2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement

# The Impact of Coverage on Bug Density in a Large Industrial Software Project

Thomas Bach*, Artur Andrzejak*, Ralf Pannemans†, and David Lo‡
*Heidelberg University, Germany
†SAP SE, Germany
‡Singapore Management University, Singapore
thomas.bach@stud.uni-heidelberg.de, artur.andrzejak@informatik.uni-heidelberg.de
ralf.pannemans@sap.com, davidlo@smu.edu.sg

***Abstract*—Measuring quality of test suites is one of the major challenges of software testing. Code coverage identifies tested and untested parts of code and is frequently used to approximate test suite quality. Multiple previous studies have investigated the relationship between coverage ratio and test suite quality, without a clear consent in the results. In this work we study whether covered code contains a smaller number of future bugs than uncovered code (assuming appropriate scaling). If this correlation holds and bug density is lower in covered code, coverage can be regarded as a meaningful metric to estimate the adequacy of testing.**

**To this end we analyse 16 000 internal bug reports and bug-fixes of SAP HANA, a large industrial software project. We found that the above-mentioned relationship indeed holds, and is statistically significant. Contrary to most previous works our study uses real bugs and real bug-fixes. Furthermore, our data is derived from a complex and large industrial project.**

## I. Introduction

Software testing is a crucial and widely deployed tool for ensuring software quality. One of the practical challenges for software testing is measuring the quality and effectiveness of test suites. Measures for adequacy of testing are typically used to identify whether a software artefact is not tested sufficiently well, and where improvement is needed. They also play an important role to indicate that a sufficient amount of testing has been done, and resource costs for testing eventually surpass the expected savings from reductions in the amount and impact of defects.

One of the most widely-used measures for adequacy of testing is statement coverage. Its direct use is to identify uncovered parts of code which potentially contain further bugs, not caught by existing test code. The code coverage (ratio), i.e. ratio of covered lines to all lines, is frequently interpreted as a metric of test quality, with numerous organisations using it to set testing requirements. On the other hand, some studies question the very existence of a relationship between coverage and test suite effectiveness [1].

If a positive correlation between coverage ratio and test suite effectiveness exists, we expect to find a smaller number of future bug-fixes in the covered parts of code, see Figure 1. This observation has been used by Ahmed et al. [2] to design a simple schema to verify the benefits of coverage as a test quality measure. Essentially, they identify the amount of bugs found in covered and not covered part of code. If the coverage ratio is meaningless, then we expect that future bugs are distributed uniformly over covered and uncovered parts of the source code. However, if coverage ratio is meaningful, then the percentage of all future bugs found in the covered parts of the code should be *smaller* than the coverage ratio. Figure 2 illustrates this *binary testedness* approach.

While Ahmed et al. [2] use mutations as surrogates for bugs, we use records of real bugs and their bug-fixes. Another essential difference is that our data is collected from a very large industrial application with high reliability requirements. Our work improves the data collection process and data evaluation by introducing multiple collection points instead of using a single snapshot like the previous study of Ahmed et al. [2]. This reduces the risk of losing track of code changes over time, and increases the size of the data set for more robust evaluation.

Our subject of study is SAP HANA, a large industrial software system from SAP developed mostly in C and C++ for more than 10 years. SAP HANA is a relational database management system and the core product of several applications offered by SAP SE [3], [4], [5].

Following the binary testedness approach, we investigate in this work how many bugs occur in covered and uncovered parts of the source code of SAP HANA. Our findings show that indeed there were less bugs in the covered parts of the source code than expected from a uniform distribution. This effect is visible in 70 out of 72 time segments of our data, and is statistically significant for the mean. This indicates that the case of our test subject, increasing coverage ratio and enforcing high coverage goals can reduce the amount of defects. In contrast to previous studies, we use a large, real world software project, and real bug data.

The structure of the remainder of the paper is as follows. Section II describes our approach and data processing steps. In Section III we discuss our findings and possible threats to validity. Section IV presents the related work, and we state conclusions in Section V.

