Code Coverage-Based Regression Test Selection and Prioritization in WebKit

Árpád Beszédes, Tamás Gergely, Lajos Schrettner, Judit Jász, László Langó, Tibor Gyimóthy
University of Szeged, Department of Software Engineering & HAS Research Group on AI, Szeged, Hungary
{beszedes,gertom,schrettner,jasy,lango,gyimothy}@inf.u-szeged.hu

***Abstract*—Automated regression testing is often crucial in order to maintain the quality of a continuously evolving software system. However, in many cases regression test suites tend to grow too large to be suitable for full re-execution at each change of the software. In this case selective retesting can be applied to reduce the testing cost while maintaining similar defect detection capability. One of the basic test selection methods is the one based on code coverage information, where only those tests are included that cover some parts of the changes. We experimentally applied this method to the open source web browser engine project WebKit to find out the technical difficulties and the expected benefits if this method is to be introduced into the actual build process. Although the principle is simple, we had to solve a number of technical issues, so we report how this method was adapted to be used in the official build environment. Second, we present results about the selection capabilities for a selected set of revisions of WebKit, which are promising. We also applied different test case prioritization strategies to further reduce the number of tests to execute. We explain these strategies and compare their usefulness in terms of defect detection and test suite reduction.**

***Keywords*-Regression testing, test case selection, code coverage, test prioritization, test quality, WebKit.**

I. Introduction

Regression testing is a widely used approach to maintain the quality of continuously evolving software systems [1]. Stakeholders often rely heavily on regression tests to ensure quality if their system changes frequently, which is often the case in traditional software engineering, but even more emphasized in different agile methods. Although the possible benefits are clear, a practical implementation and sustained reliability of regression testing are not always easy to achieve. In general quality management practice there are approaches for maintaining software quality at the process level and product level with numerous supporting methodologies and tools. However, as opposed to processes and products, the quality, hence, reliability of *regression tests* is rarely checked and controlled in a systematic way. Consequently, building the quality of the system solely upon a regression test suite may be a major risk.

The main indicator of regression test suite quality is its defect detection capability, but this is usually replaced by other characteristics, such as completeness (in terms of functional/code coverage), redundancy (whether the same defects are detected by several tests), or maintenance status (whether the test suite evolves together with the system). Unfortunately, even a properly developed and optimized regression test suite can grow extremely large if the system under test is also large and complex. The consequence is that full regression testing can become prohibitively expensive.

In this work, we report on our experiences in the application of one of the fundamental techniques in regression testing, namely *regression test selection and prioritization based on code coverage* [2]. We worked with the open source system WebKit, a popular web browser engine integrated into several leading browsers by Apple, KDE, Google, Nokia, and others [3]. It is a large, complex, and lively evolving system developed and maintained by a huge international community, making it a perfect subject.

Despite the available resources offered by the community, the WebKit regression test suite cannot always be run after each modification made to the source code, because the system and the test suite are too large and changes occur too frequently. As a first step, we wanted to investigate whether it is possible to implement selective regression test execution based on code changes as an alternative to full testing. If the selection is significant and reliable (*i. e.*, the same defects are captured), then developers will be able to check much more revisions against regressions than previously and capture defects earlier in the life cycle.

Change-based selective testing is one of the fundamental hypotheses in regression testing [4], yet few reports can be found that deal with its defect detection capability in real environments. In this experiment, we applied procedure level code coverage of the changes (on C++ functions and methods), and used data taken from the WebKit version control system, test execution and defect databases, covering a fixed interval of the system's evolution. We report on the current state of the research, and make the following contributions:

- We implemented an experimental environment to assess the regression test selection, and performed experiments for several hundred revisions of the WebKit system applying static code analysis, code instrumentation, and additional tools.
- We investigated various properties of the tests in the analyzed period and looked at how successful the selection is, *i. e.*, 1) whether the reduced test set identifies the same failures and 2) how much reduction can be achieved on average.

978-1-4673-2312-3/12/$31.00

- Since we found the initial results very promising, we extended the WebKit build system with an optimized test selection component, which is currently in live operation. It uses a procedure level coverage database with manual update, change based selection and integrated execution and reporting, and is implemented both as part of the automatic build process and as an early warning system in the issue management system.
- We present different test case prioritization strategies that can reduce the size of large selection sets, and report results about their performance compared to the unprioritized test set. We found that the strategy which selects the least covering test cases beside the changes was the best in this respect, by which we can limit the size of the selection to below 10% and still achieve about half of the inclusiveness of failures observed with respect to the non-prioritizing case.

In the next section we overview related work, then in Section III we present the basic research goals and the measurement details. Section IV presents the results of the initial measurements, while in Section V the details and evaluation of the implementation in the live system are presented. Section VI discusses our methods for test prioritization with related findings. We present threats to validity in Section VII and conclude with possible subjects of future research.

II. Related work on regression testing

An overview of regression test selection techniques has been presented by Rothermel and Harrold, who introduced a framework to evaluate the different techniques [4]. They defined the evaluation criterion called *inclusiveness*, which we rely on as well as one of the primary evaluation aspects of our methods. It is the ratio of the failing test cases included in the selection relative to the total number of failing test cases when executing the complete test suite. The other main evaluation criterion we will use is *selection size*, which is expressed as the percentage of the number of selected test cases over the total number without selection. Another survey on regression test selection and prioritization has been presented by Yoo and Harman [2].

