Beyond Code Coverage – an Approach for Test Suite Assessment and Improvement

Dávid Tengeri*, Árpád Beszédes*, Tamás Gergely*, László Vidács†, Dávid Havas*, Tibor Gyimóthy*
*Department of Software Engineering
†MTA-SZTE Research Group on Artificial Intelligence
University of Szeged, Szeged, Hungary
{dtengeri,beszedes,gertom,lac,havasd,gyimothy}@inf.u-szeged.hu

***Abstract*—Code coverage is successfully used to guide white box test design and evaluate the respective test completeness. However, simple overall coverage ratios are often not precise enough to effectively help when a (regression) test suite needs to be reassessed and evolved after software change. We present an approach for test suite assessment and improvement that utilizes code coverage information, but on a more detailed level and adds further evaluation aspects derived from the coverage. The main use of the method is to aid various test suite evolution situations such as removal, refactoring and extension of test cases as a result of code change or test suite efficiency enhancement. We define various metrics to express different properties of test suites beyond simple code coverage ratios, and present the assessment and improvement process as an iterative application of different improvement goals and more specific sub-activities. The method is demonstrated by applying it to improve the tests of one of our experimental systems.**

***Keywords*—code coverage, regression testing, test suite quality, test suite refactoring, test suite evolution, white box testing metrics**

I. Introduction

The attainable code coverage of regression test suites is often used as the main adequacy criterion, because it is associated with the defect detection capability of the test suite [1], [2]. Code coverage is therefore a traditional base for white-box test design in any life cycle model [3], [4], [5].

Studies showed that the correlation between high coverage and defect detection is not always present or is at least not evident [6], [7], but there are further risks as well in gaining confidence in a test suite based solely on high code coverage. In this work, we present our approach for test suite evaluation that goes beyond simple code coverage ratios, and address in particular the following aspects as well:

- High code coverage may have negative side-effects on the redundancy and similarity of test cases, making it more difficult to distinguish between specific roles of the test cases by what code parts they excercise.
- Different levels of code coverage criteria are typically used such as function, statement or branch coverage. However, in many situations a more detailed coverage is unnecessary (*e.g.* if a function is never invoked it is needless to work with its statement-level coverage).
- An overall coverage associated completely with a system or a module is often not precise enough to provide actionable information on coverage related to specific groups of functionality or code parts.
- Code coverage is only an indicator, and it is often difficult to see how it should be actually used to guide the test suite improvement process.

Reflecting on these issues, we developed a method for a systematic assessment and improvement of test suites (named *Test Suite Assessment and Improvement Method – TAIME*), which is based on various additional computations made on what we call *detailed coverage information*. It is essentially a binary coverage matrix, where rows represent individual test cases while columns correspond to program elements such as statements or functions according to the chosen level of granularity. Both the test cases and the program elements are decomposed into coherent logical groups, which correspond to different *functional units* in the system. This way, various analyses can be performed on the coverage matrix – in addition to identifying low coverage areas –, such as identifying coverage patterns that indicate low coherence within functional units.

In TAIME, we use these analyses to identify potential improvement points in the test suite, for instance, which test cases are candidates for removal or refactoring or where additional tests are needed. The core of the method is to iterate around different improvement goals, code coverage granularity levels and the functional units. The primary use of the method is to aid various situations where a system's regression test suite needs *maintenance* such as reflecting to code changes, test suite refactoring to enhance efficiency, and similar (without continuous maintenance, a test suite will eventually lose its main value – the defect detection capability [8]).

In this paper, we motivate our work by an industrial example we worked with earlier (Section II), then the details of the method are given in Section III, including the overall process, the metrics used for the assessment and several practical use cases. We demonstrate the methods' actual use through the systematic analysis and improvement of one of our own software modules (Section IV). We conclude the paper by discussing related work (Section V) and perspectives. Most importantly, we welcome feedback from the industry about potentials of the method and most prospective enhancement areas.

