Test Suite Reduction for Fault Detection and Localization: A Combined Approach

László Vidács*, Árpád Beszédes**, Dávid Tengeri**, István Siket*, and Tibor Gyimóthy**
MTA-SZTE Research Group on Artificial Intelligence*, University of Szeged**, Hungary
{lac,beszedes,dtengeri,siket,gyimothy}@inf.u-szeged.hu

***Abstract*—The relation of test suites and actual faults in a software is of critical importance for timely product release. There are two particularily critical properties of test suites to this end: *fault localization* capability, to characterize the effort of finding the actually defective program elements, and *fault detection* capability which measures how probable is their manifestation and detection in the first place. While there are well established methods to predict fault detection capability (by measuring code coverage, for instance), characterization of fault localization is an emerging research topic. In this work, we investigate the effect of different test reduction methods on the performance of fault localization and detection techniques. We also provide new combined methods that incorporate both localization and detection aspects. We empirically evaluate the methods first by measuring detection and localization metrics of test suites with various reduction sizes, followed by how reduced test suites perform with actual faults. We experiment with SIR programs traditionally used in fault localization research, and extend the case study with large industrial software systems including GCC and WebKit.**

I. Introduction

Using regression testing [1], [2] to maintain the overall quality of software systems and reduce the risk of regressions after changes is unavoidable for any software developed and maintained for a longer period of time. But the repetitive use and continuous maintenance of test suites used for regression testing is hard since these test suites tend to grow almost as large and complex as the system under test itself. Hence, various methods are applied to cope with this complexity such as test selection, prioritization and test suite reduction [2].

In this paper, we investigate the problem of test suite reduction whose goal is, in general, to find the test suite's representative subset that satisfies certain properties of the full test suite. A property often sought after is the *fault detection rate*, *i. e.* the goal is to reduce the test suite in such a way that one is able to detect as many faults as possible using the least resources. Predicting the fault detection capability of a test suite in general is hard, and the mostly accepted method is to use different types of code coverage as an alternative [3]. In a basic method, test cases are added to the reduced test suite in a greedy manner aiming at high absolute code coverage (referred to as *Naive coverage-based method* in this paper). A more profound method is to select test cases that increase mostly the cumulative coverage gained by previously selected test cases (called *Additional coverage-based method*).

However, coverage-based approaches may have a negative impact on another important test suite property, which is *fault localization* capability [4], [5]. Fault localization refers to an activity where, given the failing outcomes of some test executions, the most probable program parts responsible for the faults are sought. A widely researched fault localization method is based on comparing the test outcomes to code coverage (also called *program spectra*) [6]. Among other factors, the successfulness of these methods highly depends on how well a test case can differentiate between the program elements based on their coverage information. But test reductions concentrating on the highest possible code coverage may result in an opposite effect because, for instance, two test cases both having high coverage may be indistinguishable from fault localization viewpoint. Based on this observation, we defined a test reduction method that particularily aims at selecting test cases which better fit fault localization than fault detection. We call the method *Partition-based* because it is based on partitioning the program elements based on their coverage.

In this paper we contribute (1) a new combined reduction method which could be a compromise when both detection and localization aspects are important; and (2) we thoroughly evaluate coverage-, partition-based and combined methods investigating their performance on metrics and real faults with various reduction sizes. The evaluation is performed in two stages: first we define and compute suitable metrics for fault detection and localization capability assessment and compare the methods using these metrics. In the second step we verify the results on actual faults by computing detection and localization success rates and compare our findings to the metrics-based approach. For the empirical assessment we used test data from the program and test database traditionally used in the fault localization field, the SIR repository [7]. In addition, we also used real size industrial programs, including the open source systems GCC [8] and WebKit [9]. Since we wanted to verify our approach on large programs and test suites we chose to use the level of granularity of procedures for analysis. Our results suggest that in many cases it is adviseable to use special reduction methods depending on whether the goal is fault detection or localization. When both aspects are important then a combined approach could be considered.

The paper is organized as follows. We present our motivation and research goals in Section II and basic methods in Section III. In Section IV, experiments are described, while metric- and score-based measurements are presented in Sections V and VI, respectively. Results are summarized in Section VII, and we close with related work and conclusions.

II. Research Aims

We concentrate on two important activities in the test-diagnose-fix lifecycle of software defects: (1) fault detection (observing failing tests) and (2) fault localization (finding the program defects based on the failing tests). Both have a high impact on software quality and their efficiency highly depends on various – sometimes conflicting – properties of the underlying test suites. In this work, we investigate the respective properties of test suites based on their *code coverage information*. Code coverage is essentially a signature of dynamic program behavior reflecting which program parts are run during tests. We will also make use of an *error vector*, which reflects in which runs errors have been manifested.

Background on spectrum-based fault localization: In the fault localization context, code coverage information is called the *program spectrum*. Various forms of program spectra exist, refer to the paper by Harrold *et al.* for an overview [4]. Most of the spectrum-based fault localization methods are based on assigning a "risk" value to each program part and use the thus obtained ranking to localize the fault [10], [6]. In this work we will follow the Tarantula approach that assigns a risk value based on the number of hitting test cases that fail compared to the total number of hits (the more often hitting test cases fail the higher the risk). The successfulness of fault localization is usually assessed by the *diagnosis accuracy* measure, which is the "distance" between the actual program fault and the resulting program element given by the ranking.

In this paper we work with procedure level code coverage, which can be represented as a matrix whose rows are different program runs (test cases of a test suite) and columns correspond to different program parts (procedures in our case). In the matrix, 1 means that the corresponding test case executes the given program element. An example coverage matrix can be seen in Figure 1. Similarly, the error vector will be a binary vector representing the pass/fail outcomes of the test cases (0 means *pass* and 1 means *fail*). We will use P to denote the set of procedures and T for the test suite consisting of test cases.

$$\mathbf{C} = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \end{array} \begin{array}{c} \begin{array}{cccccc} p_1 & p_2 & p_3 & p_4 & p_5 & p_6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array} \quad \mathbf{e} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

Fig. 1. An example coverage matrix (program spectrum)

For fault detection and localization, usually different properties of the reduced test suites are required to hold. Traditional test suite reduction methods are optimized to maintain fault detection capability with the help of high code coverage. On the other hand, for maintaining fault localization of the reduced test suites different approaches are needed. As shown by previous research, coverage based strategies usually have a negative impact on fault localization capability [5].

