Optimizing regression testing with AHP-TOPSIS metric system for effective technical debt evaluation

Anis Zarrad1 · Rami Bahsoon1 · Priya Manimaran1

Received: 7 July 2023 / Accepted: 27 June 2024
© The Author(s) 2024

Abstract
Regression testing is essential to ensure that the actual software product confirms the expected requirements following modification. However, it can be costly and time-consuming. To address this issue, various approaches have been proposed for selecting test cases that provide adequate coverage of the modified software. Nonetheless, problems related to omitting and/or rerunning unnecessary test cases continue to pose challenges, particularly with regard to technical debt (TD) resulting from code coverage shortcomings and/or overtesting. In the case of testing-related shortcomings, incurring TD may result in cost and time savings in the short run, but it can lead to future maintenance and testing expenses. Most prior studies have treated test case selection as a single-objective or two-objective optimization problem. This study introduces a multi-objective decision-making approach to quantify and evaluate TD in regression testing. The proposed approach combines the analytic-hierarchy-process (AHP) method and the technique of order preference by similarity to an ideal solution (TOPSIS) to select the most ideal test cases in terms of objective values defined by the test cost, code coverage, and test risk. This approach effectively manages the software regression testing problems. The AHP method was used to eliminate subjective bias when optimizing objective weights, while the TOPSIS method was employed to evaluate and select test-case alternatives based on TD. The effectiveness of this approach was compared to that of a specific multi-objective optimization method and a standard coverage methodology. Unlike other approaches, our proposed approach always accepts solutions based on balanced decisions by considering modifications and using risk analysis and testing costs against potential technical debt. The results demonstrate that our proposed approach reduces both TD and regression testing efforts.

Keywords Software management · Regression testing · Technical debt · Modification-based reduction · AHP · TOPSIS

Mathematics Subject Classification 00-01 · 99-00

Extended author information available on the last page of the article

Published online: 08 July 2024
1 Introduction

Technical debt (TD) is a term coined by Cunningham (1992) that refers to the additional work and time that results from accelerating the velocity in software production and taking shortcuts for short-term savings to meet particular objectives. TD can be considered at any phase of production, including the testing and regression testing phase, which is the focus of this study.

Guo and Seama (2011) described TD as a situation when the testers and developers need to design and implement quality software, but business constraints and market pressures may limit their ability to produce high-quality products. In regression testing, TD describes a situation where testers make compromises in testing coverage adequacy and its revealing capabilities for immediate savings in testing time and effort. This is in line with the current understanding of the development community, where TD represents the differences between the system’s current state and its ideal state (Nugroho et al. 2011).

Regression testing aims to ensure the correctness and behavioral stability of the software after adding new features or modifying existing ones. In regression testing, testers rerun some or all of the test cases against the modified code to detect new bugs or problems in the updated software. The exercise is also for ensuring that the modification does not adversely affect the correctness and behavior of the unmodified part of the software. Therefore, testers often trade high coverage adequacy and revealed test results against time and budget constraints. Thorough regression testing can be time-consuming and resource-intensive. Customers, particularly those with budget constraints, may choose to limit the extent of regression testing in order to reduce their expenses. However, if errors persist across updates as carryovers from previous versions, then poor testing is responsible for the negatively impacted product quality and project expenses, particularly in the long run.

In the context of testing, TD appears when more effort and cost are spent on the follow-up fixes of the software and/or rework of a testing plan during the regression testing process. The observable impact of TD can include user inconvenience, lack of confidence, user satisfaction, and software quality. One opportunistic way to pay off TD is to rework the regression testing procedure, including the possible redesign of the testing plan and/or the test case selection. Therefore, when regression testing TD is paid off early, a project may experience cost reductions and improve maintenance quality (Brown et al. 2010). Clearly, technical debt caused by regression testing is a consequence of poor testing, such as omitting test cases that can reveal errors. Therefore, selecting test cases at random without careful consideration of code coverage and fault revealing can result in redundant test cases being selected. In general, TD disrupts the evolution and maintenance of existing systems (Martiab et al. 2018). The cost of fixing, repairing, and redistributing a defective product can hamper its development. If neglected, TD in regression testing may breed other types of TDs that can limit the software evolution. This can also easily lead to phasing out of the software when the debt becomes unmanageable.

The multitude of problems that appear in regression testing when omitting, adding, and selecting test cases can lead to TD. Most of the proposed selection
algorithms and techniques in regression testing only cover some parts of the program rather than the entire program (Li and Harrold 2008; Zazworka et al. 2011). From a business perspective, this does produce upfront savings in cost and time, but uncovered code can lead to numerous errors and create TD downstream. Another major problem occurs when a set of test cases includes the same code. Typically, in this situation, only a few test cases are selected, and the rest are omitted depending on the approach used. However, although several test cases can have the same traversal path in a control flow graph, they may generate different outputs. This is known as the “equivalent test-case” problem. A selection strategy can be used by randomly choosing test cases or choosing based on optimization methods.

Many studies have been conducted on test case selection and prioritization (Narciso et al. 2014; [9], Choudhary et al. 2018; Rubing Huang and Xia 2020; Bahsoon and Mansour 2002) using adaptive random selection and genetic and greedy algorithms. The selection approach combines code coverage and cost into a single objective to improve the test efforts for immediate saving and rapid delivery without considering TD. Testing the modified code is important to ensure that new defects are not introduced. However, testing only the modified code is insufficient because this approach can produce new errors during interactions with the older portions of the code. Xu and Rothermel (2009) discussed the idea of adding more test cases into regression testing to cover the changed code that may contain new errors that the old test cases were unable to detect. In this case, each new test may add extra TD because the newly added test case could have the same traversal path in the control flow graph (CFG) and the same output in regression testing. This may not happen immediately, and therefore, cannot be anticipated, as the tester of the next version could spend more time omitting the added test cases from the previous regression test phase. This is in line with the current understanding of the development community, where TD prioritizes speedy delivery over perfect code. Given the high cost of refactoring and future maintenance, a new method for test case selection in regression testing is required. A multi-objective approach is required to consider TD by balancing many competing objectives. In this study, we designed a new multi-objective test case selection approach based on a well-known decision making approach called the technique for order of preference by similarity to ideal solutions (TOPSIS) (Hwang and Yoon 1981) combined with an analytic hierarchy process (AHP) method (Saaty 1990; Sahar Trigui et al. 2018). The goal was to select the most effective minimal test suit using three objectives: code coverage, cost and risk. Our proposed approach extends beyond the conventional focus on test case quality (Beena and Sarala 2013; Saber et al. 2018; Yoo and Harman 2011; Harikarthik et al. 2019; Sapna and Balakrishnan 2015; Abreu and Gemund 2009; Bahsoon and Mansour 2002) by incorporating essential criteria such as cost and risk. Specifically, we introduce the following enhancements to our approach:

- **Modification-based reduction (MBR):** An improved descendent reachability information (Bahsoon and Mansour 2002) tailored for handling modified code segments.
• **Cost model:** Introducing a new cost model that considers the ratio of test case execution time to the total severity of detected faults.

• **Test case risk metric:** Incorporating a test case risk metric to enhance coverage of high-risk areas, facilitating the early detection of critical faults.

These modifications contribute to a more holistic evaluation of Technical Debt (TD) in regression testing. By allowing error detection and debugging to commence earlier, our approach facilitates faster decision-making about release schedules and resource allocation, effectively reducing TD. The main contributions of our study to the literature can be summarized as follows: (1) A multi-objective formulation of the TD problem in regression testing for test case selection. (2) A modified TOPSIS technique is proposed that combines three objectives: risk, code coverage, and testing cost to evaluate TD in regression testing. (3) Utilize AHP Analytic Hierarchy Process (AHP) to address criticism and TOPSIS problems. We introduce a reliable weighting factor model, ensuring consistent judgment and minimizing the subjective influence of human judgments. (4) Provide comparative results with three approaches discussed in [23], Beena and Sarala (2013); Rubing Huang and Xia (2020). The results show that our proposed method outperforms these methods.

