Improving Test Coverage Measurement for Reused Software

Breno Miranda
Università di Pisa
Largo B. Pontecorvo, 3 - 56127
Pisa, Italy
breno.miranda@di.unipi.it

Antonia Bertolino
ISTI - CNR
Via Moruzzi 1 - 56124
Pisa, Italy
antonia.bertolino@isti.cnr.it

***Abstract*—Test coverage adequacy measures provide a widely used stopping criterion. Engineering of modern software-intensive systems emphasizes reuse. In the case that a program uses reused code or third-party components in a context that is different from the original one, some of their entities (e.g. branches) might never be exercised, thus producing a code coverage level far from full and not meaningful anymore as a stopping rule for the program at hand. We introduce a new coverage criterion, called "Relevant Coverage", that in each testing context in which a code is reused calculates coverage measures over the set of relevant entities for that context. We provide an approach for identifying relevant entities using dynamic symbolic execution. The introduced coverage adequacy criterion is assessed in an exploratory study against traditional coverage in terms of test suite size reduction factor, cost-effectiveness ratio and rate of fault detection. The results of our study showed that relevant coverage can considerably reduce the test suite size while preserving a high cost-effectiveness ratio with respect to the traditional approach.**

I. Introduction

Test coverage measures are actively studied in academia, and largely used in industrial software development. To date many research and commercial tools exist for measuring code coverage [1], and recent surveys (see, e.g.[2]) show that coverage measures are increasingly used in several companies as an adequacy criterion, i.e., to decide when to stop testing. Moreover, when the measured coverage is not deemed sufficient, coverage information can be useful for guiding testers in enhancing their test suites so to exercise yet uncovered parts of the program. Indeed, although the relationship between code coverage of a given test suite and its ability to reveal faults is still nowadays object of research, see e.g. [3], there is consensus that having coverage information is still important as it is evident that a test suite can hardly find bugs in code that is never executed.

Even though they may target widely different entities, so far the adequacy criteria proposed in the literature are all based on the same underlying principle: a set of entities that must be covered (they could be statements, branches, paths, functions, and so on) is identified, and a program is not considered to be adequately tested until all entities (or a given percentage thereof) have been executed at least once. Coverage is then measured as the percentage of covered entities with respect to the total number of entities in the program under test. In this work we use the term "Traditional Coverage" to refer to this way of measuring coverage.

Engineering of modern software-intensive systems emphasizes reuse since it is generally recognized as a key technology for improving software productivity and quality [4]. Large complex systems may often include code parts reused in different contexts from the original one in which they were conceived. In such cases, traditional coverage measured as above might not always provide meaningful information. In fact, not all entities might be of interest in every context in which a code is reused. Therefore, by testing such composite systems from within a certain usage context might not achieve 100% coverage.

Note that we do not refer here to the well-known problem of infeasible paths, to take into account which, as early as 1988, Frankl and Weyuker [5] proposed an applicable family of data flow testing criteria. We speak of entities that are perfectly feasible, i.e. there would exist test inputs that exercise them, but such test inputs are not relevant in a usage context because this user would never invoke such inputs in operation.

Our research in this paper addresses such situation and aims at adapting test coverage measures to the specific testing *scope*. More precisely, our proposal is to take into account the way the main program in a complex system integrating existing code and third-party components interacts with them to identify the relevance of each entity to the new scope. We use the term *in-scope entities* to refer to the entities from the reused code that are exercised in the new context. The remaining ones are referred to as *out-of-scope entities*. The newly introduced coverage metric, that we call the "Relevant Coverage", measures the percentage of covered entities against the total number of in-scope entities, thus returning to the tester a more realistic information than the traditional one.

The paper is structured as follows: In Section II we introduce a motivating example, which is then referred in Section III to illustrate the approach. The rest of the paper includes our empirical evaluation of the effectiveness of the newly proposed coverage measure on the *gzip* case study taken from the SIR repository: settings, results and threats to validity of the study are presented in Sections IV, V and VI, respectively. Finally, Related Work (Section VII) and Conclusions (Section VIII) complete the paper.

II. Motivating Scenario

To introduce the notion of relevant coverage, we will use the example of the algorithm to determine the triangle type based on its angles (see Listing 1). In this example, the function `get_triangle_type` receives three integers (a, b and c) and verifies if their sum is equal to 180 (because in a triangle, the three interior angles always add to 180°). If so, the type of the triangle is returned to the user; otherwise, an error message is triggered stating that triangle formation is not possible.

```
1  char * get_triangle_type(int a, int b, int c) {
2    static char result[50];
3    if (a + b + c == 180) {
4      if (a == b && b == c) {
5          strcpy( result, "Equilateral Triangle" );
6      }
7      else if (a == b || b == c || a == c) {
8          strcpy( result, "Isosceles Triangle" );
9      }
10     else {
11         strcpy( result, "Scalene Triangle" );
12     }
13   }
14   else {
15       strcpy( result, "Triangle formation not
             possible" );
16   }
17   return result;
18 }
```

Listing 1: The triangle calculator.