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)
978-1-4799-1885-0/15/$31.00

II. Motivation and Goals

Regression test suites often become as large and complex as the software itself due to continuous software evolution. If the regression test suite does not undergo continuous maintenance, *e.g.* by the addition of new test cases and update or removal of outdated ones, it will eventually lose its basic value, defect detection potential [8]. This is especially true in the case of software project life cycles following agile principles. In agile, frequent modifications to the code base must be followed by updating regression tests, including their review, selection, retirement, update of test data and test environments [5]. Such continuous changes to the regression test suite will eventually cause significant challanges to the projects due to the lack of systematic methods and tools to aid in this process. Which tests are potentially redundant? How we could improve the efficiency of the tests? What areas of the test suite need extension the most? Etc. But firstly: what is the overall quality of the test suite beyond its coverage? We observed that questions like these are common among testers, developers and managers in various industrial projects.

In previous projects, we worked on various enhancements to WebKit, a large industrially supported open source web-browser layout engine [9]. It has about 2.2 million lines of code and a large test suite of about 27 thousand test cases. In particular, we worked on the analysis and improvement of this test suite (*e.g.* in [10]) but we soon realized that it is hard to comprehend and evaulate it without suitable methods and tools.

We first introduced the concept of *functional units* and how the test suite can be decomposed into such groups to be able to use more fine-grained analysis. The term will be used to refer to a part of the system which can be functionally separated from the others, and is a coherent set of pairs of associated *test groups* and *code groups*. A test group is a subset of test cases used to test the given functionality, and members of the code group are the respective parts of the implementation. (In general, our method does not require that the test and code groups are non-overlapping.)

Next, we developed a supporting toolset (called SoDA) with optimized data structures and algorithms suitable to handle such industrial large and complex systems as WebKit [11], [12]. With the help of WebKit developers, we determined procedure level functional units (composed of functions and methods) along with the associated test groups and measured detailed code coverage information for the test suite. Table I shows how different test groups are covering different code groups in this system. The numbers in the cells of this table represent the code coverage ratios the test cases of a given test group attain with respect to the given code group (or to the whole system as indicated in the first row and column).

We call this visualization a 'heat-map' because in it, the intensity of the red background of a cell is proportional to the ratio of the cell value and the column maximum. As expected, the test cases of a functional unit typically cover mostly their respective code groups (in the diagonal), but it is not always that evident. Using this visualization we were able to pin point potential problems where the overall coverage is too low or which functional units are less coherent.

TABLE I. 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

In the present work, we continue this line of research by providing a method to systematically assess regression test suites with the goal to find improvement points and eventually guide the improvement process (we call it TAIME – Test Suite Assessment and Improvement Method). We will shortly explain what other metrics can be used beyond coverage ratios and the visualization presented above, and how specific improvement goals can be incorporated in the TAIME process.

In particular, our goals with this paper are:

1) We give a general and systematic method for regression test suite assessment and improvement based on detailed code coverage information.
2) We demonstrate the approach by actually applying it to improve our SoDA library.
3) We discuss the possibilities of the industrial application of the approach. We already made the first step in this direction by assessing WebKit as shown above.

III. TAIME Approach

A. Method

Following the principles of the *Goal-Question-Metric* (GQM) paradigm [13], in TAIME the assessment and improvement process is centred around *goals*. By just computing various metrics we would not be able decide where the improvement efforts should be spent. Instead, we first determine our goal (such as increasing coverage or identifying redundancy), then we perform measurements and improvements, and continue with the next goal, if required.

The overall process of the approach can be seen in Figure 1. The outer loop is driven by a goal which defines the improvement in the given iteration of the specific use case. The middle loop is a technical one and it deals with the granularity level of the measurement. Measuring on coarse granularity is usually easier than measuring on finer levels, and is often a good idea to start with coarse granularity if the decisions can be made based on it (*e.g.* if low coverage is identified on a high level it is meaningless to investigate lower levels until higher level coverage has not been improved).

Next, the code and test groups are determined (or reused, if the information is still valid since the last update of the

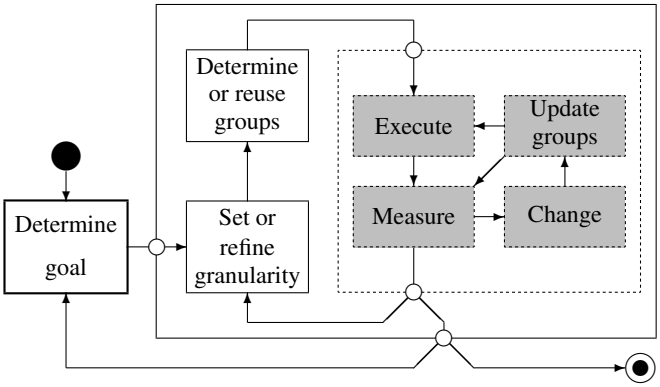

Fig. 1. Overview of the TAIME approach

groups). The process of determining the groups depends on the system under investigation and other factors; the TAIME approach does not prescribe any particular method in this step. It may follow some existing structuring of the system or may require additional manual or automatic analysis. In our WebKit and SoDA experiments we first performed an automatic decomposition based on physical file structures which was then manually refined.