In this paper, we investigate the relation of test reduction methods optimized for either fault detection or localization and, in addition, we introduce a *combined approach* where we reuse selected tests of both detection and localization methods. Our intention with the combined method is to provide a feasible alternative with a reasonable trade-off for both aspects.

Our research questions are as follows:

RQ1: Is it true that the fault localization-driven reduction method is more suitable for fault localization than fault detection-driven ones, and vice versa?

RQ2: To what extent combined reduction methods are suitable for both fault localization and detection purposes and can they be used as a suitable trade-off?

To answer the above questions we performed a series of experiments on different programs of various sizes. For in depth evaluation we applied two approaches: (1) we evaluate reduction methods by means of fault detection and localization metrics; and (2) extend the metrics-based results with the evaluation of the reduction methods on real faults.

III. Test Suite Reduction Methods

Test suite reduction means the appropriate selection of a set of test cases to reduce testing effort and at the same time maintain good testing properties. Most of test suite reduction problems can be expressed as a *minimal hitting set problem* which is known to be NP-complete [2]. Therefore heuristics are often used to find appropriate reductions. A simple yet effective general heuristic is to use a greedy approach to select those test cases that mostly satisfy the given requirement such as test cases with the highest coverage. Many reduction methods are actually based on *prioritizing* test cases and selecting them in the order implied by the prioritization. Various test suite reduction approaches rely on quite different principles – see an overview in [11]. Test prioritization methods usually concentrate on highest code coverage, however we extend the view by presenting reduction methods designed for fault localization and by combining these methods. In this section, we first introduce two suitable metrics to objectively express fault detection and localization capabilities of a test suite (or its reduced variant), which will then serve as bases for the reduction algorithms presented later in the section. The metrics will also be used for the evaluation of the reduced test suites.

A. Fault Detection and Fault Localization Metrics

Fault detection capability is traditionally predicted using the level of code coverage of test cases. In this paper we define the *Fault Detection* metric or *FD* metric as the ratio of covered elements from all elements (the metric itself merely *predicts* fault detection but we stick with this notation to be consistent with other metrics used in this study). Since we address procedure level capabilities, we define the metric for the $T' \subseteq T$ reduced set of test cases and P set of procedures as follows (clearly, bigger FD metric is better):

$$\textit{FD metric} = \frac{|\{p \in P \mid p \text{ covered by } T'\}|}{|P|}\,.$$

We define the *Fault Localization* metric or *FL* metric to express the localization capability based on the idea of counting the number of examinations one has to perform in worst case to locate a fault for each possible faulty program element (similarly to *FL*, this is also a *prediction* metric). Since the basic approach in all spectrum-based fault localization techniques is to compare the error vector to the coverage vectors of program elements, it clearly follows that the level of diversity in the coverage vectors influences the successfulness of localization. In other words, if there are many similar or same coverage vectors in the matrix, localization will take more effort because those elements have to be investigated one by one as opposed to the case when there are more distinct ones. In our research we use a simplified approach to express the similarity between the error vector and the procedure coverage vectors by looking at exact equivalence.[1]

The equivalence of coverage vectors of the different procedures will determine a *partitioning* on the procedures: procedures covered by the same test cases (e.g. having identical columns) belong to the same partition. For a program and test suite under investigation we will denote such a partitioning with $\Pi \subseteq \mathcal{P}(P)$ with the corresponding partitions $\pi_1, \dots, \pi_K$ (K being the number of partitions). This means that for a partition π_i that contains k elements we need at most $k-1$ examinations to find the faulty element. Assuming that any element of the partition can be faulty we need $k(k-1)$ examinations for π_i, and for the whole program:

$$\textit{FL metric} = \sum_{i=1}^{K} |\pi_i|(|\pi_i| - 1)\,.$$

The metric may be computed for any $T' \subseteq T$ reduced test suite by looking at the corresponding coverage vectors with the selected test cases only. It intuitively follows from the definition that small *FL* values indicate good localization capability and values close to the maximum $N(N-1)$ are worse. It can be computed effectively by determining the partitioning of the program elements using the program spectra. We use a normalized version of this metrics between [0–1].

B. Fault Detection Driven Reduction Methods

The most widely accepted fault detection driven approach aims at achieving the code coverage of the full test suite, but with a possibly minimal subset of the test cases. In this approach one selects the test cases in a greedy way based on their prioritization according to individual code coverage they achieve, starting with the biggest one. Once the desired number of test cases (or full coverage) is achieved the selection is stopped. We will refer to this traditional method as the *Naive coverage-based* method, which is often also referred to as "minimization for total coverage" or "general coverage" method [3]. The Naive coverage method needs a coverage measurement for each test case, but it is limited to a greedy local decision. A more elaborate solution is provided by the *Additional coverage-based* method. Here, in each step already selected tests are extended with a single test from the remaining set. Code coverage is computed for each candidate from the remaining test set, and the one resulting highest total coverage is selected. This requires intensive computations in each step, but in each step we select the optimal solution in terms of total coverage of the reduced test suite.

[1]Clearly, in real life situations this approach may not always be successful for fault localization, however for the purpose of fault localization capability prediction this will be a good approximation.

C. Fault Localization Driven Reduction

We elaborated a concept and implemented a novel test suite reduction method optimized for fault localization rather than fault detection. The key idea of the approach is to select those tests in the reduction that will result in the finest possible partitioning of program elements, because following the intuition behind the FL metric introduced above, fault localization is expected to be more successful with a higher number of partitions and smaller partition sizes.