Let us assume that the code from Listing 1 is being reused in a different context from that for which it was initially conceived: instead of returning a text message about the triangle type in the terminal, it is now equipped with a Graphical User Interface (GUI) that receives the triangle's angles values provided by the user and outputs the triangle type. As in many cases, the GUI will validate the users' inputs before calling the appropriate function/method and it will allow them to get the triangle type only if the sum of the values provided is equal to 180. Due to such restriction, some entities of the above code will never be reached (i.e., the `else` branch at line 14 and the statement at line 15). Hence, any structural testing of the system including the GUI and the code from Listing 1 would always return an incomplete coverage measure.

In our approach, presented in the next section, we would like to measure coverage over the set of in-scope entities (i.e., the ones that are relevant in this new context), and not over the whole set of entities available in the code being reused.

III. Relevant Coverage

The rationale behind relevant coverage is that in each scope in which a given system is being (re)used, the in-scope entities are those that could be potentially exercised according to the specific input domain restrictions. This coverage metric is defined in Equation 1.

$$\textbf{Relevant coverage} = \frac{\#\ \textit{of covered entities}}{\#\ \textit{of in-scope entities}} \cdot 100(\%) \quad (1)$$

A. Approach

As for any coverage criterion, relevant coverage measurement presupposes that the code is instrumented so to allow the identification of the entities exercised by the tester.

Relevant coverage measurement can then be summarized in the following steps: **1) Data collection**: information about possible input domain constraints is collected; **2) Identification of *in-scope entities***: the entities (e.g., functions, statements, branches, etc) that are relevant according to the input domain constraints collected in the previous step are identified; **3) Coverage measurement**: relevant coverage is calculated over the set of in-scope entities only, and the results are provided to the tester; and **4) Coverage increase**: the list of in-scope entities that have not been tested is provided along with suggestions of test cases that exercise those entities.

B. Illustration

1) Data collection:

As the first step of our approach, we collect information about possible input domain constraints. The input domain constraints provided could be coarse grained such as a list of functions that are expected to be used in that specific context; or fine grained such as the precise range of values expected to be used by a given variable. The more information is provided, the more precise is the calculation of relevant coverage. In the triangle calculator example, one of the constraints the tester would be aware of would be the fact that, passing through the GUI control, $a + b + c$ is always equal to 180. Other constraints such as the range of values expected for the variables a, b and c, for example, could also be provided.

2) Identification of *in-scope entities*:

Different approaches could be adopted to identify the entities that are relevant under the input domain constraints collected in the first step of our approach. For example, one could apply a reachability algorithm on the static call graph of the given program. Even though this is an undecidable problem [6], there exist algorithms capable of generating approximated solutions. We resolved to use Dynamic Symbolic Execution (DSE) since it has shown to be a powerful approach to analyze the code dynamically and guide its exploration based on the input domain constraints [7], [8]. Our decision was influenced by the fact that DSE is very actively investigated and several tools are available.

For this work, we adopted KLEE [9], a well-known symbolic execution tool capable of automatically generating tests that achieve high coverage even for complex and environmentally-intensive programs. First, we instrument the code from Listing 1 to guide the DSE exploration based on the input domain constraints collected in the first step of our approach. Then, when KLEE is run with the instrumented code, it tries to explore all paths, finding concrete test inputs to exercise them. Thus actually this solution not only allows to find in-scope entities, but also generates test cases that exercise them.

The set of in-scope entities are those exercised by the DSE-generated test cases[1]. Indeed, if some entities are not exercised by the DSE-generated test cases it is because they are not reachable under the input domain constraints provided.

3) Coverage measurement:

[1]Note that we "replay" the test cases generated using the original program (not the instrumented one) and use gcov so to get accurate coverage achieved by the DSE-generated test cases.

To continue with our stepwise example, let us assume that the tester of the GUI-enhanced triangle calculator has created the test suite displayed in Table I. It includes 3 test cases covering the possible ways in which an isosceles triangle can be defined and 1 test case covering the scalene type, but no test cases to cover the equilateral type.

TABLE I: Tester's test suite to assess the integration between the GUI and the triangle calculator.

Test Case #	Input Values	Comments
1	a=20, b=80, c=80	Returns "Isosceles Triangle"
2	a=80, b=20, c=80	Returns "Isosceles Triangle"
3	a=80, b=80, c=20	Returns "Isosceles Triangle"
4	a=50, b=60, c=70	Returns "Scalene Triangle"

Traditional coverage would be calculated as the ratio between the number of entities exercised and the total amount of entities available. For line coverage, for example, the tester's test suite would achieve a coverage of 77.77% (7 out of 9 executable lines, considering only the code of the function `get_triangle_type`). Note that, even though the statement at line 15 from Listing 1 can never be reached in the context in which the code is going to be used, it would still be considered when calculating coverage in the traditional way.

Using relevant coverage, the tester's test suite would instead achieve a line coverage of 87.50% (7 out 8 in-scope lines exercised in the function `get_triangle_type`). As expected, the line related to the scenario in which the triangle formation is not possible is excluded from the coverage computation as it can never be reached in that specific context.

In comparison with traditionally calculated coverage, relevant coverage gives a more meaningful measure: the point is not that the tester achieves a higher score, but that such score provides a more realistic estimate of what could be further achieved by augmenting the test suite.

