IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 29 May 2023, accepted 13 June 2023, date of publication 23 June 2023, date of current version 5 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3289073

RESEARCH ARTICLE

An Empirical Study of Greedy Test Suite Minimization Techniques Using Mutation Coverage

SEEMA JEHAN[1], (Member, IEEE), AND FRANZ WOTAWA[2], (Member, IEEE)

[1]School of Electrical Engineering and Computer Science, National University of Sciences and Technology, Islamabad 44000, Pakistan
[2]Christian Doppler Laboratory for Quality Assurance Methods for Autonomous Cyber-Physical Systems, Institute for Software Technology, Graz of Technical University, 8010 Graz, Austria

Corresponding author: Seema Jehan (seema.jehan@seecs.edu.pk)

This work was supported in part by the Austrian Federal Ministry for Digital and Economic Affairs, in part by the National Foundation for Research, Technology and Development, and in part by the Christian Doppler Research Association.

ABSTRACT Test suite minimization is the task of finding a smaller test suite that still fulfills the properties of the original test suite but which comprises fewer test cases. It is important in practice, especially in the context of regression testing, where test suites are re-executed. However, test suite minimization as a set covering problem is known as an NP-complete problem, which requires applications of heuristics. Although many test suite minimization techniques have been applied previously but obtained conflicting results primarily due to inherent differences in underlying programming languages and experimental setup. In this respect, we study traditional greedy-based algorithms for test suite minimization that allow to remove test cases in a way such that the reduced test suite satisfies all requirements. Specifically, we evaluated commonly discussed approaches on publicly available JavaScript applications using mutation coverage. We show that the discussed algorithms reduce the test suite size of the studied example programs on average to 70% without compromising the fault-detection capability of the original test suite. The suggested approach not only minimizes the test suite's size, thereby reducing the regression testing cost, but also ensures that the reduced test suite catches the same number of faults as that of the original test suite. Further, we also examine their performance in scenarios when meeting all testing requirements is not feasible due to time and budget constraints.

INDEX TERMS Test suite minimization, mutation testing, regression testing, JavaScript.

I. INTRODUCTION

Test suite minimization (reduction) is the task of identifying and, later on, eliminating redundant test cases from the original test suite. Minimizing test suites is important because software constantly evolves, which causes the test suite to grow accordingly. As a consequence, testing software over time becomes expensive, requiring a lot of resources. This is why test suite minimization has been a subject of wide interest for regression testing, which aims at re-testing software after modifications. Regression testing techniques can be divided into three categories: test suite minimization, regression test case selection, and test case prioritization [1]. In contrast, to test suite minimization, test case prioritization aims at reordering test cases without removing them from the original test suite such that faults can be detected as early as possible. In this paper, however, we focus on test suite minimization, where redundant tests are eliminated by selecting a minimal subset of tests that can satisfy all requirements. This is similar to the minimal set cover problem, which is an NP-complete problem [2].

To illustrate the necessity of test suite minimization, let us consider the case of Google. Memon et al. observed that, on average, 150 million tests are performed daily by Google's Test Automation Platform (TAP) [3]. Any new update is made to the software every few seconds leads to a "big test sets"

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

issue that is analogous to the Big Data problem [4]. It requires building robust and cost-effective strategies for test case selection or reduction during regression testing.

Previously, regression testing techniques have been studied mainly on C and Java programs [2]. Surprisingly, these studies report conflicting results as these were conducted on different programs using different test suites containing different test cases targeting different faults. For instance, Wong et al. [5] and Zhang et al. [6] concluded that test suite reduction approaches have a negligible effect on the fault-detection capabilities of the studied test suites. On the contrary, Rothermel et al. [7] observed a severe impact of test suite minimization on the test suite quality. Our contribution lies in analyzing test suite minimization approaches on JavaScript applications due to the following factors: First, these applications are typically used nowadays for building modern mobile and web applications. However, due to the **weakly typed** nature of JavaScript these applications are considered vulnerable as compared to statically typed languages such as Java. For example, 28% of JavaScript applications are reported to have *Undefined Symbol* exception whereas 9% of such applications observed *Null Exception* [8]. Second, the **asynchronous** behavior of these applications further complicates the testing process [9], [10]. Third, it is possible to inject code into these applications at runtime due to dynamic loading feature, which has naturally raised security concerns for such applications [11]. Fourth, these packages have multiple releases with updated test suites that makes regression testing harder due to continuous integration practices as the software needs to be validated after every update [12]. Consequently, Zhang et al. tailored their search-based software testing tool, namely *EVOMASTER* to enable white-box testing of JavaScript applications [13]. Likewise, Andreasen et al. discussed similar challenges observed during dynamic analysis and testing of JavaScript applications [11]. This motivated us to evaluate the efficacy of test suite minimization techniques for JavaScript applications.

In this work, we discuss five algorithms for test suite minimization and compare their effectiveness. We measure effectiveness both in terms of the reduction in the resulting test suite size as well as fault-detection loss. First, we discuss Harrold et al. algorithm, probably the first approach discussed in the literature [14]. This is followed by the greedy algorithm based on an approximation algorithm for set covering; It is typically used in test suite reduction approaches and serves as a baseline technique [15]. Likewise, the third algorithm, delayed greedy, has also been widely discussed for test suite minimization [16]. In addition to the previously mentioned traditional test suite minimization algorithms, we introduce two more algorithms: one is a variant of *Delta Debugging* algorithm by Hildebrandt and Zeller [17] originally developed for minimizing one particular test case, the second is a search-based algorithm.

To compare effectiveness, we carried out an empirical evaluation that is based on ten available open-source Node Package Manager (NPM) software packages. NPM is the largest online repository containing around two million Node.js packages, which is *"an open-source, cross-platform, back-end, JavaScript runtime environment that executes JavaScript outside a web browser"*. Since these packages are very frequently used by developers for building JavaScript applications, robustness and fault-resilience play a vital role in the quality of the developed applications [12].

We carried out an empirical evaluation in order to investigate the following research questions in the context of JavaScript applications:

RQ1: Which test suite minimization algorithm achieves the maximum reduction of the test suite size without compromising the fault-detection loss?

RQ2: Which test suite minimization algorithm performs better in the context of execution time?

In summary, the contributions of this paper include: (i) answering the question of how much fault-detection is compromised by applying test suite reduction based on killed mutants, and (ii) coming up with an exhaustive empirical evaluation comparing traditional test suite minimization algorithms on multiple versions of publicly available JavaScript packages. The latter contribution can be considered as a replication study comparing well-known and also two other algorithms for test suite minimization.

We organize the remainder of this paper as follows: In Section II, we discuss related research work. Following, in Section III we introduce the underlying foundations to be self-contained. Afterward, in Section IV, we discuss the five algorithms for test suite minimization in more detail. In Section V, we provide the details regarding the experimental evaluation along with a detailed discussion on results and threats to validity. Finally, we conclude the paper and discuss future work in Section VI.

II. RELATED WORK

Whenever software undergoes any update, it must be ensured that the new version still satisfies either all test requirements for the previous version or the latest version of the Software Under Test (SUT). The former type of regression testing is termed ''corrective'' regression testing, while the latter is named ''progressive'' regression testing [18]. Based on the actual testing requirements, *Regression testing* can be classified as *test suite minimization (reduction)*, *test case selection*, or *test case prioritization*. Test suite minimization purely focuses on the elimination of redundant test cases. Test case selection, on the other hand, does not remove any test case but rather selects only appropriate tests for testing the modified or added functionality. Test case prioritization focuses on test ordering to maximize certain properties like coverage or fault detection.

A. TEST SUITE MINIMIZATION

Test Suite Minimization can be further categorized into following approaches: Greedy-based, clustering-based, search-based and hybrid approaches [19]. Surprisingly, 67% of the discussed studies have employed greedy-based approaches for test suite minimization whereas 20% have discussed

search-based approaches. Among the earliest works on test suite minimization (reduction), Harrold et al. explain how the redundant test cases can potentially increase the regression testing cost of software under test [14]. Therefore, they suggested a heuristic for test suite reduction by iteratively selecting essential test cases, thereby reducing the test suite size. Since finding a smallest possible subset of a test suite, an ''optimal representative'' test suite, that can test all requirements is an NP-complete problem, Chen and Lau [20] proposed a greedy search algorithm for test suite reduction named GRE.

Specifically, they suggested a test suite reduction algorithm that ensures that all essential test cases are preserved while reducing the test suite. A test case is called ''essential'' that, if removed, will result in a test suite that can not meet all requirements. On the contrary, if the resulting test suite still fulfills all requirements, a test case is termed ''redundant''. Particularly, they first select all essential test cases followed by 1-to-1 redundant test cases before applying the greedy strategy. This way, they claim to achieve better reduction than Harrold et al. approach [14].

Tallam and Gupta introduced a variant of the Greedy algorithm called Delayed Greedy (DGR) using a concept analysis approach [16]. Specifically, they take a context table as input, where each row contains a test case, and each column represents a set of requirements covered by each test case. Based on concept analysis theory, the algorithm performs object, attribute, and owner reduction before making a greedy choice of removing a certain test case. Their results show that DGR achieved a smaller subset of a test suite than the traditional Greedy [20] and Harrold et al. heuristic that covers all testing requirements [14].

Zhong et al. compared four test suite reduction algorithms, that is, Harrold et al. [14], GRE [20], ILP-based approach [21], and genetic algorithm (GA) on 11 C programs [22]. They measured the complexity of a test suite accompanied by these programs by two attributes: the number of test cases in a test suite and the number of requirements a test suite needs to satisfy. Further, they evaluated the aforementioned algorithms on test suites of varying complexity with respect to the execution time and reduction in test suite size. They reported that GA by Mansour and El-Fakih [23] has the highest complexity and execution time. Further, Harrold et al., as compared to other algorithms, take the least time to generate reduced test suites. Interestingly, all approaches produced reduced test suites of similar sizes but contained different test cases. The authors recommended Harrold et al. among the other studied approaches. However, one threat to validity was the lack of real-world test cases.

Jones et al. adapted traditional test-suite reduction and prioritization to satisfy the modified-condition/coverage (MC/DC) coverage criterion that is considered to be more effective than the classical statement coverage criterion [24]. Similarly, Jena et al. discussed test suite reduction approaches using MC/DC coverage criterion for safety critical systems [25]. In addition to MC/DC coverage, combinatorial-based coverage has also been discussed in the literature for effective test suite reduction in a very specialized case [26]. In contrast, we focussed on classical greedy-based test suite reduction heuristics by employing mutation score instead of source code metrics and combinatorial coverage.

Smith and Kapfhammer carried out an empirical study on eight real-world but relatively small-sized case studies using traditional test suite reduction algorithms by including execution time [27]. They argued that a tester's main objective is to find more faults in less time, whereas the test suite reduction algorithms only consider coverage as the test adequacy and an evaluation metric. They employed three evaluation metrics: reduction in reduced test suite size, reduction in execution time, and coverage effectiveness in terms of total requirements covered in a certain time limit by ignoring the fault-detection parameter. On the contrary, our work includes both fault detection and execution time while measuring the effectiveness of an underlying approach. Their results showed that a reduced test suite might not have the lowest execution time. Moreover, Delayed Greedy performed well for both test suite reduction and prioritization, whereas Harrold et al. only showed good results for reducing test suite size. Likewise, our study on JavaScript applications confirms the superiority of the DelayedGreedy approach over the Harrold et al. technique.

Interestingly, Zhang et al. observed in their empirical study on Java programs that the key factors to be considered for reducing the test suite's size and fault-detection capabilities are the ''test case granularity'' and the ''test coverage level'' instead of any reduction approach in specific [6]. We also obtained the same result, that is, all reduction algorithms obtained a similar reduction in test suite size and fault-detection capability when all requirements must be satisfied. For example, all reduction algorithms, including greedy and Harrold et al. approach for their subject programs, obtained a 65% reduction in test suite size with fault-detection loss of nearly 5% for method-level coverage, whereas we obtained roughly 70% reduction with zero loss in fault-detection capability. However, they did not consider the execution time for evaluating studied algorithms. Also, they did not include the delayedGreedy algorithm, which seems to perform better than Harrold et al. heuristics in our analysis. Further, they studied the impact of the reduction in test suite size and the fault detection on statement-level and method-level test coverage. Similarly, for the test case type, they evaluated the same metrics on the method level and class level. Their results show that method-level test coverage should be preferred over statement-level. Likewise, method-level test cases showed better results than class-level. Our work mainly focuses on measuring the effectiveness of mutated faults generated by a third-party mutation testing tool for Node.js packages, as mutation coverage is considered a stronger coverage criterion than statement coverage.