A naïve implementation of the concept would be to select in each step the test case producing the highest *FL* metric value, however this is infeasible for large software due to the large number of computations required. We employ an algorithm which is more scalable because it does not use the *FL* metric directly but approximates it with a heuristic. The algorithm starts by selecting the first test case that best separates the procedures into two sets based on their coverage. Then, it iteratively selects new test cases that best separate the resulting procedure sets into two new sets each, and so on, until a desired number of test cases is reached (we refer to this reduction as the *Partition-based* method). In each iteration the algorithm adds two times more test cases to the reduced test suite as in the previous iteration, practically producing $1, 3, 7, 15, \dots$ test cases in iterations $1, 2, 3, 4, \dots$, respectively. This type of exponential expansion of test cases is natural due to the structure of the algorithm, but it can be implemented in such a way that the number of additions is limited to a certain limit. We store the reduced test sets in each iteration so that we can make comparisons to other approaches experimentally. This is indeed scalable: all iterations of the biggest program we used took less than an hour.

D. Combined Reduction Methods

The above mentioned reduction methods serve specific aims. Now we propose a new approach to combine the advantages of the two different reduction methods, which works as follows. For a given reduced test suite size k, we compute the intersection of the reduced sets of the two methods, and if the intersection is smaller than k we include additional elements selected interchangeably from both sets until the desired k tests are reached. More formally:

1) Select the intersection of the two test suites and add the result to the combined reduced test suite.
2) Iteration step. Select a new test case alternatingly from the first and then from the second base reduced set. The selection takes place in the order the tests were selected by the base reduction algorithms.
3) Repeat step 2 until the combined test suite reaches the desired size.

Clearly, if the intersection is large then the combined test set will be probably suitable for both tasks. However, if it is small, the combined set will be dominated by a union of the two base sets each according to their own order of elements. We will combine the Partition-based method with both coverage-based approaches, and will refer to them as *Partition-Naive* and *Partition-Additional* methods. In order to be able to make comparable measurements we will compute reduced test suite sizes following the natural sizes produced by the Partition-based method in its iterations.

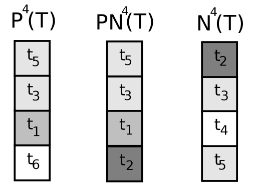
E. Summary of the Reduction Methods

We define more formally and give an example for our total of five test suite reduction methods introduced above. Let T be the set of all test cases and let k be the reduction size. Then five (base and combined) reduction algorithms selecting k test cases are the following:

$$P^k(T) = \{\text{the first } k \text{ elements of Partition-based reduction}\}$$
$$N^k(T) = \{\text{the first } k \text{ elements of Naive coverage reduction}\}$$
$$A^k(T) = \{\text{the first } k \text{ elements of Additional coverage red.}\}$$

$$\begin{aligned} PN^k(T) = & P^k(T) \cap N^k(T) \cup \\ & \{\text{alternate remaining elements from } P^k(T) \\ & \text{ and } N^k(T) \text{ in their order up to } \left|PN^k(T)\right| = k\} \\ PA^k(T) = & P^k(T) \cap A^k(T) \cup \\ & \{\text{alternate remaining elements from } P^k(T) \\ & \text{ and } A^k(T) \text{ in their order up to } \left|PA^k(T)\right| = k\} \end{aligned}$$

Figure 2 shows the reduced test suites for our example.



$$P^4(T) = \{t_5, t_3, t_1, t_6\}$$
$$N^4(T) = \{t_2, t_3, t_4, t_5\}$$
$$A^4(T) = \{t_2, t_1, t_3, t_4\}$$
$$PN^4(T) = \{t_5, t_3, t_1, t_2\}$$
$$PA^4(T) = \{t_3, t_1, t_5, t_2\}$$

Fig. 2. Example results of reduction methods.

IV. Measurements

This section gives an overview of the subject programs investigated in our experiments: a collection of small to big programs with corresponding test suites. Our set of subject systems consists of three types of programs: programs from the SIR repository [7] traditionally used in fault localization research, three medium size programs and two industrial size open source systems. The SIR set of programs includes programs from the the so-called Siemens suite as well (without 'replace', because we were unable to extract test case data from the downloaded version).

Our data set included two large industrial software systems from the open source domain. The first one was the WebKit system, which we have already used in some of our previous investigations [12], [13]. WebKit is a popular open source web browser engine integrated into several leading browsers by Apple, KDE, Google, Nokia, and others [9]. In our measurements we used the Qt port of WebKit called QtWebKit on x86_64 Linux platform. The presented data have been computed on revision r91555 (July 2011). The other large system we used was the GNU Compiler Collection (GCC), the well-known open source compiler system [8]. We chose revision r184199 (configured for C and C++ languages only) for our experiments from February 2012.

Table I summarizes the subject systems and their basic properties. We performed our measurements on the granularity of procedures, hence all statistics are given correspondingly. Column 5 shows the ratio of tests and procedures, which may be interesting for analyzing the different detection and localization results. This ratio is much higher (over 100) in the case of SIR programs than for the big programs, where these two values are comparable. The 6th column shows the number of revisions for each program. Except when noted otherwise, all following measurements were performed for each revision and average values were calculated.

A. Fault Data

The next two columns in Table I show how many and what kind of known faults were available for these programs. In the case of SIR programs, only space included real faults in the form of code annotations, the others were seeded by the repository authors. Although individual seeded faults are similar to real ones, the whole set of artificial faults is of course not as valuable as real faults. Hence from the SIR repository we mostly present results for the space program.

For WebKit, we used a data mining approach to identify most probable faults, which included the analysis of the version control repository and the bug tracking system as follows. The source code of WebKit includes ChangeLog files which contain descriptions about the changes of the system and one or more links to the corresponding entries stored in the bug tracking system of WebKit (Bugzilla). We analyzed the ChangeLog files to collect those commits which corrected bugs. To decide whether the examined bug existed in the given revision we used the Bugzilla entries. We suppose that the bug continuously existed from its reporting date – when the Bugzilla entry was created – to its correction date – when the fix was committed to the source code repository. Only those bugs were taken into account whose lifetime overlapped the date of examined revision. And finally, we analyzed the difference between the two subsequent revisions to find out which source code elements had to be changed to correct the given bug (more details can be found in [14]).

