Test Impact Analysis: Detecting Errors Early Despite Large, Long-Running Test Suites

Elmar Jürgens
CQSE GmbH
juergens@cqse.eu

Dennis Pagano
CQSE GmbH
pagano@cqse.eu

Andreas Göb
CQSE GmbH
goeb@cqse.eu

***Abstract*—Large test suites often take a long time to execute. Therefore, in practice, they are usually not executed as part of continuous integration (CI), but only less frequently and in later test phases. As a result, many errors remain unrecognized during CI and are found late, which causes high costs.**

Test Impact Analysis allows us to run only those tests affected by code changes since the last test run. As a result, it is possible to only execute that section of a large test suite during CI that is most likely to find new errors. In our studies, we were able to spot 90% of failed builds in 2% of the test execution time. This enables fast CI feedback with high error detection rates, regardless of the size and duration of the entire test suite.

I. Motivation

As software grows, its test suite must also grow in order to reliably detect bugs. But this also increases the total test execution time. In practice, we are increasingly seeing test suites that run for several hours or even days.

In our experience, however, long-running test suites are typically excluded from continuous integration (CI). Often, they are executed only in subsequent test phases that take place, e.g., before each release. Due to the less frequent execution of the test suite, however, the period between error introduction and error detection grows. This has extensive negative consequences:

- Errors can overlap each other and can only be detected after other errors have been corrected.
- Debugging regression errors becomes more complex, as an increasing number of code changes may have caused the error.
- Bug fixes take longer because developers have to invest more time understanding their own code if an error is discovered weeks after it has been introduced.

These effects increase as systems and test suites grow. Ironically, this jeopardizes the effectiveness of continuous integration, especially for large systems. These often have a particularly high strategic value, and hence short feedback cycles would be particularly important. How can we prevent this and quickly find new bugs despite longer execution times of our test suites in continuous integration?

Our Test Impact Analysis (TIA) approach addresses this problem by selecting and prioritizing test cases based on code changes. This is done by analyzing the code changes that have been made since the last successful test run. Then, only those test cases are selected that run through changed code. These tests are then sorted so that they find errors as early as possible in the test run. The more frequently TIA is performed, the less changes are made between two test runs, and the greater the expected execution time savings are.

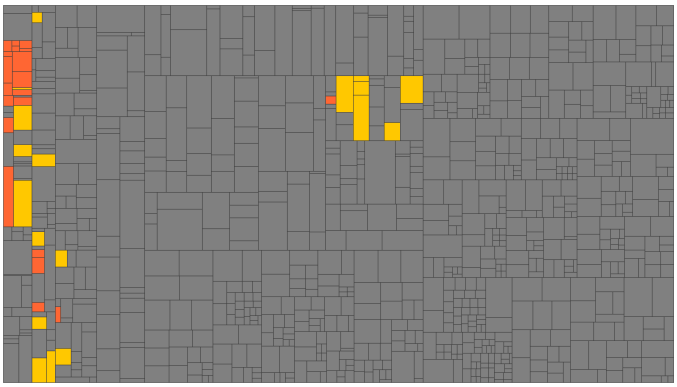

Figure 1. Changes for implementing a feature: Modified code is shown in yellow, new code is red, and unchanged code is gray.

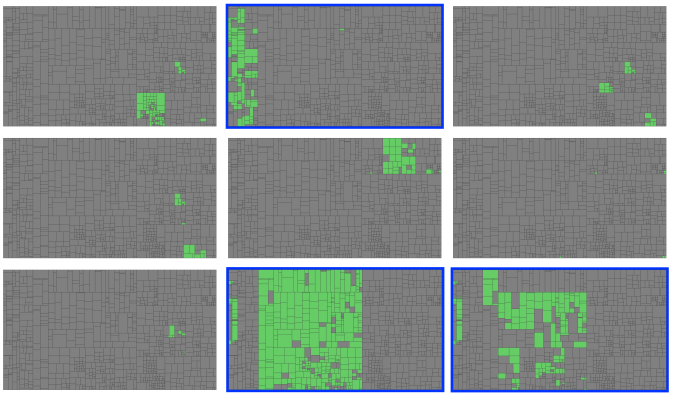

Figure 2. Test coverage of nine automated tests. If a test executes a method, it is shown in green, otherwise in gray.