978-1-5090-4039-1/17 $31.00 © 2017 IEEE
DOI 10.1109/ESEM.2017.44

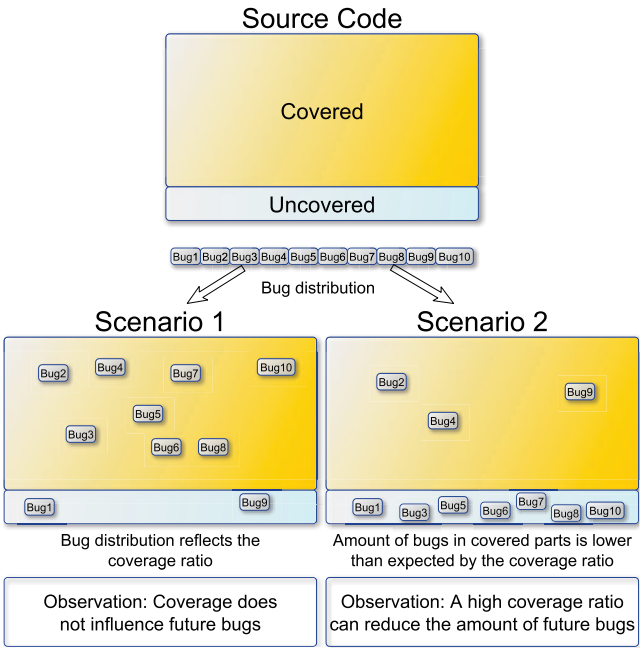


Figure 1. Exemplary coverage and bug distributions. Scenario 1 describes a situation where coverage ratio is meaningless for future code quality measured by the amount of bugs. Scenario 2 shows a situation where coverage is meaningful.

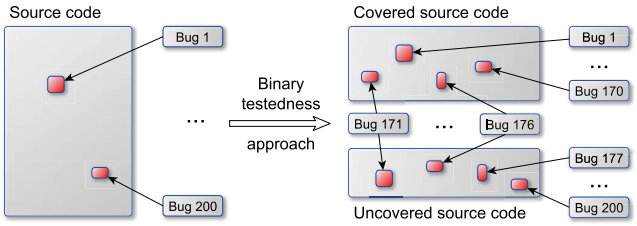


Figure 2. The binary testedness approach from [2] separates the source code in two (binary) groups and counts the number of bugs in the covered and uncovered group. In this example, 170 bugs occurred in the covered part of the source code, 24 in the uncovered part and 6 are undecided.

## II. Approach

This section describes the binary testedness approach from Ahmed et al. [2], introduces SAP HANA testing environment, and outlines our data collection and processing.

### A. Binary Testedness Approach

The binary testedness approach from Ahmed et al. [2] separates the source code in two (binary) groups: Covered and uncovered parts of the source code. Based on this separation at a given time $T$, all future bugs and the corresponding bug-fixes after $T$ are checked if they occur in the covered or the uncovered group, see Figure 2. A lower amount of bugs in covered parts (found covered bugs) compared to coverage ratio times all bugs (expected covered bugs) would indicate that coverage is *meaningful*.

Ahmed et al. use the *GumTree Differencing Algorithm* to identify the position of the original source code elements at time $T$ for a bug-fixing commit after $T$, even if the state of the elements changed between $T$ and the bug-fixing commit. For this purpose, GumTree utilizes an abstract syntax tree [6], see Figure 3 (upper part).

Instead of using coverage data for a single point $T$ in history, our approach utilizes the data of multiple coverage runs distributed from $T$ until the last observed bug-fixing commit. This subdivides the measurement period into multiple, non-overlapping *time segments*. Figure 3 (lower part) visualizes the concept and shows an example for the accuracy improvements in terms of more robust mappings between changed source code and coverage data.

Increasing the amount of coverage runs shortens the time segments, and therefore reduces the likelihood of intermediate conflicting commits between a coverage run and a bug-fix. For this study, we found that 72 time segments (roughly two coverage runs per week) ensured a mismatch rate below 1 % for all file modifications. A mismatch occurs if the original source code state of the bug-fixing commit is not equal to the state of the coverage run. The accuracy can be further improved up to a mismatch rate of zero by shortening the time segments between test runs at the cost of a higher resource investment.

### B. Testing Environment of SAP HANA

SAP performs a range of quality assurance processes to ensure the required quality of SAP HANA. This includes extensive testing with over 100 000 programmatically executed software tests [7]. The tests within the coverage profile have a cumulative runtime of over 250 hours. These tests check various aspects from functionality up to software performance and regressions.

