. of the Eighteenth International Symposium on Software Testing and Analysis*, pages 69–80, 2009.

[18] D. Schuler and A. Zeller. (Un-)covering equivalent mutants. In *ICST'10: Proc. of the 3rd International Conference on Software Testing, Verification and Validation*, pages 45–54, 2010.

[19] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *ICSE '10: Proc. of the 32nd International Conference on Software Engineering*, pages 435–444, 2010.

[20] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, July 2009.