Test Suite Evaluation using Code Coverage Based Metrics

Ferenc Horváth[1], Béla Vancsics[1], László Vidács[2], Árpád Beszédes[1], Dávid Tengeri[1], Tamás Gergely[1], and Tibor Gyimóthy[1]

[1] Department of Software Engineering
University of Szeged
Szeged, Hungary
`{hferenc,vancsics,beszedes,dtengeri,gertom,gyimothy}@inf.u-szeged.hu`
[2] MTA-SZTE Research Group on Artificial Intelligence
University of Szeged
Szeged, Hungary
`lac@inf.u-szeged.hu`

Abstract. Regression test suites of evolving software systems are often crucial to maintaining software quality in the long term. They have to be effective in terms of detecting faults and helping their localization. However, to gain knowledge of such capabilities of test suites is usually difficult. We propose a method for deeper understanding of a test suite and its relation to the program code it is intended to test. The basic idea is to decompose the test suite and the program code into coherent logical groups which are easier to analyze and understand. Coverage and partition metrics are then extracted directly from code coverage information to characterize a test suite and its constituents. We also use heat-map tables for test suite assessment both at the system level and at the level of logical groups. We employ these metrics to analyze and evaluate the regression test suite of the WebKit system, an industrial size browser engine with an extensive set of 27,000 tests.

Keywords: code coverage, regression testing, test suite evaluation, test metrics

1 Introduction

Regression testing is a very important technique for maintaining the overall quality of incrementally developed and maintained software systems [5, 21, 14, 4]. The basic constituent of regression testing, the *regression test suite*, however, may become as large and complex as the software itself. To keep its value, the test suite needs continuous maintenance, *e.g.* by the addition of new test cases and update or removal of outdated ones [12].

Test suite maintenance is not easy and imposes high risks if not done correctly. In general, the lack of systematic quality control of test suites will reduce their usefulness and increase associated regression risks. Difficulties include their ever growing size and complexity [13] and the resulting incomprehensibility. But unlike many advanced methods for efficient source code maintenance and evolution available today (such as refactoring tools, code quality assessment tools, static defect checkers, etc.), developers and testers have hardly any means that may help them in test suite maintenance activities, apart from perhaps test prioritization/selection and test suite reduction techniques [21], and some more recent approaches for the assessment of test code quality [2].

Hence, a more disciplined quality control of test suites requires that one is able to understand the internal structure of the test suite, its elements and their relation to the program code. Without this information it is usually very hard to decide about what parts of the test suite should be improved, extended or perhaps removed. Today, the typical information available to software engineers is limited to knowing the purpose of the test cases (the associated functional behavior to be tested), possibly some information about the defect detection history of the test cases and – mostly in the case of unit tests – their overall code coverage.

Furthermore, most of previous work related to assessing test suites and test cases are defect-oriented, *i.e.* the actual amount of defects detected and corrected are central [6]. Unfortunately, past defect data is not always available or cannot be reliably extracted. Approximations about defect detection capability may be used instead which, if reliable, could be a more flexible and general approach to assess test suites. In this work, we employ *code coverage*-based approximations that are based on analyzing coverage structures related to individual code elements and test cases in detail (code coverage is essentially a signature of dynamic program behavior reflecting which program parts are executed during testing, often associated with the fault detection capability of the tests [21, 22]).

In our previous work [16], we developed a method for a systematic assessment and improvement of test suites (named *Test Suite Assessment and Improvement Method – TAIME*). One of its use cases is the assessment of a test suite which supports its comprehension. In TAIME, we decompose the test suite and program code into logical groups called *functional units*, and compute associated code coverage and other related metrics for these groups. This will enable a more elaborate evaluation of the interrelationships among such units, thus not limiting the analysis to overall coverage on the system level. Such an in-depth analysis of test suites will help in understanding and evaluating the relationships of the test suite to the program code and identify possible improvement points that require further attention. We also introduce "heat-map tables" for more intuitive visualization of the interrelationships and the metrics.