SAP engineers use *DynamoRIO* to collect line based coverage data on a regular basis. Line based coverage data tells for each executable source code line if the line was executed ('hit') or not ('missed'). The coverage data is then used to identify untested parts of the software. The covered part of the source code contains all lines which were executed by at least one test, whereas the uncovered part contains lines which were not executed by any test. The uncovered part could be tested by manual tests or other test types without measured coverage, which is unknown at this level of analysis.

SAP uses a bug tracking tool to maintain defect history. For the purpose of this study, we classify defects by their detection time into *early detected* and *late detected* defects. Defects found before a change is merged into the source code repository are classified as *early*. Defects found later are classified as *late*. Early defects are detected, e.g., during development, by peer reviews or by pre-commit test runs. Late defects are detected, e.g., by extended continuous automatic regression testing, by manual testing, by internal usage of SAP HANA ('self-hosting'), by automatic tools e.g. fuzzy testing, or even by customer reports. This study

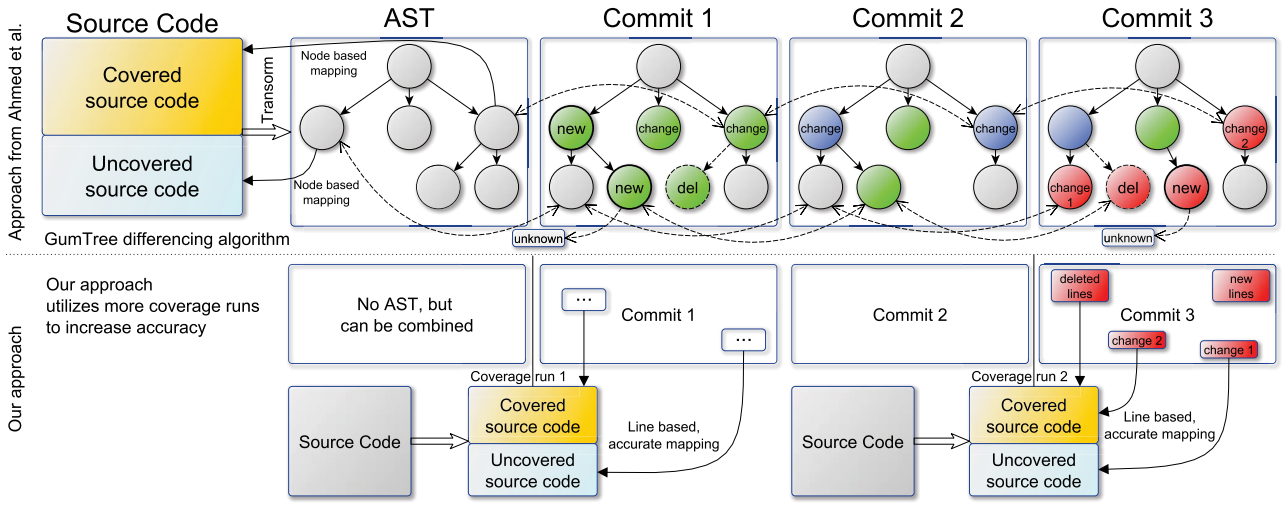


Figure 3. The GumTree approach used by Ahmed et al. [2] tracks the nodes of an abstract syntax tree (AST). This can reduce accuracy for consecutive changes. For commit 3, change 1 is uncovered, deleted and inserted nodes are unknown and change 2 is covered, but it is in fact unclear due to intermediate changes. Our approach can accurately map change 1, change 2 and the deletion from commit 3.

focuses on late defects, which tend to have a higher cost impact [8] and better documentation compared to defects during early local development. In the following, the term bug is synonymous with late detected defects.

This classification of the terms early and late bugs is a proprietary definition, which is different from common terminology, e.g. ISTQB test levels [9]. The ISTQB classification of test levels is linked to the responsibilities in a project. The distinction by responsibilities is not always possible in our case project. E.g., one test suite can contain component tests, integration tests, system tests, and regression tests. Our classification focus on the time of identification and distinguish between bugs that were detected by the current set of programmatically executed tests and bugs that were not detected by these set of tests. This allows us to apply the binary testedness approach from Ahmed et al. as described in Section II-A.

### C. Data Collection and Processing

Our experiment set-up requires coverage data, bug data, and a link between bugs and coverage. As described in Section II-B, line based coverage data is collected regularly. Each coverage-execution of a test suite creates a distinct coverage data file, which is aggregated with all other distinct coverage data files to a combined coverage data file.