Some algorithms for test case selection are based on different program representations to determine the effects of a program change [5], [6]. Our method also analyzes source code in order to determine the modified procedures, but we do not build any detailed program representations.

Inclusiveness is the focus of many researchers. Wong *et al.* suggested that regression test reduction techniques can lower the number of executed test cases without significantly reducing the fault-detection capabilities of test suites [7]. However, Rothermel *et al.* examined the costs and benefits of test-suite reduction techniques and their results show that the fault-detection capabilities of test suites can be severely compromised by test-suite reduction [8]. All subject programs of the above case studies were small programs with less than 1000 lines of code, and prepared by fault seeding. In our case a specific software, the Qt port of WebKit and its regression test suite were studied. The system source contains 1.9 million lines of investigated C++ code and the errors were not seeded but they were real failures of the software.

Another important attribute of test case selection methods is the coverage that is utilized in them. Wong *et al.* [7] used all-uses coverage, while Rothermel *et al.* [8] used all-edges coverage. The studies showed that the different granularities used in test case selection produced different reductions in the test suite size. In another study, Rotermel and Harrold [9] found that coverage-based test suites may provide test selection results superior to those provided by test suites that are not coverage-based. We used procedure level coverage in our study, but did not try to minimize the test suite in terms of eliminating "redundant" test cases.

Other studies elaborate on different issues of regression test case selection. Kim *et al.* studied the relationship between regression test application frequency and the behaviour of different techniques [10]. Their experiments exposed that as the number of changes between the program versions increased, the number of test cases selected also grew rapidly, and the effectiveness of test selection increased. We apply test case selection to all versions where important changes are present, and examine how often a full retest is required to achieve dependable results.

Test case prioritization techniques have also been studied thoroughly. Wong *et al.* proposed a hybrid technique combining modification, minimization and prioritization-based selection to identify a representative subset of all test cases that may result in different output behavior on the new software version [11]. Case studies on regression test prioritization have also been done. For example, Rothermel *et al.* [12] conclude that prioritization techniques can be effectively used, but the usefulness of a particular technique on a specific test suite depends on many attributes of the technique and the test suite. Our prioritization strategies use greedy algorithms, just as most other authors' as well, however recently there has been some research presented to use search algorithms instead [13].

III. Background and overview of the research

WebKit is a large, complex, and lively evolving software system which is maintained by hundreds of developers around the world, including several members of our department. We have recently started a long term project with the aim to enhance the internal quality of the system including the reliability and efficiency of the regression testing system. As a first step, we started with the assessment of the code coverage and other attributes of the regression test suite.

A. The WebKit system

WebKit consists of about 2.2 million lines of code, mostly C++, JavaScript and Python among others. In this research we concentrated on C++ components only, which attributes to about 86% (1.9 million lines) of the code. The system also has a relatively big collection of regression tests consisting of nearly 24 thousand test cases.

WebKit has a large, geographically distributed development community, and its development environment is a typical one for such a distributed team, which includes serious configuration management and strict integration rules. For instance, before any patch can land in any of the components in the version control repository, a set of regression tests must pass. As of April 11, 2012 there are 113914 revisions, and about 90 revisions are created on average each day. Due to the big size of the regression test suite and very frequent revisions, this requirement of always executing all the regression tests cannot be fulfilled in many cases. The automated build system [14] continuously performs regression tests, but the computation capacity of the server is not enough for testing each and every revision. This can imply that if a failure is detected it will be often hard to trace the actual revision responsible for the defect, hence making the correction more difficult.

Furthermore, the reliability and completeness of the regression test suite has never been systematically checked, so currently there is no real indication about the defect detection capability of the regression test suite. There may be defects that are either captured by several test cases, or may remain undetected. Motivated by these issues, we wanted to investigate the possibility to speed up the execution of the tests by test selection and prioritization, while keeping the same level of reliability and defect detection capability.

B. Research goals

Since the beginning of our project in autumn 2011, there were three main phases of our research. In the first round (referred to as *initial experiments* in the following) we analyzed a decent number of actual revisions of WebKit to collect data and get a deeper insight into the source code, the changes made to it, the regression test suite, the test executions, and code coverage. Section IV presents the details of initial experiments. Then, in the second phase, based on our findings we implemented an optimized and a slightly different version of test selection in the official WebKit build system, and initiated its continuous operation in parallel to the build processes. We monitored the performance of the system for a certain period of time and performed additional measurements and analyses on the collected data in order to verify the initial findings (we will call this the *live system*). The results of this analysis are presented in Section V. Our most recent research activity is to further enhance the efficiency of the implementation by applying test prioritization in addition to simple selection. In Section VI, we propose different heuristic approaches to test prioritization with which we can reduce the size of the coverage-based selection, and discuss their usefulness on the set of data collected from the live system.

