Did We Test Our Changes? Assessing Alignment between Tests and Development in Practice

Sebastian Eder, Benedikt Hauptmann, Maximilian Junker
Technische Universität München, Germany

Elmar Juergens
CQSE GmbH, Germany

Rudolf Vaas, Karl-Heinz Prommer
Munich Re Group, Germany

***Abstract*—Testing and development are increasingly performed by different organizations, often in different countries and time zones. Since their distance complicates communication, close alignment between development and testing becomes increasingly challenging. Unfortunately, poor alignment between the two threatens to decrease test effectiveness or increases costs.**

In this paper, we propose a conceptually simple approach to assess test alignment by uncovering methods that were changed but never executed during testing. The paper's contribution is a large industrial case study that analyzes development changes, test service activity and field faults of an industrial business information system over 14 months. It demonstrates that the approach is suitable to produce meaningful data and supports test alignment in practice.

***Index Terms*—Software testing, software maintenance, dynamic analysis, untested code**

I. Introduction

A substantial part of the total life cycle costs of long-lived software systems is spent on testing. In the domain of business-information systems, it is not uncommon that successful software systems are maintained for two or even three decades. For such systems, a substantial part of their total lifecycle costs is spent on testing to make sure that new functionality works as specified, and—equally important—that existing functionality has not been impaired.

During maintenance of these systems, test case selection is crucial. Ideally, each test cycle should validate all implemented functionality. In practice, however, available resources limit each test cycle to a subset of all available test cases. Since selection of test cases for a test cycle determines which bugs are found, this selection process is central for test effectiveness.

A common strategy is to select test cases based on the changes that were made since the last test cycle. The underlying assumption is that functionality that was added or changed recently is more likely to contain bugs than functionality that has passed several test cycles unchanged. Empirical studies support this assumption [1], [2], [3], [4].

If development and testing efforts are not aligned well, testing might focus on code areas that did not change, or—more critically—substantial code changes might remain untested. Test alignment depends on communication between testing and development. However, they are often performed by different teams, often located in different countries and time-zones. This distance complicates communication and thus challenges test alignment. But how can we assess test alignment and expose areas where it needs to be improved?

Problem: We lack approaches to determine alignment between development and testing in practice.

Proposed Solution: In this paper, we propose to assess test alignment by measuring the amount of code that was changed but not tested. We propose to use *method-level change coverage* information to support testers in assessing test alignment and improving test case selection.

Our intuition is that changed, but untested methods are more likely to contain bugs than either unchanged methods or tested ones. However, our intuition might be dead wrong: method-level churn could be a bad indicator for bugs, since methods can contain bugs although they have not changed in ages.

Contribution: This paper presents an industrial case study that explores the meaningfulness and helpfulness of method-level change coverage information. The case study was performed on a business information system owned by Munich Re. System development and testing were performed by different organizations in Germany and India. The case study analyzed all development changes, testing activity, and all field bugs, for a period of 14 months. It demonstrates that field bugs are substantially more likely to occur in methods that were changed but not tested.

II. Related Work

The proposed approach is related to the fields of defect prediction, selective regression testing, test case prioritization, and test coverage metrics. The most important difference to the named topics is the simplicity of the proposed approach and the fact that change coverage *assesses* the executed subsets of test suites, but does not give hints to improve them.

Defect prediction is related to our approach, because we identify code regions that were changed, but remained untested, with the expectation that there are more field bugs.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A". The responsibility for this article lies with the authors.

There are several models for defect prediction [5]. In contrast to these models, we measure only changes in the system and the coverage by tests and do not predict bugs, but assess test suites and use the probability of bugs in changed, but untested code as validation of the approach.

The proposed approach is related to [6], which uses series of changes "change bursts" to predict bugs. The good results that were achieved by using change data for defect prediction encourage us to combine similar data with testing efforts.

Selective regression testing techniques target the selection of test cases from changes in source code and coverage information. [7], [8], [9].

In contrast to these approaches, the paper at hand focuses on the assessment of already executed test suites, because often experts decide which tests to execute to cover most of the changes made to a software system [10]. However, their estimations contain uncertainties and therefore possibly miss some changes. Our approach aims at identifying the resulting uncovered code regions. Therefore, our approach can only be used if testing activities were already performed.

