PathScore-Relevance: A Metric for Improving Test Quality

Silviu Andrica and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Reliability of today's software systems hinges on developers writing test cases that exercise as much of a program as possible. Writing and running such tests is inevitably subject to a time budget. This paper addresses the question of how to maximize quality of testing, given such a fixed time budget. We define a program-path scoring metric along with a way to measure a software component's relevance, and then show how these can be combined to produce a test quality metric that is superior to the test coverage metrics in use today. The key features of our proposal are that (a) it steers testing toward code that is most in need of testing, such as frequently-used or recently-modified code, and (b) it prioritizes testing shorter program paths over longer ones. As a proof-of-concept, we augmented an automated testing tool with our test prioritization criterion and found that it explores up to 70 times more code paths in the same amount of time, with no additional human effort.

1 Introduction

A program's reliability is considered to be roughly proportional to the volume and quality of testing done on that program. Software vendors, therefore, use extensive test suites to reduce the risk of shipping buggy software.

Unfortunately, testing is subject to constraints: it is expensive, resource-intensive and under serious time-to-market pressure, so there is rarely enough time to be thorough. Such constraints call for difficult tradeoffs: development managers aim to maximize software quality within the available budgets of human and time resources. Making these tradeoffs requires weighing the benefits of more tests against the cost of writing them, and this is more of an art than a science—the best managers are "artists" relying on talent, experience, and intuition.

To reduce reliance on artists in delivering high quality products, software engineers invented quantitative ways of assessing the quality of a test suite. Metrics tend to lead to better-informed tradeoffs. The most common technique is to measure "code coverage," i.e., how much of the code is exercised by the tests. The higher the coverage, the better the test suite and the higher the expected quality of the tested product. It is therefore common for development organizations to choose a target coverage level between 75%-90% [7] and, once this is met, to declare the software ready to ship. The coverage target serves to prioritize testing of those components that fall short of the goal.

Yet, despite testing taking up more than half a typical development cycle [4], bugs still exist. Have the limits of testing been reached? We argue that there is still plenty of room for improving the *efficiency* of testing, i.e., we can achieve better testing in the same amount of time. In particular, widely-used code coverage metrics are poor approximations of test quality, and they do not take into account the fact that not all code requires the same depth of testing. A poor coverage metric leads to suboptimal tradeoffs and fails to steer the testing process toward efficiently improving software quality within the budget of available resources.

Widely-used coverage metrics have two significant drawbacks. First, code coverage often views a program as a set of statements, missing the link between these statements. A program execution is not a mere set of statements, but a sequence with a strong partial order; bugs are often the result of this partial order being wrong. Second, all code is treated equal, missing the fact that some code is more important than other. Consider, for example, error recovery paths: they typically represent small portions of the code, so testing them has minimal impact on the level of coverage. However, recovery code is critical: it runs seldom, but must run perfectly whenever it runs, because it is rescuing the system from the abyss of failure. Yet testers avoid writing recovery tests: they are more difficult than functionality tests, and, if one must improve code coverage by 5%, writing recovery tests is the least productive way to do so.

We introduce PathScore-Relevance $\mathcal{PR}$, a metric that addresses these shortcomings. It combines two measures: (a) *relevance* $\mathcal{R}$ of a code component to the overall functioning of the system, as determined both explicitly by developers and implicitly by end users, and (b) a *path score* $\mathcal{P}$ that weighs execution paths inversely to their length, encouraging testers to exercise shorter paths first. Since many of the longer paths consist of partially overlapping shorter paths, $\mathcal{P}$ discourages test redundancy, improving efficiency in a way similar to dynamic programming. While $\mathcal{R}$ helps prioritize components, $\mathcal{P}$ nudges the testing process toward lower-priority components whenever testing the higher priority ones has reached a plateau of diminishing returns.

Appears in Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep), Lisbon, Portugal, June 2009

2 Background

Before describing $\mathcal{PR}$, we illustrate four commonly used ways of measuring code coverage. We refer to the example code below and discuss the coverage reported on a test case that invokes `example(0)`:

```
         int example( int a )
         {
  1:        int res=0;
  2:        if (a==0) {
  3:           res=1;
  4:           libraryCallA();
  5:           libraryCallB();
            } else {
  6:           crash();
            }
  7:        if (a==1) {
  8:           crash();
            } else {
  9:           --res;
 10:           libraryCallB();
 11:           libraryCallA();
            }
 12:        return res;
         }
```

The most popular metric is *line coverage*, which measures the fraction of program lines (or statements) executed during a test run. Calling `example(0)` would exercise all lines except 6 and 8 (10 out of 12 lines), yielding a coverage of 83%. An organization that uses 80% coverage as a criterion for shipping would consider this code to be sufficiently well tested, despite the two undiscovered crash scenarios. A fundamental problem with line coverage is that a sequence of linear code contributes to the overall coverage proportionally to its length, so a few very long linear sequences can drown out a complex maze of short branches.