Figures 1 and 2 show an example from our own development. Each of the illustrations shows a treemap. Each small rectangle in the treemap represents a method in the source code of the *system under test*. Figure 1 shows the changes that were made as part of the development of a feature. Unchanged code is gray, modified code is orange, and new code is red. Figure 2 shows the code coverage of 9 automated test cases (unit tests and integration tests). If a test case executes a method, it is

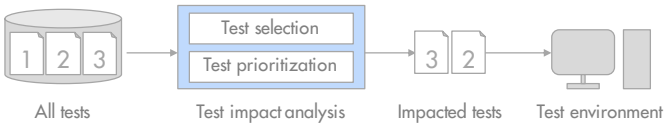

Figure 3. Phases of test impact analysis

displayed in green. Gray methods were not executed by the test case. In the example, only the three test cases whose treemaps are outlined in blue execute any modified code at all. The remaining six test cases cannot find any errors caused by the implementation of the new feature. Their duration can therefore be saved during CI.

In this article we introduce TIA, outline previous work, and then describe our approach and the empirical studies that we conducted to quantify the applicability and utility of TIA in practice. For the analyzed study objects, we were able to show that TIA can considerably reduce test times during CI: For our study objects, the test time can be reduced by 98%, but still 90% of all erroneous builds are recognized as such. The central contribution of this article is therefore showing that the proposed TIA process, which is based on research in the field of *test selection* and *test prioritization*, is ready for continuous use in practice under the conditions described in this article.

II. Test Impact Analysis

The goal of test impact analysis is to propose a subset of the entire test suite for the current code of the system under test, which can be used to find as many errors as quickly as possible. As shown in Figure 3, the analysis basically consists of two steps:

- *Test selection:* On the basis of selected metrics, a subset of the entire test suite to be executed is selected. We call this selected subset *»impacted tests«*.
- *Test prioritization:* The impacted tests are sorted so that errors are found as quickly as possible. This can be based on data from previous test runs.

TIA can be performed using only test selection or test prioritization, as well as a combination of both. The particular algorithmic implementation of these two steps is the primary distinguishing feature among various approaches. The approaches to test selection and test prioritization described in the literature often vary widely in their complexity and effectiveness. For example, there are randomized approaches or those that select or prioritize based on test case-specific coverage. We have summarized the most important approaches in Section IV. The approach we choose uses a combination of selection and prioritization. We describe this approach in more detail in Section V. The resulting selected and sorted tests are finally executed in the test environment in the defined order.

Because TIA uses only part of the entire test suite, in practice it is necessary to run all tests at regular intervals. We describe the limitations of TIA in the following section.

III. Limitations fo Test Impact Analysis

Like any analysis method, TIA has limitations. Knowing about these is critical to making good use of TIA. One such limitation is changes that are made at the configuration level without code being altered, which keeps them hidden from the analysis.

Some approaches perform selection or prioritization based on coverage that was recorded in a past run. A limitation of these approaches is the assumption of a certain test coverage stability, since changes to the code or tests can in principle lead to situations in which the next run no longer runs through the previously recorded methods.

If the selection or prioritization algorithm depends on the test coverage, then it must also be recorded for each test case. This typically requires performing the tests in isolation from each other.

Another limit of TIA is the use of indirect calls. For example, if reflection is used in a test case to test classes with a certain annotation, then such a test case is typically not selected if that annotation is added to another class.

Although there are techniques in literature to determine a subset of the tests demonstrably finding all bugs (that can be found by the entire test suite), these are often not usable in practice or do not bring any real benefit. Practical TIA can therefore typically give no guarantees for this. Therefore, all tests must be executed at regular intervals. The use of TIA, however, means that errors are usually found much earlier compared to only having dedictaed scheduled test phases, which in practice is usually associated with a significant reduction in costs.

IV. Existing Work on Test Impact Analysis

There are two established research fields related to TIA in literature: *test selection* and *test prioritization*. However, the work in the respective areas does not solve the problem presented above in practice. Our approach therefore combines findings from both areas and makes them usable in practice.