In this paper, we adapted the TAIME approach to use it for assessment purposes and verified the usefulness of the method for the comprehension of the test suite of the open source system WebKit [20], a web-browser layout engine containing about 2.2 million lines of code and a large test suite of about 27 thousand test cases. We started from major functional units in this system and decomposed test cases and procedures[3] into corresponding pairs of groups. We

[3] We use the term procedure as a common name for functions and methods

were able to characterize the test suite both as a whole and its constituents and eventually provide suggestions for improvement.

In summary, our contributions are the following:

1. We adapted the TAIME method to provide an initial assessment of a large regression test suite and present a graphical representation of the metrics computed from code coverage.
2. We demonstrate the method on a large open source system and its regression test suite, where we were able to identify potential improvement points.

The paper continues with a general introduction of the analysis method (Section 2) and an overview of the metrics used for the evaluation (Section 3). Then, we demonstrate the process by evaluating the WebKit system: in Section 4, we report how functional units were determined, while the assessment itself is presented in Section 5, where we also introduce the heat-map visualization. Relevant related work is presented in Section 6, before we conclude and outline future research in the last section.

2 Overview of the Method

Our long-term research goal is to elaborate an assessment method for test suites, which could be a natural extension to existing software quality assessment approaches. The key problem is how to gain knowledge of overall system properties of thousands of tests and at the same time understand lower level structure of the test suite to draw conclusions about enhancement possibilities. In our previous work [16], we proposed a method to balance between the two extreme cases: providing system level metrics is not satisfactory for in depth analysis; while coping with individual tests may miss higher level aims in case of large size test suites.

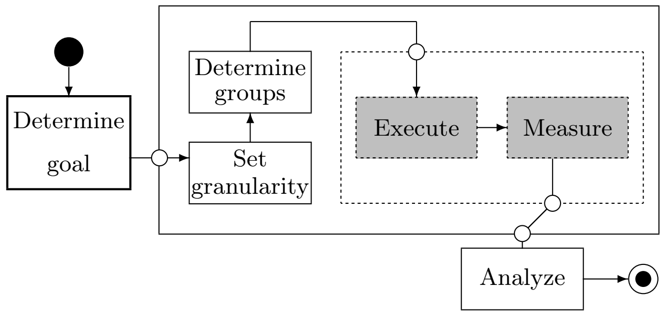

Fig. 1. Overview of the adapted TAIME approach

We adapted the TAIME approach to use it for assessment purposes. The main steps of the proposed test suite analysis method are presented in Figure 1. The basic idea is to decompose the program into features that will divide the test suite and the program code into different groups. This can be done in several ways, e.g. by asking the developers or, as we did, by investigating the tests and the functionality what they are testing. After we have a set of features we can decompose the test suite and the code into groups. Those tests, which are intended to test the same functionality, are grouped together (*test groups*). The features are implemented by different (possibly overlapping) program parts (such as statements or procedures), which we call *code groups.* In the following, we will use the term *functional unit* to refer to the decomposed functionality, which we consider as a pair of associated *test group* and *code group* (see Figure 2). The whole analysis process is centered around functional units, because by this division the complexity of large test suites can be handled more easily. The decomposition process (the number of functional units, the granularity of code groups and whether the groups may overlap) depend on the system under investigation. It may follow some existing structuring of the system or may require additional manual or automatic analysis. More on how we used the concept of functional units in our subject system will be provided later in the paper.

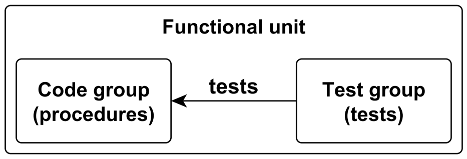

Fig. 2. Functional units encapsulate procedures and corresponding tests

According to the GQM approach [3], the aim of test suite understanding and analysis can be addressed by various questions about functional units. The proposed method lets us answer questions about the relation of code and test groups to investigate how tests intended for a functionality cover the associated procedures; and also about the relation of an individual functional unit to other units or to the test suite.

3 Metrics for Test Suite Evaluation

In this section we give an overview of metrics, coverage and partition that will be used for test suite evaluation. All metrics will be defined for a pair of a test group T and a code group P. Let T be the set of tests of a functional unit and P be the set of procedures which usually belong to the same functional unit.