We collect bugs as described in Section II-B and we assume that each entry in the bug tracking tool indicates a bug. Each bug entry either contains a link to one or multiple bug-fixing source code changes ('bug-fixing commit'), or the bug-fixing commit message contains the id of the bug.

This allows us to identify related source code changes for each bug and avoids the need to identify which commits fix bugs, and which e.g. introduce new features [2].

### D. Classifying Bug-Fixes by Coverage

For each bug, we must decide whether the corresponding bug-fixing commit changes occur in covered parts of the source code or in uncovered parts of the source code. The decision for a git commit follows the decision graph shown in Figure 4, which we explain in the following paragraphs.

The source code of SAP HANA is maintained in a git version control system [10]. Each git commit corresponds to incremental code changes as actually performed by the developers (i.e., we are not considering the delta to some fixed baseline).

A commit is considered covered if at least one chunk is covered and we call it *covered bug-fix*. This implies that the original behaviour of these covered chunks were executed by at least one test, but the test did not detect the bug.We consider at least one covered chunk as sufficient to classify the commit as a covered bug-fix, because a single covered chunk implies that the corresponding bug could have been possibly detected by at least one test. This definition can result in a larger set of covered bug-fixes than a manual decision by experts, because our covered bug-fixes may not contain covered parts of the source code which are relevant to the bug.

Analogously, if none of the existing tests executed any part of the original code of the bug-fixing commit, we call it an *uncovered bug-fix*. We use a third category for unclear cases and call a commit an *undecided bug-fix* if we cannot use any of the two previous categories.

File additions, deletions and moves are ignored. If any modification on file level occurs, we expect a change at any

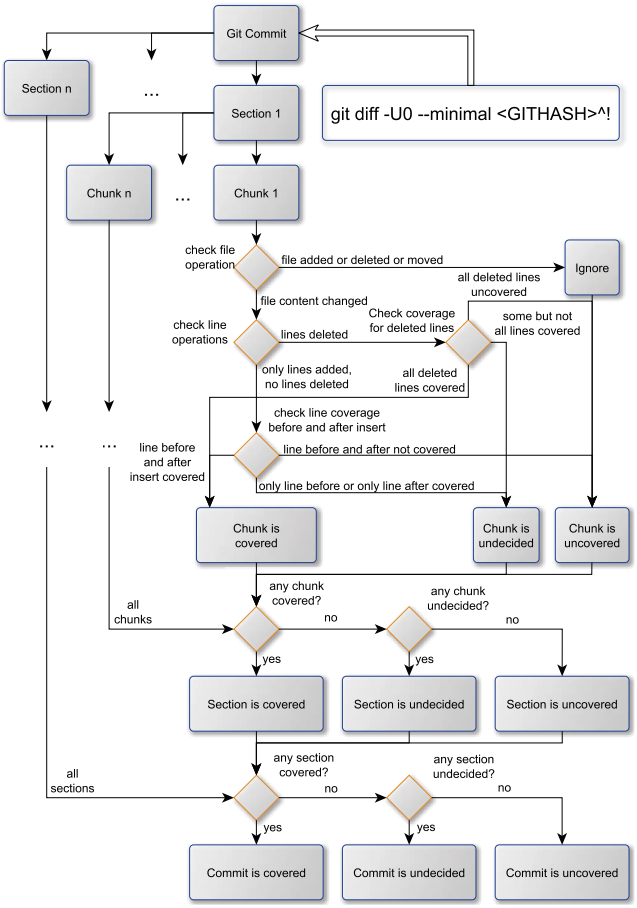


Figure 4. The process of classifying a bug-fixing git commit as covered, undecided, or uncovered (terms from Section II-D).

other point in the source code reflecting this modifications (e.g. class instantiation, function usage). We only classify this other change and ignore the file modification. There are counterexamples to this explanation, but we did not find any occurrence in practice. Additionally for file moves, the frequency of file moves is less than 1 % and the classification is complex, therefore we decided to ignore them for any classification.

Line additions are only classified as covered if the original lines before and after the addition point are covered. We found several examples invalidating a less strict approach. We only expect errors if the source code is badly formatted, e.g. there is no space between two functions and a new function was inserted or goto is used. We did not find such counterexamples.