We base our approach for test optimization on the long standing assumption that it is possible to select the more relevant tests from a complete test suite based on the set of changes applied to the system. We employ the simplest version of this principle, namely *selection based on code change coverage* of the tests, in particular computed on the level of individual procedures (functions and methods in C++). The following is an overview of the approach:

1) We determine the initial coverage information (referred to as *coverage database*) for each procedure in the system and test case in the regression test suite. (The coverage database is periodically recomputed to reflect up-to-date coverage information.)
2) We identify the set of changed procedures for a specific revision.
3) We look up in the coverage database the set of test cases which execute any of the changed procedures.
4) Optionally, if the selection is too large we apply test prioritization to reduce the overall testing time.
5) We execute the selected tests only in the regression testing of the revision in question.

Selective regression testing based on these principles has been studied extensively, constantly relying on the assumption that change based selection provides reliable defect detection. Usually the reduction rate is regarded as the most important factor, however, there is little research on actually how efficient this strategy is in terms of *inclusiveness* (how many failing tests are included in the selection) and *precision* (how many unnecessary tests are included).

We articulate the following **Research Questions**:

RQ1 Is it true that only a small portion of the test cases is responsible for revealing defects after a typical change is made to the system?

RQ2 Can a coverage-based test selection method select all or most of the failing test cases and, with the overheads of analyzing the changes, measuring the coverage, executing the selection method, etc., can it still reduce the time of the overall test execution process and hence provide opportunity to more frequent regression test execution?

RQ3 When the coverage-based selection provides an overly large set of test cases, test case prioritization can be used to reduce testing time. From the proposed heuristic prioritization strategies, which strategy can select the most defect revealing tests, and hence further reduce the test set without affecting inclusiveness significantly? Is change-based selection better than other test prioritization methods that do not depend on the changes?

C. Measurement scope and experimental tool setup

In this work, we limit our analyses to a specific configuration of WebKit, the Qt port, and to its C++ components.

The build process and the test cycles are relatively resource consuming in WebKit even for a single revision, so we had to develop a sophisticated toolchain which we used for our research. For the *initial experiments*, the measurement process included automated and manual phases as well, and basically consisted of two types of tasks: build and test execution. To collect the runtime information about the execution we used code instrumentation, which resulted in a special type of build and test execution. To be able to assess the overhead of the analysis, we experimented with both normal and instrumented types of builds and test executions.

In the implemented *live version* we slightly modified some of the tools to be suitable for continuous and real time operation, and to better conform to the required coding standards of the community. We also had to use a periodic coverage database update strategy as elaborated below.

In the following, we will overview the most important tools used in our toolchain.

- *Procedure level coverage measurement.* We used a tool that utilizes the "instrument-functions" feature of the GCC compiler that allowed us to instrument the beginning of each procedure with code that outputs a procedure identifier.
- *Identification of changes made to the source code.* To collect the set of changed procedures, initially we used the Columbus tool [15], which contains a C/C++ front end for producing a source code representation with sufficient details for us to compute structural differences between two revisions. As this kind of source code analysis was required for some other research as well [16], it was a straightforward choice. On the other hand, a build with this static analysis was 5 times longer than the native one, so we decided to switch to a less precise but more efficient approach based on fast textual difference computation. Specifically, an extended version of the `PrepareChangeLog` WebKit utility is used currently.
- *Coverage database and database update.* Initially, we computed coverage information for all revisions and for the whole program, and stored all the results in the file system. However, this approach proved to be unusable for the live system due to the huge amount of data to be processed very frequently. Furthermore, we had to use a periodically updated coverage database of selected revisions instead of computing an up-to-date full coverage information for each revision. This implies that for some intermediate revisions a slightly outdated database is used, so we performed some experiments to determine the difference between the revisions and what an acceptable database update interval would be.

IV. Initial experiments

We experimented with revisions in the range 96803–97370 which represents the development period from October 6, 2011 to October 13, 2011. Since we limited our scope of analysis to the C++ subsystem of WebKit, we had to drop 441 revisions where no relevant changes were found. There were 61347 C++ procedures in the system that we identified for use in our experiments, and 23574 test cases in the regression test suite during this period.

In this phase we selected those test cases for execution that fit into any of the following sets: 1) each new or changed test case, 2) each test case whose coverage contains a changed procedure, and 3) each test case that failed during the previous test execution. The coverage information was computed individually for each investigated revision.

A. Regression test suite statistics

First we determined what portion of the procedures is covered by test cases; the figure turned out to be just below 74%, which mostly stays at this level. The analysis of failing test cases showed that on average only 0.07% of the test cases failed per revision, the maximum is 0.31%.

To find out whether the failing test cases come from a well defined set or are they usually different, we investigated the failing tests for each revision, and calculated how much additional failing tests can be found in subsequent revisions. In Figure 1, we can see that the number of failing tests had several peaks but usually it stayed around the average value. There were a few revisions where an increased number of failing test cases did not imply new additional tests (around rev. 96960), but usually it is the opposite: failing tests are different from revision to revision.

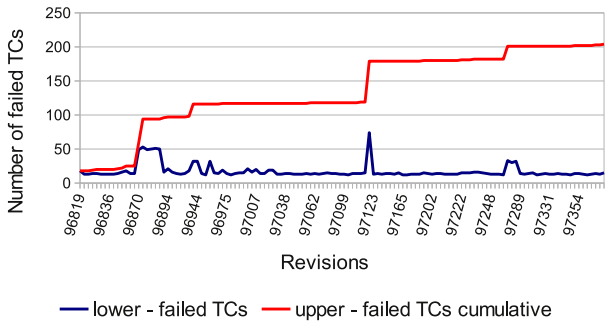

Figure 1. Failed test case variance. The lower plot shows the number of failed test cases in the individual revisions, while the upper one shows the number of test cases that failed at least once until the actual revision.