In this work we enhanced Bugzilla-based fault mining results by manual investigations. The main drawback of the Bugzilla-based method is that WebKit developers do not distinguish bugs and feature requests: formally the same kind of bug reports are used for reporting bugs and implementing new features. The bug miner tool reported 21 bugs based on pattern matching, which were corrected within 1 month period following revision r91555, our version of interest. All 21 bug reports were manually investigated at two levels: (1) whether

TABLE I
MEASUREMENT OBJECTS AND PARTITION STATISTICS (* SEE TEXT)

Program name	Lines	Procs	Tests	Tests/Proc	Vers	Faults	Fault type	Cov. %	Diff. %	avg $\lvert\pi_i\rvert$	avg $\lvert\pi_i\rvert$ %
space (SIR)	6.2K	136	13,570	99.89	38	38	Real	100%	30.00%	1.40449	1.03%
printtokens (Siemens)	726	18	4,130	229.44	5	7	Seeded	100%	1.28%	1.63636	9.09%
printtokens2 (Siemens)	570	19	4,115	216.58	10	10	Seeded	100%	0.22%	2.71429	14.29%
schedule (Siemens)	412	18	2,650	147.22	9	10	Seeded	100%	5.20%	1.28571	7.14%
schedule2 (Siemens)	374	16	2,710	169.38	10	10	Seeded	100%	2.16%	1.33333	8.33%
tcas (Siemens)	173	9	1,608	178.67	37	41	Seeded	100%	0.44%	1.5	16.67%
totinfo (Siemens)	565	7	1,052	150.29	20	20	Seeded	100%	0.58%	1.66667	23.81%
augeas	86.1K	784	273	0.35	278	–	–	100%*	70.62%	5.65185	0.72%
bison	34.3K	625	597	0.96	827	–	–	100%*	69.03%	25.934	4.15%
dateutils	18.4K	349	522	1.50	883	–	–	100%*	59.84%	2.766	0.79%
GCC	6.2M	20,372	128,230	6.29	1	–	–	100%*	44.51%	1.83399	0.01%
WebKit	4.5M	46,400	21,987	0.47	1	68	Mined	100%*	88.40%	1.98605	0.00%

the bug report is a bug or feature; and (2) whether extracted procedures really take part in the bugfix. We found that from these 17 were indeed bug fixes, while in the other cases the type of modification was more like development of new features, migration or code refactoring. Altogether 88 methods took part in the correction of the 17 bug reports, from which 68 methods were covered by the test suite. We used these 68 procedures as inputs in fault localization experiments.

B. Coverage

The next column in Table I shows the procedure level coverage ratio of the full test suites in percent. Programs of the SIR repository are provided test cases that cover all methods. The space program is the only larger program consisting of 136 methods and 13570 tests. Industrial software have two orders of magnitude larger size and they have large test suites. For the sake of easier comparison, for the larger programs where the 100% coverage is not satisfied in the basic case, we considered only those tests which were run and only those methods which were covered by test cases. For example, the r91555 revision of WebKit has 25,854 test cases in the test suite, from which 3,867 were not run by the automatic regression tests. This program consists of 72,504 procedures overall, from which only 46,400 were covered. The next column in the table shows how many different coverage vectors exist in the test suite. This number essentially shows the redundancy existing in the test cases as far as procedure level coverage is concerned. Programs with high relative ratio of test cases naturally tend to be more redundant.

C. Fault Localization Partitions

Procedure partitions will be the base for our partition-based reductions and measurements so we collected some basic statistics about them. In the last two columns in the table, the average size of partitions and their size relative to the whole program can be seen. If the programs' test suites exhibit very fine partitioning with many small partitions, fault localization could be better and vice versa. We can observe that, generally, there are a lot of small partitions in the programs. The two biggest programs' partitions have an average size smaller than two, while there are some outlier programs with an average of 14–24% of all procedures. Partition sizes showed similar distributions for the whole program set. In the case of WebKit 99.71% of the partitions were below partition size 30 and there were 2,051 partitions of size one.

V. METRIC-BASED RESULTS

In this section we compare and evaluate the five test reduction methods using fault detection and localization metrics defined earlier. These metrics rely on the coverage data and do not use the information which tests passed or failed. We thoroughly evaluate metric values measured on programs detailed in the previous section. First we present and evaluate our experiments using three representative systems. We selected the space program from the SIR repository and two large real-world systems, GCC and WebKit. For space we provide average numbers computed for all versions while for the other two programs we measured one version each.

For each program and reduction method, we computed reduced test suites with various sizes according to the sizes imposed by the Partition-based algorithm's iterations $(1, 3, 7, 15, \dots)$. The aim was to observe the internal behaviour of methods starting from small to growing test set sizes. Figure 3 shows both *FD* and *FL* metrics for space, GCC and WebKit. The x axis in the figure shows the actual iteration number, for example iteration 7 means that all algorithms are set to select $2^7 - 1 = 127$ tests. This naturally implies that the sizes on this axis are represented in a logarithmic scale. This notation is used in the remaining sections of the paper.

In the figure, *FD* values show an increasing trend as the coverage is increasing by adding new test cases, while the *FL* metric is decreasing. Examining the space program, the Additional coverage method shows the best results in the case of fault detection metric. It reaches the 100% coverage much earlier than others. Surprisingly, Naive coverage method behaves worst: it starts with a relatively high *FD* value, but the Partition algorithm reaches it in 3 iterations. From the localization point of view, the Partition algorithm produces the best results, closely followed by the combined methods. These results are in line with our intuition: each method is best the purpose it is designed for. In addition, the combined algorithms reached good results in fault localization. Large systems show slightly different picture: good fault detection and localization capability is reached only at higher reduction sizes. The Additional algorithm in initial iterations is slightly better even for localization than the Partition algorithm.