Likewise, Shi et al. raised similar concerns and discussed various trade-offs while deciding on a suitable test suite reduction strategy [28]. They highlighted the fact that all aforementioned techniques, Harrold et al. [14], GRE [20], Tallam and Gupta [16] mainly focussed on 100% attainment of statement coverage by the reduced test suite. Therefore, they also studied the quality of the reduced test suites in terms of their mutation score in addition to basic coverage. Interestingly, the mutation score-based reduced test suites prove to be more stable across multiple versions of the same program. Moreover, all previously conducted studies only considered test suite reduction for one version of the program, which might not be a realistic figure taking into account that a typical software undergoes multiple versions during its lifetime. This means that a test case deemed redundant for one version might be very useful in exposing faults in the next version. Further, they made use of multiple versions of GitHub projects for experimental evaluation as compared to majorly studied programs from Software-artifact Infrastructure Repository (SIR) [29]. As a third contribution, they also proposed inadequate test suite reduction as an alternative to adequate test suite reduction techniques to facilitate test engineers in making situation-aware decisions. Similar to them, we also employ mutation coverage criteria for evaluating test suite reduction algorithms with varying inadequacy levels. Our results seem to be consistent with their evaluation as by increasing the inadequacy level to 10, meaning 90% of the requirements are satisfied, the test suite size reduction increases on average to 80%, whereas we obtained 82.7% for the greedy algorithm. However, they did not include the execution time factor in their analysis.

Agrawal et al. proposed a fault-based test reduction algorithm and compared it with four benchmark heuristics for regression testing, namely, greedy, additional greedy, Harrold et al., and enhanced Harrold et al. algorithm on a collection of 12 programs taken from the SIR repository [30]. It was observed that this algorithm even outperforms the greedy algorithm in reducing the test suite size. However, both have similar performance in terms of execution time. Specifically, it takes as input a dummy fault matrix containing all faults and test sets covering those faults. As a next step, it assigns weight to each test case based on the number of faults detected by it. Likewise, each fault is also assigned a weight depending on the number of test cases that can detect this fault. These weights are employed to select a test case that reveals a maximum number of faults from a test set of size 1 until a total number of faults. This process is repeated till either all faults have been covered, or all sets of the original test suite are examined. The cost of generating an input matrix containing 12 faults and 15 test cases is not specified. Further, they assumed that the execution cost of each test case is one second.

In addition to greedy-based heuristics, search-based algorithms are also applied for test suite minimization. Wang et al. developed a tool,''TEst Minimization with Search Algorithms (TEMSA)'' to compare ten multi-objective search algorithms on an industrial case study of product lines [31]. They observed that the Random-weighted Genetic Algorithm outperformed other approaches when applying six different inputs for fitness functions, including Fault Detection Capability, test minimization percentage, and overall execution time.

B. TEST CASE PRIORITIZATION

Test case prioritization focuses on the permutation of test cases to maximize the testing objective. This prioritization could be coverage-based, history-based, or probability-based. Some prominent test case prioritization algorithms include *greedy algorithms*, *meta heuristics* and *evolutionary search approaches* [2]. *Greedy algorithms* follow the greedy principle of incrementally adding test cases to maximize the desired metric. But, they might fail to come up with an optimal test case ordering always. *Meta heuristic techniques* find a solution to combinatorial problems at an economical cost. The target of *evolutionary search algorithms* is to follow the survival of the fittest strategy for test suite prioritization.

Li et al. compared five algorithms, namely greedy, additional greedy, 2-optimal, genetic algorithm(GA), and hill-climbing for test suite prioritization during regression testing of programs with small and large-size test suites [15]. In the case of small-sized programs, additional greedy's performance is comparable with GA. However, additional greedy should be preferred since GA performed worse in a few examples. For large-sized test suites, the Genetic Algorithm (GA) performed better than the other studied approaches as it can handle the search space better than the rest of the approaches, whereas the greedy algorithm showed the worst performance. The reported results were based on the code coverage criterion.

Elbaum et al. discussed the importance of test case prioritization in the context of continuous integration at Google [32]. During the continuous integration development, the code written by developers needs to be tested before and after the submission to the ''code base''. Therefore, the authors suggested different regression testing strategies for the ''pre- and post-submit'' testing phase: in the pre-submit phase, they applied regression test selection approach to help developers select a subset of test suites; on the other hand, in the ''post-submit testing phase'' they recommended test case prioritization techniques for earlier fault-detection. Their approach is based on cost-effective algorithms that do not require code coverage.

Henard et al. compared twenty test case prioritization techniques concerning the white-box versus black-box approaches [33]. They studied the robustness of the generated test suite during the entire development life cycle encompassing various releases of the software under test. Their study revealed only 2% degradation among the top three white- and black-box regression techniques in the thirty versions of the subject programs.

Miranda et al. pointed out that the existing testing tends to fail as the test suite size increases in the context of industrial

systems [4]. For that, they proposed a set of similarity-based test case prioritization techniques. These techniques are based on the already proven algorithms from the big data domain. They claim that their approach can select the most effective test cases from a set of one million test cases in less than twenty minutes. Cruciani et al. [34] is one of their latest work on test suite reduction. They tested their approach on C and Java programs. Their approach is based on similarity-based test case prioritization. Our approach also evaluates a variant of the delta debugging algorithm for removing redundancy from test suites [35]. In addition, we employ mutation scores to judge the effectiveness, whereas they apply the Average Percentage of Faults Detection (APFD). This determines how fast the approach detects faults but does not take into account the time taken by the prioritization technique itself. Both these criteria aim at judging the ability of a test case to detect potential faults. They applied their approach to both white-box and black-box testing approaches, whereas our work is mutation-score based that is only applicable to white-box testing.

Groce et al. applied delta debugging minimization to reduce the size of test cases contained in a test suite as opposed to previously discussed approaches that aim at reducing the size of an entire test suite [36]. The aim was to reduce the test suite's size based on code coverage criterion rather than just minimizing only the failing test case as proposed in the original Delta Debugging algorithm. Thus, increasing the overall test suite's ''efficiency'', that is, to ensure improved coverage per unit time by minimizing the size of individual test cases. They evaluated the approach on a randomly generated test suite containing both passing and failing test cases. The rationale behind the idea was to enable *ddmin* to detect new faults rather than reducing the previously known faults. In this respect, they first generated a random test suite on ''SpiderMonkey'' tool. Later on, they reduced the test suite's size based on coverage criteria like statement and branch. They also applied mutation score for evaluating their approach on ''YAFFS2'' [37], an open-source NAND flash file system. They showed that their approach, when executed on SpiderMonkey, a JavaScript engine results in better fault detection. Our work, on the other hand, is based on the reduction of manually generated test suites available in npm packages to find a smaller set of test cases that achieve the same mutation score as that of an original test suite.

C. TEST CASE SELECTION

Test case selection, given two versions of the software under test and the original test suite T, aims at selecting those test cases in a test suite T, which executes the altered parts of the software under test. However, unlike test suite minimization, the selected test cases are not removed from the test suite. Romano et al. is one of the recent works done on test case selection of Java programs [38]. Interestingly, they also used faulty versions of the original programs (mutants) for injecting faults. Their approach outperformed the baseline techniques in almost 50% of the studied examples. Yoo and Harmann observed that meta-heuristics search techniques are better suited for multi-objective test case selection [39].

III. PRELIMINARIES

In order to be self-contained, we define the related regression software testing concepts required for understanding the rest of the paper.

Rothermel et al. [7] formally defined *test suite reduction* as follows: Given a test suite T for a program Π, a set of test case requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired test coverage of the program, and subsets of T, i.e., $T_1, T_2 \ldots, T_n$, one associated with each of the $r_i s$ such that any one of the test cases t_j belonging to T_i can be used to test r_i. The problem is to find a representative set of test cases from T that satisfies all $r_i s$. The optimal test suite reduction is the one that contains at least one test case requirement t_j from each subset T_i. The problem is considered analogous to the finding of the minimal hitting-set problem. Considering the fact that this itself is an NP-complete problem, different heuristics have been suggested to reduce software maintenance costs [2].

The key objective of these heuristics is two folds: First, they must include ''essential'' test cases in the reduced test suite. This is needed as an essential test case is the one that, if removed from the original test suite, T, the representative test suite, T_{rep} can not meet all test case requirements $r_i s$. In other words, an essential test case must be a part of every minimal representative set T_{rep}. Second, such heuristics must ensure the removal of ''redundant'' test cases. This is due to the fact that if a *redundant* test case, t_r is removed from the original test suite, T, the representative test suite, T_{rep} can still meet all test case requirements $r_i s$. Therefore, removing redundant test cases will not affect the fault detection capability of the representative test suite, T_{rep}.

Most of these heuristics are evaluated on code coverage criterion. For code coverage like statement or method coverage, we are interested in knowing how many statements or methods of a program Π are executed using a test suite T in relation to the total number of statements or methods. Besides these classical *coverage measures*, the mutation score of a test suite can also be used as a quality measure for test suites [28]. As a matter of fact, 74% of the previous studies employed mutation-score to assess independent and dependant variables such as cost for evaluating different approaches [19]. **Mutation testing** is a fault-based testing technique that evaluates test suites T via the injection of faults in the subject program *Pi* using mutation operators, thereby making different copies of the original program called mutants [40], [41]. For example, have a look at Fig. 1 showing a snippet from an original *UUID* NPM package. After applying *math mutation operator*, we see the altered version of the program in Fig. 2.

The mutants are executed using the test suite T and classified accordingly to the outcome. If there exists a test case t in T where a mutant Π^M computes an unexpected output, i.e., fails, Π^M is said to be killed. Otherwise, a mutant is said to survive. The mutation score measures how many mutants are

```
1: for var i=0; i<N; i++ do
2:     M[i] = newArray(16);
3:     for var j=0; j<16; j++ do
4:         M[i][j] = bytes[i * 64 + j * 4] << 24|
5:         bytes[i * 64 + j * 4 + 1] << 16|
6:         bytes[i * 64 + j * 4 + 2] << 8|
7:         bytes[i * 64 + j * 4 + 3];
8:     end for
9: end for
```

FIGURE 1. UUID – original program snippet.

```
1: for var i=0; i<N; i++ do
2:     M[i] = newArray(16);
3:     for var j=0; j<16; j++ do
4:         M[i][j] = bytes[i * 64 + j * 4] << 24|
5:         bytes[i * 64 + j * 4 − 1] << 16|
6:         bytes[i * 64 + j * 4 + 2] << 8|
7:         bytes[i * 64 + j * 4 + 3];
8:     end for
9: end for
```

FIGURE 2. UUID – Mutated program snippet.

killed using the test suite T and is defined as:

$$m(\Pi, \mathrm{T}) = \frac{number\ of\ killed\ mutants}{number\ of\ mutants}. \tag{1}$$

This definition is ambiguous because of several issues regarding the number of mutants to be counted. First, we have mutants that do not change the behavior of a program. Such mutants are called equivalent mutants. Second, we may have mutants that cannot compile or may lead to runs that are terminated due to given execution time limits. In all these cases, we may not consider the corresponding mutants to be counted. Therefore, we introduce the concept of *discarded mutants* that comprises all mutants that cannot be compiled or terminated due to exceeding execution time limits. In this paper, we define the *number of mutants* as the *total number of mutants* minus the *discarded mutants*, ignoring the effect of equivalent mutants. Hence, from here on $m(\Pi, T)$ is assumed to deliver the mutation score considering discarded mutants.

For different programming languages, there are language-specific mutation tools available. In our case, we focus on JavaScript applications and make use of the Mutode tool, which employs 48 mutation operators for generating ''mutants'' of the various NPM packages for our experimental evaluation [12]. The mutation testing incurs high costs as it requires to re-execute all mutants whenever any update is made to the software under test. Therefore, we generate *mutation matrix* to store the output of each mutant upon execution of a test suite. This matrix needs to be computed once for each version of JavaScript package, which reduces the execution time of computing mutation coverage for each algorithm. In addition, the mutation matrix can be used to determine the efficacy of the test suite across multiple versions of the same package.