We also analyzed the number and composition of the changed procedures in the investigated period. In Figure 2, we can observe how many procedures were changed over the range of the revisions. The lower plot shows this data individually (the number ranges between 0 and 48 while the

average is 5.13 procedures). The two other plots correspond to the cumulative number of changed procedures and the sum of the changes (disregarding repetitions). Because the two latter graphs run in parallel and the difference is small, we can conclude that the changes are usually disjunct and there is little repetition among the different revisions.

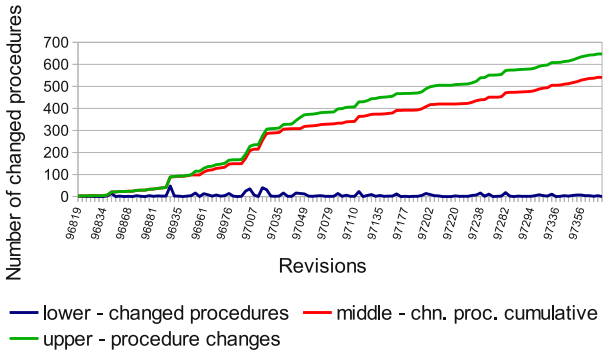

Figure 2. Number of changed procedures. The lower plot shows the number of changed procedures in the individual revisions, the middle one shows the number of all procedures changed at least once until the actual revision, while the upper plot shows the number of procedure changes until the actual revision.

Finally, we wanted to find out whether the changed procedures affect only a few or a high number of test cases. It turned out that most changes affect a smaller number of test cases, 1887 on average. The distribution of this data is also very skewed at the lower end, so there are relatively few changes that cover many test cases.

This data and Figures 1–2 show that there is a high diversity (little repetition) in the changes and the failures over a range of subsequent revisions. Furthermore, having in mind that the overall coverage is relatively high, we concluded that it was overly redundant to execute the whole test suite upon every change, and change based test selection could be used instead.

B. Test case selection efficiency

To find out whether we can achieve similar defect detection ratio using a significantly reduced set of test cases executed after a change, we measured the inclusiveness and the selection size of test selection as defined in Section II. Note that we do not deal with the precision rate [4] in this article, since it is not the property of the test selection method but of the development and is rather arbitrary, hence it is of lesser value in the present experiment.

In this phase of the research we could use a rather accurate way of determining the coverage for each revision (as opposed to the implementation in the live system in Section V), and we treated reoccurring failures as always selected. This way we could achieve quite a high rate of inclusiveness overall; it was 100% in 58.7% of the cases, and over 90% in 88.9% of the cases (the average inclusiveness was 95.08%). It is interesting to see why the inclusiveness was not perfect in some cases, although theoretically it would have been expected. This can be attributed to a combination of factors that limit the approach (exclusion of non-C++ code, flaky tests [3], imperfect determination of the changes and their correlation to the C++ procedures, etc). We addressed some of these issues in the live system, while there are still some problems to solve in the future to make the approach more reliable.

At the same time the selection size was quite significant: in more than 87% of the cases it was below 1%, and for half of the cases it was below 0.06% (the average selection size was 20.57%).

As mentioned earlier, we use a periodic coverage database update strategy in the live system. But before we implemented this feature, we analyzed a set of existing revisions for the complete coverage, and compared these coverage data to each other to find out how different they were from one revision to another. Although there were some variances in the consecutive revisions, the overall difference with respect to the base revision was steadily growing. On average, one week of development makes the way procedures are covered by test cases change by approximately 15%. (While the total coverage percentage did not change notably.)

V. The Live System – Measurements and Results

The results presented sofar were obtained using an unoptimized toolchain with custom solutions and on a limited set of revisions. So we implemented the approach in a slightly modified way in the actual WebKit build system, initiated its continuous application tied to the actual changes from the version control system, and started the observation of the actual results in this "live" environment.

Beside the implementation in the automatic build system, the method is also implemented in the so called Early Warning System of WebKit, which analyzes the submitted patches to the issue tracking system before submission to the version control system. In this article we mainly deal with experiences about the build system implementation.

A. Technical details of the implementation

Our department is an active contributor to the WebKit community both in development and quality assurance related tasks. As part of the latter, the department hosts and supervises one of the two official *Build Master* servers for continual integration tasks [14]. These servers are running "buildbots" that continuously check the version control repository and when a change appears they initiate its build and regression testing for different build-configurations of WebKit. The build configurations include a variety of ports and different platforms (for example Linux, ARM, Windows, Mac OS X). In this research we were interested in the Qt port and used the x86-based Linux platform for the measurements. Since the frequency of new revisions is fairly

high due to the large development community and intensive development (about one revision per 16 minutes happens on average), the continuous build and test could not necessarily be completed for each and every revision. On average, every second revision is skipped, while in the worst case this number goes up to 20.