A different metric is *basic block coverage*, which reports the fraction of basic blocks (not statements) exercised during the test. In our case, 5 out of the 7 basic blocks would be exercised: line sets {1,2}, {3,4,5}, {7}, {9,10,11}, and {12}. Basic block coverage is thus 71%. This metric is in some sense better than line coverage, because statements in a linear piece of code are collapsed into a single entity (the enclosing basic block). However, the behavior of a program is determined not only by its statements or basic blocks, but also by the sequence in which these are executed; basic block coverage fails to capture this behavioral component.

Branch coverage, also known as decision coverage, is often considered a more informative metric. It reports the fraction of branches that were taken during the test, i.e., to what extent the branch conditions evaluated to both their true and false values. In our example, there are 4 condition branches and the `example(0)` test takes 2 of them ($2 \rightarrow 3$ and $7 \rightarrow 9$), resulting in 50% branch coverage. Similar to the earlier metrics, this one views the branch points as being independent, with the sequence in which branches are taken not affecting the coverage metric at all.

A better test quality metric is *path coverage*, which reports what fraction of all execution paths was explored during a test. In our example, there are 4 potential execution paths (of which 3 are feasible); the test exercises only one of them (sequence $\langle 1,2 \rightarrow 3,4,5,7 \rightarrow 9,10,11,12 \rangle$), resulting in 25% path coverage. Path coverage, however, is virtually never used in practice: the number of paths through a program's control flow graph increases exponentially with the number of branches, and input-dependent loops make the number of such paths infinite.

As can be seen, the difference between test quality reported by the most popular metric (83%) and the "true" value of that quality ($\leq$25%) can be significant.

3 Guiding the Test Process with $\mathcal{PR}$

A good metric for assessing the quality of a test suite complements efforts aimed at improving software reliability, such as better testing tools and programming practices. The test metric acts as a guide for developers, pointing them toward the parts of the code that are in most need of additional tests. This need can be due to the corresponding code being highly critical and/or due to existing tests not exercising enough of the possible paths through that code. Time-to-market pressures require fast, smart choices in allocating quality assurance resources, so a metric that is closer to "the truth" than basic coverage is imperative.

We propose measuring test quality with a combination of two sub-metrics: *component relevance* $\mathcal{R}$, which captures the importance of a component from both the developers' and from the end users' perspective, and *path score* $\mathcal{P}$, which prioritizes the testing of shorter execution paths over longer ones. As will be discussed later, $\mathcal{R}$ indicates which components deserve most attention, while $\mathcal{P}$ helps the testing process move on, when a point of diminishing returns is reached. In the context of this paper, we generically refer to an arbitrary unit of code as "component"; this could be a module, class, method, block of code, etc.

3.1 Component Relevance $\mathcal{R}$

Not all code is created equal. An off-by-one bug in the query engine of a DBMS can have a substantially worse effect on quality than, say, a bug in the DBMS's output formatting code. This argues for being able to differentiate the relative importance of testing depending on the target code.

As suggested in our earlier example concerning recovery paths, even a well-intentioned developer will, when under time pressure, aim to "check off" as many program issues as possible, regardless of how important they are; this is confirmed both by intuition and systematic studies [8]. By directly weighing the test quality result by the importance of the tested component, we can motivate solving problems in critical code instead of merely reaping low-hanging fruit in less critical code.

The goal of $\mathcal{R}$ is to rank components according to their importance, to an arbitrarily fine granularity. Since software testing is a perpetual compromise, greedily focusing on the most important components can improve the chances of being left with good software when testing resources run out.

There exist two primary stake-holders involved in determining the relevance of a component: developers and end users. In the absence of a widely accepted method for classifying components, we aim for a low-overhead way of taking the opinion of both of these classes of stake-holders into account, while minimizing subjectivity as much as possible.