Compared to [11], we are validating our approach by measuring field defects, and do not take defects into account that were found during development.

Test coverage metrics give an overview of what is covered by tests. Much research has been performed in these topics [12] and there is a plethora of tools [13] and a number of metrics available, such as statement, branch, or path coverage [14]. In contrast to these metrics, we focus on the more coarse grained method coverage. Furthermore, we do not only consider static properties of the system under test, but changes.

Empirical studies on related topics focus to the best of our knowledge mainly on the effectiveness of test case selection and prioritization techniques [9], [15]. In our study, we assess test suites by their ability to cover changes of a software system, but do not consider sub sets of test suites.

III. Context and Terms

In this work, we focus on *system testing* according to the definition of IEEE Std 610.12-1990 [16] to denote *"testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements"*. System tests are often used to detect bugs in existing functionality after the system has been changed. In our context, many tests are executed manually and denoted in natural language.

Our study uses *methods* as they are known from programming languages such as Java or C#. Methods form the entities of our study and can be regarded as units of functionality of a software system. They are defined by a signature and a body. To compare different releases of a software system over time, we create *method genealogies* which represent the evolution of a single method over time. A genealogy connects all releases of a method in chronological order [17].

In the context of our work, the life cycle of a software system consists of two alternating phases (see Figure 1). In the *development phase*, existing functionality is maintained

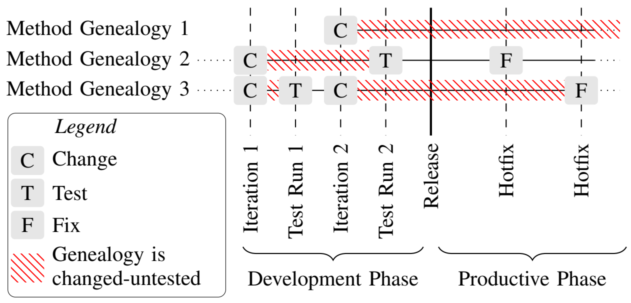

Fig. 1. Development life-cycle

or new features are developed. Development usually occurs in *iterations* which are followed by *test runs* which are the execution of a selection of tests aiming to test regressions as well as the changed or added code. A development phase is completed by a *release* which transfers the system into the *productive phase*. In the productive phase, functionality is usually neither added nor changed. If critical malfunctions are detected, *hot fixes* are deployed in the productive phase.

We consider a method as *tested* if it has been executed during a test run. If a method has been changed or added and been tested afterwards before the system is released we consider it as *changed-tested*. If a method change or addition has not been tested before the system is transferred in the productive phase, we consider the method as *changed-untested* (see genealogy 1 and 3 in Figure 1).

IV. Change Coverage

To quantify the amount of changes covered by tests, we introduce the metric *change coverage (CC)*. It is computed by the following formula and ranges between [0,1].

$$\text{change coverage} = \frac{\text{\#methods changed-tested}}{\text{\#methods changed}}$$

A change coverage of 1 ($CC = 1$) means that all methods which have been changed since the last test run have been tested after their last change. On the contrary, a coverage of 0 ($CC = 0$) indicates that none of the changed methods have been covered by a test.

V. Case Study

A. Goal and Research Questions

The goal of the study is to show whether change coverage is a useful metric for assessing the alignment between tests and development. We formulate the following research questions.

RQ 1: ***How much code is changed, but untested?*** The goal of this research question is to investigate the existence of changed, but untested code, to justify the problem statement of this work. Therefore, we quantify changed and untested code.

RQ 2: ***Are changed-untested methods more likely to contain field bugs than unchanged or tested methods?*** The goal of this research question is to decide whether change coverage can be used as a predictor for bugs in large code regions and is therefore useful for maintainers and testers to identify relevant gaps in their test coverage.

B. Study Object

We perform the study on a business information system at Munich Re. The analyzed system was written in C# and its size are 340 kLOC. In total, we analyzed the system for 14 months. The system has been successfully in use for nine years and is still actively used and maintained. Therefore, there is a well implemented bug tracking and testing strategy. This allows us to gain precise data about which parts of the system were changed and why they were changed.

We analyzed two consecutive releases of the system. Release 1 was developed in five iterations in two months, and release 2 was developed in ten iterations in four months. Both releases were deployed to the productive environment due to hot fixes five times and were in productive use for six months. Note that one deployment may concern several bugs and changes in the system. The system contained 22123 (release 1) respectively 22712 (release 2) methods.