The innermost loop of the process includes the measurement and modification of the tests. We execute the test suite and measure the different coverage-based metrics (see next section) and evaluate the results. Depending on the evaluation, we might leave the inner iteration, or we can change the test suite in which case we might need to update the code and test groups as well before re-executing and measuring the test suite. Note, that re-execution of the test suite is not required if such changes are made that do not affect the actual coverage information. For example, if the change includes only test case deletion, re-execution can be ommitted as coverage data for the remaining test cases will not change, but re-calculation of some metrics will be necessary.

B. Metrics

As part of the TAIME process, a set of basic and derived metrics may be computed from the detailed coverage information gained upon test suite execution. Note, that different kinds of metrics can be used in the approach as long as they may be derived from the detailed coverage information. Also, TAIME does not prescribe any particular method for computing the coverage; usually it can be done by existing tools. All metrics we use are defined for a pair of a test group T and a code group P. Here, T is a subset of all test cases of the test suite, while P contains program code elements according to the given granularity level, such as statements or procedures. The two basic metrics we used are the traditional code coverage ratio and the so-called *partition metric* which we defined in previous work in the context of fault localization and test reduction [10].

The *Coverage Metric (*COV*)* refers to the traditional coverage ratio on the chosen level of granularity. It is given as:

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

Possible values of COV fall into $[0,1]$ (clearly, bigger values are better).

The *Partition Metric (*PART*)* characterizes how well a set of test cases can differentiate between the program elements based on their coverage information. It is an important aspect for certain activities but especially for fault localization whether we can make difference between program elements based on the test cases covering them. Those program elements that are indistinguishable (have the same coverage information) 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

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

Having fault localization application in mind, $|\pi_p| - 1$ will be the number of code elements "similar" to p in the program, hence to localize p in π_p we would need at most $|\pi_p| - 1$ examinations [10]. Based on this observation, PART is formalized as follows:

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

This metric takes a value from $[0,1]$ similarly to COV, bigger meaning better.

In addition to these basic metrics, we experimented with derived ones as well to quantize the efficiency and uniqueness of the tests. The simplest measure for assessing efficiency is *Tests per Program elements (*TPP*)*, which shows how many test cases have been created on average to test a set of program elements (procedures, statements, etc.):

$$\text{TPP}(T,P) = \frac{|T|}{|P|}.$$

To characterize the level of overlapping and cohesion of functional units, we used two related metrics. They measure the unique contribution of test groups to the coverage of a code group compared to all other test cases in other test groups. Let a test group be $T \subseteq \mathcal{T}$, where $\mathcal{T}$ is the set of all test cases. The *Specialization metric (*SPEC*)* shows how specialized a test group is to a code group:

$$\text{SPEC}(T,P) = \frac{|\{t \in T \mid t \text{ covers } P\}|}{|\{t \in \mathcal{T} \mid t \text{ covers } P\}|}.$$

A small SPEC value shows that other test groups have the task to test the code group, while a high value reflects that the given test group is responsible for testing the code group. A related metric is the *Uniqueness metric (*UNIQ*)*, which measures what portion of the covered elements are covered only (uniquely) by a particular test group:

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

A small UNIQ value shows that the program elements of a functional unit are covered by many test cases of other test groups, while a high value indicates that the given test group uniquely covers the code group, because there are few test cases in other groups that cover the same program elements.

C. Use cases

The basic process from Figure 1 can be applied in several practical scenarios, most notably:

- **White-box test design.** The method can be applied during white-box test design which usually aims high coverage of the tested code. We can start with fine granularity and no test cases, and then add new test cases to different functional units while continuously monitoring the overall and group-level coverage. Since we have coverage information for different groups we can use this information to change test cases and design new ones for different test groups.