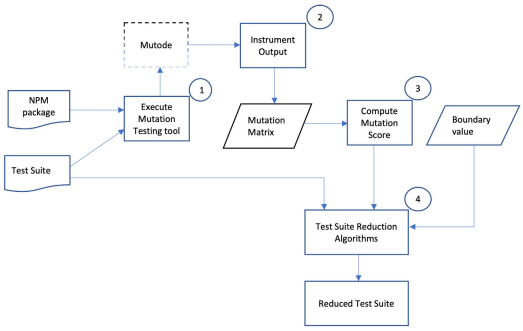

FIGURE 3. The methodology flow diagram.

A. EVALUATION METRICS

Previously, two evaluation metrics have been studied for comparing traditional test suite minimization algorithms: Test suite size reduction TS_{red} and Fault detection loss TF_{red} [6] and [28]. We can represent the reduction in the test suite size when applying a test suite minimization algorithm as follows:

$$TS_{red} = \frac{|T|\text{-}|T'|}{|T|} * 100. \tag{2}$$

where $|T|$ stands for the size of the original test suite, and $|T'|$ denotes the size of the reduced test suite. We can measure fault detection in a similar way:

$$TF_{red} = \frac{|M|\text{-}|M'|}{|M|} * 100. \tag{3}$$

where $|M|$ represents mutation score of the original test suite T, and $|M'|$ denotes the mutation score of the reduced test suite T' of the same program Π, i.e., $M = m(\Pi, T)$, and $M' = m(\Pi, T')$.

B. METHODOLOGY

This section describes an overview of the methodology depicted by Fig. 3. The process starts by executing a third-party mutation testing tool named Mutode on a test suite with a given NPM package. As described earlier, the mutation testing tool generates mutants of the underlying program. Thereafter, the original program is executed on each of these mutants. The output of the mutation testing tool is further instrumented to generate a mutation matrix that contains information regarding the number of faults killed by each mutant. Table 3 contains the time taken by the tool to generate a mutation matrix for each studied subject. The next step is to compute the mutation score for each test suite describing the number of faults detected by the original test suite. The only input required for this step is the mutation matrix generated in the previous step. Finally, the test suite minimization algorithms are supplied with three inputs: mutation matrix, boundary value, and the original test suite. The details about the algorithms are explained in the next section IV. The final output of each algorithm is a reduced test suite. The algorithms are evaluated based on two evaluation metrics: the size of the reduced test suite and the fault-detection loss of the reduced test suite.

IV. TEST SUITE MINIMIZATION ALGORITHMS

After describing preliminaries, we discuss five algorithms for test suite minimization. The first algorithm was suggested by Harrold et al. that ensures that all essential test cases are included in the representative test suite [14]. Next, we have a greedy algorithm that comes up with a minimized test suite using local means for optimization [15]. Hence, the greedy algorithm does not guarantee finding a minimal test suite, but also, the others do not come up with such a guarantee. The third algorithm is the delayed greedy algorithm, as it delays the greedy selection [16]. Since all previously mentioned algorithms do not provide any guarantee that the resulting representative test suite has the same fault detection capability as that of the original test suite, we study the impact of tolerance in the desired mutation coverage criterion during test suite minimization. Further, we also suggest two additional algorithms: the first is search-based and employs random selection; The second is the modified version of the Delta-Debugging algorithm by [17]. For both algorithms, we also allow that the reduced test suite has a (slightly) smaller mutation score, where we use the parameter α for stating the maximal accepted mutation score difference. The first three algorithms have already been discussed for coverage-based test suite minimization and serve as baseline algorithms.

A. HARROLD ET AL. *HEURISTIC*

Harrold et al. suggested a heuristic that aims at selecting "essential" test cases first by grouping all requirements into subsets of increasing cardinality [14]. Fig. 4 explains the original heuristic where requirements represent mutants generated for a program Π. First, test sets T_l containing all test cases that kill a certain mutant are generated from the mutation matrix (lines 2-6). Next, all single element test sets T_l are added to T_m as they select the test cases that can satisfy a maximum number of requirements of cardinality $k = 1$ (lines 7-10). This is followed by the selection of test cases that can meet the maximum number of unsatisfied requirements (non-killed mutants) of the next higher cardinality (lines 13-15). The selection is continued till either maximum cardinality is achieved or desired mutation score is obtained. In case two test cases meet the same requirement (same mutant) of cardinality k, the decision is made by selecting the test case which also satisfies the requirements of cardinality k+1. However, if a test set T_l of max cardinality is selected, the *MaxCardinality* is reset to the maximum cardinality of the remaining sets (lines 19-27). This way, the heuristic iteratively selects the test cases required to meet the remaining unsatisfied requirements, thereby reducing the test suite size. The time complexity of the algorithm will be $O((|s|+|m|)|s|*c)$ where $|s|$ represents the size of the test suite, $|m|$ denotes the number of mutants (testing requirements), and c stands for the maximum cardinality.

```
1:  procedure Harrold et al.(T = {t_1, ..., t_n}, a program Π, a function m, and a boundary value α )
2:      Let maxRR ← m(Π, T).
3:      Let U be the empty set.
4:      for Each mutant m_t of Π do
5:          Compute T_l containing all test cases t_{1..n} that kill mutant m_t.
6:          Add T_l to the set U.
7:      end for
8:      Let T_m be the empty set.
9:      compute MaxCard ← T_l.
10:     T_m = ∪T_l s.t. k = 1.
11:     Mark all test sets T_l containing elements in T_m.
12:     CurrCard = 1.
13:     while CurrCard ≠ MaxCard or (m(Π, T_m)+ α < maxRR) do
14:         CurrCard = CurrCard + 1.
15:         while there exists T_l in U of CurrCard that are not marked do
16:             List ← testcases t_l ∈ T_l.
17:             NextTest = SelectNextTest(CurrCard, List).
18:             add NextTest to the T_m.
19:             MayReduce = false.
20:             for Each NextTest ∈ T_l do
21:                 mark T_l containing NextTest.
22:                 if Card(T_l) = MaxCard then
23:                     MayReduce = true.
24:                 end if
25:             end for
26:             if MayReduce then
27:                 MaxCard = MaxCard(T_l) where T_l not marked.
28:             end if
29:         end while
30:     end while
31:     return T_m
32: end procedure
```

FIGURE 4. Harrold et al. – Test suite minimization adapted from [14].

B. Greedy *ALGORITHM*

The idea behind **Greedy** originates from a polynomial time approximation algorithm for computing a set cover [15]. In the set cover problem, we have a finite set of elements called the universe and a set of elements only from the universe. The problem relies upon finding a subset of the set of sets such that all elements of the universe are captured. The set cover problem is known to be NP-complete. The test suite minimization problem considering the mutation score of particular test cases, can be easily mapped to the set cover problem. The universe is the set of mutants to be covered. For each test case t of the test suite T, we add a set to the set of sets if and only if the mutant is killed by t. A set cover is now nothing else than all the test cases required to cover all mutants.

Greedy is depicted in Fig. 5. In the first FOR loop at the beginning, we are computing the set of sets for which we want to obtain the set cover. Afterward, we take the largest element of this set, which covers most of the mutations, add it to the result set T_m, and remove it from the set of sets. We also

1: **procedure** Greedy($T = \{t_1, \ldots, t_n\}$, a program Π, a function $\mathbf{m}$, and a boundary value α)
2: Let $maxRR \leftarrow \mathbf{m}(\Pi, T)$.
3: Let U be the empty set.
4: **for** $t \in T$ **do**
5: Compute all mutations m_t of Π that are killed using t.
6: Add m_t to the set U.
7: **end for**
8: Let T_m be the empty set.
9: **while** U is not empty or $(\mathbf{m}(\Pi, T_m)+\alpha < maxRR)$ **do**
10: Let m_t be the largest set in U.
11: Remove m_t from U and add t to T_m.
12: Remove the elements from m_t in all elements of U. If a set becomes empty, remove this set from U.
13: **end while**
14: **return** T_m
15: **end procedure**

FIGURE 5. Greedy –Test suite minimization adapted from [15].

remove all elements for this set from the other sets to find another set that covers different mutations. The algorithm terminates when we have considered all mutations or the desired mutation score is achieved. The runtime complexity has to be polynomial because we are only iterating over sets of sets.

C. DelayedGreedy *ALGORITHM*

The delayed greedy algorithm in Fig. 6 was presented by [16] and is based on a concept lattice theory [42]. This theory classifies objects based on their attributes, where an object represents a "test case," and the attributes denote the "test requirements" satisfied by that particular test case. Generally, the test requirements are represented by some coverage criterion, like statement coverage. The approach performs several reductions on the input matrix containing test cases and corresponding requirements, thereby delaying the greedy selection of the test cases that meet the most requirements. Hence, the delayed greedy algorithm is a variant of the already discussed greedy algorithm.

In the following, we discuss the differences using an example. For instance, consider four test cases t_1, t_2, t_3 and t_4 required to cover five requirements (mutants): m_1, m_2, m_3, m_4 and m_5. In our context, the test requirements represent "killed mutants" for a particular test case. The following table indicates the mutation covered by each test case setting the context of the greedy search:

	m_1	m_2	m_3	m_4	m_5
t_1			×	×	×
t_2	×		×		
t_3		×		×	
t_4	×		×		

The greedy algorithm **Greedy** selects t_1 as the first test case because t_1 kills more mutants than the other test cases. After removing t_1 and all mutants killed by t_1, the reduced context table consists of three test cases t_2, t_3, and t_4 along with the two mutants m_1 and m_2. Since they all cover one mutant so any one of them will be chosen in random order. In case t_4 gets selected, requirement m_1 will be removed from the context table, leading finally to a reduced test suite $= \{t_1, t_3, t_4\}$. The other possible reduced test suite would be $\{t_1, t_2, t_3\}$.

Let us now have a look at the delayed greedy algorithm, which postpones the greedy selection by performing certain reduction steps. First, *object reduction* is performed. In this reduction step, a test case t_y is killed if there exists another test case t_x, where the requirements of t_x are a superset or equivalent to the requirements of t_y. For example, in our case, t_2 and t_4 share the same mutations. Hence, in accordance with object reduction, we can either eliminate test case t_2 or t_4. In the case of selection t_4, we reduce the context table to:

	m_1	m_2	m_3	m_4	m_5
t_1			×	×	×
t_2	×		×		
t_3		×		×	

In the delayed greedy algorithm, the object reduction is followed by the *attribute reduction*. If a mutant (i.e., a requirement), m_x is killed by a set of test cases that is a subset of or equivalent to a set of test cases covering mutant m_y, then m_y is marked obsolete and removed from the context table. In our example, m_3 is covered by the tests $\{t_1, t_2\}$, and m_4 by $\{t_1, t_3\}$. Because m_1 is covered by $\{t_2\}$ alone, which is a subset of $\{t_1, t_2\}$, we can eliminate m_3. Similarly, we can remove m_4, resulting finally in the following context table:

	m_1	m_2	m_5
t_1			×
t_2	×		
t_3		×	

This context table comprises three test cases, each of which covers exactly one mutant. This is termed the strongest concept. In the last reduction, i.e., the *owner reduction* rule, each test case that covers one mutant only has to be removed from the context table, followed by the removal of all columns comprising the mutants (attributes) killed by the particular test case. Note that in this step, the removed test cases are added to the resulting test suite. After this step, all remaining test cases are minimized using **Greedy**. In our example, owner reduction removes all entries in the context table, leading to a reduced test suite: $= \{t_1, t_2, t_3\}$.

In Fig. 6 we state the pseudo-code of the delayed variant of the greedy test suite minimization algorithm. The **Delayed-Greedy** algorithm first starts with object reduction (lines 9-11), continues with attribute reduction (lines 12-14), and finally applies owner reduction (lines 15-19) before starting with greedy minimization. Traditionally, this process gets repeated until the entire context table is empty. However, we also allow stopping this search if the desired mutation score is obtained.