In the following, we present various existing approaches and position our approach. Since the contribution of this article is primarily investigating TIA's suitability for practical use, due to the limited space, we refrain from differentiating the various approaches in detail.

A. Test Selection[1]

Engström et al. [1] present a meta-study classifyig existing work on test selection in different categories. Following this classification, we will present representative existing work on the topic of automated TIA.

1) DejaVu-based Selection: DejaVu-based techniques [2] use fine-grained test coverage data to derive which tests run through modified code. These tests are then selected. The underlying assumption here is that errors are likely to be caused by these changes. Most of these Deja-Vu techniques use control flow graphs for each individual method to select

[1]Also called *»Selective Regression Testing«* in literature.

the minimum set of tests that run through the changed code. The disadvantage of this technique is that it requires high computational effort to process the fine-grained coverage data. The approach we evaluated in practice (see Section V) uses a DejaVu-based selection phase, which, however, records coverage only on method granularity. We also limit the computational burden by deducting simple refactorings [3].

2) Firewall-based Selection: Firewall-based selection approaches were first described by Leung and White [4] and are limited to integration tests. They build a dependency graph between the modules of a system and then mark all modules that have been changed or have a direct dependency on changed modules as *»inside the firewall«*. Subsequently, all integration tests that execute code within the firewall are selected. This approach is *safe* [5], so it guarantees that the same errors can be found with the selected tests as with all tests. However, this safety has its price, because in general a lot of test cases are selected, even if they do not execute any changed code. Existing studies have shown that a very large proportion of all tests are often selected, even if the changes themselves are very small [6].

3) Dependency-Based Selection: Dependency-based selection approaches [7] use a static function call graph to describe dependencies between parts of the software system. They then select those tests that directly or transiently execute changed functions. The advantage of this technique is that it is coverage-agnostic. Its drawback lies in the creation of the function call graph, which is language-specific and even depends on frameworks used (e.g., Dependency Injection).

B. Test Prioritization

Test prioritization aims at sorting a set of tests so that errors are found as quickly as possible. For this purpose, there are various approaches in the literature that calculate such an order using different factors. In any case, errors should be found faster than if the tests are run in random order. Rothermel et al. [8] describe several approaches, which we group into four categories:

- *Total Coverage:* Test cases are prioritized based on their combined coverage, so that those test cases are executed first, which produce the highest coverage. The granularity of the recorded coverage may vary (e.g., branch coverage or method coverage).
- *Additional Coverage:* Test cases are ordered by their additional coverage. Starting from a coverage level, the test case that generates the most additional coverage is selected. Again, the granularity of the coverage may vary.
- *Potential for error detection:* Test cases are arranged so that those test cases are executed first that have the most potential to actually detect errors. In order to estimate this potential, mutation tests are performed beforehand.
- *Total error:* those test cases that have failed most often in the past are executed first.

In the experiments conducted by Rothermel et al. [8], a combination of additional statement coverage and error detection potential turned out to be the best prioritization technique.

The approach we evaluated in practice (see next section) uses a combination of additional line coverage and expected test execution time as the prioritization technique.

V. Our Test Impact Analysis Approach

Figure 4 (see next page) gives a structural overview of TIA as we use it. As shown, the analysis can roughly be divided into two phases.

The goal of phase 1 is to obtain *coverage* for individual test cases and to measure their *execution time*. To do this, test cases are executed individually and fully automatically in the test environment using a profiler. The profiler measures which parts of the source code are executed per test case and how much time it takes. This *»test impact data«* is stored in a database.

The goal of phase 2 is to actually generate suggestions on which test cases should be executed in which order. This phase is performed whenever changes have been made to the source code of the *system under test* and tests are about to be executed. Such a proposal is generated based on these changes and the test case-specific test impact data collected in the previous phase:

- *Test Selection:* selects all tests executing the given *changes* (as well as new or modified tests)
- *Test Prioritization:* The selected test cases are sorted so that all changes are covered as quickly as possible. This problem can be reduced to the set coverage problem, which is $\mathcal{NP}$-complete. Therefore, we use a greedy heuristic that sorts tests by their additional coverage and is fast enough in practice even with large test sets.