- **Change-oriented test suite evolution.** In this case, we already have an active test suite and the goal is to maintain its quality (in terms of the metrics used) by following software changes. We can detect and examine changes in the test metric values on a chosen granularity level after a software change, and decide whether we need to modify the test suite if worsening of some metric is observed.

- **Assessment.** The quality of the test suite needs to be assessed periodically. In particular, any improvement process should be started by an initial assessment. Our primary goal in this case is to calculate different metrics in a single measurement and detect any issues that require further investigation. This could then serve as the initial goal in the improvement phases, or just to provide input for a more general software product quality assessment.

- **Refactoring a test suite.** An assessment of the test suite may indicate the presence of so-called test smells – problematic areas among the test cases based on "bad" metric values that require further investigation. In a continuously evolving software, one may want to refactor the test suite if much of test smells are found and the overall quality of the test suite is seen as problematic. We can start with a goal of eliminating some of the test smells using coarse granularity level. We then change (add, delete, modify) tests (possibly modifying functional unit grouping as well) and measure the effect of the changes on the metric values indicating the chosen test smell. Then we may refine the granularity and group associations and restart the inner iteration of the process. After reaching the goal of the finest granularity level, we may choose another goal and start the iteration over, this time continuously checking all the previously addressed goals as well. Note, that in the basic process we do not explicitely define the model how the metric values should be interpreted; what values are to be treated as "good" or "bad". Rather, it depends on the particular situation and is usually assessed in a relative manner.

To implement the TAIME approach, any suitable tooling may be used. Our library and toolset called SoDA and TAM have been designed specifically to accomodate the proposed approach, and is freely available as open source [11], [12].

IV. Improvement of SoDA

To assess and improve the regression test suite of the SoDA library itself, we followed the TAIME approach. In this section, we describe how we instantiated the method in this use, and demonstrate its benefits, which could serve as an example and motivation.

SoDA (Software Development and Analysis framework) is an open source library and toolset that aims to provide researchers and practitioners a framework with which various code coverage-based analyses can be performed in a unified environment. It provides a set of efficient data structures, algorithms and a graphical user interface called TAM, implemented in C++. SoDA and TAM may be used, among others, to implement the TAIME approach.

The library can be divided into two parts. The first defines efficient data structures that stores coverage, test results, and related information for different revisions of a system. The other part implements various algorithms as plugins, which use and manipulate the basic data structures.

SoDA includes a set of unit tests for testing its basic low-level functionality. These tests serve as a regression test suite, which is applied upon changing the library code. This test suite was a good subject to demonstrate the TAIME approach because its initial version was developed in a rather ad hoc manner and we were not aware of how complete it was, and what were its other quality properties. As the first step, we defined the main functional units in the system and assessed the quality of the test suite at procedure level granularity. Table II shows the basic data about the library, and its 8 functional units we identified. At the time of the first assessment (column one), the library had 112 test casess and there was a functional unit that did not have any tests associated to it. The second column shows the final state after performing the TAIME improvement.

TABLE II. Groups and their sizes in the SoDA library

Func. unit	Tests (before)	Tests (after)	Procedures	Statements
cluster	1	10	36	263
data	86	89	213	1588
fl-technique	2	4	16	175
io	13	16	56	429
metric	3	18	60	549
prioritization	2	6	21	159
reader	4	13	35	431
reduction	0	8	33	414
other	1	1	145	331
SoDA	112	165	615	4339

A notable functional unit is *other*, which includes some general utitlity code (*e.g.* exception classes), which is used commonly in the library. This code group contains a number of C++ template functions that are instantiated by the compiler, and this is the reason why this group consists of many procedures (145) but relatively few lines (331).

In the experiment, we applied the proposed method in three phases which are shown in Figure 2.