1: **procedure** DelayedGreedy($T = \{t_1, \ldots, t_n\}$, a program Π, a function $\mathbf{m}$, and a boundary value α)
2: Let $maxRR \leftarrow \mathbf{m}(\Pi, T)$.
3: Let U be the empty set.
4: **for** $t \in T$ **do**
5: Compute all mutations m_t of Π that are killed using t.
6: Add m_t to the set U.
7: **end for**
8: Let T_m be the empty set.
9: **while** U is not empty or $(\mathbf{m}(\Pi, T_m)+\alpha < maxRR)$ **do**
10: **for** each set of mutants covered by test cases t_i and t_j s.t. $M_{ti} \supseteq M_{tj}$ **do**
11: Remove t_j from U
12: **end for**
13: **for** each set of test cases covered by mutants m_i and m_j s.t. $T_{m_i} \subseteq T_{m_j}$ **do**
14: Remove m_j from U
15: **end for**
16: **for** Each mutant m_k that is killed by only one test case (strongest concept) **do**
17: Remove the row for test case t from U.
18: Remove columns for mutants killed by t.
19: add t to T_m.
20: **end for**
21: **if** U is not empty **then**
22: Let m_t be the largest set in U.
23: Remove m_t from U and add t to T_m.
24: Remove the elements from m_t in all elements of U. If a set becomes empty, remove this set from U.
25: **end if**
26: **end while**
27: **return** T_m
28: **end procedure**

FIGURE 6. DelayedGreedy – Delaying greedy search for test suite minimization [16].

D. LinMIN ALGORITHM

The **LinMIN** algorithm depicted in Fig. 7 is a search algorithm for randomly selecting a subset that is smaller by one element in each step. There are four inputs to the algorithm: the program Π, the original test suite T of Π, a function $\mathbf{m}$ returning the mutation score, and the mutation score threshold α.

The **LinMIN** algorithm first unmarks all test cases in a test suite. The while loop in Step 4 iteratively checks all unmarked test cases and whether removing them has no unwanted decline in the mutation score. For this purpose, a selected test case is marked in Step 5 and removed from the test suite in Step 6. In Step 7, **LinMIN** performs the check. If the new mutation score of the reduced test suite, together with the given threshold α is smaller than the original mutation score, the selected test case is added back to the test suite. Through parameter, α, a tester can opt for inadequate test suite reduction by allowing reduced test suite of size with

1: **procedure** LinMIN($T = \{t_1, \ldots, t_n\}$, a program Π, a function $\mathbf{m}$, and a boundary value α)
2: Let $maxRR \leftarrow \mathbf{m}(\Pi, T)$.
3: Unmark all test cases in T.
4: **while** There exists an unmarked test case $t_i \in T$ **do**
5: Mark t_i.
6: Remove t_i from T.
7: **if** $(\mathbf{m}(\Pi, T) + \alpha < maxRR)$ **then**
8: Add t_i to T.
9: **end if**
10: **end while**
11: **return** T
12: **end procedure**

FIGURE 7. LinMIN - A linear search procedure for test suite minimization.

mutation score $\mathbf{m}(\Pi, T) - \alpha$. This leads to a higher reduction in test suite size but also influences the fault-detection loss as depicted in Table 9.

Obviously, **LinMIN** has to terminate because we only consider test suites comprising a finite number of test cases. The worst-case complexity is linear in terms of the number of test cases n, i.e., $O(n)$ because each test case is only checked once for removal when assuming that computing the mutation score can be done within a constant time. Note that the complexity of the function calls $\mathbf{m}(\Pi, T)$ for computing mutation scores is $O(|s| * |m|)$, where $|s|$ represents the size of the test suite, and $|m|$ is the total number of mutants for a program Π. It is worth noting that **LinMIN** not necessarily comes back with the smallest test suite. Hence, for our experimental evaluation, we executed **LinMIN** 10 times and computed average values.

E. DeltaMIN ALGORITHM

The original Delta-Debugging algorithm aimed at reducing the size of a failing test case [17]. It takes a fairly large failing test case as input that causes failure and aims at finding a smaller test case that still triggers the fault. The faulty test case includes all possible changes and aims at finding the minimal change set (configuration) causing the fault. It marks each test as one of three statuses: passing, failing, or unresolved. It starts by dividing the input into two parts and continues to increase the granularity of the search space until the failure-causing input is identified. The target is to find software changes that lead to failures. Thus, automating the "scientific" way of debugging [43]. The approach is based on the *divide and conquer* principle and helps in locating faults in software. The worst-case complexity of the original minimization algorithm is $O(|c|^2)$, where c represents several changes to the program. In the best case, it has the same complexity as that of a binary search algorithm.

In Fig. 8 we illustrate the use of delta debugging for test suite minimization. The **DeltaMIN** algorithm calls the original delta debugging algorithm but tailors the test function used for minimization to fit the purpose. The test function **test** returns pass ($\surd$) if the mutation score of the selected test

1: **procedure** DeltaMIN($T = \{t_1, \ldots, t_n\}$, a program Π, a function $\mathbf{m}$, and a boundary value α)
2: Let $maxRR \leftarrow \mathbf{m}(\Pi, T)$.
3: Define the testing function **test** as follows:

$$\mathbf{test}(ts) = \begin{cases} \times & \textit{if } (\mathbf{m}(\mathbf{set}(ts), \Pi) + \alpha \geq maxRR) \\ \surd & \textit{otherwise} \end{cases}$$

4: **return** ddmin($\langle t_1, \ldots, t_n \rangle$)
5: **end procedure**

FIGURE 8. DeltaMIN – using delta debugging for test suite minimization.

cases is within the expected range considering the mutation score of the original test suite and the threshold α. Otherwise, **test** returns fail ($\times$). With this test function, the original delta debugging algorithm **ddmin** is called for returning a minimized test suite.

It is worth noting that there will be a reduction of at least one test case if possible due to the properties of **ddmin**. Moreover, **DeltaMIN** has to terminate and the same worst-case complexity as **ddmin**, neglecting the time required for computing the mutation score using the function *m*. However, in practice computing, the mutation score might be a limiting factor. Note that **ddmin** and, therefore, also **DeltaMIN** come with no guarantees regarding finding the globally minimal test suite. Hence, we also have to consider several runs in the experimental evaluation.

V. EXPERIMENTAL EVALUATION

In this section, we discuss the setup and results of an experimental evaluation to answer the previously mentioned three research questions. We first discuss the setup comprising the details regarding the used subject programs and the tools used for mutation. Afterward, we discuss obtained results.

A. SUBJECT DETAILS

It is interesting to note that nearly all of the previous work done on regression testing makes use of either C or Java benchmark programs for evaluation [4], [33]. Thus, there is an acute shortage of case studies about the viability of regression testing approaches in the context of JavaScript-based applications. Particularly, we were interested in evaluating the redundancy in test suites supplied with various open source *Node.js* and *NPM* packages. Since *NPM* is the largest software repository comprising more than 700,000 packages with an average of 5 billion weekly downloads [44], this demands stringent testing of these packages to avoid faults.

Before discussing the details about the subject packages, we describe the overall experimental setup: In the first step, we computed the statement coverage and the time taken by the accompanied test suites of studied NPM packages as reported in Table 1. As a next step, we executed a mutation testing tool, namely *Mutode* on the subject programs, and reported mutation details such as mutation score and the time to run all mutants on a given test suite in Table 3. Note that for our *UUID* example, the original test suite took 65 ms while the mutation testing tool took 44 minutes to execute 2,872 mutants on the same test suite. It is due to this high execution cost that mutation testing tools are not widely practiced in the industry. We try to reduce this cost by storing the result of the mutation testing tool in a fault matrix (CSV file) containing the result of executing all test cases against respective mutants. This way, we know exactly which mutant (fault) was killed by which test case. This fault matrix becomes the input for the test minimization algorithms. Hence, the time mentioned in Tables 6, 7, 8, 9 and 10 is the execution time of corresponding minimization algorithms.

In Table 1 we provide statistical information regarding our subject programs denoted by *Program*: *UUID* is a *Node.js* package for generating "Universally Unique Identifier". It has around 26 different versions with "37,796,670" weekly downloads [45]. *Debug* is a small debugging utility for Node.js [46]. *Body Parser* is a *Node.js* middleware for parsing req.body property [47]. *Express* is a popular web framework for building *Node.js* web applications with "14,181,677" weekly downloads [48]. *Passport* is authentication middleware for Express-based Node.js applications [49]. *Cheerio* is an npm package for parsing DOM model [50]. *ShortId* generates unique ids. These "URL-friendly" ids are used to keep track of log messages [51]. *Async* is a utility package for asynchronous JavaScript [52].

The specific version of the studied NPM package is denoted by *Version*. The original test suite size is denoted by $|T|$. Similarly, the statement coverage attained by the coverage tool named *NYC* is shown as *Cov* [53].

TABLE 1. Subject programs' details.

Program	*Version*	*LOC*	$\|T\|$	*Cov* (%)	*weekly downloads*
UUID	3.2.1	471	17	92.3	37,796,670
debug	3.1.0	616	4	68.6	96,302,522
body-parser	1.16.0	864	204	100.0	14,566,529
body-parser	1.18.2	919	218	99.7	14,566,529
express	4.15.0	3,965	838	98.6	14,181,677
express	4.16.3	4,051	858	98.6	14,181,677
passport	0.4.1	1,201	494	98.9	869,109
cheerio	1.0.0	3,289	650	98.6	4,483,242
shortid	2.2.16	306	17	94.3	1,182,155
async	2.6.0	4,767	511	98.4	31,751,368

B. FAULT INJECTION

For the experiments, we employed the Mutode tool [12] for seeding faults into our subject programs. Mutode is a general-purpose, open-source mutation testing tool for Node.js and NPM packages. It supports 43 mutation operators, including mutations for boolean, string, and numeric literals, in addition to arithmetic, relational, conditional, shift, logical, assignment, and deletion operators. A brief description of the Mutode mutation operators is mentioned in Table 2. Further details about each mutation operator can be found in [12].

Furthermore, in Table 3, we summarize information regarding the mutation score *M* using the Mutode tool. There are the specific versions of the NPM packages (*Version*), the

TABLE 2. Description of used mutation operators.

Mutationoperator	*Description*
BL	Boolean Literals
CB	Conditionals Boundary
I	Increments
IN	Invert Negatives
M	Math
NC	Negate Conditionals
NL	Numeric Literals
RAE	Remove Array Elements
RC	Remove Conditionals
RFCA	Remove Function Call Arguments
RFDP	Remove Function Parameters
RF	Remove Functions
RL	Remove Lines
ROP	Remove Object Properties
RSC	Remove Switch Cases
SL	String Literals

TABLE 3. Original mutation details obtained using Mutode on our subject programs.

Program	*Version*	$\|T\|$	*Time*	*M*	*Tot.*	*Kill.*	*Sur.*	*Dis.*
			(h:m:s:ms)	(%)				
UUID	3.2.1	17	0:43:53:0	92.7	2872	2644	206	22
debug	3.1.0	4	0:20:29:0	3.6	1284	45	1177	62
body-parser	1.16.0	204	0:0:26:22	77.3	858	537	157	164
body-parser	1.18.2	218	0:0:20:13	74.4	889	535	184	170
express	4.15.0	833	4:0:13:0	76.1	3411	2233	698	480
express	4.16.3	858	2:36:0:0	73.2	3216	2004	733	479
passport	0.4.1	494	0:29:33:0	86.1	759	412	66	281
cheerio	1.0.0	650	2:8:0:0	87.9	2861	2343	320	198
shortid	2.2.16	17	0:27:59:0	54.1	335	132	112	91
async	2.6.0	511	6:51:0:0	81.5	3551	1109	251	2191

original test suite size ($|T|$), the time required to execute Mutode (*Time*), the number of generated, killed, survived, and discarded mutants (*Tot.*, *Kill.*, *Sur.* and *Dis.* respectively) are given.

As already mentioned, we computed the mutation score not considering equivalent mutants but discarded mutants. This differs from the Mutode tool, which explains observed differences in resulting figures. Moreover, in contrast to Mutode, which assumes a test case causes a time out if execution takes longer than 2.5 times the execution time of the original program, we considered such a time limit for each of the subject programs that are at least 50 seconds. In Table 4 we depict the time limits used in the experimental evaluation.

C. RESULTS OBTAINED

Before explaining our results, we summarize key parameters studied in previous empirical studies in Table 5. Most of these prior studies employed either *statement coverage* or *method coverage*. Further, these studies were conducted mostly on Java programs where the typical evaluation metrics were *test suite size* or *execution time*.