Beyond these main points, in Sect. 2, we discuss some related work covering the concept of TD and regression testing. In Sect. 3 we present our proposed technical debt evaluation model (TDEM) to evaluate TD in regression testing. An example illustrates the approach in Sect. 4 with the results of the analysis. Section 5 concludes the study and discusses future proposals for improvement.

### 2 Related work

Software testing is an important part of quality assurance. In general, software testing requires a large number of tests, which is time-consuming and expensive. Therefore, test selection, minimization, and prioritization methods have been suggested in the literature.

When selecting test cases, the objective is to minimize the number of test cases in a specific test set while maintaining the efficiency of the original test set in fault detection. Test case selection is usually applied when there is no time or resources available to test the entire system after modifications. Minimization shares many similarities with the test case selection approach, except that the reduction is permanent compared to the temporary selection of test cases for a specific program version.

Test case prioritization is to run the most important tests earlier based on predefined coverage criteria. Test case prioritization usually guarantees earlier greater coverage to reduce the testing cost. There are numerous test case prioritization and selection methods in the literature (Bahsoon and Mansour 2002; Harikarthik et al. 2019; Bryce and Memon 2011; Sapna and Balakrishnan 2015; Zheng Li and Hierons 2007; Abreu and Gemund 2009). However, in contrast to the approach presented in this study, they are mostly code-coverage based and modification-based, and they mainly focus on regression testing without considering the TD aspect. Cunningham
(1992) led early work on technical debt (TD) in 1992. Nevertheless, software TD has only recently emerged. Several aspects have been discussed in the literature to determine the trade-off between rapid delivery benefits and long-term value. TD can appear in many forms in a software development life cycle (SDLC), such as in requirement debt (Lenarduzzi and Fucci 2019; Supatsara Wattanakriengkrai et al. 2018), design debt (Supatsara Wattanakriengkrai et al. 2018), coding debt (Daniel 2017; Arcelli Fontana et al. 2015), testing debt (Wiklund et al. 2012; Tornhill et al. 2018; Tornhill 2018), and documentation debt (Soares et al. 2015). Clearly, software testing impacts TD in several ways; if neglected, TD increases which lead to poor productivity. Several studies have examined the relationship between TD and software testing (Wiklund et al. 2012; Shakeri et al. 2016). Some studies (Tornhill et al. 2018; Tornhill 2018) investigated what constitutes adequate testing tools when focusing on TD.

Bahsoon and Mansour (2002) proposed a selection approach called modification-based reduction (MBR) to identify the requirements that might be congested by the modification, using the descendent reachability information concept in the control flow graph (CFG). Requirements that were not potentially affected by the modification were ignored, thus requiring no further testing. Omitting a test case that will test a sub-path could lead to errors, requiring further time and expenditure, which can increase software TD. Abreu and Gemund (2009) developed a multiple-fault localization technique based on a dynamic-spectrum-based approach. This algorithm uses statistical fault localization methods, combined with a probabilistic reasoning approach from a model-based diagnosis. The authors of Harikarthik et al. (2019) discussed a test case prioritization method that generates and forms a cluster of test cases using the fuzzy c-means clustering technique. This technique creates a relevant and irrelevant cluster of test cases to maximize fault detection probability. Moreover, this approach uses both relevant and irrelevant clusters to compute fault detection probability. A genetic model was discussed in Bryce and Memon (2011) to study GUI and web applications together. A unified theory was developed based on the abstract model and shared prioritization criteria to show how event-driven software (EDS) should be tested. They prioritized existing test suites using t-way interaction coverage. In Leitner et al. (2007); Yoo et al. (2009), the authors introduced random selection approaches for regression testing to reduce the test suite size, thereby decreasing the testing phase. There has been a concern regarding the effect of test suite minimization on the error detectability of the selected test suites. An evolutionary search algorithm was discussed in Zheng Li and Hierons (2007) to order test cases according to the fitness function. The proposed function focuses on code coverage, including block, decision (branch), and statement coverage. The authors of Sapna and Balakrishnan (2015) presented a black-box approach using the Steiner tree algorithm to generate a minimal test set. The original test cases were produced from system functionalities using the unified modeling language (UML). The resulting test suite could only be used for quick testing to ensure that the basic functionality worked correctly. Agrawal et al. (2020) focused on test suite minimization using a safe regression test case selection method based on an artificial bee colony by determining the optimal fitness value among a range of values. They also compared various approaches with many other nature-based computing algorithms,
such as bat search and ant colony optimization. The proposed method lacks an information-based global procedure to select the best solution. Multi-objective evolutionary algorithms (MOEAs) have been widely used in the regression testing literature (Saber et al. 2018; Yoo et al. 2009; Yoo and Harman 2011) to produce Pareto-efficient subsets of test sets. The proposed approach emphasizes the maximization of code coverage. In their obtained results, Pareto ranking provided a good achievement compared to greedy algorithms. One limitation is that new solutions are generated using fully randomized recombination operators. Yoo and Harman (2007) described a Pareto efficient multi-objective optimization problem for selecting test cases. Pareto optimally defines the Pareto dominance for assessing the quality of the solutions using a two-objective formation (coverage, and execution time). Beena and Sarala (2013) proposed a code coverage method to remove redundant test cases and reduce the number of test case sets. The required test cases are selected based on the modification, revealing using modification, and deleting the matrix.

Dario Di Nucci et al. (2020) presented a test case prioritization technique based on a hypervolume-based genetic algorithm. The results show better performance compared to the three algorithms used for test case prioritization. Authors in Huayao Wu et al. (2020) presented an empirical comparison of three test generation techniques, namely, combinatorial testing (CT), random testing (RT) and adaptive random testing (ART) under different test scenarios. The goals of this study are to investigate the efficiency of the three techniques. Their results show similar behavior among all proposed techniques. However, this study does not consider the impact of technical debt when used in regression testing, and the analysis assumes that the tester is aware of the parameters and constraints for the system being tested.

Wiklund et al. (2012) considered that the maintenance and development of test automation tools commonly encountered difficulties due to unexpected and unanticipated problems. They further noted from their case study on a telecommunication subsystem that the reduction of TD in automation testing is heavily dependent on the fields of interaction design and general software design principles. Test automation indicates that its use leads to higher code quality and shorter lead times. Automated testing (Wiklund et al. 2012) is beneficial to projects and organizations. This may reduce costs, improve quality, and offer continuity. However, this may incur the required training and new software costs. Tornhill et al. (2018); Tornhill (2018) proposed a “CodeScene” framework to identify TD in an automated manner based on their TD interest rate. “CodeScene” works well in practice for TD prioritization with a stable product version (Tornhill 2018). The authors in Shakeri et al. (2016) hypothesized that there is no considerable difference between the percentage of existing TD and the time required to reduce TD during testing. Reducing TD will require an increase in testing and development efforts. To the best of our knowledge, the technical debt evaluation model (TDEM) represents the first regression testing model that prioritizes test cases in a more effective way to control TD. Clearly, prioritizing test cases based on fault-code coverage and/or testing effort will have an impact on TD because modified code can introduce new faults in other parts of the system.

Prior studies on regression test case prioritization and selection have primarily focused on greedy algorithms. These techniques are debt-unaware and may pose a
substantial problem in the long run. Clearly, if the debt is considered among the selection criteria, the algorithm would have come up with different solutions. It is known that these algorithms may produce suboptimal results and carry extra debt because they may produce results that denote only the local minima within the search algorithm. Test-case prioritization approaches order test cases, such that those with the highest priority according to a particular criterion (a “fitness metric”) are executed first. For example, concerning coverage alone, testers may wish to schedule test cases to achieve code coverage at the fastest rate possible.