After the initial assessment, we found that the library had low COV, so we started with the primary goal to improve this attribute of the test suite. While we were refactoring the test suite, we also computed and recorded how other metrics

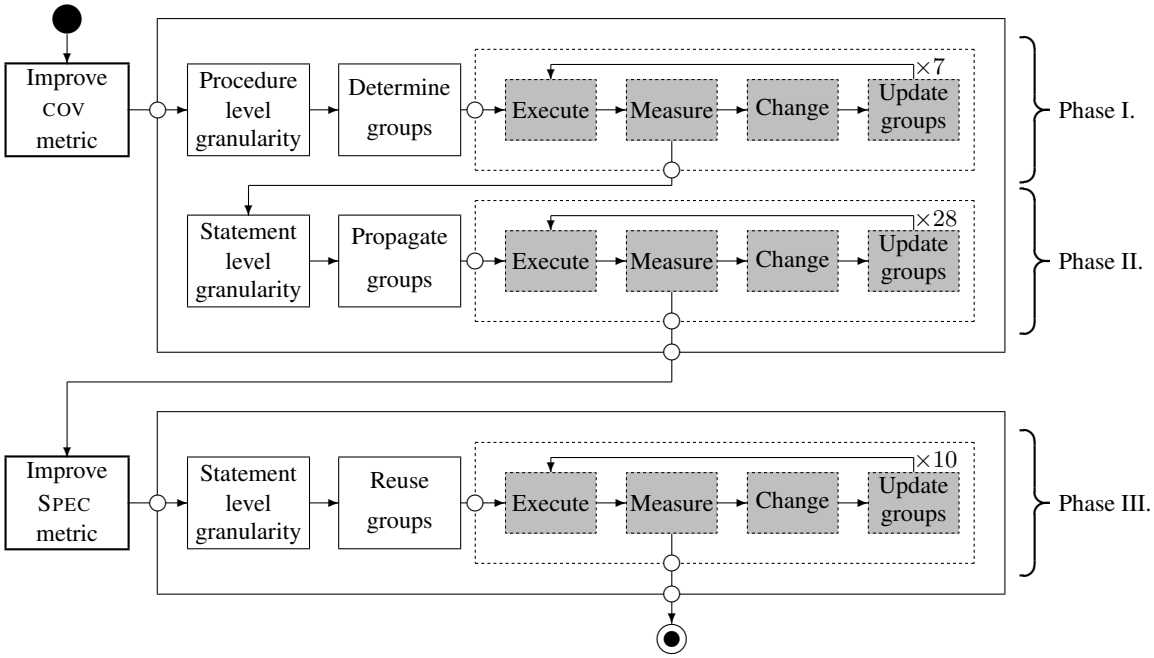

Fig. 2. Application of TAIME for the improvement of test suite coverage and specialization using procedure and statement level analysis

changed. Throughout the improvement process, we changed only the test suite, we did not modify the source code of the library.

In Phase I, we worked on procedure level granularity. We performed 7 improvement steps in this phase, concentrating on a single functional unit in each step. The *reader* group had a very high COV value (0.96), hence we did not make improvements in this group. At the end of this phase, the number of tests has been increased to 131 and the overall coverage improved from 0.58 to 0.69. Figure 3 shows graphically the effect of the improvement on the whole test suite; individual group improvements are not indicated here. Specifically, in the 'Phase I.' section we can see that apart from the coverage, PART also improved from 0.82 to 0.9.

The goal in Phase II. was to improve coverage on a lower granularity level, *i.e.* statements (Table II lists the number of statements in each unit). First, we assigned individual statements to the functional units (simply as the corresponding procedures). In this phase, we performed 28 improvement steps. The process was the same: we improved only one group at a time and did not modify the source code of the library. We found that in many cases a unit with 100% coverage on procedure level granularity could be further improved when measuring coverage on a lower level. The overall improvement of the system's coverage and partition metrics are shown in the 'Phase II.' section of Figure 3 (during this phase we did not measure the changes of the metrics at procedure level). A total of 34 new tests were added to the test suite and 6 tests were modified throughout this phase of the improvement process. At the end of this phase we reached our goal of improving the coverage, but unintentionally the partition metric improved as well. We measured procedure level metrics at this point and we found that COV improved to 0.74 (from 0.69), while PART went up to 0.93 from 0.9.

During improving the coverage of the test groups in Phase II., we also investigated how the changes made in this phase affect the values of COV, PART, and TPP metrics. The statement level metrics of the *cluster* unit showed an interesting phenomenon, which can be seen in the 'Phase II.' section of Figure 4. In the first 5 steps we worked on the *cluster* unit only, and all of its metrics improved; in fact COV, PART and TPP reached their highest values at this point. After the fifth step, however, we started working on the improvement of other units and we noticed that the SPEC metric of *cluster* started to decrease. By the end of Phase II. it reached the original level it had at the beginning of this phase. This indicated that because other units are using code of the *cluster* unit, improving the coverage of other units decreased the specialization of tests related to *cluster*. We evaluated this finding as a 'test smell', which should be corrected because unit tests should be focused on the given unit and test its functionality alone, without being influenced by other units.