All experiments encompassing minimization algorithms were run on a MacBook Pro with Apple M1 Pro and 16 GB memory running the operating system Monterey (version 12.1). The Mutode tool was executed using a Samsung notebook 7 spin (model 740U3M), with 2.54 GHz Intel i5-7200U and 12 GB RAM. We made the source code of the NPM packages along with the test suite available on GitHub [54] for future research and assuring that the results can be reproduced.

TABLE 4. Time out limits used in the experimental evaluation.

Program	*Time out limit (s)*
UUID-3.2.1	10
debug-3.1.0	10
body-parser-1.16.0	20
body-parser-1.18.2	10
express-4.15.0	20
express-4.16.3	20
passport-0.4.1	50
cheerio-1.0.0	50
shortid-2.2.16	50
async-2.6.0	130

To answer the first research question RQ1: "Which test suite minimization algorithm performs better in reducing the test suite size?", we applied five test suite minimization algorithms discussed in Section IV to compare their reduction capabilities.

In Tables 6, 7, 8, 9 and 10, we give the test suite reduction achieved by the redundancy-elimination algorithm **Harrold et al.**, **Greedy**, **DelayedGreedy**, **LinMin**, and **DeltaMin** where *Min* represents the size and time taken by the minimum test suite for a particular *alpha*. The reduced test suite size is labeled by $|T'|$. Similarly, *Max* denotes the maximum test suite size and the corresponding time achieved during ten random samples. Likewise, *Avg* represents the average test suite's size and time over ten runs, and $s.d.$ stands for the standard deviation of the test suite's size. Further, TS_{red} denotes the reduction percentage in test suite size, and TF_{red} represents the percentage loss in the fault-detection capability of the reduced test suite. We summarize the obtained average values for all algorithms in Table 11.

From the results, we see the following for $\alpha = 0$: **Greedy**, **DelayedGreedy**, **Harrold et al.**, **LinMIN** and **DeltaMIN** obtained the same or similar reduction on average. The difference between the average test suite reduction values is only minor. Hence, there seems to be no clear evidence that one of the five algorithms performs better for minimization because fault-detection loss remains the same.

For $\alpha = 5$: **Greedy** results in the highest reduction in test suite size as depicted by Figure 10, but when comparing **Harrold et al.** with **LinMIN** and **DeltaMIN** regarding the smallest reduced test suite and fault-detection loss, there is no clear difference. The lowest reduction is observed for **DelayedGreedy**, namely, 65%, However, on average **DelayedGreedy** provided the minimum fault-detection loss as shown in Figure 9.

By increasing the tolerance value to $\alpha = 10$, the test suite size reduction does not change in case of **DelayedGreedy**, **Harrold et al.**, **LinMIN**, and **DeltaMIN** but the fault detection loss is almost doubled for **Harrold et al.**, **LinMIN**, and **DeltaMIN**. By increasing the tolerance value, the **Greedy**

TABLE 5. Key attributes studied in previous studies.

Empirical Study	Algorithm Studied							Coverage criteria			Subject Program		Metrics	
	GR	GRE	DGR	HGS	ILP	GA	$2Opt$	$Statement$	$Method$	$Mutation$	$Java$	C	$TSSize$	$TSExec.Time$
Shi et al. [28]	✓	✓		✓	✓			✓		✓	✓		✓	✓
Zhang et al. [6]	✓	✓		✓	✓			✓	✓	✓	✓		✓	✓
Zhong et al. [22]		✓		✓	✓	✓						✓	✓	✓
Smith and Kapfhammer [27]	✓		✓	✓			✓				✓		✓	✓

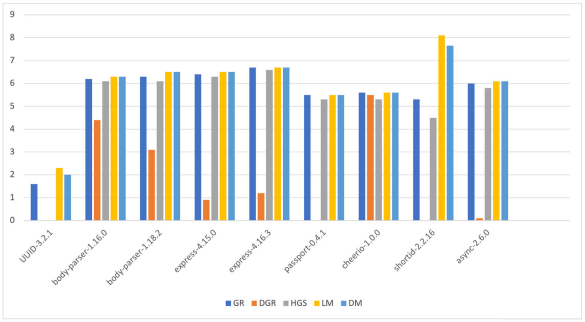

FIGURE 9. Average % reduction in fault-detection loss for alpha = 5.

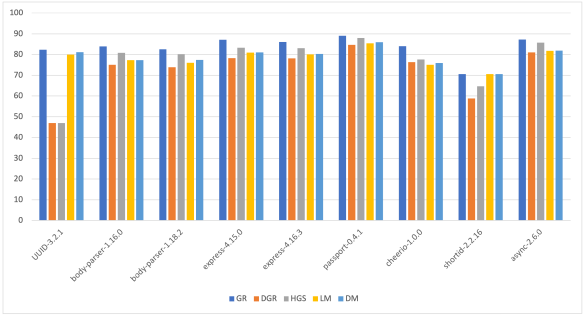

FIGURE 10. Average % reduction in test suite size for alpha = 5.

algorithm results in a higher reduction in test suite size along with higher fault detection loss. However, **DelayedGreedy** has the smallest fault detection loss among all algorithms.

The obtained results can be used to answer RQ1 as follows. Considering that **DelayedGreedy** has the highest average test suite reduction for $\alpha = 0$, acceptable reductions for the other tolerance values, and always the smallest fault detection loss, we justify the superiority of the **DelayedGreedy** test suite minimization algorithm.

To answer the second research question *RQ2: "Which test suite minimization algorithm performs better in the context of execution time?"*, we look at the execution time of all five algorithms given in Table 11 where the average runtime t is given in milliseconds. If we compare the average execution time of all five algorithms,**Greedy** and **DelayedGreedy** seem to perform equally well. Interestingly, **LinMIN** performs better than **Harrold et al.** in terms of execution time with similar test suite size reduction and fault-detection loss. Hence, research question *RQ2* can be answered as follows: *Greedy* is the fastest algorithm, followed by **DelayedGreedy**, **LinMIN**, **Harrold et al.** and **DeltaMIN**. The reason behind the larger amount of time

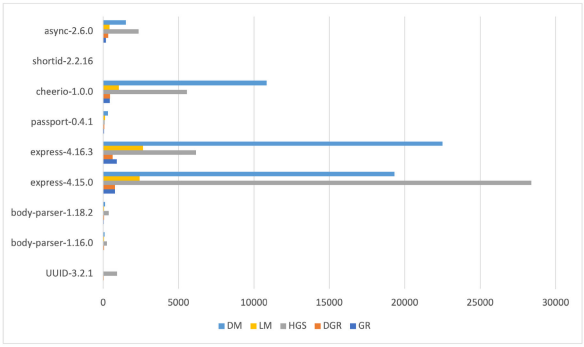

FIGURE 11. Average execution time (ms) for alpha = 5.

TABLE 6. Test suite minimization results for ***Harrold et al.*** approach.

					Min		Max		Avg			s.d.	TS_{red}	TF_{red}
$Program$	$Ver.$	$\|T\|$	M	α	$\|T'\|$	t (ms)	$\|T'\|$	t (ms)	$\|T'\|$	t (ms)	M'			
UUID	3.2.1	17	92.7	0	9	893	9	958	9.0	920.9	92.7	0.0	47.0	0.0
UUID	3.2.1	17	92.7	5	9	905	9	937	9.0	918.2	92.7	0.0	47.0	0.0
UUID	3.2.1	17	92.7	10	9	895	9	925	9.0	905.4	92.7	0.0	47.0	0.0
debug	3.1.0	4	3.6	0	1	144	1	148	1.0	145.9	3.6	0.0	75.0	0.0
debug	3.1.0	4	3.6	5	1	143	1	147	1.0	145.5	3.6	0.0	75.0	0.0
debug	3.1.0	4	3.6	10	1	145	1	149	1.0	146.6	3.6	0.0	75.0	0.0
body-parser	1.16.0	204	77.3	0	54	379	56	545	54.8	452.7	77.3	0.6	73.1	0.0
body-parser	1.16.0	204	77.3	5	39	162	39	444	39.0	272.8	72.6	0.0	80.8	6.1
body-parser	1.16.0	204	77.3	10	33	112	33	253	33.0	172.1	68.3	0.0	83.8	11.6
body-parser	1.18.2	218	74.4	0	60	425	60	781	60.0	545.5	74.4	0.0	72.4	0.0
body-parser	1.18.2	218	74.4	5	43	181	43	461	43.0	304.6	69.8	0.0	80.2	6.1
body-parser	1.18.2	218	74.4	10	38	136	38	408	38.0	239.3	65.2	0.0	82.5	12.3
express	4.15.0	833	76.1	0	188	33044	193	52620	190.1	42450.6	76.1	1.4	77.1	0.0
express	4.15.0	833	76.1	5	138	18029	139	55371	138.6	28378.7	71.4	0.8	83.3	6.2
express	4.15.0	833	76.1	10	123	7393	123	18548	123.0	11414.5	66.5	0.0	85.2	12.6
express	4.16.3	858	73.2	0	196	9964	201	12826	198.2	11703.0	73.2	1.4	76.9	0.0
express	4.16.3	858	73.2	5	145	4265	146	9667	145.9	6157.4	68.3	0.4	83.0	6.6
express	4.16.3	858	73.2	10	129	1836	130	4393	129.4	2748.5	63.4	0.4	84.9	13.3
passport	0.4.1	494	86.1	0	79	106	81	145	79.4	122.0	86.1	0.6	83.9	0.0
passport	0.4.1	494	86.1	5	58	90	60	112	59.3	100.6	81.5	0.6	87.9	5.3
passport	0.4.1	494	86.1	10	50	81	52	107	51.2	93.7	76.8	0.6	89.6	10.8
cheerio	1.0.0	650	87.9	0	186	9927	189	19745	187.2	13230.2	87.9	0.8	71.2	0.0
cheerio	1.0.0	650	87.9	5	145	3622	145	9416	145.0	5548.4	83.3	0.0	77.6	5.3
cheerio	1.0.0	650	87.9	10	136	1105	135	2205	136.0	1530.1	78.6	0.0	79.0	10.6
shortid	2.2.16	17	54.1	0	7	8	7	9	7.0	8.3	54.1	0.0	58.8	0.0
shortid	2.2.16	17	54.1	5	6	7	6	8	6.0	7.6	51.6	0.0	64.7	4.5
shortid	2.2.16	17	54.1	10	6	7	6	8	6.0	7.9	51.6	0.0	64.7	4.5
async	2.6.0	511	81.5	0	100	3409	103	7034	101.4	5176.6	81.5	0.9	80.1	0.0
async	2.6.0	511	81.5	5	72	1602	73	3273	72.9	2339.4	76.7	0.3	85.7	5.8
async	2.6.0	511	81.5	10	63	1095	64	2038	63.3	1363.4	71.6	0.4	87.6	12.0

required for **Harrold et al.** (HGS) and **DeltaMIN** (DM) may rely on the larger number of mutations to be generated. It is worth noting that **Greedy** is faster than **DelayedGreedy** with two exceptions for program *express*. This might be due to the test suite size, which is large compared with the other test suites. Hence, in these two cases delaying the greedy step and applying reductions lead to faster computation. Similarly, **Harrold et al.** take a longer time in examples with larger test suite size as it generates subsets of all possible cardinalities. If we look at the average execution time, there is a visible decrease both in **Harrold et al.** and **DeltaMIN** by increasing the tolerance value while the average time taken by **LinMIN** remains the same as depicted in Figure 11. Therefore, **DelayedGreedy** should be preferred over **Harrold et al.** for larger test suites.

TABLE 7. Test suite minimization results for *Greedy*.