For both releases, test suites containing 65 system test cases covering the main functionality were executed three times.

C. Study Design and Execution

For all research questions, we classify methods according to the categories shown in Figure 2: Tested or untested, changed or unchanged, and whether methods contain field bugs.

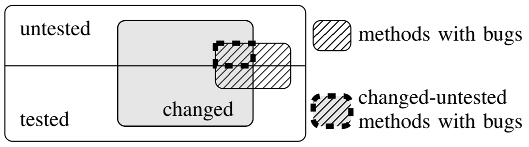

Fig. 2. Method categories used to evaluate change coverage

Study Design: First, we collect coverage and program data, then we answer RQ 1 and RQ 2 based on the collected data.

For answering RQ 1, we build method genealogies and identify changes during the development phase and relate usage data to these genealogies. With this information, we identify method genealogies that are changed-untested.

For answering RQ 2, we calculate the probability of field defects for every category of methods by detecting changes in the productive phase of the system in retrospective. This is valid for the analyzed system, since only severe bugs are fixed directly in the productive environment, which is defined by the company's processes.

We gain our results by identifying methods that are changed in the productive phase, which means they were related to a bug. We then categorize methods by change and coverage during the development phase. Based on this, we calculate the bug probability in the different groups of methods.

Study Execution: We used tool support, which consists of three parts: An ephemeral [18] profiler that records which methods were called within a certain time interval, a database that stores information about the system under consideration, and a query interface that allows retrieving coverage, change, and change coverage information. The same tool support was used in earlier studies [17], [19].

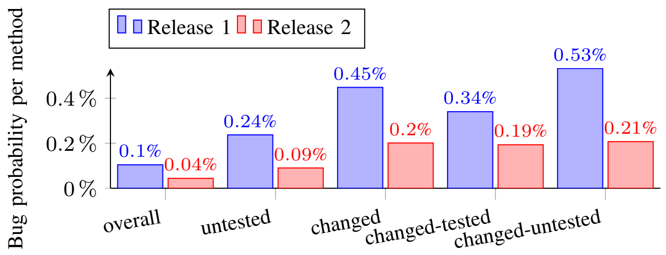

Fig. 3. Probability of fixes in both releases

Validity Procedures: We focus on validity procedures and not on threats to validity due to space limitations.

We conducted manual inspections to ensure that every bug that is identified by our tool support is indeed a bug.

To confirm the correctness of method genealogies we build based on locality and signatures, we conducted manual inspections of randomly chosen method genealogies. We found no false genealogies and have therefore a high confidence in the correctness of our technique. We also used the algorithm in our former work [17], which provided suitable results as well.

D. Results

RQ 1: Untested methods account for 34% in both releases we analyzed. 15% of all methods were changed during the development phase of the system, also in both releases. The equality of the numbers for both releases is a coincidence.

8% respectively 9% of all methods were changed-untested. Considering only changed methods, only 44% were tested in release 1 and 45% of these methods were tested in release 2. These numbers constitute that there are gaps in the test coverage of changed code in the analyzed system.

RQ 2: We found 23 fixes in release 1 and 10 fixes in release 2. The distribution of the bugs over the different change and coverage categories of methods is shown in Table I. The biggest part of bugs occurred in methods categorized as changed-untested with 43% of all bugs in release 1 and 40% of all bugs in release 2. In both releases, there are considerably less bugs in unchanged regions than in changed regions.

The probabilities of bugs are shown in Figure 3. With 0.53% in release 1 and 0.21% in release 2, the probability of bugs is higher in the group of methods that were changed-untested. This confirms that tested code or code that was not changed in the development phase is less likely to contain field defects.

E. Discussion

RQ 1: With 15% of all methods being changed and 34% of all methods being not tested, untested code and changed code plays a considerable role in the analyzed system. The high amount of changed methods results from newly developed features, which means that many methods were added during the development phase of both releases.

TABLE I
DISTRIBUTION OF FIXES OVER THE DIFFERENT CATEGORIES

	Release 1		Release 2	
Category	Absolute	Relative	Absolute	Relative
changed-tested	5	22%	3	30%
changed-untested	10	43%	4	40%
unchanged-tested	0	0%	0	0%
unchanged-untested	8	35%	3	30%

43% respectively 40% of the changed methods were not tested in the analyzed system. These high numbers also result from features that are newly developed during the development phase. For these new features, there was only a very limited number of test cases.