These findings and considerations motivated us to further improve the test suite, so we started a new phase, in which our goal was to improve the SPEC metric of the unit tests of the *cluster* group. In Phase III., we performed 10 improvement steps, eliminating the usage of *cluster* procedures from other test groups. Actually, two test groups depended on the *cluster* code, namely the *fl-technique* and *metric*. This dependency was introduced in the set up phase of the unit test where the initial input data of the tests were prepared. The last section in Figure 4 shows how the SPEC metric increased during this phase, until it reached 1.0. We made sure that the values of COV and PART metrics do not decrease during this improvement, furthermore, the related UNIQ metric also increased (in two steps) and reached its maximum at 1.0. This metric raised from 0.64 to 0.67 when the usage of *cluster* code were removed from the *fl-technique* test group and it reached 1.0 in the last step, when we removed the dependency of these plugins from all of the tests of the *metric* unit. This is a good example of how improvements in a test group with respect to

a particular metric might have side-effects on other units and metrics.

Table III summarizes the changes of all of the investigated metrics for the overall system as well as the individual functional units. Values shown in boldface font mean an increase in the metric value, while italic font indicates that the metric became worse. The SPEC and UNIQ values of the *data* and *io* units decreased after the three phases. These two units contain the basic building blocks of the algorithms and as we added more and more tests to the test suite, the program code of these units were used more by other test groups, hence the metric values decreased. Aside from these two cases, we observed improvement in metric values both at procedure and statement level for all of the units. As we can see however, there is still room for improvement: for instance, we could set a new goal to increase the SPEC value of the *io* group by adding more specialized test cases and removing too general ones.

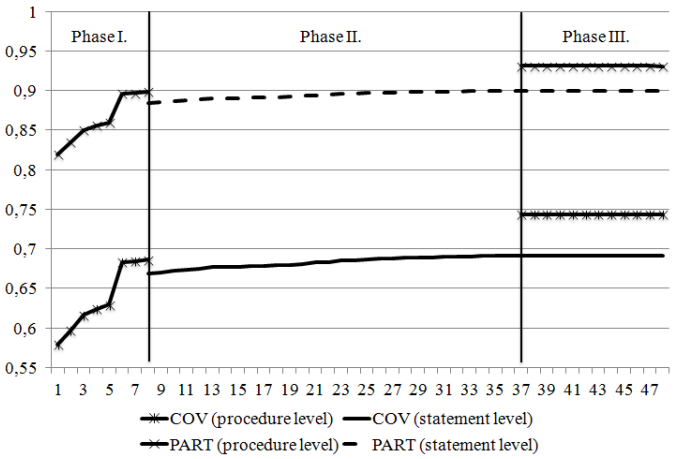

Fig. 3. Basic metric changes at procedure and statement level

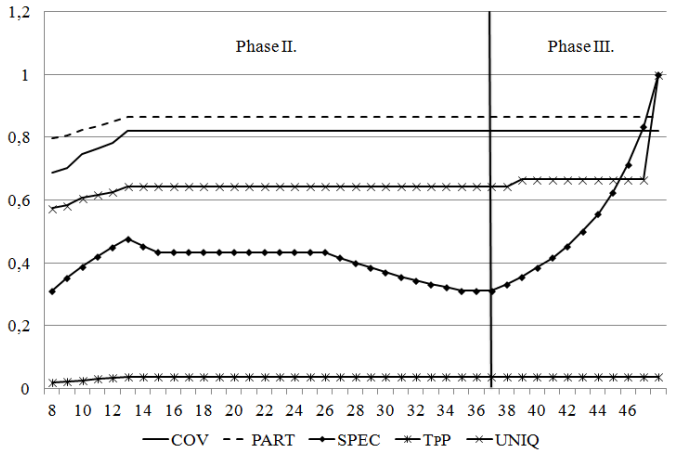

Fig. 4. Improvement of *cluster* group in phases II. and III. with side-effects

V. RELATED WORK

Assessing test quality (as well as software quality) is not an easy task. Researchers started to move towards test oriented metrics only recently, which strengthens our motives to work towards a more systematic evaluation method for testing. There are many aspects and different criteria that can be considered [14] in test quality assessment. The main approach to assess the adequacy of testing has long been the fault detection capability of tests, which is traditionally predicted by coverage metrics, though other approaches have been proposed as well (e.g. the output uniqueness criteria defined by Alshahwan and Harman [15]). Pinto *et al.* performed an extensive empirical study on test suite evolution [8]. As a result, they proposed the development of "intent-preserving" test repair techniques. A similar process can be performed using our approach provided that the "intents" can be expressed by different metrics.

