Checked Coverage for Test Suite Reduction – Is It Worth the Effort?

Roxane Koitz-Hristov*
Lukas Stracke
Franz Wotawa
rkoitz@ist.tugraz.at
l.stracke@student.tugraz.at
wotawa@ist.tugraz.at
Graz University of Technology, Institute for Software Technology
Graz, Austria

ABSTRACT

As the size of software projects increases, their test suites usually grow accordingly. Test suite size, however, has a direct impact on the efficiency of software testing. Hence, test suite reduction (TSR) procedures aim at removing redundant test cases while maintaining the suites fault detection capabilities (FDC). This paper explores *checked coverage* as a coverage metric for TSR; checked coverage not only investigates if a part of code was executed but also if it was checked by a test oracle. Previously, this metric has been applied successfully as an indicator for oracle quality. To assess how suitable checked coverage is in comparison to traditional metrics, such as line or method coverage, we developed a TSR tool for Java programs. In an empirical evaluation, we performed TSR based on different reduction algorithms, coverage metrics, and open-source Java projects. Our study investigates both the efficiency of the TSR as well as effectiveness in regard to the FDC and size of the reduced test suites.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

test suite reduction, checked coverage, experimental evaluation, software testing

ACM Reference Format:
Roxane Koitz-Hristov, Lukas Stracke, and Franz Wotawa. 2022. Checked Coverage for Test Suite Reduction – Is It Worth the Effort?. In *IEEE/ACM 3rd International Conference on Automation of Software Test (AST '22), May 17–18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524481.3527216

*Authors are listed in alphabetical order.

1 INTRODUCTION

For keeping software in use, it has to be evolved over time. This is due to the necessity to cope with changes of requirements, its runtime environment comprising other software and hardware, and other causes arising during operation. One consequence is that usually the test suite (TS) used for verifying and validating the software also adapts and increases, therefore, requiring optimization for keeping its execution within predefined resource boundaries.

Test Suite Reduction (TSR) is such an optimization. TSR focuses on eliminating TCs such that the resulting TS still has similar fault detection capabilities than the original TS. The test suite fault detection capability (FDC) is usually measured using code coverage obtained when executing the TS on the software under test (SUT). The underlying assumption behind using coverage is that executing more source code increases the likelihood of revealing a fault. There are many different coverage metrics such as statement coverage, i.e., the number of statements executed by a TS, or branch coverage, i.e., the number of branches in the program executed by a TS.

It is worth noting, that a fault is revealed, during the execution of a TC if the resulting behavior of the SUT is not the expected one. The classification of the behavior as being expected or not is usually carried out using a test oracle. In its simplest form a test oracle compares the output values obtained when executing a TC, with the expected values for the same TC. In case of a deviation, the TC is called a failing test case and, otherwise, a passing one [2].

In this paper[1], we contribute to the research field of TSR. In particular, we introduce a new tool *The Java Test Suite Reduction Framework (JSR)* allowing to compute minimized test suites considering different code coverage metrics ranging from method coverage, over to line coverage, to the more recently introduced checked coverage. The objective behind checked coverage is to only consider statements that are executed and contribute to the evaluation of a condition used by a test oracle. For computing those statements, checked coverage relies on dynamic slicing on all variables used in the test oracle condition. In addition, we also performed an experimental evaluation comparing the three implemented test suite minimization algorithms (i.e., Greedy HGS, Delayed Greedy and a genetic algorithm), considering the three different code coverage metrics and the time required for TSR. Our evaluation is based on ten Java projects from available repositories. The tool, which can

[1]Portions of the paper have been previously published as part of a Master thesis of one of the authors [22].

be extended, and all results are available for other researchers and also practitioners for further evaluation and study.

We organize this paper as follows: In Section 2, we discuss the underlying preliminaries. Afterwards in Section 3, we introduce the JSR tool, its architecture, implemented coverage metrics and algorithms. In Section 4, we outline the evaluation in detail, and in Section 5, we discuss the obtained results. Finally, we give an overview of related work (Section 6), and summarize the paper (Section 7).

2 PRELIMINARIES

Within this section, we formally define TSR and briefly describe some common TSR approaches. Afterwards, we explain checked coverage and give a toy example.

2.1 Test Suite Reduction

With the increasing importance of regression testing, various techniques for improving and optimizing regression test suites have been explored by the scientific community. TSR is one of those techniques.

Definition 1. ***Test Suite Reduction [28]:*** *Given a test suite T consisting of test cases $\{t_1, ..., t_n\}$, a set of test requirements $R = \{R_1, ..., R_m\}$ for T, and subsets $\{T_1, .., T_m\}$ of T, where each T_i is associated with an R_i such that every $t_j \in T_i$ satisfies R_i.*
Find a reduced test suite T' containing test cases from T such that each R_i is satisfied by at least one test case and the number of test cases in the reduced test suite is minimal.

In other words, the reduced test suite (RTS) must contain as few TCs as possible while still satisfying all requirements that the original TS satisfied. The important prerequisite for TSR is that we need to establish the relation between fulfilled requirements and TCs so that we know which TC satisfies which requirement. There are several code coverage metrics as TSR requirements such as line coverage, method coverage or—as we propose in this paper—checked coverage. A TC t_i thus satisfies a requirement r_j iff r_j was reached (i.e., covered) during the execution of t_i.

The TSR research community has been actively proposing new ideas, algorithms, and methodologies on how to identify and remove redundant or obsolete TCs from a TS. TSR approaches can be classified into (1) search-, (2) clustering-based, and (3) greedy approaches, with many of them relying on different heuristics to approximate optimal solutions [14, 28].

(1) In search-based reduction approaches, a set of potential solution candidates is evaluated by a fitness function, assessing the quality of all candidates. Most techniques focus on finding a global optimal solution that satisfies either one or multiple objectives [11, 14].

(2) Clustering-based approaches try to group the TCs of a TS by similarity via various clustering algorithms. The clusters are subsequently sampled via a sampling algorithm, meaning that TCs are selected from the individual clusters, forming the RTS [4, 5].

(3) Greedy algorithms greedily select relevant TCs while iterating through the set of TCs based on a selection heuristic. The basic greedy heuristic is to select the TC covering the most requirements [23]. More sophisticated heuristics that perform better exist [10, 18].

Some approaches use combinations of the three mentioned classes, creating hybrid reduction algorithms. Addtionally, TSR approaches *adequate* or *inadequate*. Inadequate methods produce a RTS that does not cover all requirements anymore that were initially covered by the original TS, while adequate ones guarantee that all original requirements are still covered in the RTS. It should further be noted that finding the optimal TSR solution is known to be an NP-hard problem as it is equivalent to the *minimum set cover* problem [10, 14].