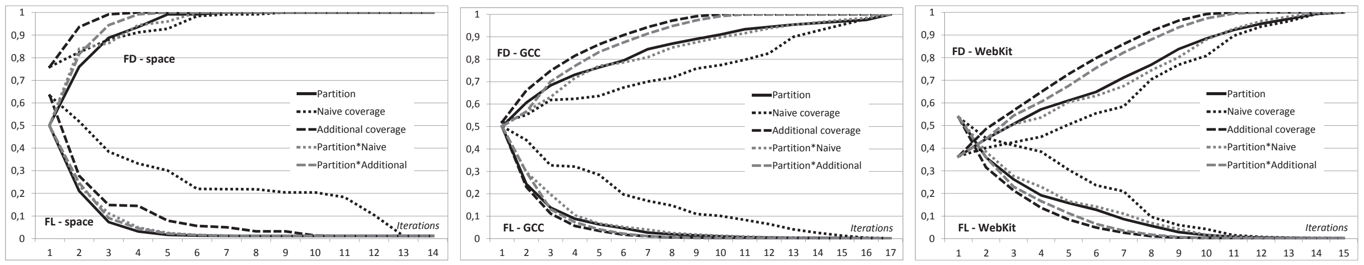


Fig. 3. FD and FL metric changes for the different reduction sizes for space, GCC and WebKit

Given the low success of the Naive coverage algorithm, in the following we concentrate on the Partition and Additional coverage algorithms and their combination. For deeper investigations we narrow down the observed reduction sizes. In Figure 4 results of selected iterations are presented as bar-charts; *FD* iterations are listed in the upper part, while *FL* results are presented in the bottom part. For space the results are almost the same after 5-6 iterations (63 tests), so in Figure 4 only the initial iterations are shown. At this phase the Additional coverage algorithm produced the best *FD* metrics, while the best *FL* metric had the Partition-based algorithm. The combined Partition-Additional algorithm is a close second during all iterations. For GCC and WebKit, initial iterations of both *FD* and *FL* metrics dominated by the Additional algorithm, while the Partition-Additional algorithm was always close to the best result. After some iterations the combined algorithm produced the best *FL* metric values both for GCC and WebKit, which emphasizes the role of the Partition-based algorithm. For large programs these reduction sizes are closer to a real-life scenario: from more than 20,000 test cases a reduced test plan likely contains at least a few hundred test cases (iterations 8 and above). The coverage-centric Additional algorithm sometimes precedes the Partition-based algorithm, which is probaly due to the lower relative number of test cases of WebKit compared to GCC and space.

TABLE II
TWO BEST REDUCTION METHODS IN TERMS OF FL AND FD METRICS

Program name	FL 1st	FL 2nd	FD 1st	FD 2nd
space	Part	Part*Add	Add	Part*Add
printtokens	Part	Part*Add	Add	Naive
printtokens2	Part	Part*Add	Add	Naive
schedule	Part	Part*Add	Add	Naive
schedule2	Part*Add	Part	Add	Naive
tcas	Part	Part*Add	Add	Naive
totinfo	Part	Part*Add	Add	Naive
augeas	Part	Part*Nai	Add	Part*Add
bison	Part	Part*Add	Add	Part*Add
dateutils	Part*Add	Part	Add	Part*Add
GCC	Part*Add	Add	Add	Part*Add
WebKit	Add	Part*Add	Add	Part*Add

To give an overall view of all programs, we prepared brief statistics about which algorithms performed best for the two scenarios. For each program we counted the winners in all iterations and in Table II we provide the two best algorithms in terms of our metrics. For fault localization purposes the Partition algorithm is the best method while usually the combined Partition-Additional version is the second best. Results slightly differ for large programs: Additional method is the second for GCC and first for WebKit. For fault detection the Additional algorithm produced best results – not surprisingly, the algorithm is constructed so that in each step the best available test case is selected for higher coverage. The Naive coverage algorithm is the second for several small programs, but for larger and industrial size programs the combined Partition-Additional method immediately follows the best result.

We summarize and compare our findings with the score-based measurements in Section VII.

VI. SCORE-BASED RESULTS

Our second approach to experimentally investigate the reduction methods was to measure the fault detection and localization accuracies of the reduced test suites on actual faults (called *scores*). This may provide additional insights to the performance of the methods, however these results will particularily depend on the used test sets and related faults and hence should be generalized with caution.

A. Measuring Detection and Localization Scores

The successfulness of a reduced test suite in terms of fault detection (the *FD score*) will be measured by a metric defined by Rothermel and Harrold as *inclusiveness* [11]. It is the ratio of the failing test cases included in the reduced test suite relative to the total number of failing test cases when executing the complete test suite:

$$FD\ score = \frac{failing\ tests\ included}{total\ failing\ tests}$$

Naturally, the FD score is best if it has the value 1 and in the worst case it is 0. For the purposes of the present article we rely on one specific fault localization technique, Tarantula [10], [6]. The *FL score* will be calculated as the *diagnosis accuracy*, the term often used in fault localization. It essentially means measuring the distance between the actual program fault and the resulting program element given by a technique's ranking. There have been various evaluation metrics proposed for this purpose, which are usually based on the amount of program elements one has to investigate before finding the (first) fault. The following method uses the average ranking position for each program element in their respective groups of same values to alleviate the problem of two or more elements with the same ranking value. More precisely, for our selected ranking technique we first compute the average