A comprehensive survey of measures used in software engineering by Gomez *et al.* [16] show that only a small fraction of the metrics is directed towards testing. Chernak [17] 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.

Athanasiou *et al.* [18] give an overview on the state of the art regarding the code-based quality criteria. They state that different coverage values as well as static software metrics (applied on test code) are utilized to assess test quality; however they conclude that although some aspects of test quality has been addressed, basically it remains an open challenge. The authors provide a model for test quality based on the software code maintainability model of Software Improvement Group [19]. However, their approach (and similar code-based approaches, e. g. [20]) cannot be applied on tests that are not implemented in the programming language of the systems, which is only typical for unit tests. Furthermore, these approaches examine the structure of the test suite alone, and not in connection with the code it is intended to test. In our approach we combine structure and specification-based information, utilize code coverage and point out additional aspects that may contribute to the quality of a test suite.

Redundancy is another commonly used aspect for test suite optimizations. Abreu *et al.* used code coverage to detect similarity between tests or program code, and used this information for program comprehension [21] and fault localization [22] purposes. Our approach is different: instead of automatic detection of similar items, we use the functional units as a priori information and utilize code coverage to gain more in-depth knowledge about the test suite and its relation to the system under test.

In their paper, van Deursen *et al.* [23] defined test code smells for unit tests, and used them for systematic refactoring of test suites. Bavota *et al.* [24] have shown that most of these smells have negative impact on test code comprehensibility. Such test smells should be detected automatically, and it can be done effectively using different metrics, as it has been studied by van Rompaey *et al.* [25]. Although test code smells are originally defined for test code, some of them can also be interpreted on non-coded tests.

Apart from these related approaches we are not aware of any documented systematic approach to assess and improve test stuites based on the analysis of detailed coverage information. Clearly, our approach can complement the mentioned non coverage-based test suite quality measurement methods.

TABLE III. METRIC CHANGES SUMMARY

	SoDA	cluster	data	fl-technique	io	metric	prioritization	reader	reduction
COV (procedure level)	0.58 → **0.74**	0.33 → **0.89**	0.82 → **0.83**	0.83 → **1.00**	0.88 → **0.89**	0.50 → **0.83**	0.67 → **1.00**	0.96 → **0.97**	0.00 → **0.88**
COV (statement level)	0.56 → **0.69**	0.16 → **0.82**	0.68 → **0.69**	0.78 → **0.80**	0.49 → 0.49	0.41 → **0.68**	0.59 → **0.79**	0.74 → **0.78**	0.00 → **0.77**
PART (procedure level)	0.81 → **0.93**	0.47 → **0.89**	0.96 → **0.97**	0.80 → **0.83**	0.95 → **0.96**	0.69 → **0.86**	0.74 → **0.79**	0.83 → **0.94**	0.00 → **0.88**
PART (statement level)	0.80 → **0.90**	0.27 → **0.86**	0.89 → **0.90**	0.72 → **0.75**	0.73 → 0.73	0.60 → **0.83**	0.71 → **0.80**	0.81 → **0.85**	0.00 → **0,83**
SPEC (procedure level)	1.00 → 1.00	0.17 → **1.00**	0.87 → *0.59*	1.00 → 1.00	0.35 → *0.20*	1.00 → 1.00	1.00 → 1.00	1.00 → 1.00	0.00 → **1.00**
SPEC (statement level)	1.00 → 1.00	0.16 → **1.00**	0.76 → *0.61*	1.00 → 1.00	0.12 → *0.09*	1.00 → 1.00	1.00 → 1.00	1.00 → 1.00	0.00 → **1.00**
UNIQ (procedure level)	1.00 → 1.00	0.00 → **1.00**	0.57 → *0.47*	1.00 → 1.00	0.37 → *0.28*	1.00 → 1.00	1.00 → 1.00	1.00 → 1.00	0.00 → **1.00**
UNIQ (statement level)	1.00 → 1.00	0.00 → **1.00**	0.55 → 0.55	1.00 → 1.00	0.42 → *0.36*	1.00 → 1.00	1.00 → 1.00	1.00 → 1.00	0.00 → **1.00**
TPP (procedure level)	0.18 → **0.27**	0.05 → **0.28**	0.40 → **0.41**	0.17 → **0.25**	0.23 → **0.28**	0.07 → **0.30**	0.13 → **0.29**	0.15 → **0.37**	0.00 → **0.24**
TPP (statement level)	0.03 → **0.04**	0.00 → **0.04**	0.05 → **0.06**	0.01 → **0.02**	0.03 → 0.03	0.00 → **0.03**	0.01 → **0.03**	0.00 → **0.01**	0.00 → **0.02**