2.2 Checked Coverage

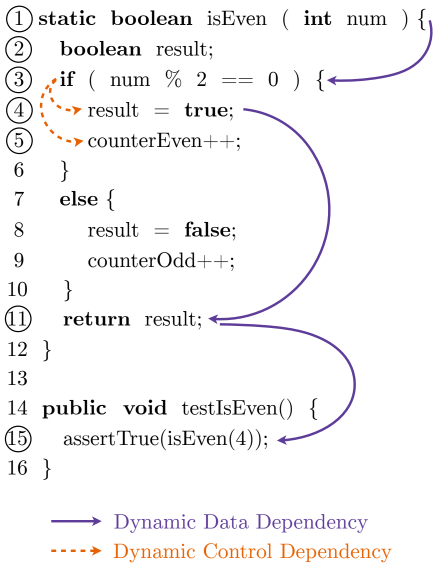
Coverage is a collective term for various techniques to track which parts of a system were actually reached (i.e., covered) during execution. Coverage metrics are measured on different granularities, ranging from coarse-grained *module* or *class* or *method* coverage to fine-grained *line* or *statement* coverage [27]. An interesting alternative to conventional coverage metrics is checked coverage [20]: Checked coverage is a code coverage metric that tries to measure TS effectiveness by investigating the test oracle quality of a TS. In conventional unit testing, an oracle is usually specified as a set of assertions that check the behavior and results of the SUT against pre-defined values or specifications. Hence, the general idea of checked coverage is to determine, which lines of code were actually checked by these assertion statements and thus influence the oracle decision (i.e., the outcome of the assertion statements) [20].

To compute the check-covered lines, an execution trace (ET), which is a chronologically ordered list of executed statements, is created during TS execution. Based on this ET the dynamic backward slice for the assertion statement is computed. Dynamic backward slicing is generally used to identify statements that had an influence on the contents of a given slicing criterion (i.e., a set of lines and variables of interest) during a specific execution. To establish this influence two types of dependencies are used: dynamic control and dynamic data dependencies [15].

Definition 2. ***Dynamic Control Dependency [20]:*** *A statement s is dynamically control-dependent on a statement t for an execution e, if s is executed during the program execution and t is a conditional statement that controls the execution of s.*

Definition 3. ***Dynamic Data Dependency [20]:*** *A statement s is dynamically data-dependent on a statement t for an execution e, if there is a variable v that is defined (written) in t and referred to (read) in s during the program execution without an intermediate redefinition of v.*

Example 1. *In Figure 1, we see a simple code example of a method that determines whether an integer is even or odd as well as a test for this method. The ET would contain all statements that are executed given the test* `testIsEven()`*. The line numbers that are circled identify the lines that are part of the ET. We can further observe the two different types of dependencies. On the one hand, there are two dynamic control dependencies from Statement 3 to Statement 4 and to Statement 5, respectively, indicating that the execution of Statement 4 and 5 depend on the condition in Statement 3. On the other hand, we can see three dynamic data dependencies; e.g., from Statement 4 to Statement 11 as the value of the variable* `result` *is assigned (written) in Statement 4 and used (read) in Statement 11.*



```
1  static boolean isEven ( int num ) {
2    boolean result;
3    if ( num % 2 == 0 ) {
4      result = true;
5      counterEven++;
6    }
7    else {
8      result = false;
9      counterOdd++;
10   }
11   return result;
12 }
13
14 public void testIsEven() {
15   assertTrue(isEven(4));
16 }
```

Figure 1: Dynamic data and control dependencies for a method and its test. Circled line numbers indicate that a line is part of the ET given the execution of `testIsEven()`.

Definition 4. ***Dynamic Backward Slice [20]:*** *A dynamic backward slice for a dynamic slicing criterion is the transitive closure over all dynamic data and control dependencies for the slicing criterion within the execution e.*

Given Definition 4, a dynamic backward slice can be easily computed by starting at the slicing criterion and following all dependencies in a backward manner. Every statement visited is added to the slice. By computing the dynamic backward slice on the assertion, we essentially collect all assertion-influencing statements. The checked coverage then is the ratio of statements within this slice relative to all coverable statements [20].

Example 1 (continued). *In our example, the slicing criterion contains the assertion* `assertTrue` *in Statement* 15. *By following the dependencies backwards, we obtain the final slice* $\{1, 3, 4, 11, 15\}$. *The coverable statements are* $\{1, 2, 3, 4, 5, 8, 9, 11\}$[2]. *Yet, only the statements that are within the slice are considered to be covered, i.e.* $\{1, 3, 4, 11\}$[3]. *Statements* 8 *and* 9 *are not executed and Statements* 2 *and* 5 *are executed but not checked. In this example, the checked coverage score would be* 50% *as* 4 *of* 8 *coverable statements are actually checked by our TC.*

3 JSR - THE JAVA TEST SUITE REDUCTION FRAMEWORK

Despite the considerable amount of published TSR research, the adaption of the proposed techniques in industry has only seen little success. In an attempt to bridge this gap between research and practice, we developed our TSR tool *JSR - The Java Test Suite Reduction Framework*[4]. JSR should serve as a platform for integrating TSR in a practical software development workflow, while also allowing to advance research in TSR techniques without having to create the framework around the item of research (e.g., a new reduction algorithm). With these intentions in mind we created a standalone, open source tool that can be easily maintained, improved and extended in the future.

3.1 Design and Architecture

Our tool currently consists of three modules, the *JSR Core* module as well as two frontend modules, the *JSR IDE Plugin* and the *JSR CLI* (see Figure 2). The main idea behind this modular architecture is that the core module holds the necessary backend logic for TSR while additional frontend modules can use it similarly to a library. Using this approach, we can extend JSR easily in the future by, e.g., implementing different reduction algorithms or coverage metrics or by adding a new frontend module. In terms of software design, we used various well-known and established design patterns introduced by Gamma et al. [8] to create a well encapsulated, maintainable and extendable framework.

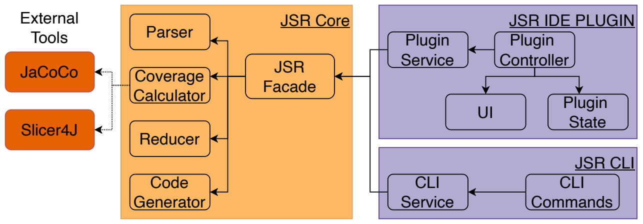


Figure 2: A high-level overview of the JSR architecture.

JSR currently features a command line interface (CLI) as well as a GUI implemented as a Jetbrains IntelliJ plugin. Further, the tool can compute three different coverage metrics (checked coverage, line coverage, and method coverage) and three reduction procedures (Greedy HGS, Delayed Greedy, and a simple genetic algorithm).

JSR handles JUnit 4[5] testing infrastructures of Java projects up to Java 11[6]. Given that we developed JSR in Java and we only use external tools written in Java, our tool is platform-independent, thus running under every JVM-supporting operating system. The following list provides an overview of relevant used tools and libraries[7]:

- `Slicer4J`[8] by Ahmed et al. [1] to perform `jar` instrumentation, execution tracing and dynamic slicing when calculating checked coverage (JSR-Core).
- `JaCoCo`[9] to calculate line and method coverage (JSR-Core).
- `Jenetics`[10] as a GA library (JSR-Core).

[2]Schuler und Zeller [20] consider only the "sliceable" statements instead of coverable statements. A sliceable statement is any statement that reads or writes a variable or manipulates the control flow and thus could become part of a slice. I.e., Statement 2 in our example would not be a "sliceable" statement. However, as we compare checked coverage to other coverage metrics, we opt to compute the ratio on the basis of all covered statements similar to line and method coverage.