Our regression test selection method is implemented as a new build configuration referred to as *Selective Test*. In this new configuration, the steps up to the end of compilation work as usual, but after that a test selection step was inserted, and the test execution runs only the selected tests. The coverage database used for test selection is a PostgreSQL relational database, which was initially prepared for the latest revision we had coverage information for at the time when we started the continuous measurement (rev. 97370). Since then, the database has been updated on a mostly irregular basis, but we are currently implementing an automatic update method. The list of changed procedures is determined using a modified version of the `PrepareChangeLog` script (the original helps developers to find the locally modified procedures when they make their revision comments). The user interface of the build system, and hence of Selective Test as well, is a web site that shows the latest results of the build and tests [14] in various formats.

B. Current status and general statistics

As of April 11, 2012, the Selective Test configuration performed the analysis of 9690 revisions since its launch date on November 16, 2011. This corresponds to about 72% of the commits between the revisions 100422–113914. It is interesting to note that the full test configuration for the same platform analyzed only 6005 revisions in the same period, which means that about 61% more revisions were checked by the Selective Test builder. Table I shows some performance figures for the original and the Selective Test processes. This shows that testing with selection saves a significant amount of time even taking into account the necessary overheads.

Table I
FULL AND SELECTIVE TEST PERFORMANCE. PERCENTAGES IN PARENTHESES ARE RELATIVE TO THE FULL TEST TIMES.

	Full Test	Selective Test
average compilation time	196 secs	111 secs
average selection time	-	33 secs
average test suite execution time	1059 secs	126 secs (11.9%)
average total time	1339 secs	287 secs (21.4%)

The overall coverage statistics for the live system are generally similar to what we found in our initial experiments. The coverage for all procedures in the system was about 68% and the number of failures per revision varied between 0 and 254, the average being 2.1. On average, 8.6 procedures changed per revision, and an average procedure was covered by less than 10% of the test cases. Since we limit our tools to C++ code only, it is important to see the overall amount of changes that affect C++ code (86% of the code base is C++). 18% of the revisions contained only C++ modifications, 56% was mixed and the rest did not contain C++ code modifications.

C. Manual verification of test selection

We investigated the effectiveness of test selection in two phases: first by manually investigating a subset of the builds and then by computing overall statistics.

For the manual investigation we needed a set of revisions shared by both the full and the selective build bots. As mentioned, they ran in parallel for the investigated period, both skipping some of the revisions, but because their build queues were not synchronized, only a certain number of revisions were built by both of them, which resulted in 1665 revisions for comparison. The next filtering we made was whether there were any new failures in them, leading to 119 revisions (with a total of 876 failures). Next, we classified the revisions also according to whether they contained C++, non-C++ or mixed changes. Out of the 119, there were only C++ changes in 5 revisions, while 90 revisions were mixed, and the rest contained no C++ changes.

From the C++-only revisions there was a build problem with one of them, but in the remaining 4 revisions the selective test captured all failures (30 altogether) of the full test (inclusiveness was 100%). Since this number of revisions is quite low to be representative, we calculated this ratio for the mixed revisions too. One revision also crashed from the 90, and from the remaining there were 76 revisions where the selective test did not find all failures. However, from the total 302 failures from these 89 revisions, an overall of 60% (181) was correctly identified. Most of the missed failures by the selective test could probably be attributed to changes in some non-C++ code and the slightly outdated coverage database. Regarding the selection size, the overall distribution of the selection sets, though it was a bit larger, resembled what we observed previously. In 67% of the cases the size was below 1% of the total number of the test cases, and the average was 19.3%.

D. Overall test selection statistics

We also verified the selection capabilities for the remaining ca. 8000 revisions that were not subjects for manual investigation. The selection was performed "offline" for these revisions in a batch process specifically created for this purpose. For this, we used the same coverage database, list of changes and set of tools as the Selective Test bot.

To be able to compare the selection capabilities of the live system to our initial experiments, first we selected the closest settings to the initial results, which is the following: we investigated only those revisions that contained changes to C++ code only, and looked only at new failures in the revisions. There were 855 revisions that satisfy these criteria.

The average inclusiveness for these revisions is similarly quite high, 96.63%, while the selection size is larger than we observed previously: the average is 9473 test cases, which is about 40% of the total number (in contrast to 20.57%), and only in about 28% of the cases is the size below 1% (compared to more than 87% in the initial experiments). This difference can be attributed to two factors: tools changed in the live system leading to functional differences, and there was a bigger change in the system code and the related test cases during the investigated live period.

There is, however, an important issue with this kind of interpretation of the average inclusiveness. Namely, we included those revisions too that did not have any new failures in them. Naturally, the selection inclusiveness is perfect in these cases, so we calculated the data also only for those revisions where these non-failing revisions were filtered out (and keeping only C++). There were 70 of this kind of revisions, 37 revisions with perfect inclusiveness and 27 with zero, while there were relatively few values between these two extremes. The relationship between individual inclusiveness and selection size values can be seen in Figure 3.