Developer Perspective: The first way to determine relevance of a component is through *annotations*: next to the name of each function, class, or subsystem, developers can include an annotation that informs the testing tool suite about the criticality of that code. Simple scales, such as high / medium / low, seem to work best in practice, and are routinely used in the case of log messages, to indicate their criticality. Annotation-based relevance information can be augmented with simple *static analysis* to identify, for example, exception handlers; depending on programming language, more general error recovery code can also be identified statically.

A second source for determining relevance is *code and bug history*. First, recently added or modified code ought to be emphasized in tests, because it is more likely to contain new bugs; studies have shown a direct correlation between the age of code and the number of bugs it contains [11]. Second, after the system has been delivered to the customer, bug reports filed by end users can be used as indicators of fragile components that require more attention. A recent survey found that there exists a strong correlation between bugs found by users and bugs detected using a static analysis tool [8], which suggests that the number of problems found through static analysis can also be used as a low-cost indicator for relevance, if collecting such statistics from bug databases is not possible. The bottom line is to test (i.e., look for bugs) where bugs are most likely to be found.

Each of the methods above can provide a relevance measure for each component or block of code. If more than one measure is used, such as age of code together with whether it is or not recovery code, we must unify the measures into a single metric. The easiest approach is to normalize the individual metrics and ensure their semantics are uniform (i.e., value 1 should mean the same thing for all metrics).

In this paper, we do not prescribe a specific way to combine the metrics, as each development team can decide on its own relevance metrics and aggregation method. Nevertheless, we do provide an illustrative example. Say for each line of code we represent code age as a flag *newCode* indicating 0 for "old" (code has not been modified since the last release) or 1 for "new" (code is fresh or has been modified since last release); this can easily be computed based on a `diff` in the source code base. Similarly, a recovery flag *recovery* indicates 1 for "yes" (code is meant to do recovery) or 0 for "no" (otherwise); this can be inferred based on annotations or static analysis.

For each component c in program P, we can compute a sum of the flags $FlagSum(c) = \sum_{l \in \text{code lines}} (newCode_l + recovery_l)$ and express relevance as:

$$\mathcal{R}(c) = \frac{FlagSum(c)}{\max_{c_i \in P} FlagSum(c_i)}$$

$\mathcal{R}$ captures the fact that a component with lots of new code or lots of recovery code should be tested more than components with less code of this kind. Code that is both new and performs recovery must be tested even more.

End User Perspective: The second way to determine relevance is based on which parts of the software are most used by end users. Regardless of how the system is constructed, whatever code the user most depends on should be viewed as the most important and thus relevant to testing.

We rely on usage profiles to determine which components are most frequently used in the field. One might argue that a widely-used component is implicitly tested extensively by the users themselves, but that is true only of the particular exercised paths, not true of the component as a whole. For example, users may make frequent use of a network discovery component, but not exercise the recovery paths within that component. Usage profiles would indicate that component as being highly relevant, making it a high priority test target, including its recovery paths.

Usage profiles can be obtained during validation testing (e.g., from alpha and beta release users) or even from systems running in production. These profiles indicate, for each component c, how many times it has been exercised. Standard low-overhead profiling, like OProfile [9], is sufficient in most cases. For widely-used software, a statistical cooperative runtime profiler could leverage the many copies of the software running around the world to make profiling overhead negligible through sampling, like in Holmes [6]. Sketching [2] is also a good candidate technique. Profiling can be turned off once the usage profile converges.

If we think of a target program P as a collection of components, then we can use the information from the profile to define the relevance of each component c :

$$\mathcal{R}(c) = \frac{NumTimesExecuted(c)}{max_{c_i \in P}\ NumTimesExecuted(c_i)}$$

$\mathcal{R}$ captures the frequency with which c was executed by end users, relative to that component which was executed the most.

Regardless of whether we employ the developers' or the end users' perspective, relevance always varies from 0 (completely irrelevant) to 1.0 (most relevant=100%). It is

therefore trivial to combine the two perspectives either in a multiplicative or a weighted additive fashion, depending on what the development organization feels is most accurate for their needs.

Having expressed relevance quantitatively, we now turn our attention to associating a path score with tests.

3.2 Assigning Path Scores to Tests

In contrast to path coverage, which treats all paths equally, $\mathcal{P}$ aims to reward tests that exercise shorter paths over those that exercise longer ones. The rationale is that longer paths are often composed of shorter paths that overlap, so testing done on the shorter paths can be leveraged in testing the longer paths. As mentioned before, this reasoning is similar to dynamic programming.