VI. PERSPECTIVES

The presented approach for test suite assessment and improvement is part of our long term research oriented towards a more general test suite quality and evolution framework. In the code quality and evolution area there are numerous advanced methods (such as refactoring tools, code quality assessment tools, static defect checkers, etc.), but similar solutions are largerly unexplored in the testing domain.

The small case study we presented on the improvement of SoDA demonstrates the actual use of the method, but its industrial application needs further verification and refinement. We have different plans on how to improve TAIME, but most importantly we plan to apply it to other projects and in various test suite evolution scenarios. In particular, the actual improvement of the WebKit test suite is among our plans, but we are looking for projects where the method would be applied in industrial setting. That said, we particularily welcome any feedback from the industry on the future directions of our efforts with TAIME.

REFERENCES

[1] 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.

[2] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997.

[3] L. Copeland, *A Practitioner's Guide to Software Test Design*. Artech House, 2004.

[4] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 85–103.

[5] L. Crispin and J. Gregory, Eds., *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2009.

[6] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ACM, 2009, pp. 57–68.

[7] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 2014, pp. 435–445.

[8] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, pp. 33:1–33:11.

[9] "The WebKit open source project," http://www.webkit.org/, last visited: 2015-02-06.

[10] L. Vidács, Á. Beszédes, D. Tengeri, I. Siket, and T. Gyimóthy, "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*. IEEE, 2014, pp. 204–213.

[11] "Soda framework and repository," http://soda.sed.hu/, last visited: 2015-02-05.

[12] D. Tengeri, Á. Beszédes, D. Havas, and T. Gyimóthy, "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)*, Sep. 2014, pp. 47–52.

[13] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, *Goal Question Metric (GQM) Approach*. John Wiley & Sons, Inc., 2002.

[14] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997.

[15] N. Alshahwan and M. Harman, "Coverage and fault detection of the output-uniqueness test selection criteria," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 181–192.

[16] O. Gómez, H. Oktaba, M. Piattini, and F. García, "A systematic review measurement in software engineering: State-of-the-art in measures," in *Software and Data Technologies*, ser. Communications in Computer and Information Science. Springer, 2008, vol. 10, pp. 165–176.

[17] Y. Chernak, "Validating and improving test-case effectiveness," *IEEE Softw.*, vol. 18, no. 1, pp. 81–86, Jan. 2001.

[18] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *Software Engineering, IEEE Transactions on*, vol. 40, no. 11, pp. 1100–1125, Nov 2014.

[19] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, Sept 2007, pp. 30–39.

[20] N. Nagappan, L. Williams, J. Osborne, M. Vouk, and P. Abrahamsson, "Providing test quality feedback using static source code and automatic test suite metrics," *2005 IEEE 16th International Symposium on Software Reliability Engineering (ISSRE)*, vol. 0, pp. 85–94, 2005.

[21] A. Perez and R. Abreu, "A diagnosis-based approach to software comprehension," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 37–47.

[22] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 89–98.

[23] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the International Conference on Extreme Programming and Flexible Process in Software Engineering (XP 2001)*, 2001, pp. 92–95.

[24] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, Sept 2012, pp. 56–65.

[25] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *Software Engineering, IEEE Transactions on*, vol. 33, no. 12, pp. 800–817, Dec 2007.