We classify bug-fixes according to< coverage before the corresponding tests for bug identification are created and measured in coverage runs. SAP engineers create tests for bugs which are used to reproduce the bug and assure the removal of the bug. We use coverage data before such tests are included within the coverage runs, therefore such tests do not influence the classification of a current commit. They can influence the classification of future commits, e.g. if a second patch is necessary to fully fix a bug.

## III. Empirical Results

This section presents the results of data processing and attempts to answers our main research question on the impact of code coverage. We will first discuss the results and implications and then highlight possible threats.

### A. Results

**Data processing.** For the time frame from May 2016 to April 2017, we collected 72 coverage runs and 16 215 bug-fixing commits which represent the same amount of bugs. For these bug-fixing commits, we extracted 76 979 sections from the diff output from git with cumulatively 376 364 chunks. Among them, 239 119 chunks modify files contained in the coverage data and only 4 482 chunks cannot be used because of source mismatch. We found that 94 891 chunks occur in covered parts of the source code, 101 907 chunks occur in uncovered parts of the source code, and for 37 839 chunks it cannot be decided. Based on these results we identified 24 571 sections as covered, 15 210 sections as uncovered, and 7 483 sections as undecided.

As shown in Table II, cumulatively 8 348 (or 51.48 %) of all bug-fixes occurred in covered parts of the source code, 6 171 (38.06 %) are uncovered, and 1 696 (10.46 %) of all bug-fixes are of type undecided.

**Testedness results.** To implement the testedness criterion from Section II-A we compare the observed number $N_{obs}$ of bugs found in tested code against the *expected* number $N_{exp}$ of bugs within the covered code, where $N_{exp}$ is computed under the assumption that coverage level is meaningless (null hypothesis). Obviously $N_{exp}$ can be approximated as number of all found bugs times the coverage ratio. For example, in Figure 1 $N_{exp} = 8$ based on Scenario 1, but $N_{obs} = 3$ based on Scenario 2. If $N_{obs}$ is lower than $N_{exp}$, we can conclude that higher coverage levels correlate with higher software quality in our test subject.

We approximate the location of bugs in the source code by the location of bug-fixing commits (Section II-C). Furthermore, to be on the safe side, we conservatively assume that undecided bug-fixes are covered bug-fixes. In other words, we compute $N_{obs}$ as the sum of covered bug-fixes and undecided bug-fixes.

Contrary to [2] we perform this comparison for *each* of the 72 time segments described in Section II-A, and not only for a single snapshot. For each segment we use 'local' coverage ratio (and local numbers of covered / uncovered / undecided bug-fixes). For confidentiality reasons, we cannot explicitly state these coverage ratios.

Figure 5 shows a comparison of the numbers of covered bug-fixes plus undecided bug-fixes (sums $N_{obs,i}$) versus the numbers $N_{exp,i}$ of covered bug-fixes expected under the null hypothesis, for each segment $i = 0, 1, \ldots, 70$. Only

Table I
PREPROCESSING STATISTICS FOR THE BUG-FIXING GIT COMMITS COLLECTED FROM MAY 2016 TO APRIL 2017 (1 YEAR).

| Metric | Number |
|---|---|
| Lines of executable code | Several millions |
| Number of source files | > 25 000 |
| Full coverage runs | 72 |
| Bug-fixing commits | 16 215 |
| For bug-fixing commits: | |
| Diff sections | 76 979 |
| Sections covered | 24 571 |
| Sections uncovered | 15 210 |
| Sections undecided | 7 483 |
| Files added | 5 595 |
| Lines in new files | 1 044 451 |
| Files deleted | 1 156 |
| Lines in deleted files | 279 674 |
| Files moved | 89 |
| Files with content changes | 70 139 |
| Lines added in changes | 770 325 |
| Lines deleted in changes | 471 926 |
| Files with coverage information | 47 264 |
| Files without coverage information | 22 875 |
| For sections: | |
| Diff chunks | 376 364 |
| Average number of chunks per section | 4.89 |
| Average number of chunks per commit | 23.21 |
| Skipped chunks source mismatch | 4 482 |
| Percentage | 0.01 % |
| Chunks with coverage information | 239 119 |
| Chunks covered | 94 891 |
| Chunks uncovered | 101 907 |
| Chunks undecided | 37 839 |

Table II
NUMBERS OF BUG-FIXING COMMITS (BUG-FIXES) IN OUR DATA SET BY CATEGORIES DEFINED IN SECTION II-D.

| Metric | Number | Percentage |
|---|---|---|
| Total number of bug-fixes | 16 215 | 100% |
| Bug-fixes in covered source code | 8 348 | 51.48 % |
| Bug-fixes in uncovered source code | 6 171 | 38.06 % |
| Bug-fixes with undecided coverage | 1 696 | 10.46 % |

the first 71 segments are shown for presentation reasons. Figure 6 shows the differences $N_{exp,i} - N_{obs,i}$ in greater detail.