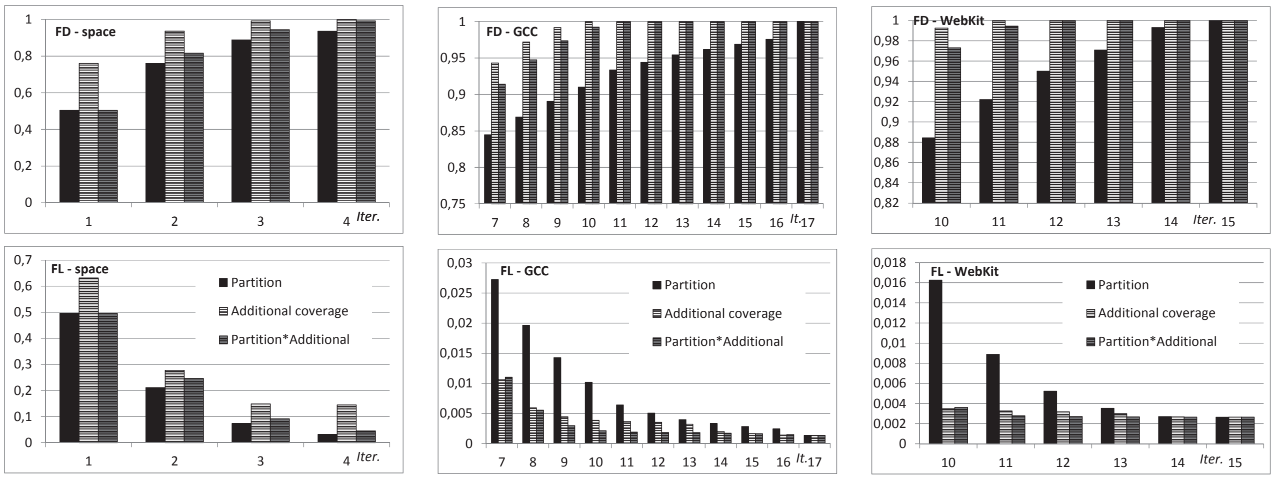


Fig. 4. FD and FL metric changes for selected range of iterations

ranking position $0 \leq R \leq N-1$. Let r_i denote the actual ranking value calculated by the technique for p_i ($i = 1 \ldots N$), and let $1 \leq d \leq N$ be the index of the program element we know to contain the fault. Then:

$$R = \frac{|\{i|r_i > r_d\}| + |\{i|r_i \geq r_d\}| - 1}{2}$$

The FL (or diagnosis accuracy) score is then computed as the relative number of program elements that need not be investigated by traversing the ranking starting from the highest rank and ending in the average ranking position R:

$$\mathit{FL\ score} = 1 - \frac{R}{N-1}$$

Similarly, FL score is best with value 1 meaning that 100% of the program elements can be skipped, and 0 means that none can be skipped, making it the worst value.

B. Results

In this set of experiments we will present detailed results for two programs, space and WebKit. Program space and its test suite are sufficiently large and, as opposed to the other programs in the SIR set, it includes real faults. For this program, we could use 38 faults, one for each revision, and in all cases exactly one procedure was faulty. From the bigger programs we had access to fault data only for WebKit, but in this case the faults were gained by semi-automatic heuristic mining methods, as presented earlier in this paper. We worked with 68 faulty procedures that belonged to 17 actual bugs (we used only one revision in WebKit).

First, we measured the FD and FL scores for the two subject programs for all reductions produced by the 5 methods without any additional filtering of the data and calculated their total average. Figure 5 shows the results for program space. We do not show the detailed results for WebKit at this point, instead we will present related results later in this section. The FD scores mainly showed the expected values with some interesting details. In the charts we included an additional data point, called *Statistically expected* that corresponds to the expected score value of a random reduction of the given size (assuming uniform distribution of failing test cases it will be proportional to the reduction size). In almost all cases all reduction methods produced slightly better scores but there are two different winners for the two programs. For space, Naive coverage proved to be the best while for WebKit it was the Additional coverage. The result for space was surprising because we expected Additional coverage here as well. We speculate that it was due to the unusually large number of test cases compared to the program size (13,570 test cases compared to 125 procedures). Namely, since the total coverage for this program was reached relatively early with the Additional coverage method (8 test cases covered all 125 procedures), all subsequently added test cases were selected arbitrarily, as opposed to the Naive coverage method where in each step the highest coverage was chosen hence there were more chances to include failing test cases.

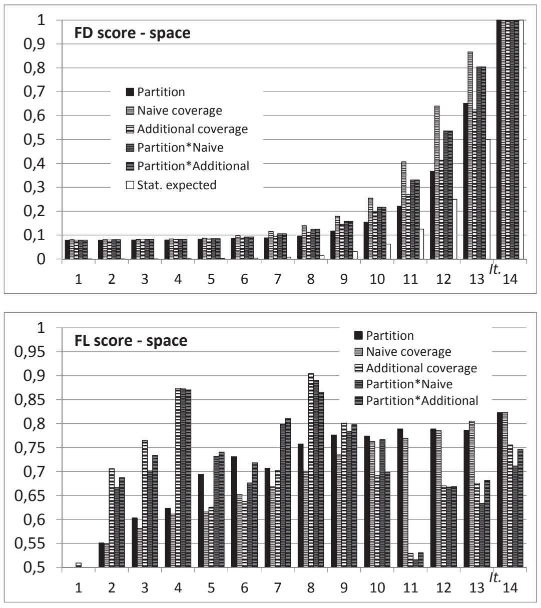


Fig. 5. FD and FL score changes for the different reduction sizes for space

Looking at the charts for the FL scores, we could not observe a remarkably big difference among the different reduction methods, contrary to our findings for the metrics in the previous section. Additional coverage showed best results in the highest number of cases but also the partition and the respective combined method won several times. Similarly, for WebKit we did not observe any significantly better method either. Therefore, we performed various filtering to the data to discover the most relevant cases. First, we decided that we should consider only reduction of certain size. Namely, looking at the FD scores we observed that in the first several iterations of the reduction algorithms there were very few failing test cases included. Only around iteration 10 (which is 1023 test cases) the best performing reduction included about 25% of the failing test cases. This means that with this and even smaller rate of failing test cases, fault localization could begin with a quite big handicap and chances to get acceptable results would be low. Indeed, in the first 5 iterations with different sizes for WebKit the FD score was actually 0 making the corresponding FL scores exactly 0.5. In the last few iterations reduction was too small anyway, hence we decided to concentrate on iterations 10–12 with both programs (sizes 1023, 2047 and 4095).

The second adjustment we performed was that we ignored from FL score calculation those program elements that could definitely be excluded as potentially faulty ones because they were not covered by any of the test cases in the reduced sets. This was required as we noticed that without this filtering, for WebKit the scores were quite low; close to the 0.5 value, which is the score given to program elements that all have the same risk value of 0. With this filtering instead of 68 we got 63, 64 and 67 procedures for WebKit in these three sizes, respectively. For space there were no such elements. The corresponding results can be seen in Figure 6. In these charts we also exluded the two Naive coverage methods because they showed the worst results overall. FD scores are the same as in the previous charts but we can observe changes with this data set regarding the FL score. Namely, in all of these cases the Partition-based method outperformed the others, which was not the case for WebKit without the filtering.