To summarize, in studying the cited work on regression test case prioritization and selection, we found no clear, detailed, or easy-to-use guidelines on how to select test cases to reduce TD while providing the maximum benefit to the organization. Most studies focused on greedy algorithms and random selection with a code-coverage concern. Other researchers considered the cost with code coverage (Yoo et al. 2009; Yoo and Harman 2011; Cibulski and Yehudai 2011) in their selection approach. However, these approaches conflate these two objectives into a single objective. Thus, the proposed solutions are not reliable for achieving long-term goals. Given the cited problems related to software testing and TD, we conclude that there is an actual need to develop a TD model for testers that aims to improve TD in existing systems after modifications. Compared to the above-mentioned related works, the main benefit of the suggested selection techniques is to further the knowledge of the main sources of TD with respect to test case selection. A multi-objective formulation of regression test case selection is proposed to cover the main source of TD. We considered TOPSIS and AHP approaches to balance our three competing objectives: code coverage with modification revealing, testing cost, and risks against the likely potential TD.

3 New technical debt evaluation model TDEM

Many proposed testing approaches (Bahsoon and Mansour 2002; Leitner et al. 2007; Yoo et al. 2009) are inadequate for identifying and quantifying TD. A new model was implemented to address TD in regression testing by selecting test cases whose omission may contribute to a high TD value, instead of choosing random test cases or those based on some form of single objective and expert judgments (Biswas and Mall 2011). The proposed model is described in the following sequence.

- **Step 1:** Owing to the particularity of regression testing, the selection method in modification-base reduction (MBR) (Bahsoon and Mansour 2002) was improved to restrict our attention to TD reduction. We evaluated the ability of the test cases based on several metrics: code coverage, cost, and risk.
- **Step 2:** Compute the TD award value for all test case candidates using the TOPSIS and AHP methods. The multi-objective formulation that combines risk, code coverage, and test case cost facilitates the theoretical treatment of the optimal TD award value for each test case.
- **Step 3:** Make decisions by selecting a final test suit to be executed for the next testing run with a low TD risk.
In regression testing, it is important to retest the software following modifications. In this context, there is a need for a method to select all test cases affected by the requirement modifications and formulate the TD reduction problem and measurement algorithms.

### 3.1 Reduction selection algorithm

We implemented significant modifications to the modification-based reduction method (MBR) to adeptly respond to evolving requirements (Bahsoon and Mansour 2002). Consequently, we built a crafted candidate test set tailored for regression testing, with dedicated consideration for Technical Debt (TD). This involved the integration of an algorithm designed to accurately compute the TD value. The computation of TD value is rooted in a thorough consideration of three pivotal objectives: (1) assessing the risk associated with each test case, (2) evaluating code coverage, and (3) determining test case cost measured in terms of execution time per fault severity. This representation of Technical Debt (TD) encapsulates not only the potential risks entailed by individual test cases but also their impact on code coverage and the resource cost associated with their execution, relative to the severity of faults they aim to detect. Further details regarding the TD computation are discussed in Sect. 3.2.

In our methodology, test data are extracted and compiled from prior testing cycles. We maintain a comprehensive historical database that meticulously records the outcomes of previous test executions. This dataset includes crucial details, such as the types of faults identified, their severity levels, and the specific test cases responsible for their detection. To ensure the reliability of our testing history in the current version, testers are required to conduct impact analysis using modification-based reduction (MBR). This analysis assesses how a change in one part of the software may influence other interconnected components. In the case of significant modifications, the test case data undergoes updates and reevaluation to accurately reflect the current state of the code and the test case severity.

This iterative method acts as a robust mechanism, guaranteeing the accuracy and relevance of test case information across revisions. It establishes a proactive approach to adapt the test suite to changes in the software, enhancing the effectiveness of our regression testing strategy.

The TOPSIS-AHP approach is used to call a decision algorithm whenever two appropriate test cases are found. Regression testing was conducted on test cases that had the highest TD values.

Additionally, all source codes that have not been tested in the most recent version need to be printed out for the tester and have an uncovered label attached to them. In this case, we can select test cases from the testing set history based on the highest code coverage. This approach eliminates the need to re-run redundant test cases for the uncovered codes. The proposed approach can achieve better results than other random and greedy algorithms and works as follows:

To identify all the requirements that might be covered in the modified version, we use reachability information retrieved from the Control Flow Graph (CFG). In
this context a requirement refers to specific conditions or capabilities needed by the system to solve a problem. The CFG represents the possible execution paths within the program, and by analyzing these paths, we can determine which requirements are affected by the modifications. A requirement that is not covered in the modified code is not potentially affected by the modification and does not need to be tested at this stage.

This should also include the unmodified requirements of a program that are affected by the modifications. Thus, A requirement traceability matrix can be used for this purpose.

The new proposed MBR method uses descendent reachability information to select a subset $TS'$ of test cases from $TS$ that satisfies all modified requirements presented as a set $R' = \{r'_1, r'_2, r'_3, ..., r'_n\}$, obtained from the modified requirement. The selected set $TS'$ must include test cases that satisfy all the influenced $r_i$ to guarantee that adequate coverage of the modification and potentially affected requirements are attained. The test cases are relevant to the modification-traversing test case concept.

To construct $TS'$, we start a reduction process by marking every requirement $r_k$ (where $k \in \{1...n\}$) not in $R'$ as unmodified, and then ignoring each marked one. It is important to note that there is a precomputed mapping between each test case in $TS$ and the requirements $r_k$ in $R$. This mapping is established during the initial analysis phase and ensures that each test case is associated with the specific requirements it covers. For all unmarked requirements in $R$, select all the singletons $TS_i$; then test the selected requirements and mark all $TS_i$, including the test cases, as satisfied.

Next, all unmarked test cases in the $TS_i$ of Cardinality 2 were considered. The test case with the maximum number in $TS_i$ was selected and added to the $TS'$ set. Again, all unmarked $TS_i$ values related to these test cases are marked satisfied. This process is repeated for $TS_i$ with cardinality $[3, 4, ..., Max - Card]$, where $Max - Card$ is the maximum cardinality of $TS_i$. When there is a tie between several test cases covering the modification requirements, cardinality is increased by one. If we reach $Max - Card$, the MBR algorithm transfers these test cases to the TD computation model described below to select the test case with a high TD value and add it to the $TS'$ set. This model aims to potentially reduce the selected regression test suite by omitting test cases covering unmodified parts along with redundant test cases.

### 3.2 TD Multi-objective formulation of regression test case selection problem

Multi-objective optimization is based on the TOPSIS and AHP methods. With multiple objectives, a set of test cases $A$ is better than another set $B$ only when $A$ excels $B$ in the combined objectives. In this problem, we used a three-objective formulation that combined (1) risk related to the test case, (2) code coverage, and (3) test case cost measured in execution time per fault severity. More formally we define TD as:

$$TD(t) = (Risk(t), Coverage(t), Cost(t))$$

where $t$ represents the test case $t = \{t_0, t_1, ..., t_n\}$. When the TOPSIS and AHP methods are applied to multi-objective test case selection, they produce a single ordering set with the maximum TD award value. This set of solutions is said to represent the best solution that jointly
optimizes the code coverage to reveal errors, risks related to the TD, and the test case cost for assessing test case adequacy based on TD. More details on these optimization objectives are provided below.

3.2.1 Objective 1: Risk

We assessed the risk factors of all test cases by determining the likelihood of errors. In this context, an error in a test case refers to any unexpected behavior from the expected outcome in the software under consideration. Essentially, errors represent defects or mistakes in the software or system being tested. The risk in this context represents the potential negative outcomes or consequences associated with discovering errors or defects in a specific test case \( t_i \). This assessment takes into consideration both the Number of Errors (N) and Higher Impact Errors (I), as indicated by the calculated probability of errors.