[3]Statement 15 is part of the slice, but not within the method; hence, it is not considered.

[4]https://github.com/Lms24/JSR

[5]https://junit.org/junit4/

[6]The limitations on the test framework and Java side are based on the current version of the slicing tool.

[7]Our `github` page contains a complete list of libraries.

[8]https://github.com/resess/Slicer4J

[9]https://www.eclemma.org/jacoco/

[10]https://jenetics.io/

- PIT[11] as a mutation testing tool to assess FDC of the RTSs and original TSs during our benchmark.

3.2 Coverage Metrics and Algorithms

JSR can compute three different coverage metrics, i.e., checked coverage, line coverage and method coverage. Our most coarse-grained coverage metric is *method coverage*. On TS execution, methods of the SUT are marked as covered, if at least one statement inside the respective method was executed. A more fine-grained metric is called *line coverage*, marking lines of the SUT covered, if they were executed during TS execution [27]. These two types of metrics were chosen as they are widely used in TSR research[12]. Furthermore, we use checked coverage as a third coverage metric (see Section 2.2).

As our initial set of reduction procedures, we implemented three approaches: the Greedy HGS reduction algorithm, the Delayed Greedy algorithm, and a simple genetic algorithm. Greedy HGS represents our baseline, while the Delayed Greedy approach is a more effective variant of a greedy procedure. We included the genetic algorithm to also feature an *inadequate* TSR procedure.

3.2.1 Greedy HGS. Harrold et al. [10] proposed to favor TCs covering requirements that are hard to fulfill, i.e., requirements having a low number of TCs that satisfy them. By selecting such requirements, many easy-to-fulfill requirements would also be covered. Iteratively, the hardest-to-fulfill requirements are identified, forming a temporary set of TCs covering them. Next, they evaluate for each TC in this set how many other hard-to-fulfill requirements are covered by it. The TC covering most hard-to-fulfill requirements is selected. Each time a TC is selected, the TC is marked as well as all requirements covered by the TC, since they are now satisfied by the RTS. The algorithm terminates, once all requirements are covered by the RTS and the set of selected tests composes the RTS. The HGS algorithm does not always find an optimal solution. However, studies show that it yields overall good results that are mostly close to the optimal solution in terms of RTS size [28]. Hence, the Greedy HGS algorithm is our base line for our evaluation in Section 4.

3.2.2 Delayed Greedy. The algorithm proposed by Tallam et al. [23] is called "Delayed Greedy" algorithm, as it *delays* the application of a greedy selection heuristic by first applying reductions if possible. The aforementioned reductions can be used to systematically eliminate TCs and requirements that are not necessary for TSR, thus decreasing the TS representation size as well as the selection space when a greedy selection is necessary.

It should be noted that as long as only reductions are used, the resulting RTS is guaranteed to be of minimal cardinality. Once the algorithm resorts to the greedy heuristic, the outcome can however not be guaranteed to be optimal anymore, due to the early selection nature of greedy algorithms. In most cases, the delayed greedy algorithm needs more steps and iterations than the HGS algorithm; yet, according to Tallam et al. [23], the sacrifice in time is compensated by the algorithm's ability to find an optimal solution far more often than other greedy approaches.

[11]https://pitest.org/

[12]Other commonly used metrics, such as *branch coverage*, can be easily integrated into JSR.

3.2.3 Genetic Algorithm. Genetic algorithms (GAs) belong to the class of search-based algorithms and were first explored in the field of TSR by Ma et al. [16]. The general idea behind GAs is to apply phenomena found in nature, biology and genetics, such as mutation, selection, survival, crossover and randomness, to problem-solving approaches in computer science. The initial solution candidates form—in the terms of GAs—an *initial population*. Each member of the population (a *gene*) is evaluated w.r.t their quality via a *fitness function*, returning a comparable measure of how well the member solves the given problem [9]. GAs try to improve the solution candidates iteratively; each iteration produces a new *generation* of the population where the individual genes' fitness score is calculated and depending on the chosen operations, the *offspring* (i.e., the next generation) is produced.

We implemented a simplified genetic algorithm inspired by the work of Coviello et al. [7]. Our initial population is made up of genes representing different combinations of selected TCs. A gene is a bit vector, with the length of the original TS size. Upon creation of a gene for the initial population, we assign a 1 to the corresponding bit, if the TC is part of the proposed solution and a 0 otherwise. To control the size of the population, we selected an initial population size of $N * 5$ with N denoting the number of TCs in the original TS. The decision whether a TC should be part of the solution candidate, is governed by the probability $P_{initSelect}(t) = 0.97$.

The crucial part of a GA is the fitness function as this is the point where we specify what a *good* solution is and how we evaluate candidates. Our fitness function for a solution candidate (gene) g is defined as

$$\text{fitness}(g) = \begin{cases} -1 & \text{if } g \text{ does not cover all} \\ & \text{TSR requirements} \\ max(u - \frac{m}{M} * \alpha, 1) & \text{otherwise} \end{cases} \tag{1}$$

where u denotes the number of uniquely covered requirements, that is the number of requirements covered by only one TC in g. Furthermore, $\frac{m}{M}$ represents the average frequency of multiply covered requirements, where m is the sum of requirement duplications in g, and M denotes the number of requirements covered by more than one TC. The idea is to reward good essential TCs in g and to punish redundancy via subtraction. In addition, α is used as a duplicate punishment or relaxation factor to control the impact of the redundancy punishment on the final fitness score. During a pre-evaluation, we found that a punishment factor of $\alpha = 10$ encouraged the best results. Ultimately, we employed a roulette wheel selection strategy to preserve good candidates for the next generation with single-bit mutation ($P_{mut}(g) = 0.4$) and single-point crossover ($P_{cross}(g_1, g_2) = 0.15$).

4 EMPIRICAL EVALUATION

In this section, we present our empirical evaluation, which investigates the following research questions:

- **RQ1:** Is checked coverage a suitable TSR requirement criterion, and how does it compare to other coverage metrics for TSR?
- **RQ2:** Which combinations of coverage metrics and reduction algorithms perform best w.r.t. efficiency (i.e., runtime) and effectiveness (i.e., reduction in TS size and FDC)?

All experiments were performed using our JSR tool as described in Section 3 on a Dell XPS 15 7590 (2019) with a 4.50GHz Intel Core i7-9750H and 16GB RAM running ArchLinux (last update on October 4, 2021).

4.1 Benchmark Projects

One important aspect of our benchmark was to test JSR under real-world conditions as we wanted to assess how it performs in actual software development workflows. Therefore, we evaluated JSR with ten widely used, open-source software projects. All projects used in our benchmark are currently publicly available and several of them are frequently used in software testing [13] and specifically in other TSR-related research publications [3, 17, 29]. Table 1 lists all ten projects and their basic properties.

Table 1: Basic properties of all projects used in the benchmarks, showing the project version we used as well as properties of its source code and TS. The project size is measured in lines of code for source (LoC$_{Src}$), TS (LoC$_{TS}$) code and number of TCs (#*TCs*). In addition, the table contains the average (Avg.) of the size measurements.