					Min		Max		Avg			s.d.	TS_{red}	TF_{red}
$Program$	$Ver.$	$\|T\|$	M	α	$\|T'\|$	t	$\|T'\|$	t	$\|T'\|$	t	M'			
						(ms)		(ms)		(ms)				
UUID	3.2.1	17	92.7	0	9	12	9	16	9.0	13.4	92.7	0.0	47.0	0.0
UUID	3.2.1	17	92.7	5	3	11	3	12	3.0	11.3	91.2	0.0	82.3	1.6
UUID	3.2.1	17	92.7	10	2	9	2	11	2.0	10.1	85.1	0.0	88.2	8.2
debug	3.1.0	4	3.6	0	1	2	1	4	1.0	2.6	3.6	0.0	75.0	0.0
debug	3.1.0	4	3.6	5	1	2	1	3	1.0	2.4	3.6	0.0	75.0	0.0
debug	3.1.0	4	3.6	10	1	2	1	3	1.0	2.2	3.6	0.0	75.0	0.0
body-parser	1.16.0	204	77.3	0	55	33	56	36	55.4	33.7	77.3	0.4	72.8	0.0
body-parser	1.16.0	204	77.3	5	32	29	34	31	32.8	30.0	72.5	0.8	83.9	6.2
body-parser	1.16.0	204	77.3	10	23	27	24	29	23.4	28.1	67.6	0.4	88.5	12.5
body-parser	1.18.2	218	74.4	0	63	38	64	41	63.8	38.7	74.4	0.4	70.7	0.0
body-parser	1.18.2	218	74.4	5	38	33	38	35	38.0	33.6	69.6	0.0	82.5	6.3
body-parser	1.18.2	218	74.4	10	26	30	26	31	26.0	30.5	64.6	0.0	88.0	13.0
express	4.15.0	833	76.1	0	183	1067	184	1148	183.3	1116.7	76.1	0.4	77.9	0.0
express	4.15.0	833	76.1	5	106	786	108	807	107.0	794.5	71.2	0.4	87.1	6.4
express	4.15.0	833	76.1	10	75	639	76	749	75.3	664.8	66.2	0.4	90.9	13.0
express	4.16.3	858	73.2	0	193	1202	194	1209	193.8	1205.9	73.2	0.4	77.4	0.0
express	4.16.3	858	73.2	5	119	894	120	945	119.5	909.2	68.2	0.5	86.1	6.7
express	4.16.3	858	73.2	10	88	744	88	754	63.0	88.0	63.3	0.0	89.7	13.5
passport	0.4.1	494	86.1	0	76	60	78	63	76.7	60.8	86.1	0.6	84.4	0.0
passport	0.4.1	494	86.1	5	53	56	54	58	53.4	56.4	81.3	0.4	89.1	5.5
passport	0.4.1	494	86.1	10	38	52	39	54	38.6	52.8	76.4	0.4	92.1	11.3
cheerio	1.0.0	650	87.9	0	186	547	188	558	187.0	551.9	87.9	0.7	71.2	0.0
cheerio	1.0.0	650	87.9	5	103	421	104	440	103.6	427.9	83.0	0.4	84.0	5.6
cheerio	1.0.0	650	87.9	10	72	359	73	376	72.6	364.2	78.1	0.4	88.8	11.2
shortid	2.2.16	17	54.1	0	7	2	7	4	7.0	2.6	54.1	0.0	58.8	0.0
shortid	2.2.16	17	54.1	5	5	1	5	2	5.0	1.7	51.2	0.0	70.5	5.3
shortid	2.2.16	17	54.1	10	3	1	3	2	3.0	1.4	45.9	0.0	82.3	15.1
async	2.6.0	511	81.5	0	104	213	106	217	104.7	214.8	81.5	0.6	79.5	0.0
async	2.6.0	511	81.5	5	64	182	66	189	65.1	185.1	76.6	0.5	87.2	6.0
async	2.6.0	511	81.5	10	47	166	47	173	47.0	168.2	71.7	0.0	90.8	12.0

TABLE 8. Test suite minimization results for *DelayedGreedy*.

					Min		Max		Avg			s.d.	TS_{red}	TF_{red}
$Program$	$Ver.$	$\|T\|$	M	α	$\|T'\|$	t	$\|T'\|$	t	$\|T'\|$	t	M'			
						(ms)		(ms)		(ms)				
UUID	3.2.1	17	92.7	0	9	38	9	40	9	39.3	92.7	0.0	47.0	0.0
UUID	3.2.1	17	92.7	5	9	39	9	41	9	40.0	92.7	0.0	47.0	0.0
UUID	3.2.1	17	92.7	10	9	39	9	40	9	39.3	92.7	0.0	47.0	0.0
debug	3.1.0	4	3.6	0	1	64	1	67	1	65.0	3.6	0.0	75.0	0.0
debug	3.1.0	4	3.6	5	1	117	1	120	1	118.6	3.6	0.0	75.0	0.0
debug	3.1.0	4	3.6	10	1	117	1	120	1	118.4	3.6	0.0	75.0	0.0
body-parser	1.16.0	204	77.3	0	54	69	54	70	54	69.6	77.3	0.0	73.5	0.0
body-parser	1.16.0	204	77.3	5	51	56	51	57	51	56.3	73.9	0.0	75.0	4.4
body-parser	1.16.0	204	77.3	10	48	53	48	54	48	53.8	72.0	0.0	76.4	6.8
body-parser	1.18.2	218	74.4	0	60	73	60	75	60	74.1	74.4	0.0	72.4	0.0
body-parser	1.18.2	218	74.4	5	57	61	57	63	57	61.6	72.0	0.0	73.8	3.1
body-parser	1.18.2	218	74.4	10	51	57	51	60	51	58.3	68.8	0.0	76.6	7.4
express	4.15.0	833	76.1	0	182	812	182	834	182	825.8	76.1	0.0	78.1	0.0
express	4.15.0	833	76.1	5	180	782	180	794	180	787.3	75.4	0.0	78.3	0.9
express	4.15.0	833	76.1	10	147	748	147	772	147	755.6	66.4	0.0	82.3	12.8
express	4.16.3	858	73.2	0	189	659	189	702	189	671.0	73.2	0.0	77.9	0.0
express	4.16.3	858	73.2	5	187	628	187	650	187	635.2	72.2	0.0	78.2	1.2
express	4.16.3	858	73.2	10	155	595	155	614	155	603.6	65.2	0.0	81.9	10.8
passport	0.4.1	494	86.1	0	76	74	76	79	76	76.0	86.1	0.0	84.6	0.0
passport	0.4.1	494	86.1	5	76	74	76	83	76	76.3	86.1	0.0	84.6	0.0
passport	0.4.1	494	86.1	10	67	70	67	74	67	70.9	76.3	0.0	86.4	11.4
cheerio	1.0.0	650	87.9	0	183	474	183	492	183	479.3	87.9	0.0	71.8	0.0
cheerio	1.0.0	650	87.9	5	153	443	153	457	153	447.1	83.1	0.0	76.4	5.5
cheerio	1.0.0	650	87.9	10	153	443	153	453	153	446.6	83.1	0.0	76.4	5.5
shortid	2.2.16	17	54.1	0	7	7	7	8	7	8.4	54.1	0.0	58.8	0.0
shortid	2.2.16	17	54.1	5	7	7	7	9	7	7.9	54.1	0.0	58.8	0.0
shortid	2.2.16	17	54.1	10	7	7	7	9	7	7.7	54.1	0.0	58.8	0.0
async	2.6.0	511	81.5	0	98	339	98	350	98	343.4	81.5	0.0	80.8	0.0
async	2.6.0	511	81.5	5	97	327	97	335	97	329.7	81.4	0.0	81.0	0.1
async	2.6.0	511	81.5	10	97	326	97	341	97	332.1	81.4	0.0	81.0	0.1

TABLE 9. Test suite minimization results for *LinMIN*.

					Min		Max		Avg			s.d.	TS_{red}	TF_{red}
$Program$	$Ver.$	$\|T\|$	M	α	$\|T'\|$	t	$\|T'\|$	t	$\|T'\|$	t	M'			
						(ms)		(ms)		(ms)				
UUID	3.2.1	17	92.7	0	9	13	9	15	9.0	14.1	92.7	0.0	47.0	0.0
UUID	3.2.1	17	92.7	5	3	13	6	14	3.4	13.5	90.5	0.9	80.0	2.3
UUID	3.2.1	17	92.7	10	2	13	2	14	2.0	13.1	85.1	0.0	88.2	8.2
debug	3.1.0	4	3.6	0	1	8	1	14	1.0	9.5	3.6	0.0	75.0	0.0
debug	3.1.0	4	3.6	5	0	8	0	10	0.0	8.5	0.0	0.0	100.0	100.0
debug	3.1.0	4	3.6	10	0	8	0	11	0.0	9.1	0.0	0.0	100.0	100.0
body-parser	1.16.0	204	77.3	0	54	51	58	55	55.5	52.6	77.3	1.1	72.7	0.0
body-parser	1.16.0	204	77.3	5	37	50	50	57	46.2	52.0	72.4	3.6	77.3	6.3
body-parser	1.16.0	204	77.3	10	39	50	50	52	43.4	51.0	67.4	3.2	78.7	12.8
body-parser	1.18.2	218	74.4	0	60	59	63	62	61.2	60.5	74.4	0.9	71.9	0.0
body-parser	1.18.2	218	74.4	5	49	56	58	60	52.5	58.3	69.5	2.4	75.9	6.5
body-parser	1.18.2	218	74.4	10	36	56	45	58	40.8	57.4	64.5	2.9	81.2	13.2
express	4.15.0	833	76.1	0	184	2360	190	2518	187.3	2422.0	76.1	1.8	77.5	0.0
express	4.15.0	833	76.1	5	150	2347	166	2485	159.2	2412.5	71.2	4.1	80.8	6.5
express	4.15.0	833	76.1	10	127	2298	145	2435	136.8	2354.4	66.1	4.5	83.5	13.1
express	4.16.3	858	73.2	0	191	2576	196	2776	193.2	2634.6	73.2	1.7	77.4	0.0
express	4.16.3	858	73.2	5	164	2524	179	2748	170.7	2654.9	68.2	5.4	80.1	6.7
express	4.16.3	858	73.2	10	140	2419	157	2599	150.4	2542.1	63.2	4.2	82.4	13.6
passport	0.4.1	494	86.1	0	79	135	81	144	80.2	138.1	86.1	0.7	83.7	0.0
passport	0.4.1	494	86.1	5	68	131	76	140	71.7	134.9	81.3	2.4	85.4	5.5
passport	0.4.1	494	86.1	10	61	132	70	140	65.1	135.3	76.3	2.7	86.8	11.4
cheerio	1.0.0	650	87.9	0	184	1040	188	1099	186.6	1066.4	87.9	1.4	71.2	0.0
cheerio	1.0.0	650	87.9	5	156	1028	169	1095	162.0	1057.2	82.9	4.1	75.0	5.6
cheerio	1.0.0	650	87.9	10	135	991	150	1089	143.6	1041.9	77.9	4.1	77.9	11.3
shortid	2.2.16	17	54.1	0	7	4	7	5	7.0	4.6	54.1	0.0	58.8	0.0
shortid	2.2.16	17	54.1	5	5	4	5	5	5.0	4.2	49.6	0.0	70.5	8.1
shortid	2.2.16	17	54.1	10	3	4	3	5	3.0	4.6	45.0	0.0	82.3	16.6
async	2.6.0	511	81.5	0	103	418	108	444	105.4	424.8	81.5	1.3	79.3	0.0
async	2.6.0	511	81.5	5	88	401	100	427	93.0	414.7	76.5	3.8	81.8	6.1
async	2.6.0	511	81.5	10	69	404	91	420	79.7	411.1	71.5	6.9	84.4	12.2

TABLE 10. Test suite minimization results for *DeltaMIN*.