We define the length of a path to be the number of basic blocks contained in that path. It is possible to use the number of statements as well, but the overhead of computing the metric in practice would be higher, in exchange for no increase in precision.

We denote by $NumPathsExercised_L(c)$ the number of paths in c of length L that were exercised by the test suite (derived dynamically), and by $TotalNumPaths_L(c)$ the number of possible execution paths of fixed length L in the tested code (derived statically). We define a test suite's path score for a given component c as the length-weighted amount of path exploration done in c by that test suite:

$$\mathcal{P}(c) = \sum_{L=1}^{max} \frac{1}{2^L} \times \frac{NumPathsExercised_L(c)}{TotalNumPaths_L(c)}$$

where max is the upper bound on the path length of interest. This formula computes the fraction of fixed-sized paths in c covered by the test suite. The value of $\mathcal{P}(c)$ ranges from 0 to 1.0 (0% to 100%), with 1.0 indicating the best. The $1/2^L$ ratio serves to weigh longer paths exponentially less, according to their length; 2^L is the number of leaves one would expect in a typical execution tree of depth L.

In theory, the sum starts with paths of length 1 (as if measuring line coverage) and ends with paths of length max (as in bounded-length path coverage). However, the $1/2^L$ ratio limits the impact that long paths have on the score, so we expect that in practice L will be limited to a small number. E.g., $1/2^7 < 1\%$, so it is unlikely for paths of length $L > 7$ to improve $\mathcal{P}$ in any significant way. In that sense, path score $\mathcal{P}$ represents a practical compromise between line coverage and path coverage.

3.3 Aggregate PathScore-Relevance $\mathcal{PR}$

The PathScore-Relevance metric $\mathcal{PR}$ combines relevance $\mathcal{R}$ and path score $\mathcal{P}$ such that, the higher the relevance of a component c, the higher the path score must be, in order to achieve the same value of $\mathcal{PR}$ as less relevant components:

$$\mathcal{PR}(c) = \frac{\mathcal{P}(c)}{\mathcal{R}(c)}$$

Aiming to maximize overall $\mathcal{PR} = \sum_{c_i \in P} \mathcal{PR}(c_i)$ in a greedy fashion will guide the testing efforts to first improve $\mathcal{P}$ for components with high $\mathcal{R}$, as this is the quickest way to increase overall PathScore-Relevance $\mathcal{PR}$.

Advantages: The PathScore-Relevance metric ensures that both the way customers use the system and the way developers understand the system helps decide what level of test quality is sufficient. Our approach does not require end users to write test suites or specifications, rather they implicitly define what is important by using the system. $\mathcal{PR}$ prioritizes testing critical code (e.g., error recovery) or potentially unstable code (e.g., a recent patch). As will be seen in §4, this prioritization can be done even for automated test generators.

At the same time, $\mathcal{PR}$ combines the tractability of computing line or basic block coverage with the accuracy of path coverage. $\mathcal{PR}$ treats components as sets of sequences of statements and is thus deeper semantically than basic block coverage, which only considers paths of length 1, but shallower than full path coverage.

Drawbacks: Like anything resulting from an engineering tradeoff, $\mathcal{PR}$ also has shortcomings. First, $\mathcal{P}$ emphasizes short paths, but some bugs may lie on long paths within a component. Whether it is better to spend resources searching for that bug or finding other more shallow ones in other components depends entirely on how relevant that component is. If it is critical, $\mathcal{R}$ will keep the testing within that component, encouraging deeper testing, otherwise $\mathcal{P}$ will nudge testing toward less critical components, where the return on time invested may be higher. If, however, the overall value of $\mathcal{PR}$ plateaus before QA resources run out—indicating that a point of diminishing returns was reached while resources were still available—one can replace the aggressive exponential decay $1/2^L$ with a more slowly (but still monotonically) decreasing function f, as long as $\sum_1^\infty f(l) = 1$. This way, the relative influence of path length decreases in favor of $\mathcal{R}$.

Second, $\mathcal{P}$ treats a branch condition as a single true-or-false predicate. However, most predicates have multiple boolean sub-expressions, each of which may influence the subsequent path structure. The metric could be enhanced to support modified condition/decision coverage [7], which requires a test suite to set every sub-expression that can affect the result of the decision to both true and false.