RQ 2: With a probability of bugs in untested-changed methods of 0.53% respectively 0.21%, this group of methods contains most of the bugs. This means that the system itself contains few bugs at the current stage of development and bugs are brought into the system by changes.

Furthermore, the probability of bugs in untested code is, in both releases, less than half of the probability in changed-untested code. Hence, we conclude that only considering test coverage is not as efficient as considering change coverage.

The probability of bugs in changed code regions is also considerably higher than in untested regions. But the combination of both metrics, test coverage and changed methods points to code regions that are more likely to contain bugs than others.

Is Change Coverage Helpful in Practice? We employed the proposed approach also in the context of Munich Re in currently running development phases. We showed the results to developers and testers by presenting code units, like types or assemblies ordered by change coverage. During the discussion of the results, we conducted open interviews with developers to gain knowledge about how helpful information about change coverage is during maintenance and testing.

Developers identified meaningful methods in changed but untested regions by using the static call graph to find methods they know. With these methods, the developers were able to identify features that remained untested. For example the processing of excel sheets in a particular calculation was changed, but remained untested afterwards. In this case, among some others, the (re-)execution of particular test cases and the creation of new test cases were issued. This increased the change coverage considerably for the code regions where the features are located. This shows that change coverage is helpful for practitioners.

VI. Conclusion and Future Work

We presented an automated approach to assess the alignment of test suites and changes in a simple and understandable way. Instead of using rather complex mechanisms to derive code units that may be subject to changes, we are focusing on changed but untested methods and calculate an expressive metric from these methods. The results show that the use of change coverage is suitable for the assessment of the alignment of testing and development activities.

We also showed that change coverage is suitable for guiding testers during the testing process. With information about change coverage, testing efforts can be assessed and redirected if necessary, because the probability of bugs is increased in changed-untested methods. Furthermore, we presented our tool support that allows us to utilize our technique in practice.

However, the number of bugs we found is too small to derive generalizable results. Therefore, we plan to extend our studies to other systems to increase external validity. But the first results that we presented in this work point out that the consideration of code regions that are modified, but not very well tested is important. This motivates future work on the topic and the inference of improvement goals.

One challenge is the identification of suitable test cases from code regions to give hints to testers and developers which test case to execute to cover more changed, but untested methods. Therefore, we plan to evaluate techniques related to trace link recovery to bridge the gap to test cases.

References

[1] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE*, 2005.

[2] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality," in *ICSE*, 2008.

[3] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, 2000.

[4] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ISSTA*, 2004.

[5] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, 2012.

[6] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *ISSRE*, 2010.

[7] V. Channakeshava, V. K. Shanbhag, A. Panigrahi, R. Sisodia, and S. Lakshmanan, "Safe subset-regression test selection for managed code," in *ISEC*, 2008.

[8] Y.-F. Chen, D. Rosenblum, and K.-P. Vo, "Testtube: a system for selective regression testing," in *ICSE*, 1994.

[9] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," in *ICSE*, 1998.

[10] M. Harrold and A. Orso, "Retesting software during development and maintenance," in *FoSM*, 2008.

[11] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *ISSTA*, 2002.

[12] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, 1997.

[13] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing tools," in *AST*, 2006.

[14] Y. Malaiya, M. Li, J. Bieman, and R. Karcich, "Software reliability growth with test coverage," *IEEE Trans. Rel.*, vol. 51, no. 4, 2002.

[15] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, 2001.

[16] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," New York, USA, 1990.

[17] S. Eder, M. Junker, E. Jurgens, B. Hauptmann, R. Vaas, and K. Prommer, "How much does unused code matter for maintenance?" in *ICSE*, 2012.

[18] O. Traub, S. Schechter, and M. D. Smith, "Ephemeral instrumentation for lightweight program profiling," School of engineering and Applied Sciences, Harvard University, Tech. Rep., 2000.

[19] E. Juergens, M. Feilkas, M. Herrmannsdoerfer, F. Deissenboeck, R. Vaas, and K. Prommer, "Feature profiling for evolving systems," in *ICPC*, 2011.