**Number of errors (N):** This attribute represents the count of defects or deviations identified within a specific test case \( t_i \). In our assessment, we systematically identify and log each error encountered during the testing process. These errors encompass a range of issues, including coding errors, logic flaws, and deviations from specified requirements. The number of errors serves as a quantitative measure of the issues found within the boundaries of the test case.

**Higher impact errors (I):** This attribute refers to the potential consequences and disruptions that certain errors can introduce when they are present in the software system. These impacts are typically assessed based on the likelihood of the errors causing significant issues. Here are some key factors that might be considered as higher impact causes in test case risk: higher-impact errors are those with the potential to result in critical malfunctions, security vulnerabilities, or major setbacks in the functionality of the system. This is represented as a percentage.

The proportion percentages of the number of errors and higher impact errors were set at 10% and 2%, respectively. A meticulous analysis of prior research, as discussed in references (Lessmann 2008) [44,45], was conducted to derive an average indicative of the error impact, expressed as a percentage. This approach enables us to address critical areas that have the potential to significantly impact the operational integrity of the system.

The risk factor for the specific test case \( t_i \) is defined through the following equation:

\[
\text{Probability for Errors}(E) = \text{Number of Errors}(N) \times \text{Higher Impact Errors}(I)
\]

The Probability for Errors is a metric that reflects the likelihood of encountering errors or defects in a specific test case or the software system as a whole. Its computation involves a comprehensive assessment, considering factors like complexity, code coverage, and the overall quality of the testing process. This probability serves as a foundational input for the subsequent calculation of risk.

The risk factor associated with a particular test case \( t_i \) is then determined by multiplying the Probability for Errors \( (E) \) by 100:
This risk calculation estimates the potential negative outcomes or consequences associated with the discovery of errors or defects in the specified test case. A higher risk value indicates a greater likelihood of encountering issues, underscoring the importance of thorough testing and effective mitigation strategies. Conversely, low-risk scenarios, characterized by minimal impact on technical debt (TD), are typically excluded as test cases during the mutation testing phase. This strategic approach ensures that testing efforts are directed towards areas with a higher potential impact, contributing effectively to software quality assurance.

### 3.2.2 Objective 2: code segment coverage

In our study, we incorporated the segment-coverage metric, considering it as a form of code coverage, facilitated by the control flow graph (CFG). The term “segment coverage” refers to the coverage of specific sections of code rather than a single line of code (Fig. 1). In our approach, segment coverage is deemed more meaningful than line coverage in unit and integration testing because it focuses on covering logical segments of code that may span multiple lines. While line coverage offers insights into executed lines of code, it may fall short of capturing the nuances of complex control flow structures or interactions between different components. In contrast, segment coverage provides a more comprehensive view of how these logical segments of code are exercised during testing, aligning with the intricacies of integration scenarios and enhancing the depth of testing analysis.

The probability of detecting errors escalates with the increasing number of segments in a test case. The segment coverage metric for test case $t_i$ is denoted by $\text{Coverage}$. This is linked to the required coverage and CFG. Segment coverage is another important metric that defines the extent to which the software functionalities are covered. It measures the percentage of test case coverage against the total number of segments in the CFG. The testing team mainly calculates this. $\text{Coverage}$ is defined as

$$\text{Coverage}(t_i) = \frac{\text{Number of segments covered by } t_i}{\text{Total number of segments}} \times 100.$$ 

The number of segments can be retrieved from the test case traversal table as shown in Table 3.

### 3.2.3 Objective 3: Test case cost

Another important measure to consider is the cost of test case execution per fault severity. We estimate the cost by measuring the actual time required to execute each test case. We formally define the test case cost as follows:

$$\text{Risk Factor}(t_i) = E \times 100$$
The Cost \((t_i)\) is a function that maps the fault severity and cost of \(t_i\) into a value. Where the execution time of the test case measured the actual time required by test case \(t_i\). Execution time was recorded using testing tools for each test case. We used a test annotation (for example, \@Test in JUnit) to determine the execution time of the individual test case. The time was measured in seconds. The total fault severity represents the summation of the severity values for all faults detected by \(t_i\).

Unlike other approaches (Gupta and Mahapatra 2021; Yoo and Harman 2007, 2011) that solely consider execution time as a cost parameter, our comprehensive perspective incorporates fault severity to estimate the cost associated with each test case. The computation of this cost value is grounded in the severity of faults detected by each test case, utilizing a historical test case outcomes dataset, as explained in Sect. 3.1. Severity values are retrieved from the test case historical data, detailing the impact of faults detected in past testing cycles.

To ensure continual accuracy and relevance in severity assignments, testers update test case severity in the historical dataset as new data emerges. Periodic reviews allow reassessment and adjustment of severity levels based on an evolving understanding of the impact of defects. This iterative process ensures that the historical test case dataset remains accurate and reflective. We utilize severity values assigned to each fault based on their impact, as outlined in Table 1. This ensures a nuanced evaluation that not only considers the number of faults detected but also weighs them based on their potential impact on the system.

The decision to use odd numbers for severity values in our approach was guided by the need to establish a clear and reasonable distinction among different severity levels. Assigning odd values, such as 1, 3, 5, etc., helps avoid ambiguity and ensures a noticeable separation between severity categories. The utilization of these severity values allows us to allocate a weighted measure to the faults identified by each test case, underscoring the significance of recognizing high-severity issues. This methodology ensures that test cases detect faults with substantial potential impact and contribute significantly to the overall evaluation. The assignment of severity values establishes a clear and objective foundation for determining the number of faults detected by each test case using historical data in our evaluation.

\[
\text{Cost}(t_i) = \frac{\text{Execution time of the test case } t_i}{1 + \sum_{n=1}^{k} \text{Total fault severity } t_i}
\]

### Table 1 Fault severity values

<table>
<thead>
<tr>
<th>Severity level</th>
<th>Severity value</th>
</tr>
</thead>
<tbody>
<tr>
<td>High severity</td>
<td>7</td>
</tr>
<tr>
<td>Medium severity</td>
<td>5</td>
</tr>
<tr>
<td>Less severity</td>
<td>3</td>
</tr>
<tr>
<td>Least severity</td>
<td>1</td>
</tr>
</tbody>
</table>
3.3 TOPSIS-AHP based approach to evaluate TD

This approach is proposed as a test case selection solution to TD problem in software testing. This technique attempts to improve the retest-all model by selecting a subset of the entire test set based on TD evaluation.

To evaluate the TD for a given test case, we adopted a computational approach based on an AHP-TOPSIS analytical hierarchy with a technique for order preference organized by similarity to an ideal solution. The basic idea of the proposed method is to select suitable test cases with the shortest distance from the Positive Ideal Solution $V^+$ and the farthest distance from the Negative Ideal Solution $V^-$. 

In TOPSIS, guaranteeing the consistency of the weighted normalized matrix is often difficult. AHP (Eji and Lung 1995) can be used to solve problems related to subjectivity and expert opinion by checking the consistency ratio of the judgment matrix (John 2002).

\[ Consistency\; Index = (\lambda_{\text{max}} - n)/(n - 1). \]

\[ Consistency\; Ratio = \frac{Consistency\; index}{\text{Random\; Index}}. \]

The AHP determines the weight to be given for each criterion, and the multi-criteria TOPSIS method is used to compute the TD score of each test. As mentioned, we focus on three objectives: risk related to a test case, code coverage, and test case cost. Therefore, in our model the \text{RandomIndex} (Alonso and Lamata 2006) is equal to 0.58 when $n = 3$. The AHP comparative judgment is shown in the following matrix.

\[
\begin{bmatrix}
\text{Risk} & \text{Coverage} & \text{Cost} \\
1 & 3 & 5 \\
\frac{1}{3} & 1 & 3 \\
\frac{1}{5} & \frac{1}{3} & 1 \\
\end{bmatrix}
\]

The matrix inputs were analyzed to determine the percentage weights for $w(\text{Risk}, \text{Coverage}, \text{Cost})$ such that:

\[
\sum_{n=1}^{3} w^n = 1.
\]

The results indicated the following: \text{Risk} = 63\%, \text{ Coverage} = 26\%, and \text{Cost} = 11\%. The next step was to measure the consistency of objective preferences arranged in the comparison matrix. The consistency ratio was evaluated to be 0.017, which is less than the permissible limit (Eji and Lung 1995). Our comparison were consistent. The aforementioned weights are also used in the TD computation for $\alpha$, $\beta$, and $\gamma$, to balance the importance of the three objectives.