Third, even a $\mathcal{P}$ value of 100%, meaning that all paths have been exercised, does not mean the component has been

sufficiently tested. Consider, for instance, the following definition of the mathematical absolute function:

```
#define abs(x)  (x<0) ? (-x) : (x)
```

There are two paths, one for $x < 0$ and another for $x \geq 0$, so test cases `abs(-5)` and `abs(5)` will achieve complete coverage. However, `abs(INT32_MIN)` will wrongly return a negative number on most platforms. Factoring semantics of the input values into a coverage metric would be an interesting extension, to be pursued in conjunction with results from symbolic executions. An easier-to-automate approach is to make $\mathcal{PR}$ aware of equivalence classes, i.e. sets of values that cause the same branch to be taken, and their boundary values.

Finally, unlike a simple line coverage metric, the connection between writing a test and the expected improvement in $\mathcal{PR}$ is not immediately obvious to a developer. E.g., writing a test that will additionally cover a path with 10 statements will clearly increase the number of covered statements by 10, but it's effect on $\mathcal{PR}$ is less clear. We expect, however, that developers' intuition can develop quickly, since $\mathcal{PR}$ is a relatively simple metric.

4 Preliminary Experimental Exploration

To evaluate whether $\mathcal{PR}$ can improve the efficiency of test development, it should be applied to a test development process. Given that it would be difficult to discount the human factor, we opted instead to use PathScore-Relevance in the context of an automated test generation tool. We chose KLEE [5], which takes a program and, without modifying it, can generate tests that exercise the program along different paths. KLEE's explicit goal is to maximize test coverage and, to our knowledge, represents the current state-of-the-art. KLEE relies on a symbolic execution engine and a model of the filesystem to discover ways in which the target program can be exercised. During symbolic execution, KLEE relies on a *searcher* to decide which statement it should explore next. The default searcher in KLEE employs random search along with some heuristics. When reaching an exit point, be it a normal exit or due to a bug, KLEE generates a test case that drives the application through that same execution path.

We implemented a new searcher for KLEE that, when a new choice is to be made, selects a statement inside the most relevant function, in accordance with $\mathcal{R}$. However, when the path score $\mathcal{P}$ for the respective function achieves a target goal, another statement from a function with a lower $\mathcal{R}$ is chosen, to increase that function's $\mathcal{PR}$. The process repeats either until overall $\mathcal{PR}$ reaches a set target or time runs out, thus emulating real-world testing.

We ran KLEE for 5 minutes on five COREUTILS [5] programs (*cat, sort, mkdir, tail* and *tr*) in three scenarios: one with KLEE's random searcher, that randomly picks the next statement to be symbolically executed, and two with the $\mathcal{PR}$-based searcher.

The two $\mathcal{PR}$ scenarios differ in how relevance is determined. In the first one, we collect usage profiles by compiling the applications with profiling support and exercising a sample workload for these tools. In the second one, we annotate some of the error recovery paths in the five programs, setting the *recovery* flag for the respective lines to 1. To ensure that error recovery paths are executed, we configured KLEE to generate a maximum of 50 system call errors along each execution path.

In Figure 1, we report the average number of paths explored by KLEE with the two searchers in the three scenarios. For each program, we normalized the numbers relative to the default KLEE searcher; the absolute values are shown above the bars.

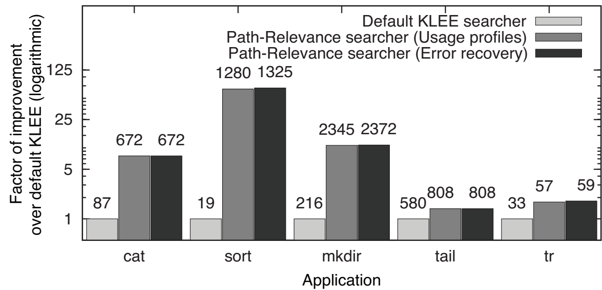

Figure 1. Number of explored paths resulting from 5-minute KLEE runs in 3 scenarios for 5 programs.

The $\mathcal{PR}$ searcher makes KLEE explore 1.7x to 70x more paths than the default KLEE searcher, in the same amount of time. A key reason for the speed-up is that the test generation process is steered toward exploring shorter paths before exploring the longer ones, so KLEE can find exit conditions more quickly.

Another reason is that the $\mathcal{R}$ component of the metric guides KLEE to explore more relevant code. In the case of usage profiles, KLEE was guided to exercise paths (and variants thereof) that the benchmark exercised, thus keeping KLEE from getting "trapped" in potentially infeasible parts of the execution tree—this effect is particularly visible in the case of *sort* and *tr*, where the default KLEE searcher explored only few paths. In the case of error recovery annotations, KLEE was steered toward paths that reached exit points quickly (e.g., program exit due to error), so interesting conditions that result in test cases were found more quickly.