ID	Project	Version	LoC$_{Src}$	LoC$_{TS}$	#TCs
1	`commons-lang`	3.4 (2015)	68685	55477	2622
2	`commons-io`	2.6 (2017)	28691	26493	1067
3	`opencsv`	3.3 (2016)	7371	9819	286
4	`commons-cli`	1.4 (2017)	6268	5393	318
5	`commons-csv`	1.6 (2019)	4793	6086	311
6	`minimal-json`	0.9.6 (2016)	3978	4423	457
7	`java-tuple`	1.2.1 (2020)	3389	5696	793
8	`json-simple`	1.1.1 (2021)	2505	1048	26
9	`confucius`	1.3 (2021)	1658	956	90
10	`ascii-table`	1.2 (2021)	410	551	21
Avg.			12775	11594	599

4.2 Independent and Dependent Variables

Our focus lies on identifying trends based on the TSR results of all coverage metrics and TSR algorithms combinations on open-source projects. We found that some real-world projects contain *zero-coverage* TCs, i.e., TCs that do not have any coverage information. While there are rarely any zero-coverage TCs when using line and method coverage, they often occur under checked coverage: In case a TC does not include any assert statements, its checked coverage record is empty. This occurs frequently in JUnit TCs, when the TC expects an exception to be thrown. To check this behavior, JUnit offers a test oracle via an annotation instead of an assert statement. Furthermore, we observed that some projects used in our benchmark occasionally included TCs without any oracle, either for reasons unknown to us or because the TC was (ab)used to benchmark SUT performance without additional correctness checks. Hence, we performed TSR with all combinations of test requirements (i.e., line, method, and checked coverage) and TSR algorithms (i.e., Greedy HGS, Delayed Greedy, and a genetic algotihm) with and without the inclusion of zero-coverage TCs in the RTSs. Making the *coverage metric*, the *TSR approach* and the *handling of zero-coverage TCs* our independent variables.

We exploit three evaluation metrics, i.e., dependent variables: *runtime* as an efficiency measure as well as the *reduction factor F* and the *mutation score* in regard to the effectiveness. The reduction factor F is the ratio of TCs in the RTS in comparison the amount of TCs in the original TS. To evaluate the FDC of each project's original TS and the RTS we compute the mutation score. The mutation score is the number of killed mutants relative to the number of created mutants in mutation testing and the gold standard of TS evaluation; the higher the mutation score, the higher the FDC of a given TS [12].

4.3 Threads to Validity

This section presents potential internal[13], external[14], and construct[15] threats to the validity of the experiments [24]. First, our choice in test subjects, i.e., Java projects, can pose as a validity thread. We try to mitigate this by relying on ten real-world Java projects—portions of which have been previously used in software testing [13] and TSR-related research [3, 17, 29]. Although our subjects may not be completely representative of all other possible projects, they are diverse and extensively used in practice. Still, in future research evaluating JSR and its methods and metrics on different datasets, such as the *Software-artifact Infrastructure Repository* [16], would be of interest.

Second, the implementations of the TSR approaches and coverage metrics threaten the internal validity. Hence, we implemented typical TSR approaches in their simplest form all in Java as well as using well-tested tools whenever possible. In addition, we performed all experiments on the same machine with identical evaluation procedures.

Third, the measures to determine the efficiency and effectiveness of the approaches and coverage metrics pose a thread to the construct validity. Yet, runtime is a simple efficiency indicator that can be applied to compare the different TSR algorithms. Additionally, the combination of the TS reduction ratio (i.e., the reduction factor F) and the FDC (i.e., the mutation score) is common in TSR research [14].

5 EVALUATION RESULTS AND DISCUSSION

In this section, we present and discuss the results of our benchmark. The first portion is dedicated to the initial results and properties from the projects' original TSs. In the second portion, we explain in depth the results from the reduction approaches based on the different test requirements and finally we provide answers to our research questions.

5.1 Coverage Results

Figure 3 shows that the vast majority of the projects have very high line and method coverage scores, whereas the lower checked coverage scores have a higher variation and a more diverse distribution. The coverage score of our main metric of interest—checked

[13]Threats to internal validity are factors that can unintentionally affect the dependent variables.
[14]Threats to external validity limit the ability to generalize experimental results.
[15]Construct validity concerns itself whether the measures taken adequately capture the concepts they are supposed to measure.
[16]https://sir.csc.ncsu.edu/

coverage—ranges from 39% to 81%, with most projects having a score between 50% and 60%. In contrast, the line and method coverage scores are significantly higher, with most projects having scores roughly between 85% and 95%.

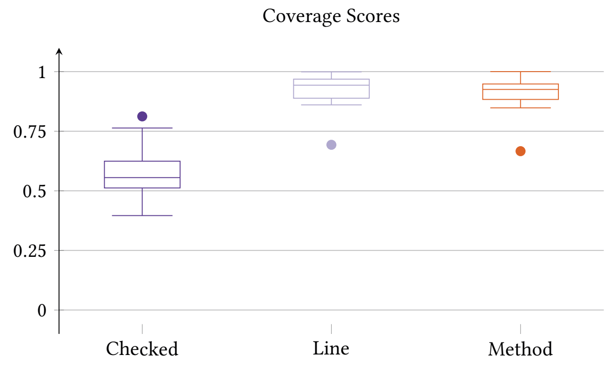


Figure 3: Boxplot diagram of the distribution of the coverage (checked, line, and method) scores of all projects.

The considerably lower checked coverage scores (w.r.t. the other coverage metrics) were to be expected as one has to consider that the set of lines marked as checked-covered is always a subset of the set of lines marked as line-covered. Considering the lack of correlation between traditional coverage metrics and FDC, it seems reasonable that not every executed line from our real-world projects is also checked. An overall interesting observation is that our checked coverage scores are mostly in the range of the results of previous research on checked coverage [21, 29].

Table 2 lists the runtimes of the coverage computations per project. We can observe that the computation times were very long, especially for checked coverage that took almost two days to complete for the biggest project (Project 1). The reasons are mainly due to the high computational effort involved with dynamic slicing. This is by far the biggest contributing factor to the long computation time.

5.2 TSR Results

We performed 18 TSR runs per project as we tested every combination of coverage metric, TSR algorithm and in-/exclusion of zero-coverage TCs. Subsequently, we evaluated the resulting RTSs via the mutation score to measure and assess the change in FDC and to determine which combination of TSR parameters produce the best results.

5.2.1 Greedy Approaches. What we can observe in Table 3 is that the reduction based on the two greedy approaches produced RTSs containing mostly between 50% and 70% of the original suite's TCs. Based on the results from this table, TSR with method coverage led to the smallest RTSs (average reduction factor of 77%), followed by checked coverage (66%) and the biggest RTSs were obtained when using line coverage as a TSR requirement (58%). This was to be expected, as the algorithms are from the same class and they both rely on similar heuristics. However, we can still observe that the delayed approach generates on average a slightly larger reduction, suggesting that the selection space reductions, which are additionally performed by the delayed greedy algorithm, do in fact contribute to a more optimal solution.

Table 2: Coverage computation times of all projects' original TSs with average (Avg.), median (Med.) and standard deviation (Std.).