VI. Empirical Study: Applicability of Test Impact Analysis

The empirical study examines the applicability and usefulness of TIA in practice. First, we describe the study objects and the study design. Then we describe the individual research questions and their results. Since performing the calculation of selection and prioritization requires only fractions of a second for all study objects, the study focuses on the utility of the results rather than on the TIA calculation time.

A. Study Objects

The study was conducted on 12 systems, all implemented in Java. To make the study reproducible, we selected 11 open source systems. In addition, we included the commercial software analysis tool Teamscale[2] because we, as Teamscale's developers, know it very well and can best validate the study results here. The study objects differed in size, history length and number of test cases. The smallest systems comprise 7k LOC (application and test code), the largest over 300k, history lengths go from 44 to 82,164 commits. Due to the large differences between the systems, it is unlikely that a system-specific single effect dominates the study results. A detailed overview of the study objects can be found in Table I.

[2]https://www.teamscale.io

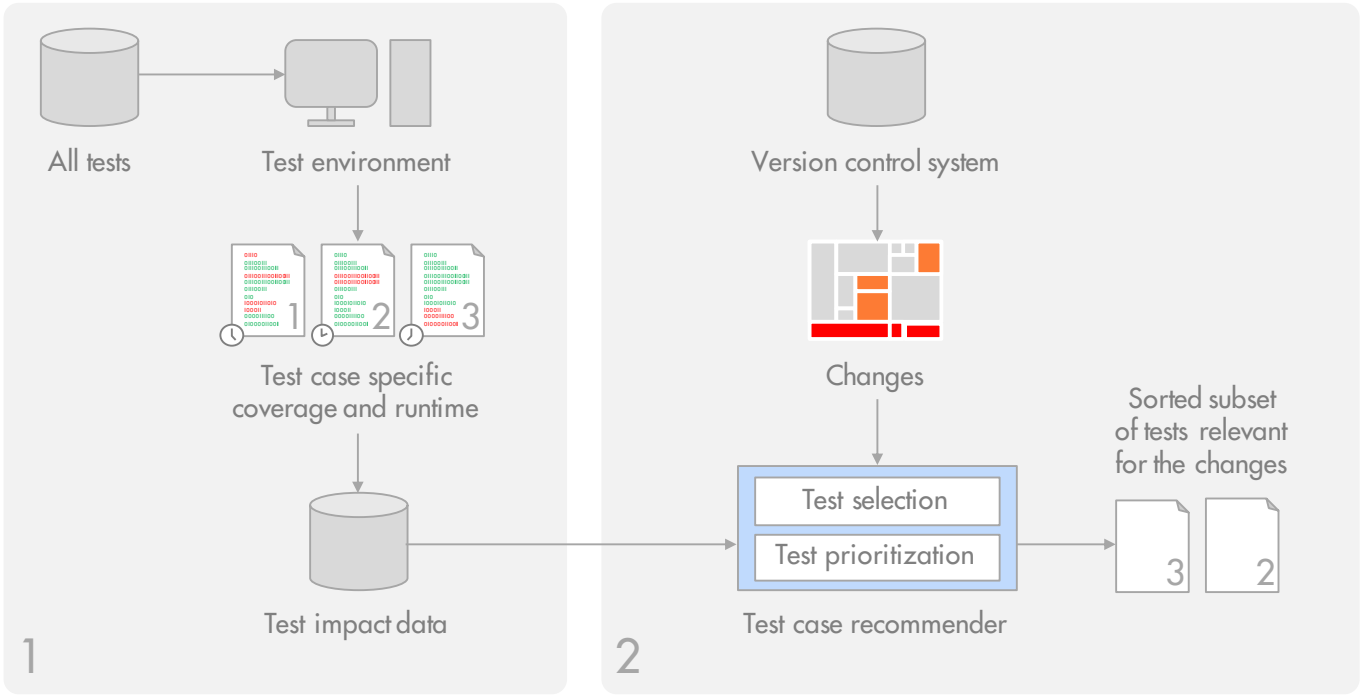

Figure 4. Test Impact Analysis as we use it

Table I
OVERVIEW OF STUDY OBJECTS

Study Objects	Σ	kLOC Application	Test	Commits
Apache Commons Collections	62	31	31	3,235
Apache Commons Lang	75	27	48	5,486
Apache Commons Math	178	87	91	7,156
Histone Template Engine 2	14	12	2	1,133
JabRef	122	94	27	10,645
Joda-Time	83	28	55	2,105
Lightweight-Stream-API	23	8	15	529
LittleProxy	9	4	5	1,037
OkHttp	52	26	26	3,548
RxJava	242	84	158	6,000
Symia Commons Math Parser	7	6	2	44
Teamscale	336	270	67	82,164

B. Study Design

TIA always compares two versions of a system. We call the first version the *baseline*, the second the *working copy*. For the study, we used mutation testing to automatically incorporate errors into the study objects. As a result, we know all the errors we have and can determine what percentage of them can be found by TIA how quickly. We did the following in our study to create the study objects:

1) As baseline, we selected an official release of each study object. As the initial version of the working copy, we then took a version that follows the baseline in the version history and made sure that there were a substantial number of changes between versions. Then we incorporated an error via mutation testing into this version to create the working copy. The particular location of the error in the code was chosen randomly.
2) We generated between 100 and 1,000 pairs of baseline and working copy per study object. (We chose the number so that for every study object, all these pairs could be evaluated in less than one day of computation time).