Formally, for our problem we selected the ideal point $V^+ = (V^R_{\text{max}}, V^C_{\text{max}}, V^T_{\text{min}})$ and the negative ideal point $V^- = (V^R_{\text{min}}, V^C_{\text{min}}, V^T_{\text{max}})$, where $V^R_{\text{max}}, V^C_{\text{max}},$ and $V^T_{\text{min}}$ represent the best ideal points for risk, coverage, and cost respectively, and $V^R_{\text{min}}, V^C_{\text{min}},$ and $V^T_{\text{max}}$ represent the worst ideal points for risk, coverage, and cost respectively.

The value of TD is computed according to its distance from positive and negative ideal points. Positive and negative ideal points were defined based on their
beneficial characteristics. Time is a non-beneficial attribute, and a lower value (min) was desirable. Risk and coverage are beneficial attributes; therefore a higher value (maximum) is desirable. Therefore, the problem can be formulated as:

$$TD'(t) = \alpha \frac{\text{Risk}(t) - V_{\text{max}}^R}{V_{\text{min}}^R - V_{\text{max}}^R} + \beta \frac{\text{Coverage}(t) - V_{\text{max}}^C}{V_{\text{min}}^C - V_{\text{max}}^C} + \gamma \left(\frac{\text{Time}(t) - V_{\text{min}}^T}{V_{\text{max}}^T - V_{\text{min}}^T}\right)^{-1}$$

Weighting factors $\alpha$, $\beta$, and $\gamma$ are used to balance the importance of the three objectives: risk, code-segment coverage, and test-case cost. Here, we adopt the same weighting factors considered by AHP during the weighted normalized matrix in the TOPSIS. A test case with a higher TD is selected as the final decision. When $TD'(t)$ is large, this means that test case $t_i$ can test more segments/requirements and detect more errors during a unit interval of time. In other words, the higher the $TD'(t)$, the better the test case $t_i$.

Test cases $t_i$ with a high TD need to be integrated into the regression-testing test set. The proposed model is stable and credible because it combines subjectivity and objectivity when computing the TD value for $t_i$. Traditional TOPSIS usually constructs a weighted normalized matrix based on expert judgment. Therefore, AHP is integrated to check consistency and stabilize the model.

3.4 The technical debt computation algorithm TDCA

The algorithm executed a test case in the original test set $\{t_1, t_2, ..., t_n\}$, which was initially established by the tester. Each test case $t_i$ has the execution Cost$_i$ and coverage percentage. All the TD are stored in an array $\{TD_{t_1}, TD_{t_2}, ..., TD_{t_n}\}$. The algorithm is described in detail below. Please note that the algorithm pseudocode is abstract, and not all details are presented.

Algorithm TDCA
Input: Set of initial reqs $R$, set of modified reqs $R'$, set of test cases $TS$
Output: Test suite $TS'$

1. $TS'$ is empty, $C = 1$,
2. Ignore unmodified requirements (reach-ability analysis)
3. For (every element in $R'$) // loop through all $r \in R'$
4. Generate $T_i$: test cases set covering the requirement $R'_i$
5. End For
6. Max-Card = $|T_i|$
7. While (there are unmarked Requirements $R_i$)
8.   If ($C < $ Max-Card)
9.     select $T_i$ with smallest cardinality $C$
10.    Declare $t'$ and $t$
11.   For (every element $t \in T_i$)
12.      Compute number of occurrences where ($t \in T_i$)
13.      Select $t$ with max
14.      If (there is a tie between test cases $t'$ and $t$)
15.        Increase the cardinality // go back to line 9
16.     Else
17.         For (all $R_i$ that include $t$)
18.             Mark $R_i$ as satisfied
19.         Add $t$ to $TS'$
20.         End For
21.     End If
22.     End For
23. Else
24.   If ($t = t'$) // tie between $t'$ and $t$
25.      getTechnicalDebt($t$)
26.      getTechnicalDebt($t'$)
27.      If ($t$ score $\geq$ $t'$ score)
28.          Add $t$ to $TS'$
29.      Else
30.          Add $t'$ to $TS'$
31.      End If
32.   End If
33. End If
34. End While
35. Output $TS'$

The TD analyzer described below quantified the TD for each test case. This is called when similarity is detected among many test cases.
4 Case study and performance analysis

In this section, we describe an example and evaluate of the proposed TDCA model.

4.1 Example

To illustrate the proposed algorithm, we consider program $P$ and its control-flow graph in Fig. 1. The program reads the lengths of the three sides of a triangle and classifies the triangle as scalene, isosceles, right, or equilateral. The triangular area is then computed. The program displays a triangular class and area. As shown in Fig. 1 the code was segmented into $S_1 ... S_{11}$.

Table 2 lists the initial test cases used to test program $P$. Table 3 lists the test segment traversal information.

Table 4 establishes the test requirements based on the information inherent to a particular study. Table 5 presents the probability of error as a risk factor for each test case. The number of errors in each test case was determined as 10 percent of the number of segments traversed. Subsequently, higher impact-causing errors were identified, constituting 2 percent of the total number of errors detected. This implies that a small proportion (2%) of the overall errors carries a significantly greater impact compared to the remaining errors. The probability of errors is then calculated as the product of the percentage of the total number of errors and the percentage of higher-impact errors. The risk factors listed in Table 5 were computed using the following formula:

\[
TD'(t) = \alpha \cdot \frac{Risk(t) - V_{max}^R}{V_{min}^R - V_{max}^R} + \beta \cdot \frac{Coverage(t) - V_{max}^C}{V_{min}^C - V_{max}^C} + \gamma \cdot \left(\frac{Time(t) - V_{max}^T}{V_{max}^T - V_{min}^T}\right)^{-1}
\]

where $\alpha$, $\beta$, and $\gamma$ are weight factors, and $V_{min}$ and $V_{max}$ represent the minimum and maximum values of the respective metrics.

```plaintext
Input: Set of test cases $TS$
Output: Array Score []

1. Score [] = new array ||TS||
2. For (every element $i \in TS$) // loop through all elements
3. \[TD'(t) = \alpha \cdot \frac{Risk(t) - V_{max}^R}{V_{min}^R - V_{max}^R} + \beta \cdot \frac{Coverage(t) - V_{max}^C}{V_{min}^C - V_{max}^C} + \gamma \cdot \left(\frac{Time(t) - V_{max}^T}{V_{max}^T - V_{min}^T}\right)^{-1}\]
4. Score[$i$] = TD$_i$
5. End For
6. return Score []
7. getTechnicalDebt(index)
8. return Score[index]
```

Compute Technical Debt Algorithm
Fig. 1 Control flow graph for the triangle example

Table 2 Test cases for program P

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>t₁</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>t₂</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>t₃</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>t₄</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>t₅</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>t₆</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3 Segment traversal of test cases in program P

<table>
<thead>
<tr>
<th>Test case</th>
<th>Segment traversal information</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>S1, S2, S3, S5, S6, S7, S9, S11</td>
</tr>
<tr>
<td>t₂</td>
<td>S1, S2, S3, S5, S7, S10, S11</td>
</tr>
<tr>
<td>t₃</td>
<td>S1, S3, S4, S5, S7, S8, S11</td>
</tr>
<tr>
<td>t₄</td>
<td>S1, S3, S5, S7, S10, S11</td>
</tr>
<tr>
<td>t₅</td>
<td>S1, S2, S3, S5, S6, S7, S9, S11</td>
</tr>
<tr>
<td>t₆</td>
<td>S1, S2, S3, S5, S7, S10, S11</td>
</tr>
</tbody>
</table>
As mentioned earlier we used the TOPSIS method to obtain the best alternative for test case selection, which is closest to the ideal solution. In other words, the best test case has the most distant solution from the anti-ideal solution.