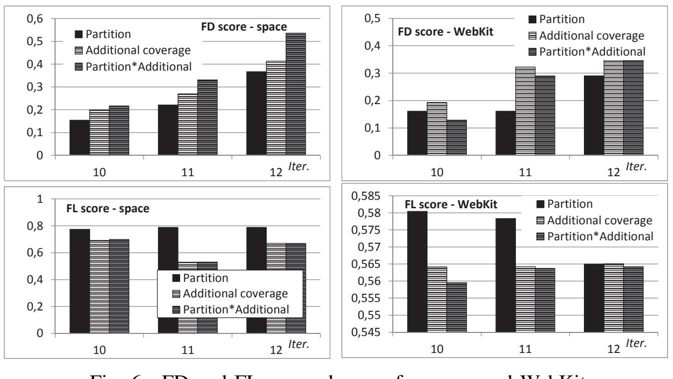


Fig. 6. FD and FL score changes for space and WebKit

A closer investigation of the results for both FD and FL scores will underline the superiority of the Partition-based method for fault localization-aware reduction over coverage-based approaches. Namely, compare any case when the Partition-based reduction produced the best FL score to the corresponding FD score, which was smaller than any other reduction in its category. This means that better localization could be obtained by a test set with *fewer failig tests* of the same size reduction using the partition-based approach than with the coverage-based ones. In other words, the Partition-based algorithm selected fewer failing test cases but they were better in terms of localization. For instance, in the case of the 1023 test suite size for space, the Partition-based reduction selected an average of 181 failing test cases, while the Additional coverage selected 201, yet Partition-based reduction resulted in better fault localization.

In our last set of investigations we looked at the relatively low average FL scores obtained for WebKit. It turned out that it was generally due to the fact that there were several fault localization cases with really low scores compared to the smaller number of high scores and these data points influenced badly the average values. We believe that the relatively high number of such low scores were due to: (1) WebKit's faults were mined using heuristic approaches with no guarantee for their validity, (2) the faulty procedures were all taken together regardless of the bugs they were related to and (3) there could usually be assigned more than one faulty elements to each bug. To alleviate this problem we excluded from FL score computation those program elements whose Tarantula risk values were smaller that 0.5, in other words we worked only with those elements that had more failing test cases than passing ones. The modified scores obtained with this filtering can be seen in Figure 7. We can see that the FL scores were usually above 0.9, but the relative difference between the reduction methods was similar as with the unfiltered version.

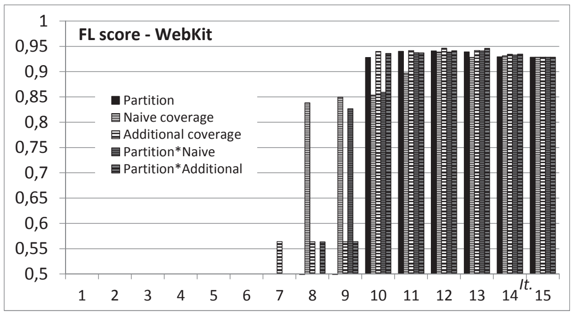


Fig. 7. FL score changes for the different reduction sizes for WebKit when only high risk values are used

VII. Discussion

To address the research questions set forth at the beginning of the article, we followed a dual approach: using conceptual metrics that could predict the actual fault detection and localization (Section V) and with actual faults on some of the subject systems (Section VI). The two sets of measurements showed slightly different results at some points, which is not that much surprising since the metrics are based on heuristics that could behave differently in real scenarios and, second, we had access to a limited set of actual faults, which make difficult the overall generalization of the results. However, for the most parts of the results they align so we can answer the research questions as follows.

RQ1 – Reduction for fault detection vs. fault localization: In terms of fault detection, coverage-based methods seem to be the clear winners, as expected. In the case of metric-based measurements we found that Additional coverage is always the best, which was also the case in most score-based measurements, except that Naive coverage outperformed the rest in some cases (see results for space, for instance). For fault localization, our expectation was that the Partition-based method will be the best performing. In metric-based measurements it was true for smaller programs while for bigger ones and the most relevant reduction sizes its combined version with Additional coverage was slightly better. Score-based measurements supported that Partition-based method is the best for the most relevant reduction sizes in the relevant subjects with faults, space and WebKit. In this case, however, filtering non-covered program elements after reduction was necessary. In summary, coverage-based reduction is usually better for fault detection, while for fault localization Partition-based reduction showed better results. As Figure 6 shows, fault localization was more successful with fewer failing test cases coming from Partition-based reduction than failing test cases reduced for fault detection.

RQ2 – Combined reduction methods: When both fault detection and localization are important for a reduced test suite, a suitable trade-off could be to use a combined reduction method. As our results show, the combination of Partition-based and Additional coverage-based reduction is a viable trade-off. Our metrics-based measurements showed that this combined method usually produced results that are between its two base methods, furthermore the metric was usually closer to the better one. The combined approach was always among the best or second best performing methods in terms of fault localization, and it was the second best for fault detection in the case of bigger programs. In score-based measurements this trend was not always true but the combined method performed relatively good as well, in some cases it even outperformed the other ones. An additional conclusion from our experiments was that the Naive coverage-based reduction usually performed very bad, despite it is the most widely cited approach in the literature. This is probably due to its simplicity of computation (compared to the Additional coverage method, for instance), however we propose to use at least its combination with the Partition-based approach method if the Additional coverage is not feasible. Naive coverage combined with Partition reduction showed comparable results to the other combined method.

A. Threats to Validity

In contrast to many other works on fault localization and fault detection, we performed our experiments on the level of granularity of procedures (functions and methods) because we wanted to apply our approach on large programs and test suites. However, we expect that our approach and the results can be generalized to other granularities as well. As other authors also found, finer granularity analysis outperformed courser granularity only by a relatively small margin overall [15]. Some of our experiments involved "known" faults, in other words program locations annotated as being faulty. Some data of this kind were originated in the SIR repository and were actually seeded faults. The other set (for WebKit) was based on data mining techniques, and its absolute reliability cannot be guaranteed. The main reason for this uncertainty was that the data was mined from the textual comments in the bug reporting database and revision logs, which may include noise. When possible, we used manual verification of the data.