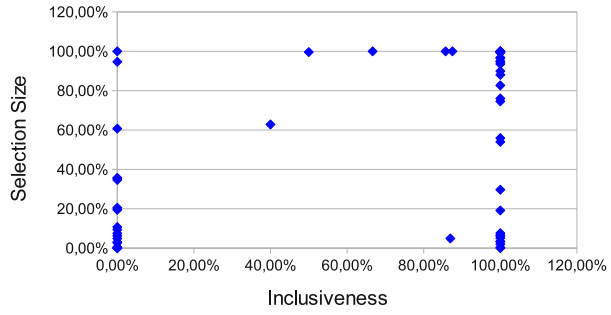

Figure 3. Inclusiveness and selection size compared (excluding non-failing revisions). Note, that in reality there are no cases with less than 100% inclusiveness for 100% selection size. Where this may be observed in the Figure it is a visualization weakness.

The interesting area is where the size is small with large inclusiveness, while there are some cases when the inclusiveness was achieved by a big selection set. These latter cases are good candidates for test prioritization which is the topic of our next section. The overall inclusiveness (by counting the failures individually over all revisions) was **75.38%**, which corresponds to 294 failures identified of a total of 390 in all revisions. The selection in this case is a bit bigger, 11743 test cases on average (49%).

We concluded that the inclusiveness data are comparable to the initial experiments, and that the differences are due to a slightly different way of coverage computation and processing. There are different reasons for the imperfect inclusiveness, most notably, uncertainties with the GCC-based instrumentation, the slightly outdated coverage database and the imperfections of the change determination method.

VI. Selection reduction by prioritization

Using a pure change-based selection, sometimes there will be too many (if not all) test cases selected. If the selection size is important, for instance, in applications for embedded platforms where computation capacity is limited, we can enhance the method by applying prioritization on the selected test cases. We elaborated several strategies to prioritize the test cases based on different attributes, and observed their effects on the defect detection capability. We plan to implement these features in the near future into the live regression test selection system in WebKit.

A. Overview

The basic approach to the prioritization is to calculate various attributes about the test cases and determine an "award" value based on them, which will establish the order of execution of the test cases. Currently, we rely on code coverage only, but in the future we plan to include other attributes as well like defect frequency.

Once we have the ordering, we can set a fixed threshold to select only the first N tests for execution. We experimented with different N values to get a picture of what threshold would be small enough while maintaining reasonable inclusiveness ratios. We compare the different strategies to find out which provides the best inclusiveness with this same number of tests. As a baseline, we also used random prioritization to find out whether any of the more advanced prioritization strategies shows significant difference.

Initially, we worked on the prioritization of only the change-based selected test cases. But another interesting research question that we asked ourselves was whether change-based selection is any better in the first place than some other general approach which ignores the change but relies only on some global property of the test cases. In this case the same prioritization will apply regardless of the change, so this experiment could support or reject the relevance of the very principle of change-based selection.

B. Notations

We introduce some basic notations to make the description of the methods more concise.[1]

R	Set of revisions investigated
T	Set of all test cases
P	Set of all procedures
$ch(r) \subseteq P$	Changed proc. at revision $r \in R$
$chcov(r) \subseteq T$	Covered test cases by the changed procedures at revision $r \in R$
$tcov(r,t) \subseteq P$	Covering procedures of test case $t \in T$ at revision $r \in R$

[1]For simplicity of description, we assume that T and P are the same for all revisions in R. In practice, one would use the actual sets at the given revision. The computation of coverage information of changes at a revision r, hence $chcov(r)$ and $tcov(r,t)$ are subject to any approximation, such as our periodically updated database.

C. Prioritization strategies

The first three strategies are based on the changes while the rest are their counterparts that ignore the changes (note that *Specific* is meaningless for the ignoring case).

General: cover most procedures besides the changed ones. Here, the assumption is that test cases with higher overall coverage are better, which is one of the most often discussed prioritization strategies. More precisely, at a given revision r, prioritize $\forall t \in chcov(r)$ test cases according to the descending order of $|tcov(r,t) \setminus ch(r)|$.

Additional General: adds most additional coverage to the global coverage. Following other authors (*e.g.*, Rothermel *et al.* [12]), this strategy is a refinement of *General* in the sense that it favors those test cases that yield greatest addition to the overall coverage. Formally, at a given revision r, prioritize $\forall t \in chcov(r)$ test cases according to the descending order of $|tcov(r,t) \setminus ch(r) \setminus tcovcumulative|$, where $tcovcumulative$ is a cumulative set of covered procedures by incrementally adding the coverage for already processed test cases.

Specific: cover least procedures besides the changed ones. This is the opposite of *General* in that it selects those test cases first which cover little outside of the changes, in other words, they are supposedly specific to the changes. More precisely, at a given revision r, prioritize $\forall t \in chcov(r)$ test cases according to the ascending order of $|tcov(r,t) \setminus ch(r)|$.

General Ignore: cover most procedures. This strategy is the traditional most covering strategy that does not take into account the changes. For all revisions $r \in R$, prioritize $\forall t \in T$ test cases according to the descending order of $|tcov(r,t)|$.

Additional General Ignore: adds most additional coverage. Similarly, this takes into account the greatest additions to the coverage but ignoring the changes. For all revisions $r \in R$, prioritize $\forall t \in T$ test cases according to the descending order of $|tcov(r,t) \setminus tcovcumulative|$.

In addition, we will also use *Random* prioritization on the change-covering test cases, *i. e.*, at a given revision r, we prioritize $\forall t \in chcov(r)$ test cases randomly.