For all but 2 out of the 72 time segments we have $N_{exp,i} > N_{obs,i}$ which suggests that coverage is a meaningful metric. At the same time, the relative reduction of the number of bug-fixes per segment is not large. However, this can be in part attributed to our conservative way of treating undecided bug-fixes as covered bug-fixes.

We also applied the Wilcoxon signed-rank test to reject the null hypothesis that the mean of expected numbers of covered bug-fixes $N_{exp,i}$ and the mean of numbers of covered plus undecided bug-fixes $N_{obs,i}$ are equal. The test confirms this with $p$-value less than $2.2e-16$, and effect size $r = 0.612$ which is considered as large ($r > 0.5$).

This non-parametric test is a paired difference test, i.e. we assume that the samples of covered bug-fixes

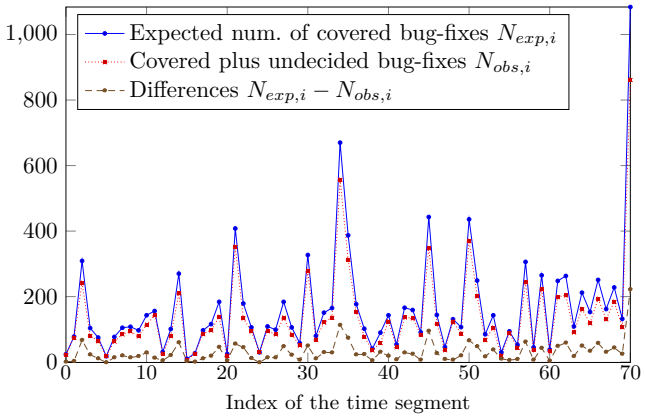


Figure 5. Expected numbers of bug-fixes $N_{exp,i}$, numbers of covered plus undecided bug-fixes $N_{obs,i}$, and their differences for time segments $i = 0, \ldots, 70$.

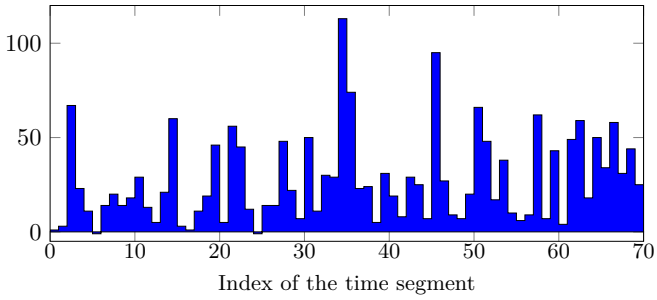


Figure 6. Differences $N_{exp,i} - N_{obs,i}$ for 71 time segments. $y$-values > 0 indicate that less bugs occurred than expected.

and uncovered bug-fixes are dependent. This is quite likely and indicated by Figure 5. However, we have also applied the Wilcoxon-Mann-Whitney-Test which assumes two independent samples. Also here the null hypothesis was rejected ($p$-value = 0.06322) but with small to medium effect size ($r = 0.13$).

The numbers of bugs (represented by bug-fixes) in covered code are smaller than expected if code coverage would be meaningless. This holds for 70 ouf of 72 time segments and is statistically significant for the means.

### B. Discussion of the Results

Our results provide a fairly good support for the thesis that higher coverage levels imply lower levels of future bug-fixes for our test subject. This is in particular strengthened by the fact that in 70 out of 72 time segments the results agree with the hypothesis. In other words, the positive effect could be observed consistently over time, for large variation of bug numbers. This supports the argument that setting (and reaching) goals for coverage ratio can have a

positive effect on the overall quality of our test subject by reducing the amount of future bugs.

There are multiple possible explanations for these results. Of course, the most plausible and widely assumed one is that the tests covering code detect bugs often, and since most of these defects are fixed, less of them remain.