3.1 Coverage Metric (COV)

We define the *Coverage metric* (COV) as the ratio of the number of procedures in the code group P that are covered by test group T. We consider a procedure as covered if one of its statement was executed by a test case. This is the traditional notion of code coverage, and is more formally given as:

$$\textsc{cov}(T, P) = \frac{|\{p \in P \mid p \text{ covered by } T\}|}{|P|}.$$

Code coverage measures are widely used in white-box test design techniques, and they are useful, among others, to enhance fault detection rate, drive test selection and test suite reduction. This metric is easy to calculate based on the coverage matrix produced by test executions. Possible values of COV fall into $[0, 1]$ (clearly, bigger values are better).

3.2 Partition Metric (PART)

We define the *Partition metric* (PART) to express the average ratio of procedures that can be distinguished from any other procedures in terms of coverage. The primary application of this metric is to support fault localization [19]. The basis for computing this metric are the coverage vectors in a coverage matrix corresponding to the different procedures. The equality relation on coverage vectors of different procedures will determine a *partitioning* on them: procedures covered by the same test cases (*i.e.* having identical columns in the matrix) belong to the same partition. For a given test group T and code group P we denote such a partitioning with $\Pi \subseteq \mathcal{P}(P)$. We define $\pi_p \in \Pi$ for every $p \in P$, where

$$\pi_p = \{p' \in P \mid p' \text{ is covered by and only by the same test cases from } T \text{ as } p\}.$$

Notice that $p \in \pi_p$ according to the definition. Having fault localization application in mind, $|\pi_p| - 1$ will be the number of procedures "similar" to p in the program, hence to localize p in π_p we would need at most $|\pi_p| - 1$ examinations. Based on this observation, the PART metric is formalized as follows, taking a value from $[0, 1]$:

$$\textsc{part}(T, P) = 1 - \frac{\sum_{p \in P}(|\pi_p| - 1)}{|P| \cdot (|P| - 1)}.$$

The numerator can be also interpreted as the sum of $|\pi| \cdot (|\pi| - 1)$ values for all different partitions. It will be best (1) if test cases partition procedures so that each procedure belongs to its own partition, while it will be the worst (0) if there is only one big partition with all the procedures.

$$\mathbf{C} = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \end{array} \begin{array}{c} \begin{array}{cccccc} p_1 & p_2 & p_3 & p_4 & p_5 & p_6 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Fig. 3. An example coverage matrix

The COV and PART metric values for our example coverage matrix (see Figure 3) can be seen in Table 1. The metrics were calculated for four test groups and for the whole matrix, the code group always consisted of all procedures. This example exhibits several cases where either COV is higher than PART, or the other way around. Although in theory there is no direct relationship between these two metrics, with realistic coverage matrices we observed similarities for our subject system as described later.

Table 1. Metrics of the example coverage matrix in Figure 3

Test group	COV	PART	Partitions
$\{t_1, t_2\}$	0.33	0.60	$\{\{p_1\}, \{p_2\}, \{p_3, p_4, p_5, p_6\}\}$
$\{t_3, t_4\}$	1.00	0.33	$\{\{p_1, p_2, p_3, p_4, p_6\}, \{p_5\}\}$
$\{t_5, t_6\}$	0.50	0.73	$\{\{p_1, p_2\}, \{p_3\}, \{p_4, p_5, p_6\}\}$
$\{t_7, t_8\}$	1.00	0.80	$\{\{p_1, p_2\}, \{p_3, p_4\}, \{p_5, p_6\}\}$
$\{t_1 \ldots t_8\}$	1.00	1.00	$\{\{p_1\}, \{p_2\}, \{p_3\}, \{p_4\}, \{p_5\}, \{p_6\}\}$

4 Data Extraction from the WebKit System

The first two steps in the adapted TAIME method is to set the granularity and determine the test and code groups. In this section we present the extraction process of these groups from the WebKit system [20], a layout engine that renders web pages in some major browsers such as Safari. WebKit contains about 2 million lines of C++ code (this amounts to about 86% of the whole source code) and it has a relatively big collection of regression tests, which helps developers keep code quality at a high level.