ID	Checked Coverage Time$_{chk}$	Line Coverage Time$_{ln}$	Method Coverage Time$_{mth}$
1	47h 38m 52s	42m 34s	43m 35s
2	17h 24m 25s	12m 14s	11m 06s
3	9h 24m 25s	2m 29s	2m 26s
4	48m 56s	2m 47s	2m 48s
5	2h 29m 02s	3m 22s	3m 19s
6	2h 08m 29s	4m 03s	4m 12s
7	1h 34m 42s	6m 43s	7m 25s
8	59m 59s	13s	13s
9	9m 30s	48s	80s
10	3m 51s	10s	10s
Avg.	8h 16m 12s	7m 32s	7m 39s
Med.	1h 51m 35s	3m 4s	3m 3s
Std.	14h 52m 48s	12m 49s	13m 3s

When comparing the computation times of the two algorithms in Table 3, one can see that the slight decrease in RTS size found in the delayed greedy algorithm's RTSs comes with a significant increase in runtime up to two orders of magnitude. A reason for this increase in runtime are the additional reduction steps in which the selection space is reduced but no TC is selected. We could also observe multiple delayed greedy runs in which a greedy selection was never used, meaning that the reduced TC set is indeed the optimal solution.

5.2.2 Genetic Algorithm. In Table 4, we can see significantly smaller reduction factors and thus less reduced suites compared to the greedy algorithms. Depending on the coverage metric used as a requirement criterion, the original TSs were reduced by around 15% to 20%. Furthermore, the bigger original TSs (Projects 1, 2 and 7) led to inadequate solutions meaning that the RTSs did not satisfy all initial requirements anymore. A possible explanation of the inadequacy is the large search space (for Project 1 for example 2^{2622} possible combinations) which makes it likely that most (or all) solution candidates were inadequate from the beginning despite our high initial TC selection probability. In addition, potentially still adequate temporary solutions could have been destroyed by our mutation and crossover operators.

In addition to the inadequate RTSs, we can also observe notably longer execution times compared to the greedy TSR runs, which means that our GA did not only produce less reduced suites but it also took longer to come to this solution. GAs are more efficient when the solution candidates can be evaluated and altered in parallel. The degree of parallelism, however, was quite limited in our testing environment that significantly decreased the algorithm's performance. However, if TSR should be run, for example, on the local computers of software engineers, they will encounter the same problem when using GAs. In addition, most runtime was spent

Table 3: RTS results of TSR with the Greedy HGS and Delayed Greedy algorithm of all coverage metrics and without zero-coverage TCs. The table shows the reduction factor (F) and the reduction runtime (T) in seconds.

	Greedy HGS						**Delayed Greedy**					
	Checked Coverage		Line Coverage		Method Coverage		Checked Coverage		Line Coverage		Method Coverage	
ID	F_{chk}	T_{chk}	F_{ln}	T_{ln}	F_{mth}	T_{mth}	F_{chk}	T_{chk}	F_{ln}	T_{ln}	F_{mth}	T_{mth}
1	0.563	5.671s	0.486	12.999s	0.638	3.171s	0.564	770.81s	0.486	1640.27s	0.640	713.42s
2	0.746	3.423s	0.593	2.592s	0.732	0.790s	0.746	97.27s	0.597	466.20s	0.738	66.28s
3	0.790	0.137s	0.682	0.368s	0.839	0.127s	0.790	1.83s	0.682	7.21s	0.846	0.99s
4	0.827	0.234s	0.739	0.485s	0.915	0.183s	0.833	1.41s	0.748	5.38s	0.918	0.99s
5	0.894	0.178s	0.768	0.373s	0.907	0.163s	0.894	0.78s	0.772	5.83s	0.910	0.50s
6	0.700	0.208s	0.685	0.403s	0.816	0.158s	0.705	3.21s	0.689	7.06s	0.821	1.68s
7	0.576	0.439s	0.444	0.512s	0.464	0.272s	0.576	22.25s	0.444	60.54s	0.464	70.15s
8	0.192	0.130s	0.192	0.118s	0.538	0.042s	0.192	0.26s	0.192	0.30s	0.538	0.10s
9	0.689	0.057s	0.611	0.073s	0.711	0.051s	0.689	0.24s	0.611	0.23s	0.711	0.13s
10	0.619	0.134s	0.571	0.049s	0.810	0.041s	0.619	0.10s	0.571	0.06s	0.810	0.04s
Avg.	0.660	1.061s	0.577	1.797s	0.737	0.5s	0.661	89.816s	0.579	219.308s	0.74	85.428s
Med.	0.695	0.193s	0.602	0.388s	0.771	0.161s	0.697	1.62s	0.604	6.445s	0.774	0.99s
Std.	0.196	1.915s	0.17	4.005s	0.152	0.964s	0.197	241.167s	0.172	519.69s	0.153	222.438s

Table 4: RTS Results of TSR with the genetic algorithm of all three coverage metrics and without zero-coverage TCs. Entries marked with an asterisk (*) represent TSR runs in which the genetic algorithm produced an inadequate RTS.

	Checked Coverage		Line Coverage		Method Coverage	
ID	F_{chk}	T_{chk}	F_{ln}	T_{ln}	F_{mth}	T_{mth}
1	0.166*	1329.97s	0.044*	2461.12s	0.075*	509.89s
2	0.231*	181.10s	0.036*	365.50s	0.062*	91.03s
3	0.091	61.69s	0.070	251.43s	0.168	24.36s
4	0.362	22.50s	0.151	97.85s	0.107	7.22s
5	0.418	11.87s	0.174	322.03s	0.119	5.24s
6	0.037	2.83s	0.072	214.01s	0.053	31.46s
7	0.074*	44.25s	0.050*	90.45s	0.052*	37.31s
8	0.038	5.08s	0.154	17.15s	0.154	0.17s
9	0.467	1.03s	0.067	2.38s	0.044	0.91s
10	0.048	0.88s	0.667	11.91s	0.667	1.99s
Avg.	0.193	166.12s	0.148	383.383s	0.150	70.958s
Med.	0.129	17.185s	0.071	155.93s	0.091	15.79s
Std.	0.167	412.594	0.189	741.754s	0.187	156.696s

evaluating the solutions via the algorithm's fitness function, which means that the improvement and optimization of a GA's fitness function must be of high priority.

The overall, preliminary conclusion from the comparison of our genetic to the two greedy algorithms is, therefore, that the greedy algorithm's performance is not up to par. As mentioned earlier, as this is a simple GA more sophisticated approaches (such as Coviello et al. [7]) may produce better results.

5.2.3 Inclusion of Zero-Coverage TCs. So far, we discussed RTS results where zero-coverage TCs were excluded from the suite, conforming entirely to the definition of TSR (see Definition 1). However, especially with checked coverage, we encountered a considerable amount of zero-coverage TCs.

Table 5 shows the performance of all reduction algorithms with checked coverage including and excluding the zero-coverage TCs. For the other coverage metrics, the TSR results with and without zero-coverage TCs were close to identical as there were no to very few zero-coverage TCs identified with these metrics. Hence, the results including zero-coverage TCs are omitted here. What we can observe from the inclusion of the zero-coverage TCs in Table 5 is that the RTS size increased for many projects. While the suites' size increased, execution times stayed roughly the same (within margin of error between individual runs), which was expected as the zero-coverage TCs are only re-added to the RTS after the reduction. Naturally, the reduction factor decreased compared to the previous runs where zero-coverage TCs were excluded.