4) **Coverage increase:**

Having a coverage measure that is more meaningful but does not help testers to improve their test suite would be little useful. As the last step of our approach, we help testers to augment their test suites to exercise those in-scope entities that had been left untested. Because we are using KLEE to identify in-scope entities and as said it generates the test cases that exercise all the paths explored, in this step we can simply suggest among the DSE-generated test cases those ones that can increase coverage when compared to the tester's test suite.

IV. Exploratory Study

The research objective of this exploratory study is to *analyze* the usefulness of the proposed relevant coverage adequacy criterion *for the purpose of* comparison *with respect to* traditional coverage adequacy criterion *from the point of view of* testers *in the context of* the testing of applications that are reused in different contexts.

A. Research Questions

Having defined our general research objective, there are many specific questions that could be targeted, depending on how the generic term of "usefulness" in the above definition is assessed. For this study we have defined the following goals (following the Goal/Question/Metric method [10]):

Goals: given a specific test context (in which part of a program under test is reused), evaluate the usefulness of relevant coverage adequacy criterion, when compared to traditional coverage adequacy criterion, in terms of: test case selection ($\mathbf{G}_1$), fault detection ability ($\mathbf{G}_2$), and test case prioritization ($\mathbf{G}_3$).

The assessment of goals $\mathbf{G}_1$ to $\mathbf{G}_3$ is performed by the following research questions and metrics:

$\mathbf{Q}_1$ [test suite size]: Given a specific testing context, does relevant coverage adequacy criterion reduce the test suite size when compared to traditional coverage adequacy criterion?
$\mathbf{Q}_2$ [cost-effectiveness]: Given a specific testing context, do test suites selected according to relevant coverage adequacy criterion yield better cost-effectiveness ratio than test suites selected according to traditional coverage adequacy criterion?
$\mathbf{Q}_3$ [prioritization]: Given a specific testing context, do test suites prioritized according to relevant coverage adequacy criterion reveal faults faster than test suites selected according to traditional coverage adequacy criterion?

$$\mathbf{M}_1 = |\textit{test suite}| \tag{2}$$

$$\mathbf{M}_2 = \frac{\#\ \textit{of faults revealed}}{\#\ \textit{of test cases}} \tag{3}$$

$$\mathbf{M}_3 = APFD \tag{4}$$

Metric $\mathbf{M}_1$ computes the size of the test suites obtained when using both relevant coverage adequacy criterion and traditional coverage adequacy criterion, respectively. The cost-effectiveness metric $\mathbf{M}_2$ computes the proportion between the number of faults revealed and the size of the test suites respectively associated to each adequacy criterion. Finally, metric $\mathbf{M}_3$ measures how rapidly a prioritized test suite detects faults by computing a weighted Average of the Percentage of Faults Detected (APFD) [11]. APFD values range from 0 to 100 and higher values mean faster (better) fault detection rates.

B. Study Subject

In order to carry out our exploratory study and to investigate our research questions in a realistic setting, we looked for subjects in the Software-artifact Infrastructure Repository (SIR) [12]. For selecting our subject, some prerequisites had to be considered: first, the subject should be written in the C language; second, it should contain faults (either real faults or seeded ones) and a test suite associated with it; finally, because our approach considers the need to target specific testing contexts, we looked for a subject that would allow us to convincingly define different testing scenarios.

For this study we selected the gzip subject: a software application used for file compression and decompression. The gzip subject is available from SIR with 6 sequential versions (1 baseline version and 5 versions with seeded faults). Each variant version contains a different number of seeded faults ranging from 7 to 16. The test suite that comes with the subject

TABLE II: Test Suite Sizes for the Different Adequacy Criteria

Variant/ Scenario	Line						Branch						Function					
	Traditional		Relevant			Reduction Factor	Traditional		Relevant			Reduction Factor	Traditional		Relevant			Reduction Factor
	Suite Size	TCov	Suite Size	TCov	RCov		Suite Size	TCov	Suite Size	TCov	RCov		Suite Size	TCov	Suite Size	TCov	RCov	
V1/S1	21	74.7%	5	55.6%	95.4%	76.2%	25	59.1%	7	29.3%	92.5%	72.0%	11	87.7%	1	56.8%	97.7%	90.9%
V1/S2	21	74.7%	6	57.6%	86.1%	71.4%	25	59.1%	8	44.3%	80.8%	68.0%	11	87.7%	3	72.8%	93.9%	72.7%
V1/S3	21	74.7%	7	57.5%	94.6%	66.7%	25	59.1%	8	43.2%	90.5%	68.0%	11	87.7%	2	58.0%	97.7%	81.8%
V2/S1	21	65.6%	5	49.3%	95.6%	76.2%	26	52.5%	6	25.9%	93.1%	76.9%	11	73.5%	2	49.0%	97.7%	81.8%
V2/S2	21	65.6%	6	50.9%	87.9%	71.4%	26	52.5%	7	39.0%	82.2%	73.1%	11	73.5%	2	58.2%	93.8%	81.8%
V2/S3	21	65.6%	7	50.8%	94.8%	66.7%	26	52.5%	8	38.4%	91.2%	69.2%	11	73.5%	2	49.0%	97.8%	81.8%
V5/S1	21	67.1%	4	34.3%	100.0%	81.0%	24	52.6%	4	24.9%	99.0%	83.3%	12	75.3%	2	49.5%	100.0%	83.3%
V5/S2	21	67.1%	4	50.8%	88.5%	81.0%	24	52.6%	4	37.6%	82.7%	83.3%	12	75.3%	2	58.8%	91.2%	83.3%
V5/S3	21	67.1%	5	51.2%	94.6%	76.2%	24	52.6%	5	38.1%	91.8%	79.2%	12	75.3%	3	50.5%	95.7%	75.0%
Average:	**21**	**69.1%**	**5.44**	**50.9%**	**93.1%**	**74.1%**	**25**	**54.7%**	**6.33**	**35.6%**	**89.3%**	**74.8%**	**11.33**	**78.8%**	**2.11**	**55.8%**	**96.2%**	**81.4%**