For each study object, we performed all automated tests on each working copy created in this way. If at least one test failed, the built-in error was considered detected.

Finally, we performed TIA on each pair of baseline and working copy to determine the impacted tests. Based on this data, we will answer the research questions below.

C. How Reliable is Test Impact Analysis?

TIA only executes impacted tests, i.e., a subset of all tests in a system. In principle, therefore, errors could go undetected because they are not found by the impacted tests, but would be found by the remaining tests. The goal of this research question is to quantify the proportion of errors that are not found by the impacted tests in practice.

To do this, we calculate the percentage of errors that the impacted tests reveal in relation to the percentage of errors that the complete test suite uncovered. Of the 6,661 synthezied errors, 4,102 were detected by running all tests on the system. (Since the remaining 2,559 errors were not recognized by any test, they cannot be detected by TIA either). Of these 4,102 detected mutated-in errors, 4,073, or 99.29%, are also detected by the impacted tests alone. In 7 of the 12 study subjects, 100% of the errors were detected, in the remaining 5

Table II
TIME-SAVINGS DUE TO EXECUTION OF IMPACTED TESTS ONLY

	Execution Time (ms)		
Study Object	All Tests	Impacted Tests	Saving
Apache Commons Collections	25,277	610	97.59%
Apache Commons Lang	24,987	3,111	87.55%
Apache Commons Math	160,391	114,042	28.90%
Histone Template Engine 2	34,603	32,204	6.93%
JabRef	119,849	40,721	66.02%
Joda-Time	20,782	1,297	93.76%
Lightweight-Stream-API	1,523	480	68.48%
LittleProxy	155,334	150,406	3.17%
OkHttp	96,671	76,457	20.91%
RxJava	464,018	170,575	63.24%
Symia Commons Math Parser	528	528	0.00%
Teamscale	1,249,088	196,684	84.25%

study objects between 90.6% and 98.7% were detected[3]. TIA recognizes over 90% of the errors in all projects. In more than half of the projects, all errors were identified.

D. How Much Time Does Restricting Test Execution to Impacted Tests Save?

The purpose of this research question is to quantify by how much the duration of tests is be accelerated in practice, if only impacted tests are executed.

For this we calculate the saving of the execution time of the impacted tests compared to the execution time of all tests. This saving occurs during each test run, even if no error is found.

The savings range from 0% to 97.6%, which is very different between the study objects. On average, selection saves 52% of the test execution time, the median being 64% across all study objects. Detailed results are shown in Table II.