5.2.4 Reduction Factor. A more compact and condensed reduction factor comparison can be made when investigating Figure 4. The figure shows the reduction factor distributions for checked coverage with zero-coverage TCs (+) as well as for all metrics without zero-coverage TCs (-)[17].

The figure emphasizes the difference in reduction factors between our GA and the two greedy algorithms. In addition, we can observe that the distribution of reduction factors of the GA's RTSs is much more uniform than the distributions of the greedy algorithms' reduction factors where the interquartile ranges are mostly more than twice as high.

From Figure 4 we can again deduce that the inclusion of zero-coverage TCs decreased the reduction factors of the checked coverage RTSs (and hence increased the RTS size). While for RTSs without zero-coverage TCs, checked and method coverage had a

[17]We omitted the result for line and method coverage in case of the inclusion of the zero-coverage TCs. As mentioned early, they are nearly identical to the one's excluding those tests.

Table 5: RTS Results of TSR via all three algorithms with checked coverage with zero-coverage TCs.

	Greedy HGS		Delayed Greedy		Genetic	
ID	F_{hgs}	T_{hgs}	F_{del}	T_{del}	F_{ga}	T_{ga}
1	0.443	6.39s	0.444	782.38s	0.046	1781.02s
2	0.558	1.27s	0.558	92.50s	0.030	140.81s
3	0.790	0.16s	0.678	1.84s	0.091	53.47s
4	0.629	0.18s	0.635	1.31s	0.267	46.45s
5	0.621	0.14s	0.621	0.96s	0.122	7.16s
6	0.700	0.19s	0.646	4.13s	0.037	41.42s
7	0.537	0.37s	0.537	25.12s	0.050	44.30s
8	0.192	0.06s	0.192	0.21s	0.038	5.84s
9	0.389	0.59s	0.389	0.17s	0.133	0.88s
10	0.619	0.02s	0.619	0.03s	0.048	0.81s
Avg.	0.548	0.937s	0.532	90.865s	0.086	212.216s
Med.	0.589	0.185s	0.589	1.575s	0.049	42.86s
Std.	0.171	1.951s	0.151	244.669s	0.073	552.799s

mostly similar reduction factor distribution, checked coverage distributions with zero-coverage TCs were more in line with the line coverage distributions.

Overall, we can conclude from analyzing the reduction factors that greedy algorithms led to more drastically reduced suites, suggesting that they were able to more often reach close to optimal TSR solutions. The higher variation in reduction factors of the greedily obtained RTSs can be explained by the fact that the ten projects probably had a varying degree of redundancy in their TSs, hence leading to more diverse distributions of reduction factors.

5.2.5 *Mutation Score.*

From the reduction factor we move on to the FDC by discussing the mutation score changes between the original and the RTS. Figure 5 shows the mutation score distribution of the original TS as well as the one's obtained via all three algorithms based on checked coverage (with and without zero-coverage TCs) as well as on line and method coverage[17]. Regarding our original TSs' mutation scores, we can see that—similarly to the traditional coverage scores—most projects have very high mutation scores, with the average score being slightly below 90%. Only Project 8 had a significantly lower score (65%). Therefore, our results indicate that the original TSs of our ten projects have overall high FDCs.

We can observe that the reduction based on method and checked coverage without zero-coverage TCs are the least promising as they have the highest average mutation score drops (-16% and -17%, respectively). For method coverage, this was to be expected, as it is the most coarse-grained coverage metric. We therefore assumed that it would yield RTSs with limited FDC as the coverage of a method cannot express much about what was covered inside the method.

For checked coverage without zero-coverage TCs, however, the drop in mutation score compared to the original score is also exceptionally high. When including zero-coverage TCs, the drop is reduced to 12% on average. An explanation for this drop are the overall lower coverage scores of checked coverage, which means that there are overall less requirements to be satisfied by the TSR algorithm. The difference between the in- and exclusion of zero-coverage TCs suggests that although zero-coverage TCs are not deemed valuable in terms of checked coverage, they do indeed contribute to the TS' FDC.

The RTSs based on line coverage yielded the mutation scores with the lowest drops. For the greedy algorithms (which drastically

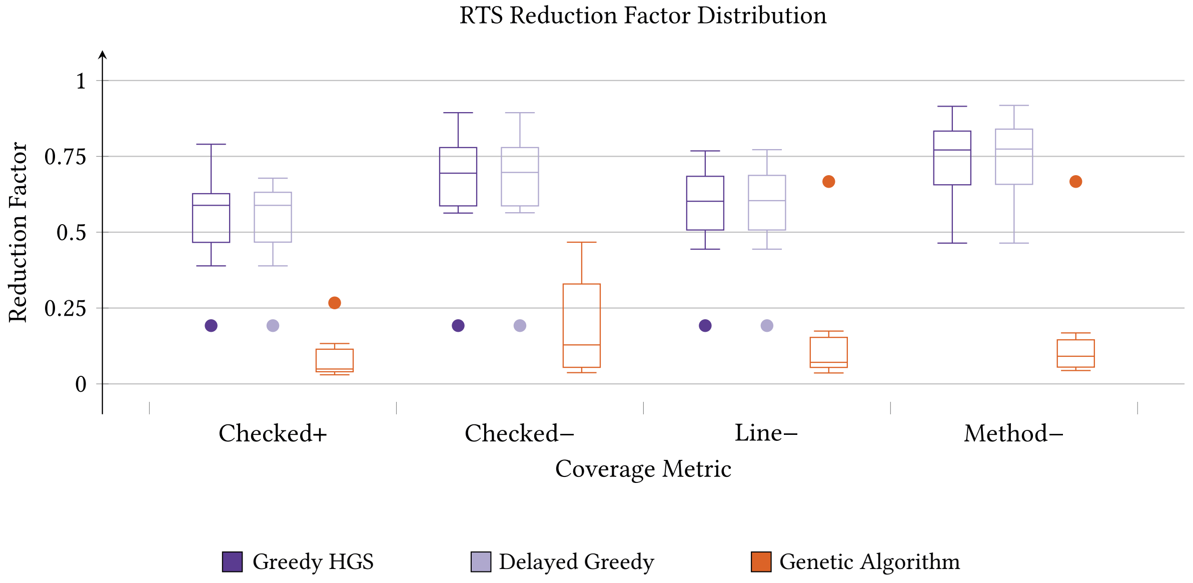


Figure 4: Distribution of the RTSs' reduction factors for each coverage/algorithm TCs combination without (-) zero-coverage TCs and with (+) for checked coverage.