contains 214 test cases, which overall can reveal only a subset of the seeded faults available on each version. We refer to this test suite as the *SIR test suite*.

C. Tasks and Procedures

We defined three realistic scenarios in which our subject could be reused: *scenario #1* (gzip is reused, in a bigger system, for compressing files only); *scenario #2* (gzip is used by an online service only for decompressing the files submitted by the service's users); and *scenario #3* (gzip is reused for compressing not only files but also whole directories recursively). These scenarios are used to provide the input domain constraints for our approach (step 1 of Section III-A).

In this study we considered three types of entity: line, branch, and function, which correspondingly identify three coverage criteria. Then, for each entity and for each variant version of gzip available we performed the following tasks:

1) Collected the input domain constraints from the testing scenarios previously defined (e.g., because the scenario #1 is related to file compression, one of the constraints collected is the fact that the parameter `-d`, required for *decompressing*, will never be used).
2) Instrumented the code with KLEE methods reflecting the input domain constraints to guide the DSE exploration (as illustrated in the Section III-B).
3) Used the output from the previous step to identify the set of in-scope entities (line, branch or function) for that specific testing scenario (in the same way as explained in the Section III-B).
4) Used a greedy algorithm for generating two test suites. The first test suite is derived aiming at achieving maximum coverage for a given adequacy criterion. We refer to it as *traditional test suite*. The second test suite is derived aiming at achieving maximum coverage over the set of in-scope entities only and we refer to it as *relevant test suite*.
5) Evaluated, for each possible combination of gzip variant, test scenario and adequacy criteria, the performance of the relevant test suite when compared to the traditional one using the metrics $\mathbf{M}_1$ to $\mathbf{M}_3$.

The greedy algorithm used for deriving the test suites in Task 4 is very simple: given a set of entities that should be covered (branches, for example), the algorithm selects, from the SIR test suite, the test case that covers the highest number of entities. When a test case is selected, the set of entities to be covered is updated to remove the entities covered by the chosen test case, and the algorithm keeps looking for the next text case with the highest coverage for the remaining entities until it is not possible to increase coverage anymore. In case of a tie, the algorithm selects the test case that covers the lowest number of out-of-scope entities (this step is applicable for the relevant test suite only); if the tie persists, then the algorithm orders the set of tied test cases alphabetically and picks the first one (these steps allow us to have a deterministic algorithm for deriving the test suites).

D. Execution

During our exploratory study we noticed that according to the original fault-matrix provided by SIR along with the gzip subject, none of the test cases available in the SIR test suite would reveal the faults in variant 3. Besides that, for variant 4 there were no faults that could be revealed in 2 out of the 3 testing scenarios we had defined. For being able to apply all the metrics defined for this study, we proceed with our study considering only variants 1, 2, and 5.

V. Analysis

A. Test Suite Size ($\boldsymbol{M}_1$)

Table II shows the test suite sizes as well as the structural coverage achieved by the traditional and the relevant test suites generated by the greedy algorithm as explained in IV-C. Note that for relevant coverage, information is provided in two different columns; the "TCov" column reports the coverage that would be achieved by the relevant test suite if it was calculated in the traditional way, whereas "RCov" reports the relevant coverage calculated according to Equation 1. The column "Reduction Factor" shows, for each coverage criterion, the reduction factor achieved by the relevant test suites when compared to the traditional ones.

As expected the relevant test suites are much smaller as they are derived aiming at achieving maximum coverage over the set of targeted entities only. The average number of test cases for the relevant test suite varied from 2.11 to 6.33 depending on the adequacy criterion chosen, while the average reduction achieved varied from 74.1% (for line coverage) to 81.4% (for function coverage). Considering all the possible combinations of variant, scenario, and adequacy criterion, the reduction factor was never smaller than 66.7%.

Not surprisingly, the absolute coverage of traditional test suites was bigger in all the cases varying from 52.5% to 87.7%. The structural coverage of the relevant test suite varied from 24.9% to 72.8% (when measured in the traditional way) and from 80.8% to 100% when measured in the relative (relevant) way. 100% relevant coverage has been achieved only by V5/S1 (for line and function coverage), which means that for the vast majority of the cases, in-scope entities for our testing scenarios had been left untested.