We also consider an alternative interpretation of the results: that uncovered parts of the source code might include code for which it is difficult to achieve coverage. One well-known example are execution paths in face of rare error conditions, e.g. if `malloc` returns a null pointer. Such return values might not be checked correctly in code because they are not trivial to test, leading to failures if these rare error conditions occur during production usage. A second example is code with a high amount of dependencies. This code requires additional effort to test, because all of the dependency interaction must be simulated. A third example comprises code handling external input, e.g. SQL queries. In this case is it hardly feasible to test all possible valid and invalid input combinations and fuzzing techniques can require a high time and resource costs.

### C. Threats to Validity

Possible threats to our results contain (non-)causality, bad bug classification, and wrong coverage granularity.

Based on our results, we can conclude that fewer bug-fixes occurred in covered parts of the source code then expected. However, we cannot confirm the causality, i.e., we can not conclude that testing caused the reduced numbers of bugs.

We used an existing bug database to retrieve all bugs. These bugs represent only late defects (see Section II-B). We argue that these bugs are the more interesting defects, because they are more expensive to fix [8], and apparently harder to detect. Analysing all defects could generate different results for our research question. From our experience, it seems impractical to log all 'easy' and early defects during the development, and distinguishing between early and late bugs is hardly possible for young projects.

The bug classification within the bug tracking tool could be wrong. Developers could misuse the bug category for enhancements or other tasks. The bug tracking tool contains different categories for entries, therefore we do not expect that developers mislabel entries intentionally. But their decision could be wrong. 88 % of all entries are bugs, the remaining 12 % consists of the categories feature, enhancement, and performance. In addition, SAP enforces a strict policy of bug-labelling. These measures reduce the likelihood of this mislabelling to happen.

We also investigated the possibility that a bug-fixing commit does not only contain the bug-fix, but also unrelated code changes. Herzig et al. found that 15 % of all bug-fixes in five open-source projects contain unrelated changes according to [11]. One of the reasons why such unrelated changes happen is the boyscout-principle: 'Leave the campground cleaner than you found it' [12]. According to SAP engineers, unrelated changes happen rarely, because developers focus on fixing the bug and avoid to introduce unrelated regressions in the same bug-fixing commit. Enforced commit based code reviews also reduce the probability that unrelated changes are introduced in the same commit as the bug-fixing change. If unrelated changes happen, this could only increase the amount of bugs in covered code, and can not decrease it.

Our coverage data is line based. A finer granularity (e.g. statement coverage) could produce different results for our research question. We expect only a minor effect, because bug-fixing commits occur rarely in a fine granularity.

Coverage from different test types is mixed. The large corpus of tests for coverage creation contains a mix of e.g. component tests, integration tests, regression tests, system tests, performance tests. It is not clear how this affects the result.

Finally, we only investigate one large industrial system. We do not have access to a second industrial software projects with this size and similar test environment and test data. It is unclear if our results can be reproduced in other large industrial software systems.

## IV. Related Work

There are a number of research publications analysing the question whether code coverage and coverage goals are beneficial for measuring and improving software quality.

**Works supporting benefits of code coverage.** These publications often study correlation between code coverage ratio and number of bugs in different test subjects. For example, Mockus et al. investigate the correlation between coverage ratio and probability of defects affecting a component on two industrial software projects and the effort required to increase the coverage ratio [13]. They find that an increase in coverage leads to a proportional decrease in defects without any indication of diminishing returns. Ahmed et al. analyse 49 open source projects to understand the correlation between coverage and bugs [2] (see also Section II-A). They find a weak but significant correlation between statement coverage and number of bug-fixes. Kochhar et al. investigate two large software systems to compute correlations between coverage, test suite size, and its effectiveness. They find moderate to strong correlation between coverage and test suite effectiveness [14]. A metastudy by Inozemtseva et al. showed that eight out of 12 studies confirmed a positive correlation between coverage and effectiveness of a test suite [1].

**Works contradicting the benefits of code coverage.** While also here correlation analysis is used, some authors consider more and other variables than only coverage and bugs. For example, Inozemtseva et al. study correlation between coverage and test suite effectiveness in five open source systems, but they also control the test suite size for their experiments [1]. They find that it is generally not safe to assume that test suite effectiveness

is strongly correlated with coverage. In fact, their results indicate that test suite size is correlated with test suite effectiveness and code coverage does not show a strong correlation with test suite effectiveness if the test suite size is controlled. Namin et al. indicate similar results for smaller study subjects and investigate multiple models for the correlation [15]. Groce et al. provides an extensive overview for existing work highlighting limitations [16].