We chose to work on the granularity of procedures (C++ functions and methods) due to the size and complexity of the system, but our approach is generalizable to finer levels as well. (In [16], we applied the TAIME method on both statement and procedure level.)

The next step was to determine test and code groups. WebKit tests are maintained in the `LayoutTests` directory. It contains test inputs and expected results to test whether the content of a web page is displayed correctly (they could be seen as a form of system level functional tests). In addition to strictly testing the layout, it contains, for example, http protocol tests and tests for JavaScript code as well. Tests are divided into thematic subdirectories to group tests of particular topics. These topics are the different features of the WebKit system and this separation made by the WebKit developers gave the base of the decomposition of the program into functional units. Based on this separation, the decomposition of the test suite into test groups was a natural choice.

At the separation of the program code, it is important to note that the associated code groups were not selected based on code coverage information, rather on expert opinion about the logical connections between test and code. Code groups could be determined automatically based on the test group coverage, but in that case the coverage of code groups would be always 100%. Our choice to involve experts helps to highlight the differences between the plans of the developers and the actual state that the test suite provides.

Table 2. Functional units in WebKit with associated test and code group sizes

Functional unit	Number of Tests	Number of Procedures
WebKit	27013	84142
canvas	1073	400
css	2956	1899
dom	3749	3761
editing	1358	1690
html5lib	2118	4176
http	1778	454
js	8313	8113
svg	1955	6336
tables	1340	2035

The resulting functional units can be observed in Table 2. We identified a total number of 84142 procedures in WebKit, and the full test suite contains 27013 tests, as shown in the first row of the table.[4] The rest of the table lists selected functional units and the statistics of the associated test and code groups, which are detailed in the following subsections. We asked 3 experienced WebKit developers – who have been developing WebKit at our department for about 7 years – to determine functional units of the WebKit system.

4.1 Test Groups

As the experts suggested, the test groups were determined based on how WebKit developers categorize tests. The `LayoutTests` directory contains a separate directory for each functionality in the code that is intended to be tested. These directories (and their sub-directories) contain the test files. A test case in WebKit is a test file processed by the WebKit engine as a main test file, which can include other files as resources. The distribution of the number of tests in these main test directories can be seen in Figure 4. There are several small groups starting form the 12th largest directory in the middle of the diagram. We could either put them into individual groups, which would make the analysis process inefficient (with too many groups but less gain), or we could threat them as one functional unit breaking the logical structure of the groups. Finally we decided to exclude these tests, so about 91% of tests were included in our experiments.

[4] Actually, there are more tests than this, but we included only those tests that are executed by the test scripts. We used revision *r91555* of the Qt (version 4.7.4) port of WebKit called QtWebKit on the x86_64 Linux platform. Also, only C++ code was measured.