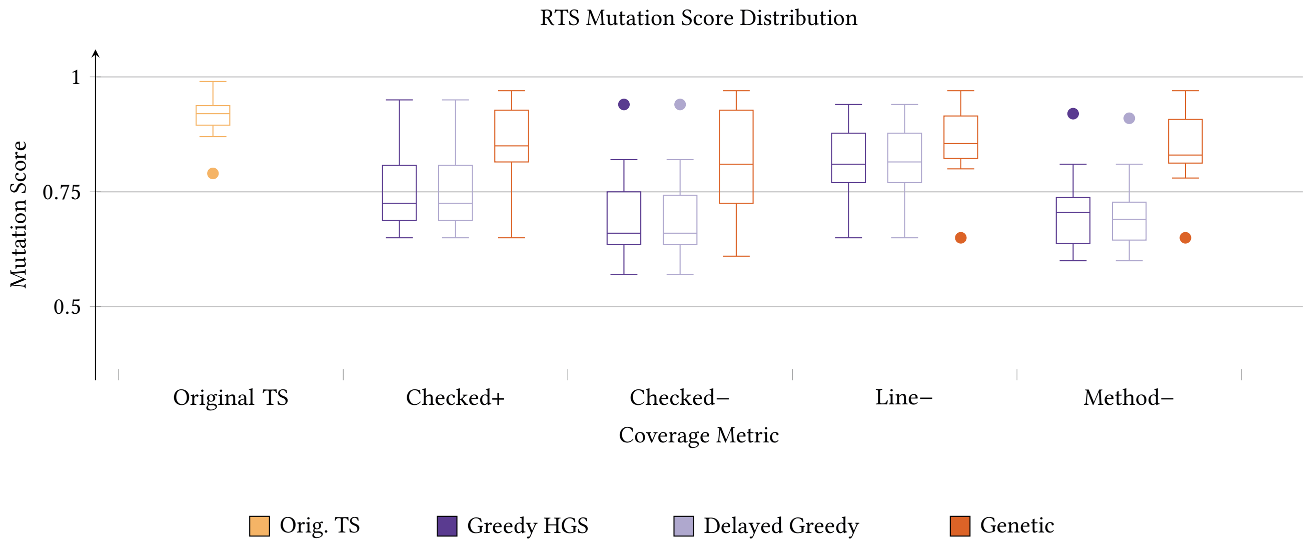


Figure 5: Distribution of the RTSs' mutation scores of the original TS and of the RTS for each coverage/algorithm TCs combination without (-) zero-coverage TCs and with (+) for checked coverage.

reduced the TS size) the average drop was around 6%, meaning that FDCs did not decrease much. When compared to checked coverage, line coverage-based RTSs have higher mutation scores. Again, the explanation for this can be the much higher original line coverage scores compared to the original checked coverage scores plus the fact that the set of lines deemed as covered with line coverage is a super-set compared to the set from checked coverage. This means that TSR was based on a higher number of requirements that could have led to more or other TCs in the RTS than when using checked coverage.

One observation from Figure 5 is that the interquartile ranges of the RTSs' mutation scores have increased dramatically in comparison to the original suite's. This holds true for all combinations of coverage metrics and algorithms. What we can conclude from this fact is that TSR does not work equally well for all projects in terms of FDC loss. While some projects (e.g. Projects 3 and 8 with line coverage) still have close-to-original mutation scores with the respective RTSs, others projects' (e.g. Projects 6 and 7 with line coverage) RTSs have higher drops (12% and 7% respectively).

What we have not mentioned explicitly so far is the much lower drop in mutation scores of RTSs produced with our GA, as one can see in Figure 5. While average drops of 1% to 3% percent (depending on the coverage metric) seem to justify the longer runtimes of the algorithm, the fact remains that the RTS reduction factor is very small in comparison to the greedy approaches. Hence, the savings in TS execution time and resources are much smaller when using the GA to find an RTS, while at the same time the confidence in the RTS can be deemed higher. Judging from the RTS sizes of the GA, we also expected the better mutation scores.

It should be further noted, that (with the exception of checked coverage) the interquartile ranges of the GA-based RTSs mutation score distribution is considerably smaller than the RTSs' scores from the greedy approaches, meaning that the mutation results were more consistent throughout the projects. This conforms with the more consistent (but also much smaller) reduction factors shown in Figure 4.

Overall, we can state that TSR with greedy approaches yielded quite drastically (close to optimal) reduced suites, especially when compared to GA-based RTSs. As expected, the greedy reductions led to a generally higher loss in FDC as the genetic approach, suggesting that there is ultimately a trade-off to be made between RTS size and FDC loss. When comparing coverage metrics, line coverage produced the smallest FDC losses, while method coverage yielded the biggest losses. Checked coverage-reduced RTSs are in between and only provide acceptable mutation score results when including zero-coverage TCs.

5.3 Answers to the Research Questions

Based on our results and findings from the previous subsection, we can answer the two research questions as follows:

- **RQ1:** Checked coverage can certainly be used as a coverage metric and in that sense it is a suitable metric. However, it is in fact outperformed by line coverage which (1) is faster to compute and (2) the RTSs found with line coverage as a requirement criterion have a lower FDC loss (based on the measured mutation scores) than the RTSs based on checked coverage. In addition, the computation time associated with checked coverage is impractical for large projects.
- **RQ2:** Our results have shown that (at least in the scope of our tested projects and suites), out of the three investigated coverage metrics, line coverage performed best as a TSR requirement criterion. In terms of reduction algorithm, we clearly see the advantage of greedy algorithms as they produced a much more reduced suite in generally less time than

our GA. However, it should be noted that our GA is a simple algorithm in its class and there is certainly room for improvement. Furthermore, the RTSs delivered from our GA had by far the smallest FDC losses, albeit also the smallest reduction factors. To summarize, line coverage in combination with the Greedy HGS algorithm showed in our evaluation the most promising result.

6 RELATED WORK

There is a vast amount of research on methods to reduce TS size as well as associated empirical studies. For instance, Wong et al. [25, 26] assessed TSR with respect to the reduction factor and the loss in FDC in the context of C programs. Their studies were based on common Unix utilities and a program developed for the European Space Agency. Given their results, they concluded that the FDC of the reduced TS are minor. Based on the findings, Rothermel et al. [19] performed additional experiments that showed that coverage-based TSR can compromise the FDC severely. These results are consistent with our observations, i.e., FDC loss is related to the used coverage metric and while line coverage provides a negligible drop in mutation score, the same cannot be stated for checked or method coverage.

More recently, Coviello et al. [6] presented a study of adequate and inadequate TSR approaches. In their evaluation, the authors considered six adequate and 12 inadequate reduction approaches. They based their experiments on 19 Java programs from the *Software-artifact Infrastructure Repository (SIR)*, which is a public dataset used in regression testing. The data showed that inadequate methods achieve a better trade-off between the reduction factor and FDC loss. The FDC loss was computed as the ratio of the number of faults in the original TS and the number of faults detected by the RTS[18]. Their results also show that the loss in FDC is negligible for adequate TSR approaches in combination with statement or method coverage. Our results differ in the sense that while we also found that line coverage (as our most fine-grained metric) produces suitable reductions in regard to RTS size and FDC, we cannot state the same for method coverage (or checked coverage for that matter).

In regard to checked coverage, Schuler et al. [20, 21] conducted two empirical studies to assess TS evaluation based on checked coverage. In their work, they were able to improve TS effectiveness by increasing the checked coverage scores of several open-source Java projects (by adding meaningful assertions to the test cases). The effectiveness of the TS was measured via mutation testing. In addition, they showed that the checked coverage score correlates with the mutation score when purposefully removing assertions from the TSs [20, 21]. The results are confirmed in a later study by Zhang et al. [29], who also come to the conclusion that there is a strong correlation between (the number of) assertions and a TS's FDC.