VIII. Related Work

We briefly overview the main results in fault detection and localization. Fault detection is a widely researched area which includes some topics related to our research such as test selection and prioritization. An overview of regression test selection techniques has been presented by Rothermel and Harrold, who introduced a framework to evaluate the different techniques [11]. They defined the evaluation criterion called *inclusiveness*, as the ratio of the failing test cases included in the selection relative to the total number of failing test cases of the complete test suite. We call this criterion *FD score* in our work. Another survey has been presented by Yoo *et al.* [2].

Rothermel *et al.* [3] presented six coverage based prioritization techniques for fault detection. All six techniques improved the fault detection capability of baseline algorithms. 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 [16]. 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 [17]. In another study, Rotermel and Harrold [18] found that coverage-based test suites may provide test selection results superior to those provided by test suites that are not coverage-based. Elbaum *et al.* [15] conducted a set of empirical studies and assessed whether fine-grained (statement level) prioritization techniques outperform coarse-grained (function level) ones. The authors found that the latter produce only marginally worse results in most cases, in turns of effectiveness.

Yu *et al.* [5] investigated the effect of test suite reduction on fault localization capability. The authors implemeted two reduction strategies, the *Statement based reduction* and *Vector based reduction*. The goal of the former is to produce a reduced test suite that executes the same set of statements as the unreduced test suite. This reduction method is similar to our coverage methods. Vector-based reduction aims to produce a reduced test suite that executes the same set of statement vectors as the unreduced test suite. A statement vector is the set of statements executed by one test case. The authors found that if a test suite is reduced using statement-based strategies, the fault-localization technique will almost always perform worse and if a test suite is reduced using vector-based strategies, the fault-localization technique will almost always perform the same. However, no specially designed reduction methods have been presented that are more suitable for fault localization. Baudry *et al.* [19] emphasize the contrast between reduction and fault localization methods, because the localization is more precise when more trace information is available from test cases. This problem is one of the central topics of our research as well. The authors also investigated how to increase diagnosis accuracy by extending the test suite. They systematically add test cases while trying to maximize the number of test case groups which can be distinguished.

Several studies use the so-called Siemens suite to evaluate their approaches. Initially, Hutchins *et al.* [20] used seven C programs with 141 to 512 lines of code to experiment with data flow- and control flow-based test adequacy criteria. The authors increased their test suite by adding 132 versions of the programs by inserting faults into the code. The Siemens suite has been extended since this early study and it is a central database for evaluation of fault detection and localization methods, also as part of the Software Infrastructure Repository (SIR) database [7], which we used as well.

IX. Conclusions

Following upon previous results, this paper underlined that coverage-based test suite reduction traditionally used for fault detection, may have a negative impact on an also important test suite property, fault localization capability. We presented an approach to reduce test suites with this property in mind. This algorithm, the Partition-based method, is then combined with traditional coverage-based approaches to offer a compromise solution for cases when both fault detection and localization are important. Our experimental results showed that *Additional coverage-based reduction* was clearly better for fault detection than fault localization, while *Partition-based reduction* in many cases outperformed coverage-based ones in the fault localization setting. The combined approach proved to be a suitable alternative, although the distinction was not always evident. In addition, the traditional Naive coverage-based method performed poorly in both situations.

We designed our reduction algorithms, the metric and score calculations so that they could be applied for real life systems as well, not only on experimental programs and tests. Thus we were able to apply the methods to and measure industrial size programs and test suites including GCC and WebKit with real faults. In the present work, we used procedure level granularity, but in the future we plan to work on combining this approach with statement-level granularity and hence increase precision while maintaining scalability. Investigation of ways for selecting test suite size is another future research topic.

This research extends previous works towards handling the whole test-diagnose-fix cycle in terms of test suite optimization. We believe that one should always care for all the different activites in this complex process and that suitable combined approaches are required. This paper concentrated on the first two stages but on a longer perspective also the fault fixing aspect should be in focus; in other words, how test suites could be optimized to help in the fixing process as well, and not only in fault detection and localization.

Acknowledgment

This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0039.

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] G. Rothermel, R. J. Untch, and C. Chu, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[4] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," in *Proc. of the 1998 ACM SIGPLAN-SIGSOFT workshop PASTE '98*. ACM, 1998, pp. 83–90.

[5] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *Proc. of the International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 201–210.

[6] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *24th International Conference on Software Engineering*, ser. ICSE '02. ACM, 2002, pp. 467–477.

[7] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[8] "GCC, the GNU Compiler Collection," http://gcc.gnu.org/, last visited: 2013-09-11.

[9] "The WebKit open source project," http://www.webkit.org/, last visited: 2013-09-10.

[10] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. of International Conference on Automated Software Engineering*. ACM, 2005, pp. 273–282.

[11] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 529–551, 1996.

[12] Á. Beszédes, L. Schrettner, B. Csaba, T. Gergely, J. Jász, and T. Gyimóthy, "Empirical investigation of SEA-based dependence cluster properties," in *Proc. of IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'13)*, Sep. 2013, pp. 1–10.

[13] L. Schrettner, J. Jász, T. Gergely, Á. Beszédes, and T. Gyimóthy, "Impact analysis in the presence of dependence clusters using Static Execute After in WebKit," in *Proc. of IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2012, pp. 24–33.

[14] Z. Tóth, G. Novák, R. Ferenc, and I. Siket, "Using version control history to follow the changes of source code elements," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013*. IEEE Computer Society, 2013, pp. 319–322.

[15] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, Feb. 2002.

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

[17] 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.

[18] 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.

[19] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *28th international conference on Software engineering*, ser. ICSE '06. ACM, 2006, pp. 82–91.

[20] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering, ICSE-16*, 1994, pp. 191–200.