The process begins with the transformation of the original matrix shown in Table 6.

### Table 4 Test suite TS, testing requirements \( R_i \), and associated tests \( T_i \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( R_i )</th>
<th>( T_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S1</td>
<td>( { t_1, t_2, t_3, t_4, t_5, t_6 } )</td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
<td>( { t_1, t_2, t_5, t_6 } )</td>
</tr>
<tr>
<td>3</td>
<td>S3</td>
<td>( { t_1, t_2, t_3, t_4, t_5, t_6 } )</td>
</tr>
<tr>
<td>4</td>
<td>S4</td>
<td>( { t_5 } )</td>
</tr>
<tr>
<td>5</td>
<td>S5</td>
<td>( { t_1, t_2, t_3, t_4, t_5, t_6 } )</td>
</tr>
<tr>
<td>6</td>
<td>S6</td>
<td>( { t_1, t_5 } )</td>
</tr>
<tr>
<td>7</td>
<td>S7</td>
<td>( { t_1, t_2, t_3, t_4, t_5, t_6 } )</td>
</tr>
<tr>
<td>8</td>
<td>S8</td>
<td>( { t_5 } )</td>
</tr>
<tr>
<td>9</td>
<td>S9</td>
<td>( { t_1, t_5 } )</td>
</tr>
<tr>
<td>10</td>
<td>S10</td>
<td>( { t_2, t_4, t_6 } )</td>
</tr>
<tr>
<td>11</td>
<td>S11</td>
<td>( { t_1, t_2, t_3, t_4, t_5, t_6 } )</td>
</tr>
</tbody>
</table>

### Table 5 Probability for errors - risk factor

<table>
<thead>
<tr>
<th>Test case</th>
<th>Segment traversal info</th>
<th>( N )</th>
<th>( I )</th>
<th>Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>S1, S2, S3, S5, S6, S7, S9, S11</td>
<td>0.8</td>
<td>0.016</td>
<td>1.28</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>S1, S2, S3, S5, S7, S10, S11</td>
<td>0.7</td>
<td>0.014</td>
<td>0.98</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>S1, S3, S4, S5, S7, S8, S11</td>
<td>0.7</td>
<td>0.014</td>
<td>0.98</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>S1, S3, S5, S7, S10, S11</td>
<td>0.6</td>
<td>0.012</td>
<td>0.72</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>S1, S2, S3, S5, S6, S7, S9, S11</td>
<td>0.8</td>
<td>0.016</td>
<td>1.28</td>
</tr>
<tr>
<td>( t_6 )</td>
<td>S1, S2, S3, S5, S7, S10, S11</td>
<td>0.7</td>
<td>0.014</td>
<td>0.98</td>
</tr>
</tbody>
</table>

### Table 6 TOPSIS- original matrix

<table>
<thead>
<tr>
<th>Test case</th>
<th>Risk</th>
<th>Coverage</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>1.28</td>
<td>0.72</td>
<td>0.022</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>0.98</td>
<td>0.63</td>
<td>0.018</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>0.98</td>
<td>0.63</td>
<td>0.018</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>0.72</td>
<td>0.54</td>
<td>0.017</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>1.28</td>
<td>0.72</td>
<td>0.019</td>
</tr>
<tr>
<td>( t_6 )</td>
<td>0.98</td>
<td>0.63</td>
<td>0.017</td>
</tr>
</tbody>
</table>

Number of errors found(\( N \)) = \textit{10 percentage of segments traversed}

Higher impact causing Errors(\( I \)) = \textit{2 percentage of N}

Probability for Errors(\( E \)) = \(((N \times I) \times 100))

As mentioned earlier we used the TOPSIS method to obtain the best alternative for test case selection, which is closest to the ideal solution. In other words, the best test case has the most distant solution from the anti-ideal solution. The process begins with the transformation of the original matrix shown in Table 6.
into a normalized matrix (Table 7), and finally to a normalized weighted matrix, as shown in Table 8.

The test case’s execution time was measured in milliseconds using the JUnit Runner. The test case cost is computed using the objective 3 formula defined in Sect. 3.2.3.

The weight vector obtained by AHP as mentioned earlier is as follows: $W = (Risk = 63\%, Coverage = 26\%, \text{ and } Cost = 11\%)$. We apply the weight factors to the matrix in Table 7 to form the weighted matrix in Table 8.

Now, we can determine the best ideal point $V^+$ and worst ideal point $V^-$ as shown in Table 8. Because the performance of TD depends on high risk with high coverage and low execution time, we have classified this as $V^+ = (V_{max}^R, V_{max}^C, V_{max}^T)$ and $V^- = (V_{min}^R, V_{min}^C, V_{max}^T)$. Therefore, $V^+$ has a positive impact on TD and $V^-$ has a negative impact on TD.

Finally, we computed the TD value for each test case using the $TD'$ function and ranked them in ascending order according to their TD values, as shown in Table 8. As mentioned in the Introduction, cases with higher TD values must be selected during the regression testing phase.

Upon modifying S8, the reachability analysis indicated that the change in S8 might potentially influence all segments except S9 and S10. The set of modification requirements $R' = \{S1, S2, \ldots, S11\}$ is then used by the algorithm to find the set $T_{S'}$ from the regression testing that satisfies all potential modification requirements for TD. Using Table 3, the algorithm first marks segments S9 and S10 as unaffected, and ignores their links to $T_i$. RTD then searches for all potentially affected

---

**Table 7** TOPSIS- normalized matrix

<table>
<thead>
<tr>
<th>Test case</th>
<th>Risk</th>
<th>Coverage</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>0.495380373</td>
<td>0.453638382</td>
<td>0.483428924</td>
</tr>
<tr>
<td>$t_2$</td>
<td>0.379275598</td>
<td>0.396933584</td>
<td>0.395532756</td>
</tr>
<tr>
<td>$t_3$</td>
<td>0.379275598</td>
<td>0.396933584</td>
<td>0.395532756</td>
</tr>
<tr>
<td>$t_4$</td>
<td>0.27865146</td>
<td>0.340228786</td>
<td>0.373558714</td>
</tr>
<tr>
<td>$t_5$</td>
<td>0.495380373</td>
<td>0.453638382</td>
<td>0.417506798</td>
</tr>
<tr>
<td>$t_6$</td>
<td>0.379275598</td>
<td>0.396933584</td>
<td>0.373558714</td>
</tr>
</tbody>
</table>