D. Results

We applied the different strategies for test prioritization to the final data set from the last section, that is, to those 70 revisions that had C++ changes only and at least one failure. Once the priority order of the change-based test cases is produced according to a strategy, the first N test cases are kept (or all of them if there are less than N test cases in the selection). To find out the effectiveness of prioritization algorithms we measured the inclusiveness values at different N values, and compared these values to the inclusiveness of the unprioritized list (which was about 75%). We computed these values for the first 50, 100, 150, and so on, until 1000, and for 2000, 3000, until 25000 (this last value practically means no selection by prioritization).

We found that there are interesting differences between the performances of the strategies. The two most promising strategies were *General* and *Specific*, so we compared them for each mentioned N (see Figure 4). The first thing we can observe on this graph is that these two approaches behave differently, and apart from the small exception around 2000-3000 (10% selection size), the *Specific* strategy always outperforms *General*.

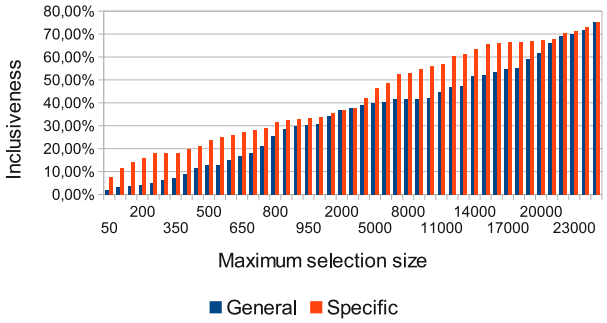

Figure 4. Inclusiveness for the *General* and *Specific* strategies at different selection size thresholds.

For selection size 250 (which is about 1% of the total number of test cases) *Specific* was more than three times better, and in the region around 50% selection size it is better by about 30%. Using the *Specific* strategy, we can obtain over 30% inclusiveness by limiting the selection to only about 3% of the test cases (the inclusiveness for the unprioritized case is about 75%).

The result was surprising because, to the best of our knowledge, there is no previous research that suggested to use a strategy similar to *Specific*; most researchers suggested high coverage as the most important prioritization strategy (*General* in our case). At this stage of the research we did not investigate the actual reasons for this finding, we just speculate that it may be due to the hypothesis that those tests are better at finding a specific fault in a change which concentrate around the change and, hence, are less general.

Table II
SELECTION EFFICIENCY FOR TOP-N PRIORITIZATION.

Strategy	*Incl.*	*Sel. size*	*Incl.*	*Sel. size*
a) General	5.13%	max 250	40.00%	max 5000
b) Add. General	7.69%	max 250	42.05%	max 5000
c) Specific	17.95%	max 250	46.67%	max 5000
d) General Ignore	4.87%	250	21.54%	5000
e) Add. Gen. Ign.	0.00%	250	30.51%	5000
Random prior.	7.69%	max 250	40.17%	max 5000
No prioritization	**75.38%**	avg **11743**	**75.38%**	avg **11743**

In Table II, we compare all strategies to the random and no prioritization cases for selection sizes of 250 (~1%) and 5000 (~20%) test cases. It can be observed that the *Additional General* strategy is better than *General*, which

corresponds to what other authors found. We implemented random prioritization to verify the usefulness of the other prioritization strategies and found that, although the difference is not so big, random prioritization shows worse performance than most of the other change-based approaches.

The differences between *Specific* and the other change-based strategies and random are biggest (*Specific* being at least twice as better) at lower selection thresholds, up to about 10%, which suggests that this strategy could be applied with success at this level. It remains for future work whether findings about this strategy can be generalized to other systems and test environments.

It can be concluded that applying *some* prioritization is advised since by limiting the selection size to only 10% we can still obtain about half of the inclusiveness compared to the unprioritized list. Furthermore, any change-based selection is indeed a good strategy for test selection and prioritization, rather than using some other global attributes only. Our two strategies that do not take into account the changes did not produce any convincing results compared to any change-based selection, including random.

VII. Threats to validity

In this work, we used the ratio of observed failures in the reduced test suite compared to the failures in the full test suite as a measure of effectiveness of the method. However, ideally one would be interested in the defect (fault) detection rate instead, as many test cases can fail all because of one single fault. Currently, we had to rely solely on test execution results as fault information was available only from the bug tracking system, which we did not analyze.

We had to modify different parts of the WebKit system to achieve our goals. Although we believe that these modifications do not influence the original functionality of the system, it may happen that certain attributes (like timing of the tests) are reported differently than without applying these modifications. Specifically, the `DumpRenderTree` component [3] had to be modified in order to mark the beginning and the end of the test cases, since more test cases are exercised during a single execution of the tested binaries. Since we are using a third-party instrumentation (a built-in function of GCC), we must rely on the correctness of this method. Indeed, we identified certain problems with it that are related to different compiler versions. We also modified the timing restrictions of the test suite because, due to the need of producing and storing the coverage data, test cases run slower. We raised the timeout value of the test cases, and it might influence the actual results of the test case executions where timing considerations are important.

The periodic coverage database update strategy is a clear limitation of the approach. We had to choose this method, since it is infeasible in this project to recompute the coverage after each revision. The effect is that test selection and prioritization algorithms may work on obsolete data. That is why we apply a regular database update as discussed earlier. We plan to implement a more frequent coverage database update by using a dedicated server for this purpose, which will not perform the actual selection.