B. *Cost-effectiveness ($\mathbf{M}_2$)*

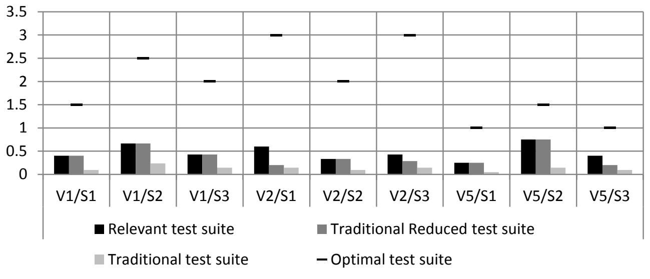
In the same manner that a specific test scenario would restrict coverage to the in-scope entities, there may be faults that are never triggered when the input domain is subject to some constraints (because they are located in out-of-scope entities). Hence to assess cost-effectiveness, we also needed a procedure to distinguish between relevant and not relevant faults within each test context. We proceeded as follows: first we analyzed the SIR test suite manually and filtered the test cases related to each given testing scenario (for the testing scenario #2, for example, we pick only the test cases related to decompression). Second, we used the SIR fault-matrix associated to that specific variant to identify the faults that are revealed by the selected test cases. The union of all the faults revealed by the selected test cases gives the set of all faults that could potentially be revealed in the testing scenario being evaluated. We refer to these faults as the *relevant faults*.

Then we evaluated the cost-effectiveness of the relevant test suite, on the one hand against the traditional one, and on the other hand against an *optimal* test suite, which constituted the baseline of the best possible performance for the given context. The optimal test suite was derived as follows: we look into the set of filtered test cases for the scenario at hand and select, each time, the test case that exposes the highest number of faults until all the relevant faults could be revealed.

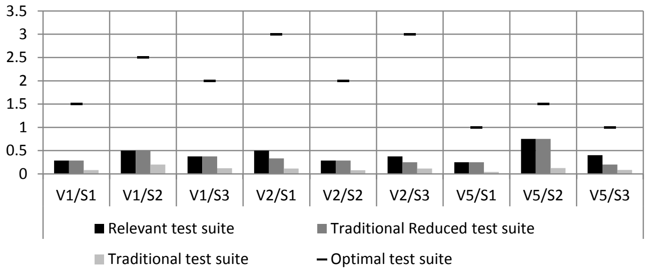
Figures 1a, 1b, and 1c show the bar graphs comparing the cost-effectiveness ratio achieved by the different test suites for line, branch, and function coverage, respectively. In the horizontal axis of each figure we have all the possible combinations of variant and testing scenario, and in the vertical axis we have the cost-effectiveness ratios. The measures related to the relevant test suite are displayed by the black bars, or the leftmost one of each group, while the measures for the traditional test suite are displayed in the light grey bars, or the rightmost one of each group (please ignore for the moment the central bar, whose meaning is explained later on). The cost-effectiveness values achieved by the optimal test suite are identified by the short horizontal lines. For the cost-effectiveness metric, the higher the bar, the better.

As expected, the optimal test suite achieved the highest cost-effectiveness ratios in all cases, except one in which the relevant test suite performed better (Fig. 1c, function coverage, V1/S1), and another case in which the result obtained by the optimal test suite tied with the relevant test suite (Fig. 1c, function coverage, V5/S2).

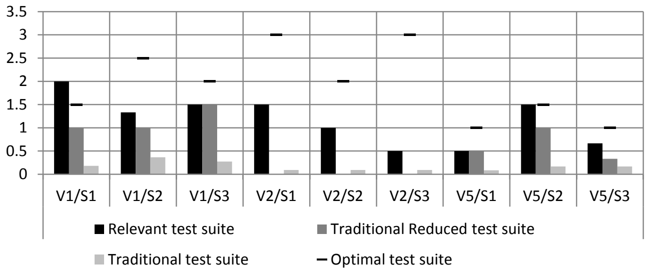
When compared against the traditional suite, the relevant test suite performed better in all the cases. However, we considered such comparison to be unfair because, as we saw in the results reported for the previous metric $\mathbf{M}_1$, the traditional



(a) **Line** adequacy criterion



(b) **Branch** adequacy criterion



(c) **Function** adequacy criterion

Fig. 1: Cost-effectiveness comparison

test suites were much bigger than the ones derived using the relevant adequacy criterion and, of course, this would impact negatively the cost-effectiveness ratio. To provide a fair comparison we reduced the traditional test suite down to the size of the relevant test suite (we took the first test cases selected by the greedy algorithm) and applied the metric $\mathbf{M}_2$ again over the newly created suite (we refer to it as *traditional-reduced* test suite). The measures related to the traditional-reduced test suite are displayed in the central bars of each group in the graphs.

Comparing the relevant test suite against the traditional-reduced one, either the test suite derived by the relevant adequacy criterion performed better than the traditional-reduced test suite or they were tied. In no case the traditional-reduced performed better than the relevant test suite.