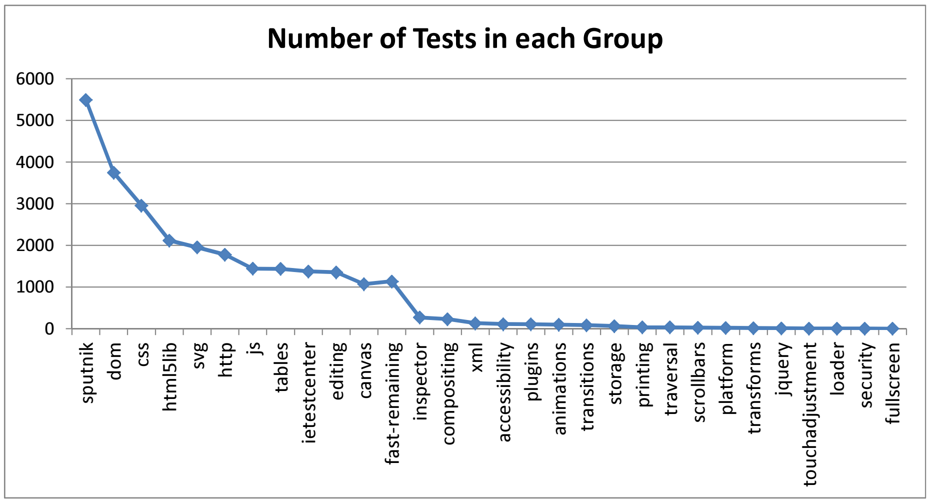

Fig. 4. Number of tests in various test directories in WebKit

However, we found some inconsistencies in the test directory structure, so made further adjustments to arrive at the final set of groups. First, the `fast` directory contains a selection of test cases for various topics to provide quick testing capability for WebKit. This directory is constantly growing, and takes about 27% of all the tests in WebKit. Unfortunately, it cannot be treated as a separate functional unit as it contains tests that logically belong to different functions. Hence we associated each `fast/*` subdirectory to some of the other groups and a small, heterogeneous part is left uncategorized, marked as `fast-remaining` (starting with this point we cut the remaining test groups, see in the middle of Figure 4). Second, there exist three separate test groups for testing JavaScript code. This language plays a key role in rendering web pages, and the project adopted two external JavaScript test suites called `sputnik` and `ietestcenter`. In addition, the WebKit project has its own test cases for JavaScript in the `fast/js` directory. We used the union of these three sources to create a common JavaScript category called `js` in our tables.

4.2 Code Groups

Once the features were formed, we asked the experts to assign source code directories/files to functionalities based on their expertise, but without letting them access the coverage data of the tests. This task required two person-day effort from the experts and the result of this step was a list of directories and files for each functional unit. The path of each procedure in the code was matched against this list, and the matching functional unit(s) were assigned to the procedures. We chose this approach to avoid investigating more than 84000 procedures one by one and so we determined path prefixes and labeled all procedures automatically. However, this method resulted in a file level granularity in our dataset, and increased the overlap between functional units. Eventually, out of 84142 procedures we associated about 28800 with one of the functional units, during code browsing only procedures strictly implementing the functionality of the units were selected. The omitted procedures usually included general functionality that could not be associated with any of the functional units unambiguously. At the end of this process, all experts accepted the proposed test and code groups.

We used several tools during this process. For the identification of all procedures in WebKit (including non-covered ones) the static C/C++ analyzer of the Columbus framework [8] was used. Furthermore, we used the SoDA library [15, 17] for trace analysis and metrics calculation after running the tests on the instrumented version of WebKit.

At the end of this process we determined the groups, executed the test suite and measured the coverage using the given metrics.

5 The Internals of a Test Suite as seen through Metrics

In this section, we present the last, analyze step of the proposed TAIME approach by first presenting our measurement results using the introduced metrics on WebKit, then at the end of the section we evaluate the system based on the result and determine possible enhancement opportunities.

5.1 Coverage metrics

In the first set of experiments, we investigated the code coverage metrics for the different functional units to find out if a test group produces higher overall coverage on the associated code group than the others. In particular, for each *test group - code group* pair we calculated the percentage of code that the corresponding test cases cover, which we summarize in Table 3. Rows in the table represent test groups while code groups are in the columns, and each cell contains the associated coverage metric COV. For example, we can observe that tests from the `canvas` functional unit cover 26% of the procedures of unit `css`.

The first row and column of this table show data computed for the whole WebKit system without separating the code and tests of the functional units. The coverage ratio of the whole system is 53%, which means that about 47% of the procedures is never executed by any of the test cases (note that for computing system level coverage all tests are used including the 9% tests omitted from individual groups). This is clearly a possible area for improvement globally, but let us not forget that in realistic situations complete coverage is never aimed due to a number of difficulties such as unreachable code or exception handling routines.

We can interpret the results of Table 3 in two ways: by comparing results in a row or in a column. In the former case, we can observe which code groups are best covered by a specific test group, and in the latter the best matching test group can be associated with a code group. Using the latter it can be observed which tests are in fact useful for testing a particular piece of code. We used a graphical notation in the matrix of two parallel horizontal lines for rows and two