In a first attempt, Chen et al. [3] investigated the role of assertions in coverage-based TSR. The authors reached the conclusion that assertions have a significant impact on TSR results when using them as a coverage metric. They calculated checked coverage (which they call *assertion coverage*) similarly to Schuler et al. [21] and Zhang et al. [29] via dynamic slicing. In a first evaluation, the authors show that checked coverage decreases significantly in reduced TSs with respect to the original TSs' values when performing TSR based on statement and method coverage. In addition, the authors conducted TSR experiments in which they simultaneously used statement coverage and checked coverage as TSR requirements. Their approach is based on integer linear programming (ILP). The results show that this combination of coverage metrics performed better than conducting TSR based on just statement coverage with higher mutation scores, albeit also with larger RTSs. While the research goal of our work is similar to Chen et al. [3], our experiments are based on individual coverage metrics only (i.e., we did not combine metrics). Further, we compare different reduction procedures on these different test requirements, while Chen et al. [3] only compared their statement-coverage-based ILP technique to their combined ILP approach.

[18]The SIR dataset contains information on faults present in the projects.

7 CONCLUSION AND FUTURE WORK

Checked coverage is an interesting alternative to traditional coverage metrics (such as line or method coverage). It differs from traditional measures as lines do not only have to be executed to be marked covered but also *checked* from a test oracle. While checked coverage was already explored more thoroughly in the TS evaluation settings, research into its abilities as a TSR requirement was rather limited. Although, test oracles contribute to fault detection within TS and hence should be considered in TSR.

Using our own open-source Java TSR tool, JSR, we performed an empirical evaluation of TSR for ten public real-world projects with three coverage metrics (i.e., line, method, and checked coverage) and three reduction procedures (i.e., Greedy HSG, Delayed Greedy, and a genetic algorithm). The overall results indicate that checked coverage is indeed theoretically an interesting coverage metric; however, its applicability as a TSR requirement criterion is rather limited. This is revealed by the fact that the RTSs based on checked coverage had higher FDC losses compared to line coverage. While the RTS reduction factors were mostly slightly higher in comparison to the factors of line coverage-based RTSs, their mutation scores dropped on average only half as much as the checked coverage-based RTSs. Furthermore, the computation of checked coverage required significantly more runtime than the computation of the other traditional coverage metrics. These observations led us to our conclusion, that line coverage in combination with a greedy algorithm provides a sufficiently reduced suite with a more acceptable FDC loss in high but acceptable runtime. This combination, therefore, turned out to be the best performing parameter combination. While our results have shown that checked coverage is not as suitable as line coverage in terms of TSR, its value for test suite evaluation is considerably higher.

As our JSR tool can be easily adapted to realize other reduction procedures, a next logical step is to compare the established algorithms and approaches against newly integrated ones and using more subjects by exploiting public datasets that have been applied previously in the testing research field. Given that the lack of comparative studies on large-scale software projects was criticized by Khan et al. [14], such studies could be of high value for the TSR research community.

REFERENCES

[1] Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. Slicer4J: a dynamic slicer for Java. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1570–1574.

[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[3] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, and Bing Xie. 2017. How Do Assertions Impact Coverage-Based Test-Suite Reduction? *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017* (5 2017), 418–423. https://doi.org/10.1109/ICST.2017.45

[4] Nour Chetouane, Franz Wotawa, Hermann Felbinger, and Mihai Nica. 2020. On Using k-means Clustering for Test Suite Reduction. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 380–385. https://doi.org/10.1109/ICSTW50294.2020.00068

[5] Carmen Coviello, Simone Romano, and Giuseppe Scanniello. 2018. CUTER: ClUstering-based TEst Suite Reduction. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 306–307.

[6] Carmen Coviello, Simone Romano, and Giuseppe Scanniello. 2018. An Empirical Study of Inadequate and Adequate Test Suite Reduction Approaches. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Oulu, Finland) *(ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/3239235.3240497

[7] Carmen Coviello, Simone Romano, Giuseppe Scanniello, and Giuliano Antoniol. 2020. GASSER: Genetic Algorithm for TeSt Suite Reduction. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 36, 6 pages. https://doi.org/10.1145/3382494.3422157

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA.

[9] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[10] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2 (1 1993), 270–285. Issue 3. https://doi.org/10.1145/152388.152391

[11] Zhen-feng He, Bin-kui Sheng, Cheng-qing Ye, et al. 2005. A genetic algorithm for test-suite reduction. In *2005 IEEE International Conference on Systems, Man and Cybernetics*, Vol. 1. IEEE, 133–139.

[12] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[13] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[14] Saif Ur Rehman Khan, Sai Peck Lee, Nadeem Javaid, and Wadood Abdul. 2018. A Systematic Review on Test Suite Reduction: Approaches, Experiment's Quality Evaluation, and Guidelines. *IEEE Access* 6, 11816–11841. https://doi.org/10.1109/ACCESS.2018.2809600 10,3,5,16,17,18,19,20,22,23,24.

[15] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Information processing letters* 29, 3 (1988), 155–163.

[16] Xue-ying Ma, Bin-kui Sheng, and Cheng-qing Ye. 2005. Test-Suite Reduction Using Genetic Algorithm. In *Advanced Parallel Processing Technologies*, Jiannong Cao, Wolfgang Nejdl, and Ming Xu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 253–262.

[17] Alessandro Marchetto, Giuseppe Scanniello, and Angelo Susi. 2019. Combining Code and Requirements Coverage with Execution Cost for Test Suite Reduction. *IEEE Transactions on Software Engineering* 45, 4 (2019), 363–390. https://doi.org/10.1109/TSE.2017.2777831

[18] Christos H. Papadimitriou and Kenneth Steiglitz. 1981. Combinatorial Optimization: Algorithms and Complexity.

[19] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. 2002. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability* 12, 4 (2002), 219–249.

[20] David Schuler and Andreas Zeller. 2011. Assessing oracle quality with checked coverage. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 90–99.

[21] David Schuler and Andreas Zeller. 2013. Checked coverage: an indicator for oracle quality. *Software testing, verification and reliability* 23, 7 (2013), 531–551.

[22] Lukas Stracke. 2022. *Test Suite Reduction with Checked Coverage*. Master's thesis. Graz University of Technology.

[23] Sriraman Tallam and Neelam Gupta. 2005. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (2005), 35–42. https://doi.org/10.1145/1108792.1108802

[24] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[25] W Eric Wong, Joseph R Horgan, Saul London, and Aditya P Mathur. 1998. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience* 28, 4 (1998), 347–369.

[26] W Eric Wong, Joseph R Horgan, Aditya P Mathur, and Alberto Pasquini. 1999. Test set size minimization and fault detection effectiveness: A case study in a space application. *Journal of Systems and Software* 48, 2 (1999), 79–89.

[27] Qian Yang, J. Jenny Li, and David M. Weiss. 2009. A survey of coverage-based testing tools. *Computer Journal* 52, 589–597. Issue 5. https://doi.org/10.1093/comjnl/bxm021

[28] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120.

[29] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, 214–224. https://doi.org/10.1145/2786805.2786858