The big differences in saving are due to the different amount of code that is executed by each test in the different study objects. In *Apache Commons Collections*, most tests go through very little code, which varies greatly between tests. This greatly benefits the selection phase of TIA. In the case of *Symia Commons Math Parser*, on the other hand, many tests start the entire parser. As a result, much of the code is executed by all tests, which reduces the usefulness of the selection phase.

E. When does Test Impact Analysis find the first failing test?

When the first test case fails, we already know that the system under test is not bug-free (and, e.g., the build result should not be deployed). We know this without having to wait for the results of the remaining test cases. This research question quantifies how quickly TIA finds the first failing test.

To do this, we calculate how fast the built-in error is found when we run the impacted tests in the proposed order.

[3]In-depth analysis of the 29 unrecognized errors revealed that the majority were related to non-deterministic test cases (and thus unstable test coverage). A customized measure of test coverage that executes test cases multiple times and only counts coverage for those methods that are executed in each pass would likely reduce those issues.

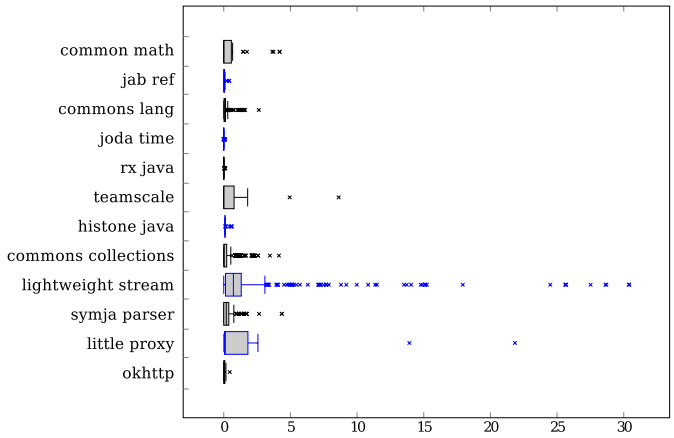

Figure 5. Distribution of Execution Times Until The First Error is Found

For all 4,073 pairs of baseline and working copy, for which the impacted tests did find the error, we have determined after which execution time the first test fails. Then, for each pair we determined the relative duration until the first error was detected. For example, if the first error was found after 3 seconds, but the entire suite takes 100 seconds, the result is 3%.

Statistical distributions are shown in figure 5. For each system, the distribution of values is shown as a box plot[4]. Since most of the values are in the range of less than 5%, only the range of 0% to 30% of the execution times is shown, in order not to squeeze the box plots even more. Each box plot represents the values of the second and third quartile in the box. The median is displayed as a vertical line inside the box. For some systems, the values are so crowded (e.g., JabRef) that the box collapses to a narrow vertical bar. The box plots clearly show that the overwhelming majority of values are between 0% and 2%.

F. How likely are errors detected using limited time?

To guarantee that test execution is completed on time during CI, it is not sufficient to stop test execution as soon as the first error is detected: If there is no error, then all impacted tests are executed (which, as seen above, will in some cases result in the execution of the entire test suite). Instead, test execution must be terminated after a predefined period of time. Therefore, this research question quantifies how reliably TIA detects a system being faulty within a limited time span.

For this we evaluated what fraction of the errors is found when only a subset of the impacted tests is executed that can fit into 1%, 2%, 5% or 10% of the test suite's total execution time. The results are shown in Table III. Using only 1% of the time, already between 60.8% and 98.1% of the errors are found, with a median of 89.6% and an average of 85.7%. Using 2% of the time, between 77.1% and 98.6% of all errors are found, with a median of 92.7% and an average of 91.4%.

[4]For a more detailed description of how to interpret box plots, see e.g. https://en.wikipedia.org/wiki/Box_plot.