C. *Prioritization ($\mathbf{M}_3$)*

Given a large test suite, test case prioritization aims at identifying an ordering of its test cases according to some goal, e.g. maximizing the rate of fault detection, and is more often used for regression test purposes (see, e.g., [13]). Although not originally conceived for prioritization purposes, as an afterthought we saw that relevant coverage could also be used as such. Hence in this study we also compared its rate of fault detection against that of the traditional-reduced test suites,

TABLE III: APFD Values Achieved by “Traditional-reduced” and “Relevant” Test Suites

Variant/ Scenario	**Line**				**Branch**				**Function**			
	# Faults Revealed		APFD Values		# Faults Revealed		APFD Values		# Faults Revealed		APFD Values	
	Traditional Reduced	Relevant	Traditional Reduced	Relevant	Traditional Reduced	Relevant	Traditional Reduced	Relevant	Traditional Reduced	Relevant	Traditional Reduced	Relevant
V1/S1	2	2	80.00	90.00	2	2	85.71	92.86	1	2	50.00	50.00
V1/S2	4	3	70.83	91.67	4	4	75.00	93.75	3	4	50.00	83.33
V1/S3	3	3	88.10	88.10	3	3	89.58	89.58	3	3	58.33	58.33
V2/S1	1	3	10.00	90.00	2	3	58.33	91.67	0	3	0	75.00
V2/S2	2	2	16.67	75.00	2	2	64.29	71.43	0	2	0	25.00
V2/S3	2	3	28.57	59.52	2	3	68.75	64.58	0	1	0	25.00
V5/S1	1	1	87.50	87.50	1	1	37.50	87.50	1	1	75.00	75.00
V5/S2	3	3	70.83	79.17	3	3	54.17	79.17	2	3	50.00	58.33
V5/S3	1	2	90.00	80.00	1	2	50.00	70.00	1	2	83.33	50.00
		Average:	**60.28**	**82.33**		**Average:**	**64.81**	**82.28**		**Average:**	**40.74**	**55.55**

because the latter by construction gave the same test suite sizes and, as shown in the previous section, in 14 out of 27 scenarios presented the same cost-effectiveness ratio.

Table III shows the APFD values for relevant and traditional-reduced test suites along with the number of relevant faults revealed by each test suite. Recall that for the APFD metric, higher values mean faster (better) fault detection rates. As shown, the relevant test suite outperformed the traditional-reduced one in 18 out of 27 cases and achieved the same APFD value in other 6 cases with an overall average APFD of 73.4 against 55.3 for the traditional-reduced. APFD values achieved by the relevant test suite were never smaller than 25 and they were never bigger than 93.75. For the traditional-reduced test suite, APFD values ranged from 10 to 90.

VI. Discussion and Threats to Validity

A. Evaluation of the results and their implications

1) Test Suite Size (M_1): The results show that our approach can help to considerably reduce the number of test cases of a given test suite if the target is to focus the testing on the areas of code that are relevant in a given context. Clearly, this cannot be an approach to advise for safety-critical systems, since we are saying a tester to reduce the test coverage only on some part of a program. Even though this should be safe, because we know that the uncovered parts will not be used in operation, with enough resources a more comprehensive testing would never hurt.

2) Cost-effectiveness (M_2): We see that, when considering only the relevant faults, our approach does not lose in terms of fault detection effectiveness. On average, a test case in traditional coverage detects 0.13 faults; 0.45 faults are revealed in traditional-reduced, whereas a test case in relevant coverage reveals 0.68. Looking into the results per adequacy criterion, for line coverage the number of faults detected by traditional, traditional-reduced, and relevant are 0.13, 0.39, and 0.47, respectively; for branch coverage the figures are (following the same order) 0.11, 0.36, and 0.41; and for function coverage the number of faults revealed are 0.17, 0.59, and 1.17.

In Section V-B we initially compared the cost-effectiveness of our test suite against the traditional one, and then against a reduced version of the traditional for the sake of providing a fair comparison. If we had not considered the number of test cases and had only looked at the absolute number of faults detected (which would favor traditional coverage because the test suites derived according to this adequacy criterion had on average, at least 4 times more test cases than our test suites), our approach would still outperform the traditional one with a total of 67 faults revealed among all the variant/scenario combinations, against 65 revealed by the traditional test suites. The results were tied in 22 cases out of 27; the traditional outperformed the relevant in two cases by revealing one extra fault on each case; the relevant, on its turn, outperformed the traditional in three cases: two times with one extra fault being revealed, and one time detecting two extra faults.

3) Prioritization (M_3): In our study the relevant test suite achieved better APFD values in the majority of the cases, being outperformed by the traditional-reduced test suite only in three cases. The results achieved seem promising, but as we said prioritization was not among the initial goals for relevant coverage, and so far we analyzed only one possible heuristic to derive the suites. More data is needed to investigate the research question associated with this metric.

4) Overall: The original motivation behind relevant coverage was to define a more meaningful notion of coverage for a given tester, assuming that part of the program under test is reusing components and code that were conceived for different usage contexts. In the course of the exploratory study, we realized that the approach seems indeed a novel promising technique for test case reduction and prioritization that adapts to a tester’s context. We intend to plan future studies both to investigate deeper the potential of relevant coverage in this respect, e.g. by comparing it against other reduction and prioritization techniques, and to assess the usefulness of test cases generated to augment coverage of in-scope entities.

B. Threats to Validity

1) Threats to internal validity: concern aspects of the study settings that could bias the observed results.