					Min		Max		Avg			s.d.	TS_{red}	TF_{red}
$Program$	$Ver.$	$\|T\|$	M	α	$\|T'\|$	t	$\|T'\|$	t	$\|T'\|$	t	M'			
						(ms)		(ms)		(ms)				
UUID	3.2.1	17	92.7	0	9	15	9	19	9.0	16.0	92.7	0.0	47.0	0.0
UUID	3.2.1	17	92.7	5	3	12	5	13	3.2	12.9	90.8	0.6	81.1	2.0
UUID	3.2.1	17	92.7	10	2	12	2	13	2.0	12.9	85.1	0.0	88.2	8.2
debug	3.1.0	4	3.6	0	1	4	1	5	1.0	4.6	3.6	0.0	75.0	0.0
debug	3.1.0	4	3.6	5	1	6	1	8	1.0	6.9	1.2	0.0	75.0	66.4
debug	3.1.0	4	3.6	10	1	6	1	8	1.0	6.9	2.2	0.0	75.0	39.1
body-parser	1.16.0	204	77.3	0	54	139	57	191	55.3	164.5	77.3	0.9	72.8	0.0
body-parser	1.16.0	204	77.3	5	41	79	50	116	46.2	98.2	72.4	2.6	77.3	6.3
body-parser	1.16.0	204	77.3	10	34	49	42	80	38.2	62.6	67.4	2.4	81.2	12.8
body-parser	1.18.2	218	74.4	0	60	191	63	264	61.6	225.2	74.4	1.0	71.7	0.0
body-parser	1.18.2	218	74.4	5	47	100	53	173	49.1	126.0	69.5	2.0	77.4	6.5
body-parser	1.18.2	218	74.4	10	33	43	45	84	39.9	67.1	64.5	3.3	81.6	13.2
express	4.15.0	833	76.1	0	184	29469	192	35144	186.3	32862.1	76.1	2.3	77.6	0.0
express	4.15.0	833	76.1	5	148	13929	170	26884	157.7	19323.4	71.2	6.1	81.0	6.5
express	4.15.0	833	76.1	10	125	7743	138	13305	132.8	10420.1	66.1	3.7	84.0	13.1
express	4.16.3	858	73.2	0	192	32546	195	44182	193.3	36579.2	73.2	1.1	77.4	0.0
express	4.16.3	858	73.2	5	162	19095	176	27832	168.2	22502.5	68.2	4.4	80.3	6.7
express	4.16.3	858	73.2	10	135	11301	157	17839	147.7	14773.5	63.2	6.4	82.7	13.6
passport	0.4.1	494	86.1	0	79	436	83	594	80.4	490.4	86.1	1.2	83.7	0.0
passport	0.4.1	494	86.1	5	64	244	74	380	69.5	312	81.3	2.9	85.9	5.5
passport	0.4.1	494	86.1	10	56	177	68	296	61.8	229.5	76.3	3.0	87.4	11.4
cheerio	1.0.0	650	87.9	0	185	17255	188	20385	186.2	18761.7	87.9	0.9	71.3	0.0
cheerio	1.0.0	650	87.9	5	150	8953	161	12353	156.7	10857.2	82.9	4.0	75.8	5.6
cheerio	1.0.0	650	87.9	10	128	5093	152	9325	140.3	7603	77.9	7.3	78.4	11.3
shortid	2.2.16	17	54.1	0	7	5	7	7	7.0	6.0	54.1	0.0	58.8	0.0
shortid	2.2.16	17	54.1	5	5	4	5	5	5.0	4.5	49.9	0.0	70.5	7.6
shortid	2.2.16	17	54.1	10	3	3	3	4	3.0	3.7	44.9	0.0	82.3	16.8
async	2.6.0	511	81.5	0	102	2269	106	2817	104.3	2487.4	81.5	1.4	79.5	0.0
async	2.6.0	511	81.5	5	85	1132	102	1890	92.2	1502.2	76.5	5.2	81.9	6.1
async	2.6.0	511	81.5	10	70	669	84	1089	76.5	842.0	71.5	3.9	85.0	12.2

D. THREATS TO VALIDITY

The threats to validity can be further classified as internal, external, and construct validity. The *Internal validity* denotes the faults during the implementation of the studied techniques. To overcome this threat, we implemented the algorithms greedy [15], delayed greedy [16] followed by testing and code-review of the implementation. Further, we made the implementation public for validation. In addition, there is a possibility that the faults introduced for experimental evaluation are not real faults. To cater to this problem, we made use of the Mutation testing tool for generating faults in the original packages automatically, which is previously proven to simulate real faults as compared to manually injected faults [6].

To address the *external validity* threat, it is important to show that the obtained results can be generalized to other languages. For that, we have conducted experiments on real-world JavaScript packages publicly available and widely used by the community for developing JavaScript applications. These packages come with test suites that a developer must execute to ensure the correctness of the package. At the moment, our approach requires the ''Mocha'' testing library, due to which we can only implement our approach on five out of twelve packages listed in [12]. This is because our redundancy elimination algorithms require the effect of each test case on the overall mutation score.

TABLE 11. Comparative analysis of all five algorithms Greedy (GR), DelayedGreedy (DGR), Harrold et al. (HGS), LinMIN (LM), and DeltaMIN (DM) summarizing the average values of fault detection loss, test suite reduction, and runtime.

		Test Suite Fault Detection loss (%)					Test Suite size reduction (%)					Average Time (ms)				
Program	*Ver.*	*GR*	*DGR*	*HGS*	*LM*	*DM*	*GR*	*DGR*	*HGS*	*LM*	*DM*	*GR*	*DGR*	*HGS*	*LM*	*DM*
$\alpha = 0$																
UUID	3.2.1	0	0	0	0	0	47.0	47.0	47.0	47.0	47.0	13.4	39.3	913	14.1	16.0
debug	3.1.0	0	0	0	0	0	75.0	75.0	75.0	75.0	75.0	2.6	65.0	145.8	4.7	4.6
body-parser	1.16.0	0	0	0	0	0	72.8	73.5	73.2	72.7	72.8	33.7	69.6	444.1	52.6	164.5
body-parser	1.18.2	0	0	0	0	0	70.7	72.4	72.4	71.9	71.7	38.7	74.1	599.5	60.5	225.2
express	4.15.0	0	0	0	0	0	77.9	78.1	77.1	77.5	77.6	1116.7	825.8	42450.6	2422	32862.1
express	4.16.3	0	0	0	0	0	77.4	77.9	76.9	77.4	77.4	1205.9	671.0	11703.0	2634.6	36579.2
passport	0.4.1	0	0	0	0	0	84.4	84.6	84.1	83.7	83.7	60.8	76.0	113.6	138.1	490.4
cheerio	1.0.0	0	0	0	0	0	71.2	71.8	71.2	71.2	71.3	551.9	479.3	13230.2	1066.4	18761.7
shortid	2.2.16	0	0	0	0	0	58.8	58.8	58.8	58.8	58.8	2.6	7.8	8.3	4.6	6.0
async	2.6.0	0	0	0	0	0	79.5	80.8	80.1	79.3	79.5	214.8	343.4	5176.6	424.8	2487.4
Average		**0**	**0**	**0**	**0**	**0**	**71.5**	**72.0**	**71.6**	**71.4**	**71.5**	**324.1**	**265.1**	**7478.4**	**682.2**	**9159.7**
$\alpha = 5$																
UUID	3.2.1	1.6	0.0	0.0	2.3	2.0	82.3	47.0	47.0	80.0	81.1	11.3	40.0	921.9	13.5	12.9
debug	3.1.0	0.0	0.0	0.0	100.0	66.4	75.0	75.0	75.0	100.0	75.0	2.4	118.6	144.8	4.4	3.9
body-parser	1.16.0	6.2	4.4	6.1	6.3	6.3	83.9	75.0	80.8	77.3	77.3	30.0	56.3	241.7	52.0	98.2
body-parser	1.18.2	6.3	3.1	6.1	6.5	6.5	82.5	73.8	80.2	75.9	77.4	33.6	61.6	369.3	58.3	126.0
express	4.15.0	6.4	0.9	6.3	6.5	6.5	87.1	78.3	83.3	80.8	81.0	794.5	787.3	28378.7	2412.5	19323.4
express	4.16.3	6.7	1.2	6.6	6.7	6.7	86.0	78.2	83.0	80.1	80.3	909.2	635.2	6157.4	2654.9	22502.5
passport	0.4.1	5.5	0.0	5.3	5.5	5.5	89.1	84.6	87.9	85.4	85.9	56.4	76.3	100.6	134.9	312.0
cheerio	1.0.0	5.6	5.5	5.3	5.6	5.6	84.0	76.4	77.6	75.0	75.8	427.9	447.4	5548.4	1057.2	10857.2
shortid	2.2.16	5.3	0.0	4.5	8.1	7.65	70.5	58.8	64.7	70.5	70.5	1.7	7.9	7.6	4.2	4.5
async	2.6.0	6.0	0.1	5.8	6.1	6.1	87.2	81.0	85.7	81.8	81.9	185.1	329.7	2339.4	414.7	1502.2
Average		**4.9**	**1.5**	**4.6**	**15.4**	**11.9**	**82.7**	**72.8**	**76.5**	**80.6**	**78.6**	**245.2**	**256.0**	**4420.9**	**680.6**	**5474.2**
$\alpha = 10$																
UUID	3.2.1	8.2	0.0	0.0	8.2	8.2	88.2	47.0	47.0	88.2	88.2	10.1	39.3	911.8	13.1	12.9
debug	3.1.0	0.0	0.0	0.0	100.0	39.1	75.0	75.0	75.0	100.0	75.0	2.2	118.4	145.4	4.4	3.6
body-parser	1.16.0	12.5	6.8	11.6	12.8	12.8	88.5	76.4	83.8	78.7	81.2	28.1	53.8	169.5	51.0	62.6
body-parser	1.18.2	13.0	7.4	12.3	13.2	13.2	88.0	76.6	82.5	81.2	81.6	30.5	58.3	215.1	57.4	67.1
express	4.15.0	13.0	12.8	12.7	13.1	13.1	90.9	82.3	85.2	83.5	84.0	664.8	755.6	11414.1	2354.4	10420.1
express	4.16.3	13.5	10.8	13.3	13.6	13.6	89.7	81.9	84.9	82.4	82.7	747.8	603.6	2748.5	2542.1	14773.5
passport	0.4.1	11.3	11.4	10.8	11.4	11.4	92.1	86.4	89.6	86.8	87.4	52.8	70.9	93.7	135.3	229.5
cheerio	1.0.0	11.2	5.5	10.6	11.3	11.3	88.8	76.4	79.0	77.9	78.4	364.2	446.6	1530.1	1041.9	7603.0
shortid	2.2.16	15.1	0.0	4.5	16.6	16.8	82.3	58.8	64.7	82.3	82.3	1.4	7.7	7.9	4.6	3.7
async	2.6.0	12.0	0.1	12.0	12.2	12.2	90.8	81.0	87.6	84.4	85.0	168.2	332.1	1363.4	411.1	842.0
Average		**11.0**	**5.4**	**8.7**	**21.2**	**15.1**	**87.4**	**74.2**	**77.9**	**84.5**	**82.6**	**207.0**	**248.6**	**1859.9**	**661.5**	**3401.8**

However, Mutode only reports the total number of passing and failing test cases. Therefore, we had to intercept the output of the underlying testing framework, Mocha, in this case, to compute the result of each test case for every mutant. Second, there were two packages, namely *Sockets.io* and *Bower*, where the execution of some test cases was skipped at run time, due to which the output of such test cases can not be ascertained. As a consequence, we had to exclude specified packages. Third, there are some mutants that, when executed by a given test suite, lead to a syntax error, thereby causing a test case to fail. We discarded all such mutants, but Mutode marked them as killed, leading to a higher mutation score. However, due to differences in the underlying programming languages a fair comparison with the previous studies in not feasible.

The *construct validity* means whether the evaluation metrics used in experiments are realistic. Therefore, we studied the effect of all algorithms based on two commonly used metrics, that is, the size of the reduced test suite and the execution time concerning two coverage criteria: statement and mutation coverage. However, the majority of previous studies have employed statement coverage for evaluating the execution cost of the studied approaches, which entails low cost as compared to mutation coverage [6]. However, mutation coverage is costly to implement but has been proven to be more stable than statement coverage when test suite minimization algorithms were studied on multiple versions of the same program [28].

Apart from that, we did not remove equivalent mutants that might increase the chance of duplicate mutants generated by Mutode [55]. Also, we did not discard minimum mutants and disjoint mutants from the Mutode. These are often removed from the mutants set to remove subsuming mutants [56].

VI. CONCLUSION

Mutation testing has limited applicability in the industry primarily due to the heavy cost involved in the mutant generation and equivalent mutant problems. Our approach addresses the first problem, that is, the mutation testing execution cost by keeping track of the output of every test case concerning all mutants. This output is afterward employed by algorithms to measure the mutation score of the reduced test suite. This way, we only need to execute *Mutode* once for each NPM package. Moreover, the execution cost of these algorithms is comparable to that of the original test suite available in subject NPM packages.

The empirical results obtained so far indicate that the proposed algorithms, **Harrold et al.**, **DeltaMIN** and **LinMIN** achieve on average 70% reduction in the original test suite size if all requirements must be satisfied. This finding is different from the previous study by Shi et al. as they obtained, on average 51% adequate reduction in test suite size based on killed mutants for all versions of studied programs. This could be due to the fact that they also included multiple versions of the same program. However, by decreasing the inadequacy level to 95%, they observed a 70% reduction in test suite size, whereas we observed an average 78% reduction in test suite size. Interestingly, the reduction percentage can reach up to 81% on average for all examples by decreasing the tolerance value to 90% in both studies [28].