**Table 8** TOPSIS- normalized weighted matrix with $Risk = 63\%, Coverage = 26\%, \text{ and } Cost = 11\%$

<table>
<thead>
<tr>
<th>Test case</th>
<th>Risk</th>
<th>Coverage</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>0.312089635</td>
<td>0.117945979</td>
<td>0.053177182</td>
</tr>
<tr>
<td>$t_2$</td>
<td>0.238943627</td>
<td>0.103202732</td>
<td>0.043508603</td>
</tr>
<tr>
<td>$t_3$</td>
<td>0.238943627</td>
<td>0.103202732</td>
<td>0.043508603</td>
</tr>
<tr>
<td>$t_4$</td>
<td>0.17555042</td>
<td>0.088459484</td>
<td>0.041091459</td>
</tr>
<tr>
<td>$t_5$</td>
<td>0.312089635</td>
<td>0.117945979</td>
<td>0.045925748</td>
</tr>
<tr>
<td>$t_6$</td>
<td>0.238943627</td>
<td>0.103202732</td>
<td>0.041091459</td>
</tr>
<tr>
<td>$V^+$</td>
<td>0.312089635</td>
<td>0.117945979</td>
<td>0.053177182</td>
</tr>
<tr>
<td>$V^-$</td>
<td>0.17555042</td>
<td>0.088459484</td>
<td>0.053177182</td>
</tr>
</tbody>
</table>
singletons: test \( t_3 \) in \( T_4 \) and \( T_8 \) is chosen first (cardinality 1) and included in \( TS' \). Because test \( t_3 \) covers segments \( S_1, S_3, S_5, S_7, \) and \( S_{11} \), the corresponding \( T_i \) value of those test cases are marked as satisfied. All unmarked \( T_i \) values of cardinality 2 should be considered in the next step. In this case, \( T_6 \) was chosen. A similarity was detected in \( T_6 \) between \( t_1 \) and \( t_5 \). Both test cases are referred to us in \( T_1, T_2, T_3, T_5, T_7, \) and \( T_{11} \), respectively. Thus, the process continues for the unmarked \( T_i \) with the next highest cardinality. Next, \( T_2 \) with cardinality 4 is considered. The test cases involved in the tie during the previous step were used again to compute the maximum coverage with cardinality 4. Again, there is a similarity between \( t_1 \) and \( t_5 \). Because the highest cardinality is reached, we call the TD analyzer.

The algorithm computes \( TD(t_1) \) and \( TD(t_5) \), as shown in Table 9. Test case \( t_5 \) has less time than \( t_1 \), and thus, its TD is more than \( t_1 \). Thus, \( t_5 \) was selected and added to the \( TS' \) set \( t_3, t_5 \). Subsequently, the algorithm continues to execute. When the algorithm is complete, it establishes an array list to store all segments \{ \( S_1, S_2, \ldots, S_{11} \) \}. First, the array list must be compared with the segment of \( t_3 \). If the segment appears simultaneously, it is removed from the array list. Next, \( t_5 \) should be compared with the array list, as \( t_3 \) above. The output is the remainder of the segments in the array list, \( S_{10} \).

Using the random selection approach (Bahsoon and Mansour 2002), selecting \( t_1 \) for inclusion in \( TS' \) results in marking \( T_2 \) and \( T_6 \) as satisfied since \( t_1 \) meets their corresponding requirements. Thus, the resulting regression sets \( TS' \) are \( t_1, t_3 \). With outputs \( t_1 \) and \( t_3 \), segment \( S_{10} \) is not tested, which can lead to a serious problem. When one of the important segments is not tested, errors that hide behind the segment cannot be detected. The random approach reduces the original test case set to use the minimized test cases to test the modification code and determine the most dangerous error. At the same time, the technology can help save time, space, and money on testing. However, TD is likely to develop when the number of test instances decreases.

### 4.2 Results analysis and comparison

We compared our Technical debt Evaluation Model (TDEM) with the customised technique for test selection presented in Beena and Sarala (2013) and multi-objective evolutionary algorithms (MOEAs) based on NSGA-II [23]. Table 10 was used as the test

<table>
<thead>
<tr>
<th>Test case</th>
<th>Segment traversal info</th>
<th>TD value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>( S_1, S_2, S_3, S_5, S_6, S_7, S_9, S_{11} )</td>
<td>4.05409978</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>( S_1, S_2, S_3, S_5, S_7, S_{10}, S_{11} )</td>
<td>3.079981401</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>( S_1, S_3, S_4, S_5, S_7, S_{8}, S_{11} )</td>
<td>3.079981401</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>( S_1, S_3, S_5, S_7, S_{10}, S_{11} )</td>
<td>2.136183552</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>( S_1, S_2, S_3, S_5, S_6, S_7, S_{9}, S_{11} )</td>
<td>4.097599867</td>
</tr>
<tr>
<td>( t_6 )</td>
<td>( S_1, S_2, S_3, S_5, S_7, S_{10}, S_{11} )</td>
<td>3.100469266</td>
</tr>
</tbody>
</table>
case statement coverage for (Beena and Sarala 2013) and as the initial parent population for [23].

The obtained output of the triangle example when using (Beena and Sarala 2013) is $T = \{t_3\}$ when the modified statement vector is equal to S8. In [23] we used two objective branch coverage (to maximize), and execution cost (to minimize). The obtained output is $T = \{t_1, t_5\}$. Considering the overall result of the triangle program example, our TDEM approach performed better than those in Beena and Sarala (2013), [23]. This suggests that if the branch coverage and cost factors are equal TDEM is a better choice to reduce the impact of TD. In addition, we offer the tester the opportunity to correct errors earlier and to reduce TD by integrating the concepts of risk percentage, cost, and test effectiveness. Table 11 shows the comparison between TDEM, the customized technique for test selection presented in Beena and Sarala (2013), Adaptive Random Testing (ART) (Rubing Huang and Xia 2020) and multi-objective evolutionary algorithms (MOEAs) based on NSGA-II [23].

Error density is defined by dividing the number of defects that have been identified during testing by the lines of code and then multiplying the result by 100 to represent it as a percentage. In our study, error density is calculated to provide a comprehensive understanding of testing effectiveness. A lower error density indicates a higher quality codebase with fewer defects per line of code, while a higher error density suggests a higher likelihood of defects within the code. In the context of regression testing, safety is defined as the effectiveness of error detection in the segments of code that have been modified.

$$\text{Error Density (\%) = \frac{\text{Number of Defects}}{\text{Lines of Code}} \times 100}$$

<table>
<thead>
<tr>
<th>Test case</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
<th>S9</th>
<th>S10</th>
<th>S11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_3$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_4$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_5$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_6$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 10** Initial parent population for triangle example

**Table 11** Triangle example - comparison between TDEM and other approaches

<table>
<thead>
<tr>
<th></th>
<th>TDEM</th>
<th>MOEA [23]</th>
<th>Test selection (Beena and Sarala 2013)</th>
<th>ART (Rubing Huang and Xia 2020)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error density</td>
<td>5.26%</td>
<td>10.5%</td>
<td>15%</td>
<td>10.5%</td>
</tr>
<tr>
<td>Branch coverage</td>
<td>91%</td>
<td>72%</td>
<td>63%</td>
<td>82%</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>0.89</td>
<td>0.99</td>
<td>0.43</td>
<td>1.39</td>
</tr>
<tr>
<td>Safety</td>
<td>100%</td>
<td>96%</td>
<td>91%</td>
<td>94%</td>
</tr>
</tbody>
</table>
The safety metric is calculated by determining the ratio of errors detected in the modified segments to the total number of test cases applied to these segments, expressed as a percentage. The objective is to ensure that the applied test cases effectively uncover errors or regressions introduced during modifications.

\[
\text{Safety} = \frac{\text{Number of Errors Detected in Modified Segments}}{\text{Total Number of Test Cases in Modified Segments}} \times 100
\]

This formula captures the essence of safety in regression testing by emphasizing the proportion of errors identified within the modified segments relative to the total number of test cases executed in those areas.

In the subsequent results analysis, we selected two Java-based systems from the GitLab open-source development platform. The first system, referred to as the Calculator program, is designed to recognize handwritten equations in images and calculate their results [50]. This project comprises a total of 526 lines of code (LOC), for which we generated a test set consisting of 56 test cases. The second project is referred to as the Online Shopping system with around 22,000 lines of code, offering a comprehensive range of features such as support for all devices, fast searchable product listings, customizable elements, category and item management, order management with various statuses (Confirmed, Preparing, On Way, Delivered), payment gateway integration, voucher support, favorite category and product management, as well as OTP authentication. In the testing scenario for this approach, our focus centers on the interaction between the OrderService and PaymentService components. More precisely, we verify that the placeOrder method of the OrderService correctly interacts with the PaymentService, covering pertinent code segments. We designed 1245 test cases for this purpose. Tables 12 and 13 illustrate the experimental outcomes for the Calculator and Online Shopping projects, showcasing metrics such as Error Density, Branch Coverage, Time, and Safety for different testing approaches. The comparison reveals varying possibilities for Error Density and Safety, highlighting the effectiveness of different testing techniques in these diverse systems.