In-scope entities identification: As stated, we used KLEE for performing the symbolic execution of the subject’s code and identifying the set of in-scope entities. Some of the KLEE’s search heuristics for path-finding are random-based, which means that the order in which the paths are explored and, consequently, the order in which the test cases are generated, may change. Because we used concrete files (e.g.: real compressed files) and because the outputs of our study subject depend on the environment in which it is executed, we created a *sandbox* and executed KLEE 10 times, under the same environment conditions to make sure that all the possible test cases would be generated. The union of the unique test cases generated after the 10 runs were "replayed" in the original variant version to identify the set of in-scope entities.

SIR test suite coverage: The SIR test suite that accompanies the subject of this study does not achieve full coverage in any of the variants studied. Different results could had been achieved if full coverage was provided, as the level of coverage may affect effectiveness, see, e.g., [3]. One possible way of controlling this threat would be adding more test cases to achieve different levels of coverage up to full. We performed a cost-benefit analysis and decided to use the subject as it is provided (i.e., not to introduce other test cases) with all of its artifacts, since it represents the *golden standard* for benchmarking purposes.

Relevant faults identification: Because the SIR test suite does not achieve full coverage, the set of relevant faults we derived for computing cost-effectiveness may not contain all the faults that could be possibly revealed in a given testing scenario. However, since we use the same matrix associating test cases to faults, we do not see how such threat could produce different impacts on different test criteria in systematic way, and thus influence the results on cost-effectiveness.

Testing costs approximation: As a measure of testing costs, we took the number of test cases, which does not account for many factors affecting the actual cost of executing and analyzing those test cases. Although some factors are the same for traditional and relevant coverage, we do not have estimates of whether and to what extent the cost of the steps that are different, e.g. the identification of in-scope entities, may impact the cost-effectiveness ratio. To assess cost more realistically we should perform studies involving human subjects.

Prioritization quality evaluation: In this study we used APFD as the metric for evaluating the speed in which faults are revealed by a given test suite. However, APFD is not the only possible measure of rate of fault detection. Control for this threat can be achieved only by conducting additional studies using different metrics for evaluating the prioritization quality of the test suites used in our study.

2) Threats to external validity: concern aspects of the study that may impact the generalizability of results.

Subject representativeness: In this work we investigated the variants of a single subject. As explained in Section IV-B the study settings imposed a set of requirements on the subject that made it not easy to identify good candidates. Control for this threat can be achieved only by conducting additional studies.

Faults representativeness: The subject program chosen contained seeded faults in all of the variants investigated. Subjects with real faults might yield different results.

Heuristic representativeness: Both for relevant and for traditional coverage we adopted a greedy heuristic to derive an adequate test suite. Other heuristics and algorithms should be investigated before more general conclusions can be drawn.

3) Threats to construct validity: concern confounding aspects by which what we observed is not truly due to the supposed cause.

Experimental design: To fully evaluate our proposed approach the exploratory study should target all the steps described in Section III-A. However, as already discussed, we did not evaluate the approach in its capability to increase coverage. Also we did not actually execute the test cases and use an oracle for fault detection, but relied on the information in the SIR repository. Overall, our exploratory study may be quite distant from a real test context. However, we never draw absolute conclusions about the properties of relevant coverage *per se*, but always in comparison with traditional (or traditional-reduced) coverage. Since most of the above approximations seem to affect both approaches in similar way, the effects of such threats should not be relevant.

VII. Related Work

In this work we have introduced a novel family of structural coverage measures, the relevant coverage. The topic of software test adequacy criteria has been extensively investigated in software testing research. Since the very notion of a test criterion was formalized in the 70's [14], [15], a lot of contributions have been made on the definition of new coverage criteria [16] that are effective in failure detection, and considerable research effort has been devoted to the comparison of multiple criteria to provide support for the use of one criterion or another [17].

Relative Coverage

In [18], Bartolini et al. argue that traditional test coverage should be revised to deal with service-oriented systems, and introduce a notion of relative coverage in which the set of covered entities is measured against a customized set of entities that can vary from a user to another. Relative coverage has later been used also in [19]. In this work, the authors propose an approach in which testable services (services instrumented to provide their clients with coverage information) are provided along with test metadata to help their testers to get a higher coverage. In both [18] and [19] the list of relevant (in-scope) entities is manually defined by the user, whereas here we provide an approach for deriving them automatically.

We stated our intent to pursue the goal of providing an approach to calculate relevant coverage measure in [20]. Besides, we proposed in [21] the "social coverage" measure, in which the list of in-scope entities is derived in an automated way based on historical coverage data collected from similar users. Therefore the two criteria of social and relevant coverage consider very different ways to characterize entity relevance.

Conceptually our notion of relevant coverage is close to Rosemblum's notion of "adequate testing" introduced in [22] with regard to component-based software testing. In this work, the author was in fact concerned with the scenario in which a component needs to be tested when it is being used by a possibly larger system. However, that paper only provided a theoretical definition and did not provide an approach to assess "adequate testing".

Test Suite Reduction

Achieving adequate coverage may require a high number of test cases, and researchers have proposed several approaches for test suite reduction or minimization, e.g. [23], [24]. The problem of test suite reduction consists of identifying a minimal set of test cases that achieves a given coverage measure. Test suite reduction is often applied in regression testing.