Table 3. Coverage metric values and heat-map for test groups in functional units

Test \ *Code*	*WebKit*	*canvas*	*css*	*dom*	*editing*	*html5lib*	*http*	*js*	*svg*	*tables*
WebKit	.53	.56	.61	.59	.67	.67	.65	.47	.50	.72
canvas	.16	.46	.26	.24	.07	.19	.00	.30	.03	.45
css	.24	.13	.51	.33	.25	.36	.00	.32	.11	.62
dom	.33	.17	.38	.52	.34	.51	.12	.35	.08	.57
editing	.23	.02	.31	.38	.66	.35	.01	.31	.06	.59
html5lib	.29	.12	.37	.43	.46	.52	.13	.34	.20	.63
http	.33	.23	.41	.42	.25	.41	.65	.39	.14	.57
js	.33	.16	.37	.47	.51	.44	.15	.44	.11	.63
svg	.26	.01	.38	.35	.17	.21	.01	.31	.50	.56
tables	.18	.00	.29	.30	.16	.31	.00	.26	.02	.62

vertical lines for columns, respectively, to indicate the maximal values. In order to have a more global picture of the results, we also present them in a graphical way in the form of a 'heat-map': the intensity of the red background of a cell is proportional to the ratio of the cell value and the column maximum, *i.e.* the column maxima are the brightest.

The most important to observe is that, except for `tables`, each code group is best covered by its own test group. This is an indicator of a good separation of the tests associated with the functional units. In the other dimension, there are more cases when a particular test group achieves higher coverage on a foreign code group, but the value in the diagonal is also very high in all cases. The dominance of the diagonal is clearly visible, however there are some notable cases where further investigation was necessary. For example code group `http`, which is very specific to its test group, and `tables`, which does not have a clear distinction between the test groups. We are going to discuss these cases in more detail at the end of this section. Another observation is that the coverage values in the main diagonal are close to the overall coverage of 53%.

5.2 Partition metrics

The partition metrics for the functional unit pairs showed surprising results. As can be seen in Table 4 – that also shows the corresponding heat-map information –, the PART metric values basically indicate the same relationship between the test and procedure groups as the coverage metrics. In fact, the Pearson correlation of the two matrices of metrics (represented as vectors) is 0.98.

Table 4. Partition metric values and heat-map for test groups in functional units

Test \ Code	WebKit	canvas	css	dom	editing	html5lib	http	js	svg	tables
WebKit	.77	.80	.84	.83	.88	.88	.80	.71	.74	.92
canvas	.29	.69	.45	.41	.13	.33	.00	.50	.06	.65
css	.39	.23	.72	.53	.37	.57	.00	.51	.15	.81
dom	.55	.30	.62	.76	.56	.76	.22	.57	.16	.79
editing	.39	.03	.51	.61	.87	.56	.00	.51	.11	.81
html5lib	.48	.22	.60	.67	.70	.76	.23	.55	.30	.84
http	.54	.40	.64	.66	.44	.64	.79	.62	.26	.80
js	.53	.28	.59	.71	.73	.64	.26	.68	.18	.84
svg	.43	.00	.55	.56	.31	.33	.00	.50	.74	.79
tables	.31	.00	.46	.50	.21	.51	.00	.45	.04	.83

In general, the coverage and the partition metric values do not necessarily need to be correlated, which is illustrated also in our example from Figure 3 and Table 1. However, certain general observations can be made as follows. When the number of tests and procedures are over a certain limit, it is more likely that a unit with high coverage will include different tests and produce high partition metric as well, but it can happen that a unit with high coverage consists of test cases with high but very similar coverage values, in which case the partition metric will be worse. On the other hand, if the coverage is low, it is unlikely that the partition metric will be high because non-covered items do not get separated into distinct partitions. In earlier work [19], we used partition metrics in the context of test reduction, and there the difference between partitioning and coverage was distinguishable. We explain this also by the fact that the reduced test sets consisted of much less test cases compared to the test groups of functional units from the present work.

5.3 Distribution of procedures

In a theoretical case of an ideal separation of functional units (code and test groups) the above test suite metric tables would contain high values only in their diagonal cells. Our experiments show that this does not always hold for WebKit test groups. We identified two possible reasons for this phenomenon: either test cases are not concentrated to their own functional unit, or procedures are highly overlapping between functional units. We calculated the number of common procedures in all pairings of functional units and it turned out that in general the overlap is small: `css` is slightly overlapped with four other groups;

and the `html5lib` group contains most of the `canvas` and is overlapped with `tables`. From the small amount of the total 567 common procedures we reason that test cases cause the phenomenon. Although the tests are aiming well defined features of WebKit, technically they are high level (system level) test cases executed through a special, but fully functional browser implementation. WebKit developers acknowledged that tests go through functional units, for example a single test case for testing an *svg* animation property will cover also *css* for style information, *js* for controlling timing and querying attributes, which also implies the coverage of some *dom* code as well.