The provided tables present a comparison of four different testing approaches (TDEM, MOEA, Test Selection, and ART) across three software scales: small (Triangle), medium (Calculator), and large (Online Shopping). The comparison is based on four metrics: Error Density, Branch Coverage, Time (in milliseconds), and Safety.

For the Calculator project, the Error Density ranges from 1.52 to 2.09%, with TDEM and ART having the lowest and highest values, respectively. Similarly, for
the Online Shopping project, the Error Density varies from 1.35 to 1.81%, with TDEM again having the lowest value and Test Selection having the highest. This indicates that TDEM generally leads to lower error density compared to other approaches for both small and large software scales.

In terms of Branch Coverage, TDEM consistently achieves higher percentages compared to other approaches for both projects. This suggests that TDEM provides more comprehensive coverage of code branches, potentially leading to more thorough testing and detection of defects.

Regarding Time, the values vary across approaches and projects. However, TDEM generally exhibits competitive performance in terms of time efficiency compared to other approaches, especially for the Calculator project.

In evaluating Safety, which represents the effectiveness of applied test cases in detecting errors or regressions, TDEM and MOEA consistently demonstrate higher percentages compared to Test Selection and ART. This implies that TDEM and MOEA are more reliable in identifying errors introduced during modifications.

The approach proposed in Beena and Sarala (2013) focuses on modified and deleted statements linked to requirements to guarantee that the selected test cases can detect errors in the affected or modified segments. However, it is important to note that the modified segments can also affect other segments, and in some cases, it is crucial to retest the unaffected segments, particularly if they represent the main parts of the program. According to our TD model in regression testing, parameter $E_{Ti}$ selects test cases with a high probability of detecting test cases with more errors. Therefore, the model could find more errors during the regression testing phase. In our proposed approach, the remaining segments are listed as printed outputs, which can help the tester know which segments are not tested, making the testing plan for the next version much easier.

The technique presented in Rubing Huang and Xia (2020) works by randomly selecting test cases from the input space, but with a bias towards untested areas of the input space. This bias is achieved by using an adaptive mechanism that modifies the probability distribution. This can easily affect the found error probability and the safety parameters, as shown in Tables 12 and 13. Once the input space becomes wider, the probability of finding errors through random testing decreases to a Mid-level, the branch coverage percentage drops below 80%, and the process becomes time-consuming.

<table>
<thead>
<tr>
<th></th>
<th>TDEM</th>
<th>MOEA [23]</th>
<th>Test selection (Beena and Sarala 2013)</th>
<th>ART (Rubing Huang and Xia 2020)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error density</td>
<td>1.35%</td>
<td>1.46%</td>
<td>1.81%</td>
<td>1.72%</td>
</tr>
<tr>
<td>Branch coverage</td>
<td>87%</td>
<td>81%</td>
<td>76%</td>
<td>71%</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>1.21</td>
<td>1.39</td>
<td>1.78</td>
<td>1.55</td>
</tr>
<tr>
<td>Safety</td>
<td>91%</td>
<td>90%</td>
<td>89%</td>
<td>86%</td>
</tr>
</tbody>
</table>
The TDEM model is generally more effective in detecting faults that are related to specific parts of the code, such as boundary conditions or modified parts. This is because the approach focuses on exercising specific parts of the code that are likely to be associated with faults. In terms of efficiency, the model selects test cases that require less testing time. Compared to [23], the TDEM selects test cases using global knowledge test case data, finding the most effective test case and placing it in the regression testing set. This makes the regression test more effective because it requires less time and detects more errors. The iterative process in [23] is time-consuming and can lead to duplicate solutions. While the TDEM model excels in identifying many types of faults, it may encounter challenges when it comes to detecting issues arising from complex interactions between different segments of the code. This characteristic highlights the need for complementary testing approaches to ensure comprehensive fault detection across all dimensions of the software architecture.

Most of the proposed testing approaches, such as Leitner et al. (2007); Yoo et al. (2009), obtained TD by employing test team experience and skills. Such models can pose potential risks in the omitted test cases. The resultant testing sets are not safe for regression testing. Safety is important in regression testing and should include the possibility of errors, coverage of segments, and the time required. The technique aims to reduce the number of test cases for regression testing. It could save more time in fixing the problem immediately before executing the next test; however, it is not sufficiently safe. As mentioned previously, if some test cases are omitted, some errors are ignored, and some core codes are not involved in the new testing. However, the TD proposed in this study is safer and generates a hierarchy of test-case importance for the tester.

Overall, the comparison highlights the varying performance of different testing approaches across different software scales. TDEM emerges as a promising approach, particularly for achieving low error density, high branch coverage, and effective error detection. However, further analysis and experimentation may be necessary to validate these findings and identify the most suitable testing approach for specific software development contexts.

5 Conclusion and future work

Regression testing is a critical aspect of software development, with recent attention focused on addressing the challenges posed by Technical Debt (TD). Despite recognizing TD’s impact on regression testing, effectively evaluating it remains a significant challenge. This study introduces a novel metric system tailored for assessing TD in regression testing, aiming to overcome issues related to random comparisons and the requisite expertise of human evaluators. The metric system endeavors to generate a test suite that serves as a dependable indicator of TD value while facilitating efficient bug resolution. By leveraging Analytical Hierarchy Process (AHP) and Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS), the paper demonstrates the modeling of the metric system, illustrating its application in familiar testing scenarios such as the testing triangle.

© Springer
and calculator problems. A comparative analysis with a specific multi-objective optimization method and standard coverage methodology further elucidates the proposed approach’s efficacy.

This approach’s strength lies in its capacity, as demonstrated by the selected test suite, to make well-balanced decisions. It accomplishes this by identifying modifications, employing risk analysis, and carefully weighing testing costs against potential technical debt. The results unequivocally demonstrate the superior performance of the proposed TD metric system over an extended period. Future research will focus on integrating additional quality metrics into the test selection process to further refine TD evaluation. Additionally, exploring the method’s applicability to various industrial-scale test cases will be a key area of investigation.

Author contributions All authors have contributed significantly to this work [Anis Zarrad] implemented the technical debt evaluation approach, conducted the experiments, reviewed and edited the manuscript, and provided suggestions for revisions.[Rami Bahsson] designed the research study, developed the methodology, supervised the overall progress of the project, reviewed the manuscript, and provided critical feedback[Priya Manimaran] contributed to the analysis of the experimental results, assisted in the analysis of the data, conducted the literature review, and contributed to the discussion section of the manuscript. All authors have reviewed and approved the final manuscript.

Funding The research presented in this paper was conducted without any external funding. We declare that no specific grants, financial support, or other forms of funding were received for the design, implementation, analysis, or publication of this research.

Declarations Conflict of interest The corresponding author confirms on behalf of all authors that we have no Conflict of interest to declare, and we affirm that this research has been conducted with complete objectivity and transparency.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

A program that uses artificial neural networks to recognize handwritten equations in images and calculates its result. https://github.com

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Anis Zarrad1 · Rami Bahsoon1 · Priya Manimaran1

✉ Anis Zarrad
a.zarrad@bham.ac.uk
https://www.birmingham.ac.uk/index.aspx
Rami Bahsoon
r.bahsoon@bham.ac.uk

Priya Manimaran
pmanimaran@conestogac.on.ca

1 University of Birmingham, Edgbaston, Birmingham B15 2TT, UK