Table III
PERCENTAGE OF FAULTY SYSTEM VERSIONS DETECTED BY EXCEUTION TIME LIMIT

Studienobjekt	1%	2%	5%	10%
Apache Commons Collections	84.25	96.58	99.83	100.00
Apache Commons Lang	94.23	96.43	98.49	99.31
Apache Commons Math	80.20	85.92	91.55	92.96
Histone Template Engine 2	88.46	88.46	88.46	88.46
JabRef	95.71	95.71	95.71	95.71
Joda-Time	98.10	98.61	99.11	99.87
Lightweight-Stream-API	60.76	90.73	94.74	97.25
LittleProxy	74.29	77.14	80.00	80.00
OkHttp	90.70	90.70	90.70	96.51
RxJava	91.89	94.59	94.59	94.59
Symia Commons Math Parser	78.46	86.15	89.23	89.23
Teamscale	91.86	95.93	96.57	96.57

VII. OUTLOOK

We plan to develop TIA in the following directions:

Teamscale Development Process: We recently integrated TIA into our own development process. Developers can decide for themselves on their feature branches whether all tests or only the impacted tests should be executed. We plan to introduce TIA for all of our branches as the default test strategy, as we gain further experience.

Programming Languages: Currently, we only use TIA for Java. We plan to extend both the tool support and the studies to other programming languages.

Benchmark: We are currently using a mutation-based benchmark to answer the research questions. We are in the process of building a benchmark based on real open source bug fixes to quantify the research questions based on the errors historically found in these projects during CI. In ongoing work, 437 faulty versions of 31 projects were analyzed. The results are very similar to the mutation-based benchmark results in this paper, but are not yet final.

Test Types: We are currently working on a research project evaluating TIA for hardware-in-the-loop testing of embedded ECU software. The initial results are very promising, but this study is also still ongoing.

TIA Strategies: We plan to evaluate further approaches to TIA based on both the mutation-based benchmark and the error data from real projects. We want to do this not only to check if there are better approaches than what we are pursuing, but also which approaches are feasible, if e.g. no test case-specific coverage can be obtained.

VIII. SUMMARY

Our approach to test impact analysis selects and prioritizes test cases based on code changes since the last test run. This allows one (small) part of a large test suite to be run for each CI pipeline. In our empirical study, we quantified the reliability and usefulness of our approach on 12 Java systems in practice.

In practice, TIA has a very high reliability of 99.26% over all study objects (in the worst individual case of 90.6%). While the savings on execution time of all impacted tests vary widely between systems, time savings until the first error is detected are significant across all systems.

TIA's greatest potential, therefore, is to shorten the time between the start of a test run and the first error being detected. With only 1% test execution time, TIA reveals a faulty system version on average (and median) in over 80% of cases, with 2% test run time in more than 90% of cases.

In other words, the test execution time can be shortened by 98% and yet only in 10% of the cases faulty builds are not recognized as such. Especially for systems whose tests run so long that they are not executed at all during CI, this represents a substantial improvement, since now 90% of all errors are detected during CI, and not revealed only in late test phases that may not take place until months later.

Since TIA does not detect all errors, all tests must still be executed at regular intervals. However, since most errors are already detected during CI, comparatively fewer errors should occur during the execution of all tests. This should also reduce the extra effort caused by the time delay and possibly overlapping errors.

ACKNOWLEDGEMENTS

This paper builds on previous work and ideas by Florian Dreier, Jakob Rott and Rainer Niedermayr, to whom we would like to express our heartfelt thanks. This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant »SOFIE, 01IS18012A«. The responsibility for this article lies with the authors.

REFERENCES

[1] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Inf. Softw. Technol.*, vol. 52, pp. 14–30, Jan. 2010.

[2] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, pp. 173–210, Apr. 1997.

[3] F. Dreier, E. Jürgens, and A. Göb, "Detection of refactorings," bachelor's thesis, Technische Universität München, 2015.

[4] H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings of the 1990 Conference on Software Maintenance*, 1990.

[5] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, pp. 529–551, Aug 1996.

[6] E. Jürgens, B. Hummel, F. Deissenboeck, M. Feilkas, C. Schlögel, and A. Wübbeke, "Regression test selection of manual system tests in practice," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, pp. 309–312, 2011.

[7] Y. Wu, M.-H. Chen, and H. M. Kao, "Regression testing on object-oriented programs," in *Proceedings of the 10th International Symposium on Software Reliability Engineering*, ISSRE '99, (Washington, DC, USA), pp. 270–, IEEE Computer Society, 1999.

[8] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, pp. 929–948, Oct. 2001.