As mentioned, regression tests are executed in a batch, and unfortunately not all test cases are completely independent from each other. It can happen that the execution of a test case can influence the outcome of some other ones, and what is more, sometimes this behavior is nondeterministic. This can be due to different specialities of the WebKit system not detailed here. The community treats these situations in such a way that if the result of a test case is fail it is re-executed to verify whether it is consistently failing. If yes, then the final result is fail, otherwise it is treated as such an "unreliable" test case in this execution. These tests are called *flakey tests*. We try to minimize the effect of flakey tests in our experiments by treating them as passing tests. It remains for future work to investigate more deeply the nature of flakey tests and their effect on the selection method.

For the identification of changed procedures we use an approximate code analysis. The `PrepareChangeLog` WebKit script is used for this purpose, which is based on a textual diff and an approximate source code parser. It has known issues in handling some specific code constructs like C++ template instances. This can result in input errors for the coverage computation. Although it could be improved in a number of ways, based on our manual analysis of the precision of this approach the error margin is quite low, well below 1% of the changed procedures.

Finally, we must note that external validity may be affected, since in this report we dealt with only one system and one of its configurations. However, we performed the analysis of the system for about a half-year period of development, which we feel is relatively long to be able to draw valid conclusions for this project. Furthermore, currently we are dealing with C++ language modules only. However, these restrictions are relatively mild with respect to the whole WebKit project, since these cover over about 86% of the whole code base, and we believe that the method would perform similarly for other configurations as well.

VIII. Conclusions and Future Work

The method for regression test optimization applied in this research is based on change-based test selection. Although the basic principle is simple, we reported a fairly complex adaptation in a real, large open source project. One of our basic experiences with this project is that it is far from trivial to achieve what is theoretically to be expected from the basic method, namely perfect inclusion. There were a lot of technical problems to solve, which were, we believe, a significant engineering result. This is supported also by the warm support for the project by the project community.

In summary, the answers to our research questions set forth in Section III of this article are the following. For the

WebKit system, change-based selection is beneficial since usually only a small portion of the tests is relevant for the changes (RQ1). Our change-based test selection method has been successfully implemented in the live environment of the project and can be used to detect failures at a high rate, although due to the limitations required by practical constraints, the inclusiveness is not perfect (RQ2). Including the overheads, the selective test provides significant improvement in total testing time (RQ2). Finally, our test case prioritization strategies proved to be useful to reduce the size of the selection set (RQ3). The strategy that selects the least covering test cases besides the changes was the best in this respect, by which we can limit the size of the selection to below 10% and still achieve about half of the inclusiveness compared to the non-prioritizing case. This result was surprising because it outperformed the most promising approach: the most covering first method.

In this ongoing project we have a set of short- and long-term plans for continuation. On a short-term, we plan with different optimizations to the framework, which regards the instrumentation framework, additional investigation of the flakey tests, inclusion of other languages besides C++, since all these pose threats to the reliability of the method. The periodic coverage database update strategy also needs further research to make it fully automatic and much more effective.

As part of our long-term research agenda, we want to investigate the overall quality of the testing system of the project from other perspectives as well, not just simple code coverage, and hence offer other improvements to the system. For example, we plan to involve the bug database in the analysis, since that way we could get access to other kinds of information which is not possible from the currently used build logs (such as fault detection rate). We also plan to investigate the benefits of applying change impact analysis methods to amend the change-based test selection.

The different heuristics for prioritization we experimented with are promising directions for future work. We plan to continue this research and define more classification types of the tests, and verify the findings on other projects as well.

The ultimate verification of our activities would be if the regression test suite is provably more efficient in finding defects, but for this further research and a longer real usage of the system is required.

Acknowledgements

The authors would like to thank Csaba Osztrogonác, Péter Siket, Béla Váncsics, John Taylor and Attila Kerék for their valuable supporting work for this research. This research was supported by the Hungarian national grants GOP-1.1.1-11-2011-0039 and OTKA K-73688.

References

[1] M. J. Harrold, "Testing: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering at ICSE'00*, 2000, pp. 61–72.

[2] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[3] "The WebKit open source project," http://www.webkit.org/, last visited: 2012-04-19.

[4] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, Aug. 1996.

[5] ——, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 173–210, Apr. 1997.

[6] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Proceedings of the 1992 Conference on Software Maintenance*, 1992, pp. 299–308.

[7] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings on the 17th International Conference on Software Engineering*, 1995, pp. 41–50.

[8] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[9] G. Rothermel and M. J. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 401–419, Jun. 1998.

[10] J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test application frequency," in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 126–135.

[11] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, 1997, pp. 264–274.

[12] 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, 2001.

[13] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[14] "WebKit QT port Buildbot," http://build.webkit.sed.hu/, last visited: 2012-04-18.

[15] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus – reverse engineering tool and schema for C++," in *Proceedings of the International Conference on Software Maintenance*, 2002, pp. 172–181.

[16] J. Jász, L. Schrettner, Á. Beszédes, C. Osztrogonác, and T. Gyimóthy, "Impact analysis using Static Execute After in WebKit," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12)*, 2012, pp. 95–104.