While these results are by no means predictive of efficiency improvements in an actual development organization, it is a data set that suggests $\mathcal{PR}$'s potential. A test suite with more tests explores more paths, so it increases the probability that it will find bugs in important parts of the code, thus improving test quality at no increase in cost.

5 Related Work

Software engineers and researchers have made several proposals for better ways to measure test quality and/or to guide testing. We survey here some of the work more closely related to our proposal.

Specified path coverage [1] is the only coverage metric similar to $\mathcal{R}$. It quantifies the percentage of explored paths relative to a set of paths specified a priori. These paths can be given explicitly or can be derived based on a use case specified by end users. Our $\mathcal{R}$ metric differs from specified path coverage in two ways: it does not draw a hard line between program paths that should and should not be tested, and it provides guidance on the order in which paths are to be tested, based on their importance. Thus, $\mathcal{R}$ is more flexible and realistic.

In practice, full path coverage is not used to assess the quality of tests, because of the potentially infinite number of paths contained in a decently sized program's control flow graph. To mitigate this problem, researchers have proposed various path coverage variants, which offer ways to reduce the number of paths that need to be tested and, thus, make it feasible to compute that variant of path coverage.

Basis path coverage [10] identifies a "basis" for a given method's paths, i.e., the smallest set of paths that can be combined to create every other possible path through a method. It then measures how many of the paths in this basis were covered by the tests. The basis set is constructed by picking an arbitrary path as a baseline and then flipping decision branches one at a time, to construct the other paths in the basis. The goal of testing the basis set is to test every decision outcome independently of one another. In contrast to basis path coverage, $\mathcal{P}$ takes a dynamic programming-like approach that takes into account all the paths, but aims to cover the shorter ones first, thus encouraging tests to cover many paths quickly.

Decision-to-decision path coverage [3] measures how many of the paths between two decision statements have been tested. This metric is subsumed by $\mathcal{P}$, because a decision-to-decision path merely corresponds to a path of length 3 basic blocks.

There are numerous other proposals for measuring test quality, surveyed in substantial reference books [1, 3]. We believe that, for a test quality metric to be successfully adopted, it must be simple enough and have proper tool support. We have implemented the necessary $\mathcal{PR}$ support and expect to evaluate its use in real development and test projects.

6 Conclusions

In the absence of accurate and practical ways to measure test quality, we run the risk of testing blindly and, as code size and complexity increase, this type of testing can miss a large number of bugs. In this paper, we identified opportunities for being smarter about testing, i.e., doing better testing with no increase in effort. We introduced a new metric to guide testing; this metric prioritizes testing of more relevant code and emphasizes testing shorter paths first. Our proposal is an advance over conventional coverage metrics, which consider all components equally important and do not take into account that longer paths may just be overlapping combinations of independently-testable shorter paths.

Our proposed metric can be used as a test selection criterion. There is plenty of room for improvement, and we think that discussion of how to better measure test quality is imperative. Decisions on when a software system is ready for wide use are routinely made based on antiquated metrics. We expect that better selection criteria will lead to improvements in testing efficiency, will result in more relevant tests, and ultimately lead to software that better answers the end user's reliability requirements.

Acknowledgments

We thank the anonymous reviewers and Tim Brecht for their help in substantially improving our paper.

References

[1] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[2] S. Bhatia, A. Kumar, M. Fiuczynski, and L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *8th Symp. on Operating Systems Design and Implementation*, 2008.

[3] R. V. Binder. *Testing Object-oriented Systems Models, Patterns, and Tools*. Addison-Wesley, 2005.

[4] F. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison–Wesley, 1975 (revised 1995).

[5] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th Symp. on Operating Systems Design and Implementation*, 2008.

[6] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Intl. Conf. on Software Engineering*, 2009.

[7] S. Cornett. Code coverage analysis. http://www.bullseye.com/coverage.html, Dec 2008.

[8] P. J. Guo and D. Engler. Linux kernel developer responses to static analysis bug reports. In *USENIX Annual Technical Conference*, 2009.

[9] Oprofile. http://oprofile.sourceforge.net.

[10] A. H. Watson and T. J. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Computer Systems Laboratory, National Institute of Standards and Technology, 1996.

[11] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Using developer information as a factor for fault prediction. In *Intl. Conf. on Software Engineering*, 2007.