5.4 Characteristics of the WebKit test suite

We summarize our observations on functional units that we found during the analysis of metrics in the previous sections.

Regarding the special cases, we identified two extremes in the system. On the one hand, there are functional units, where code groups are not really exercised by other test groups, while their test groups cover other code groups. One of these groups is `http`. By investigating this group, we found that most functionalities – *i.e.* assembling requests, sending data, etc. – are covered by the `http` test group, while other test groups usually use basic communication and small number of requests. The number of test cases in these groups could probably be reduced without losing coverage, but only with taking care of tests which cover the related code of the group.

On the other hand, there are groups like the `tables` group which is some kind of an outlier in the sense that this code group is heavily covered by all of the test groups. The reason for this is that it is hard to separate this group from the code implementing the so-called box model in WebKit, which is an integral part of the rendering engine. Thus, almost anything that tests web pages will use the implementation of the box model, which is mostly included in the `table` code group. Hence, the coverage data is highly overlapping. Although its coverage is the highest one `tables` maintains good PART metrics. Highly covered by other test groups, `tables` should be the last one to be optimized among the test groups. The number of test cases in this group could probably be reduced due to the high coverage by other modules, however, more specific tests could be used to improve coverage.

According to our analysis there is room for improving the coverage of all code groups as they are around the overall coverage rate of 53%. Another general comment is that component level testing (unit testing) is usually more effective than system level testing (as is the case with WebKit) when higher code coverage is aimed. For example, error handling code is very hard to be exercised during system level testing, while at component level such code can be more easily tested. Thus, in a long term, introducing a unit test framework and adding real component tests would be beneficial to attain higher coverage.

In summary, the experience that we gatherd from analyzing the data of the adapted TAIME method appoints two main ways to improve testing. The proposed metrics can either be used to provide guidance for the developers during the maintenance of the tests, or to focus test reduction on enhancing specific capabilities of the test suite, e.g. fault detection and localization [19].

6 Related Work

Although there exist many different criteria for test assessment [22], the main approach to assess the adequacy of testing has long been the fault detection capability of test processes in general and test suites in particular. Code coverage measurement is often used as a predictor to the fault detection capability of test suites, but other approaches have been proposed as well, such as the output uniqueness criteria defined by Alshahwan and Harman [1]. Code coverage is also a traditional base for white-box test design techniques due to the presumed relationship to defect detection capability, but some studies showed that this correlation is not always present, inversely correlated to reliability [18], or at least not evident [11, 10].

There have been sporadic attempts to define metrics that can be used to assess the quality of test suites, but this area is much less developed than that of general software quality. Athanasiou et. al. [2] gave an overview on the state of the art. They concluded that although some aspects of test quality had been addressed, basically it remained an open challenge. They provided a model for test quality based on the software code maintainability model, however, their approach can only be applied on tests implemented on some programming language.

Researchers started to move towards test oriented metrics only recently, which strengthens our motives to work towards a more systematic evaluation method for testing. Gomez *et. al.* provide a comprehensive survey of measures used in software engineering [9]. They found that a large proportion of metrics are related to source code, and only a small fraction is directed towards testing. Chernak [6] also stresses the importance of test suite evaluation as a basis for improving the test process. The main message of the paper is that objective measures should be defined and built into the testing process to improve the overall quality of testing, but the employed measures in this work are also defect-based.

Code coverage based test selection and prioritization techniques are also related to our work as we share the same or similar notions. An overview of regression test selection techniques has been presented by Rothermel and Harrold, who introduced a framework to evaluate the different techniques [14]. Elbaum *et al.* [7] conducted a set of empirical studies and and found that fine-grained (statement level) prioritization techniques outperform coarse-grained (function level) ones, but the latter produce only marginally worse results in most cases, and a small decrease in effectiveness can be more than offset by their substantially lower cost. A survey for further reading in this area has been presented by Yoo *et al.* [21].