Although like the previous study by Zhang et al., we observed minor differences in test suite size reduction and

fault-detection loss obtained by all algorithms for adequate reduction [6]. However, there is a difference between the required runtime of the five algorithms. **Greedy** performs test suite minimization faster than **DelayedGreedy**, **LinMIN** and **DeltaMIN** for the example programs. **DeltaMIN** requires, on average most of the time. Hence, for practical application of test suite minimization, **DelayedGreedy** should be preferred because it delivers, on average, the smallest test suite coming with an acceptable runtime.

Note that the original mutation score computed using the *Mutode* tool was comparatively low. This is because we studied the approach on tests available in the NPM package. These tests are most often manually generated. Therefore, a future extension relies upon the automation of test suite generation for better fault coverage. We made all artifacts available on GitHub for reproducibility and future research activities (see [54]).

As a next step, we plan to augment current baseline approaches with other traditional test suite reduction algorithms, namely GE [20], GRE [20], and ILP [21]. Another key aspect will be to evaluate the test suite minimization approaches on different versions of the same NPM package. This would help us better analyze the variance in different coverage criteria as the software under test evolves. Moreover, we plan to provide support for testing frameworks other than Mocha so that we can analyze the effect of the suggested approach on a larger number of NPM packages. Other future works include the comparison of different mutation testing tools such as Stryker [57] and Mutandis [58] to analyze the effect of equivalent mutants on test suite minimization. It would also be interesting to compare greedy approaches with meta-heuristics techniques for multi-objective test suite minimization [39].

In addition, we also intend to apply machine learning for test suite prioritization in modern development settings such as DevOps. The goal would be to analyze the combined effect of both test suite minimization and prioritization for reducing the ever-increasing regression testing costs during continuous integration. Specifically, we intend to adapt the test suite prioritization approaches discussed by Zhang et al. [59], keeping in view the time constraints involved in the regression testing for DevOps.

ACKNOWLEDGMENT

Seema Jehan thanks Saifullah (saifullah.bese17seecs@seecs.edu.pk), Fatima Tuz Zehra (fzehra.bese19seecs@seecs.edu.pk), and Areej Razzaq (arazzaq.bese19seecs@seecs.edu.pk) for implementing test suite reduction algorithms as part of paid internships.

REFERENCES

[1] C. Coviello, S. Romano, G. Scanniello, A. Marchetto, A. Corazza, and G. Antoniol, "Adequate vs. inadequate test suite reduction approaches," *Inf. Softw. Technol.*, vol. 119, Mar. 2020, Art. no. 106224. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919302393

[2] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test., Verification Rel.*, vol. 22, no. 2, pp. 67–120, Mar. 2012, doi: 10.1002/stvr.430.

[3] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng., Softw. Eng. Pract. Track (ICSE-SEIP)*. Piscataway, NJ, USA: IEEE Press, May 2017, pp. 233–242.

[4] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST approaches to scalable similarity-based test case prioritization," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 222–232, doi: 10.1145/3180155.3180210.

[5] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Softw., Pract. Exp.*, vol. 28, no. 4, pp. 347–369, Apr. 1998.

[6] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of JUnit test-suite reduction," in *Proc. IEEE 22nd Int. Symp. Softw. Rel. Eng.*, Nov. 2011, pp. 170–179.

[7] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Softw. Test., Verification Rel.*, vol. 12, no. 4, pp. 219–249, Dec. 2002.

[8] F. S. Ocariza Jr., K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *Proc. IEEE 22nd Int. Symp. Softw. Rel. Eng.*, Nov. 2011, pp. 100–109.

[9] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 35–53, Jan. 2012, doi: 10.1109/TSE.2011.28.

[10] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip, "A framework for automated testing of JavaScript web applications," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 571–580, doi: 10.1145/1985793.1985871.

[11] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for JavaScript," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 1–36, Sep. 2017, doi: 10.1145/3106739.

[12] D. Rodríguez-Baquero and M. Linares-Vásquez, "Mutode: Generic JavaScript and Node.js mutation testing tool," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*. New York, NY, USA: Association for Computing Machinery, Jul. 2018, pp. 372–375, doi: 10.1145/3213846.3229504.

[13] M. Zhang, A. Belhadi, and A. Arcuri, "JavaScript instrumentation for search-based software testing: A study with RESTful APIs," in *Proc. IEEE Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2022, pp. 105–115.

[14] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 270–285, Jul. 1993, doi: 10.1145/152388.152391.

[15] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, pp. 225–237, Apr. 2007.

[16] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *ACM SIGSOFT Softw. Eng. Notes*, vol. 31, no. 1, pp. 35–42, Sep. 2005, doi: 10.1145/1108768.1108802.

[17] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," *ACM SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, pp. 135–145, Aug. 2000, doi: 10.1145/347636.348938.

[18] H. K. N. Leung and L. White, "Insights into regression testing (software testing)," in *Proc. Conf. Softw. Maintenance*, Oct. 1989, pp. 60–69.

[19] S. U. R. Khan, S. P. Lee, N. Javaid, and W. Abdul, "A systematic review on test suite reduction: Approaches, experiment's quality evaluation, and guidelines," *IEEE Access*, vol. 6, pp. 11816–11841, 2018.

[20] T. Y. Chen and M. F. Lau, "A new heuristic for test suite reduction," *Inf. Softw. Technol.*, vol. 40, nos. 5–6, pp. 347–354, Jul. 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584998000500

[21] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *Proc. 26th Int. Conf. Softw. Eng.*, May 2004, pp. 106–115.

[22] H. Zhong, L. Zhang, and H. Mei, "An experimental study of four typical test suite reduction techniques," *Inf. Softw. Technol.*, vol. 50, no. 6, pp. 534–546, May 2008, doi: 10.1016/j.infsof.2007.06.003.

[23] N. Mansour and K. El-Fakih, "Simulated annealing and genetic algorithms for optimal regression testing," *J. Softw. Maintenance, Res. Pract.*, vol. 11, no. 1, pp. 19–34, Jan./Feb. 1999.

[24] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 195–209, Mar. 2003, doi: 10.1109/TSE.2003.1183927.
[25] A. K. Jena, S. K. Swain, and D. P. Mohapatra, "Model-based test-suite minimization using modified condition/decision coverage (MC/DC)," *Int. J. Softw. Eng. Appl.*, vol. 9, no. 5, pp. 61–74, May 2015.
[26] Q. Mayo, R. Michaels, and R. Bryce, "Test suite reduction by combinatorial-based coverage of event sequences," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation Workshops*, Mar./Apr. 2014, pp. 128–132.
[27] A. M. Smith and G. M. Kapfhammer, "An empirical study of incorporating cost into test suite reduction and prioritization," in *Proc. ACM Symp. Appl. Comput. (SAC)*. New York, NY, USA: Association for Computing Machinery, Mar. 2009, pp. 461–467, doi: 10.1145/1529282.1529382.
[28] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 246–256, doi: 10.1145/2635868.2635921.
[29] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Eng.*, vol. 10, no. 4, pp. 405–435, Oct. 2005.
[30] A. P. Agrawal, A. Choudhary, A. Kaur, and H. M. Pandey, "Fault coverage-based test suite optimization method for regression testing: Learning from mistakes-based approach," *Neural Comput. Appl.*, vol. 32, no. 12, pp. 7769–7784, Jun. 2020.
[31] S. Wang, S. Ali, and A. Gotlieb, "Cost-effective test suite minimization in product lines using search techniques," *J. Syst. Softw.*, vol. 103, pp. 370–391, May 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121214001757
[32] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 235–245, doi: 10.1145/2635868.2635910.
[33] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 523–534.
[34] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, "Scalable approaches for test suite reduction," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*. Piscataway, NJ, USA: IEEE Press, May 2019, pp. 419–429, doi: 10.1109/ICSE.2019.00055.
[35] I. Pill, S. Jehan, F. Wotawa, and M. Nica, "Analyzing the reduction of test suite redundancy," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Gaithersburg, MD, USA, Nov. 2015, p. 65.
[36] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction for quick testing," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation*, Mar. 2014, pp. 243–252.
[37] YAFFS. (2002). *Yaffs—A Flash File System for Embedded Use*. [Online]. Available: https://yaffs.net/
[38] S. Romano, G. Scanniello, G. Antoniol, and A. Marchetto, "SPIRITuS: A simple information retrieval regression test selection approach," *Inf. Softw. Technol.*, vol. 99, pp. 62–80, Jul. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918300405
[39] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2007, pp. 140–150.
[40] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: A mutation system for Java," in *Proc. 28th Int. Conf. Softw. Eng.* Los Alamitos, CA, USA: IEEE Computer Society, May 2006, pp. 827–830, doi: 10.1145/1134285.1134425.
[41] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Oct. 2011.
[42] R. Wille, "Concept lattices and conceptual knowledge systems," *Comput. Math. Appl.*, vol. 23, nos. 6–9, pp. 493–515, Mar. 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0898122192901207
[43] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proc. 10th ACM SIGSOFT Symp. Found. Softw. Eng.*, Nov. 2002, pp. 1–10.
[44] npm. (2009). *Node Package Manager*. [Online]. Available: https://www.npmjs.com/about/
[45] UUID. (2019). *UUID*. [Online]. Available: https://www.npmjs.com/package/uuid
[46] Debug. (2019). *Debug*. [Online]. Available: https://www.npmjs.com/package/debug
[47] Body-Parser. (2019). *Body-Parser Framework*. [Online]. Available: https://www.npmjs.com/package/body-parser
[48] Express. (2019). *Express Framework*. [Online]. Available: https://www.npmjs.com/package/express
[49] Passport. (2019). *Passport Authentication Middleware*. [Online]. Available: https://www.npmjs.com/package/passport
[50] Cheerio. (2018). *Cheerio*. [Online]. Available: https://www.npmjs.com/package/cheerio
[51] ShortId. (2020). *ShortId*. [Online]. Available: https://www.npmjs.com/package/shortid
[52] Async. (2020). *Async*. [Online]. Available: https://www.npmjs.com/package/async
[53] NYC. (2020). *NYC*. [Online]. Available: https://www.npmjs.com/package/nyc
[54] Saifullah, F. T. Zehra, and A. Razzaq. (2020). *Test Suite Reduction Algorithms for JavaScript Applications*. [Online]. Available: https://github.com/areejrazzaq/Test-Suite-Reduction
[55] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 936–946.
[56] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation*, Mar. 2014, pp. 21–30.
[57] N. Jansen and S. de Lang. (2018). *The JavaScript Mutation Testing Framework*. [Online]. Available: https://github.com/stryker-mutator/stryker
[58] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient JavaScript mutation testing," in *Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation*, Mar. 2013, pp. 74–83.
[59] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Trans. Softw. Eng.*, vol. 45, no. 9, pp. 898–918, Sep. 2019.

SEEMA JEHAN (Member, IEEE) received the M.S. and Ph.D. degrees in computer science from the Graz University of Technology, Austria. She was an Assistant Professor with the School of Electrical Engineering and Computer Science, National University of Sciences and Technology, Islamabad, Pakistan, from 2018 to 2022. Currently, she is a Software Team Lead with Zaheen Systems. Her current research interests include model-based software testing, software engineering, and deep learning. She is a Professional Member of ACM.

FRANZ WOTAWA (Member, IEEE) received the M.Sc. degree in computer science and the Ph.D. degree from the Vienna University of Technology, in 1994 and 1996, respectively. He is currently a Professor of software engineering with the Graz University of Technology and the Head of the Institute for Software Technology. During his career, he has written more than 435 peer-reviewed papers for journals, books, conferences, and workshops. He has supervised 100 master's and 38 Ph.D. students. His current research interests include model-based and qualitative reasoning, theorem proving, mobile robots, verification and validation, and software testing and debugging. He has been a member of various program committees and organized several workshops and special issues of journals. He is a member of the Academia Europaea, the IEEE Computer Society, ACM, the Austrian Computer Society (OCG), and the Austrian Society for Artificial Intelligence. He is a Senior Member of AAAI. For his work on diagnosis, he received the Lifetime Achievement Award of the International Diagnosis Community, in 2016.