As we hint in our exploratory study, relevant coverage can be seen as an approach for reducing the size of a test suite based on a completely different premise, i.e., we only consider the test cases covering the code entities that really matter, and discard test cases covering entities that are outside scope. Doing so, we do not maintain the same coverage measure, but we accept that the reduced test suite can achieve a lower coverage measure in absolute terms. We are not suggesting however to deliberately shrink testing effort. Our argument is rather opportunistic, we say that if only a limited number of test cases can be executed, than these should target those entities that will be executed in operation.

VIII. Conclusions and Future Work

We have introduced a novel approach for measuring and using coverage information, aimed at customizing traditional coverage testing adequacy criteria to a constrained testing context. We proposed the relevant coverage approach for computing in-scope entities (in particular lines, branches and functions) using dynamic symbolic execution constrained by those conditions that would apply in a specific testing scope. Such approach produces a more meaningful measure of coverage and can also generate test cases to cover yet uncovered in-scope entities. An exploratory study on the gzip benchmark from the SIR repository showed that relevant coverage can considerably reduce the test suite size while preserving a high cost-effectiveness ratio (measured over the *relevant* faults) with respect to a traditional approach that would address the whole set of entities. We found that relevant coverage could provide a novel approach to test suite reduction and prioritization.

However, this is a first study and many directions remain open for future work. On the one side we certainly need to perform further empirical studies, possibly also involving human testers so to fully explore relevant coverage costs and benefits. In such studies we would like to use other heuristics and algorithms for test case selection, as well as compare relevant coverage to most up-to-date reduction and prioritization approaches. On the other side, we would like to try other tools and approaches to compute in-scope entities. For example, Conditioned-Slicing [25] could provide a static and more comprehensive approach.

While in four decades of software testing literature tons of proposals can be found of new coverage criteria aimed at improving fault-finding effectiveness, the notion of relevant coverage introduced here, which in a nutshell proposes not to identify novel entities to be covered, but rather to change the denominator of the coverage ratio formula depending on the testing context, provides a completely new point of view. We believe that such a novel perspective paves the way to many new avenues for improving test cost-effectiveness, yet all to be explored.

References

[1] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing tools," in *Proc of the 2006 Int Workshop on Automation of Software Test*, ser. AST '06. New York, NY, USA: ACM, 2006, pp. 99–103.

[2] V. Garousi and T. Varma, "A replicated survey of software testing practices in the canadian province of alberta: What has changed from 2004 to 2009?" *Journal of Systems and Software*, vol. 83, no. 11, pp. 2251 – 2262, 2010.

[3] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proc. of the 36th Int. Conf. on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014.

[4] M. Morisio, M. Ezran, and C. Tully, "Success and failure factors in software reuse," *Software Engineering, IEEE Transactions on*, vol. 28, no. 4, pp. 340–357, 2002.

[5] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, pp. 1483–1498, Oct. 1988.

[6] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 2, pp. 158–191, Apr. 1998.

[7] C. S. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 4, pp. 339–353, Oct. 2009.

[8] T. Chen, X. song Zhang, S. ze Guo, H. yuan Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Gen. Comp. Systems*, vol. 29(7), pp. 1758 – 1773, 2013.

[9] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[10] V. R. Basili, G. Caldiera, and H. D. Rombach, "Goal Question Metric Paradigm," in *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994, vol. 1, pp. 528–532.

[11] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proc. IEEE Int. Conf. on Software Maintenance, 1999.(ICSM'99)*. IEEE, 1999, pp. 179–188.

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

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

[14] J. Goodenough and S. Gerhart, "Toward a theory of test data selection," *IEEE Trans. on Sw. Engineering*, vol. SE-1, no. 2, pp. 156–173, 1975.

[15] W. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. on Sw Eng.,*, vol. SE-2, no. 3, pp. 208–215, 1976.

[16] 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, 1997.

[17] N. Juristo, A. Moreno, and S. Vegas, "Reviewing 25 years of testing technique experiments," *Emp. Sw Eng.*, vol. 9, no. 1-2, pp. 7–44, 2004.

[18] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, "Whitening SOA testing," in *Proc. of the 7th joint European Sw. Eng. Conf. and the ACM SIGSOFT symposium on The foundations of Sw. Eng.*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 161–170.

[19] M. Eler, A. Bertolino, and P. Masiero, "More testable service compositions by test metadata," in *Service Oriented System Engineering (SOSE), 2011 IEEE 6th International Symposium on*, 2011, pp. 204–213.

[20] B. Miranda, "A proposal for revisiting coverage testing metrics," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 899–902.

[21] B. Miranda and A. Bertolino, "Social coverage for customized test adequacy and selection criteria," in *Proceedings of the 9th International Workshop on Automation of Software Test*, ser. AST 2014, 2014, pp. 22–28.

[22] D. S. Rosenblum, "Adequate testing of component-based software," Un. of California, Irvine, CA, Tech. Rep. UCI-ICS-97-34, 1997.

[23] J. Jones and M. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," in *Proc. IEEE Int. Conf. on Sw Maintenance*, 2001, pp. 92–101.

[24] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," in *Fundamental Approaches to Sw Eng.*, ser. LNCS, M. Dwyer and A. Lopes, Eds. Springer, 2007, vol. 4422, pp. 291–305.

[25] G. Canfora, A. Cimitile, and A. De Lucia, "Conditioned program slicing," *Inf. and Sw Technology*, vol. 40(11), pp. 595–607, 1998.