7 Conclusions

Systematic evaluation of test suites to support various evolution activities is largely an unexplored area. This paper provided a step towards establishing a systematic and objective evaluation method and measurement model for (regression) test suites of evolving software systems, which is based on analyzing the code coverage structures among test and code elements. One direction for future work will be to continue working towards a more complete test assessment model by incorporating other metrics, possibly from other sources like past defect data.

We believe that our method for test evaluation is general enough and is not limited to the application we used it in. Nevertheless, we plan to verify it by involving more systems having different properties, and in different test suite evolution scenarios such as metrics-driven white-box test case design.

Regarding our subject WebKit, our observations are based mostly on the introduced metrics and we did not take into account the feasibility of the proposed optimization possibilities of the tests. We plan to involve more test suite metrics and investigate our actual suggestions in the future.

References

1. Alshahwan, N., Harman, M.: Coverage and fault detection of the output-uniqueness test selection criteria. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 181–192. ACM (2014)
2. Athanasiou, D., Nugroho, A., Visser, J., Zaidman, A.: Test code quality and its relation to issue handling performance. Software Engineering, IEEE Transactions on 40(11), 1100–1125 (Nov 2014)
3. Basili, V., Caldiera, G., Rombach, H.D.: Goal question metric approach. In: Encyclopedia of Software Engineering, pp. 528–532. John Wiley & Sons, Inc. (1994)
4. Beck, K. (ed.): Test Driven Development: By Example. Addison-Wesley Professional (2002)
5. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: 2007 Future of Software Engineering. pp. 85–103. IEEE Computer Society (2007)
6. Chernak, Y.: Validating and improving test-case effectiveness. IEEE Softw. 18(1), 81–86 (Jan 2001)
7. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: A family of empirical studies. IEEE Trans. Softw. Eng. 28(2), 159–182 (Feb 2002)
8. Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus - reverse engineering tool and schema for C++. In: Proceedings of the 6th International Conference on Software Maintenance (ICSM 2002). pp. 172–181. IEEE Computer Society, Montreal, Canada (Oct 2002)
9. Gómez, O., Oktaba, H., Piattini, M., García, F.: A systematic review measurement in software engineering: State-of-the-art in measures. In: Software and Data Technologies, Communications in Computer and Information Science, vol. 10, pp. 165–176. Springer (2008)
10. Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). pp. 435–445. ACM (2014)

11. Namin, A.S., Andrews, J.H.: The influence of size and coverage on test suite effectiveness. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. pp. 57–68. ACM (2009)
12. Pinto, L.S., Sinha, S., Orso, A.: Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 33:1–33:11. ACM (2012)
13. Pinto, L.S., Sinha, S., Orso, A.: Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 33:1–33:11. FSE '12, ACM, New York, NY, USA (2012)
14. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. IEEE Trans. Softw. Eng. 22(8), 529–551 (1996)
15. SoDA library. `http://soda.sed.hu`, last visited: 2015-08-20
16. Tengeri, D., Beszédes, Á., Gergely, T., Vidács, L., Havas, D., Gyimóthy, T.: Beyond code coverage - an approach for test suite assessment and improvement. In: Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15); 10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART'15). pp. 1–7 (Apr 2015)
17. Tengeri, D., Beszédes, Á., Havas, D., Gyimóthy, T.: Toolset and program repository for code coverage-based test suite analysis and manipulation. In: Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'14). pp. 47–52 (Sep 2014)
18. Veevers, A., Marshall, A.C.: A relationship between software coverage metrics and reliability. Software Testing, Verification and Reliability 4(1), 3–8 (1994), `http://dx.doi.org/10.1002/stvr.4370040103`
19. Vidács, L., Beszédes, Á., Tengeri, D., Siket, I., Gyimóthy, T.: Test suite reduction for fault detection and localization: A combined approach. In: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on. pp. 204–213 (Feb 2014)
20. The WebKit open source project. `http://www.webkit.org/`, last visited: 2015-08-20
21. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2), 67–120 (2012)
22. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. 29(4), 366–427 (Dec 1997)