**Mixed results.** Some works claim that the answer depends on the exact question, definitions of the terms, and potentially on more variables [15]–[18]. For example, Groce et al. argue in their survey that there is uncertainty as to what exactly measuring code coverage should achieve, as well as how we would know if it can, in fact, achieve it. They develop a strong and weak coverage hypothesis to provide a meaningful context for further discussion and experiment design [16].

## V. Conclusions

We applied the binary testedness approach from Ahmed et al. [2] to SAP HANA, a large industrial software project. Instead of using mutants, we used a large set of real bugs and bug-fixing commits. In addition, we introduced multiple data collection points to reduce the risk of losing track of code changes over time. Our results show that a significantly lower number of bugs occur in covered parts of the source code in our test subject than expected if coverage would be meaningless. For practitioners, our results suggest that setting (and reaching) goals for coverage ratio has a positive effect on the overall quality of our test subject in terms of the amount of future bugs. This confirms previous conclusions from Ahmed et al. [2] and Mockus et al. [13].

For SAP engineers, our results confirm the expectation of engineers and management, that measuring coverage and enforcing coverage goals can be beneficial to the quality of SAP HANA. The internal impact of this study can not be measured, because changes to QA policies require more time and considerations and could require further investigations as highlighted in future work.

There are several directions to extend our results in future work. Similar to work of Mockus et al [13], the binary testedness approach could be used on a component level instead of a system level. A component level analysis could produce comparable results between the different components in SAP HANA and could reveal positive or negative factors. Also, a long term experiment over different release cycles of a software project might disclose whether different coverage goals had an impact on the number of bugs. Mockus et al. [13] found that 'there is no indication of diminishing returns (when an additional increase in coverage brings smaller decrease in fault potential)'. It would be interesting to replicate our study on large open source projects to investigate if similar findings can be observed for such projects.

Furthermore, we plan to consider the test multiplicity (i.e. number of test suites covering a code line), as it is likely that more intensely tested code shows less future bugs. For example, is there a difference in the amount of future bugs for a the source code line whether it was executed and tested by 100 tests or by only 1 test?

## References


[1] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *ICSE*, 2014, pp. 435–445. (Cited from: I and IV)

[2] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen, "Can testedness be effectively measured?" in *FSE*, 2016, pp. 547–558. (Cited from: I, 2, II, II-A, 3, II-C, III-A, IV, and V)

[3] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The sap hana database – an architecture overview." *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012. (Cited from: I)

[4] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: Data management for modern business applications," *SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, Jan. 2012. (Cited from: I)

[5] N. May, A. Böhm, and W. Lehner, "SAP HANA - the evolution of an in-memory DBMS from pure OLAP processing towards mixed workloads," in *BTW*, 2017, pp. 545–563. (Cited from: I)

[6] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ASE*, 2014, pp. 313–324. (Cited from: II-A)

[7] T. Bach, A. Andrzejak, and R. Pannemans, "Coverage-based reduction of test execution time: Lessons from a very large industrial project," in *ICSTW*, March 2017. (Cited from: II-B)

[8] B. W. Boehm and P. N. Papaccio, "Understanding and controlling software costs," *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, pp. 1462–1477, Oct. 1988. (Cited from: II-B and III-C)

[9] "International software testing qualifications board (istqb)," http://www.istqb.org/, accessed: 2017-06-21. (Cited from: II-B)

[10] "Git distributed version control system," https://git-scm.com/, accessed: 2017-06-21. (Cited from: II-D)

[11] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 121–130. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487113 (Cited from: III-C)

[12] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. (Cited from: III-C)

[13] A. Mockus, N. Nagappan, and T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *ESEM*, 2009, pp. 291–301. (Cited from: IV and V)

[14] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *SANER*, 2015, pp. 560–564. (Cited from: IV)

[15] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 57–68. (Cited from: IV)

[16] A. Groce, M. A. Alipour, and R. Gopinath, "Coverage and its discontents," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 2014, pp. 255–268. (Cited from: IV)

[17] B. Smith and L. A. Williams, "A survey on code coverage as a stopping criterion for unit testing," North Carolina State University. Dept. of Computer Science, Tech. Rep., 2008. (Cited from: IV)

[18] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997. (Cited from: IV)