EASR

Engineering and Applied Science Research

https://www.tci-thaijo.org/index.php/easr/index

Published by the Faculty of Engineering, Khon Kaen University, Thailand

Test case model for maintaining software under the concept of risk and requirement-based prioritization

Thacha Lawanna*

International College of Digital Innovation, Chiang Mai University, Chiang Mai 50200, Thailand

Received 30 March 2023
Revised 6 February 2024
Accepted 13 February 2024

Abstract

Test case growth in software maintenance poses multifaceted challenges, including extended execution times, heightened costs, maintenance complexities, and increased system intricacy. Larger test cases not only introduce redundancy but also elevate the risk of overlooking defects during testing, emphasizing the need to manage test case expansion effectively while maintaining adequate testing coverage. One approach to mitigating this issue is risk-based prioritization, which focuses on testing critical and high-risk areas of the software. By emphasizing these pivotal aspects, this method aims to streamline test cases, optimizing testing efforts. However, it has limitations, as it may not comprehensively address all functional requirements, necessitating an in-depth understanding of the software system's architecture and vulnerabilities. Furthermore, requirements-based prioritization ensures thorough testing of all functional requirements, crucial for meeting industry standards and regulatory compliance. This approach utilizes clear criteria provided by requirements to evaluate whether software functions as intended. While effective in guaranteeing comprehensive coverage, it may fall short in addressing all potential risks and vulnerabilities, potentially undermining its effectiveness in identifying and prioritizing critical areas of the software. Considering these challenges, a nuanced approach that combines risk-based and requirements-based prioritization could offer a balanced solution. By leveraging the strengths of both methods, it becomes possible to address the diverse aspects of software testing, optimizing coverage, and efficiently managing test case growth. Moreover, the proposed model demonstrates promising results in tackling these challenges. With a notable test suite size reduction of 0.61-1.05% compared to traditional methods and a faultless percentage surpassing comparative studies by 0.05-0.75% across seven C-language System Under Test (SUT) programs, the model showcases its efficacy in improving testing efficiency and reliability. These findings underscore the potential of adopting advanced models to navigate the complexities associated with test case management and software testing in general.

Keywords: Test case, Prioritization, Selection, Program, Maintenance, Reduction

1. Introduction

Software maintenance (*SM*) is the process of modifying and updating software to fix identified problems, address bugs, and improve functionality. The two main categories of *SM* sm are corrective and adaptive. The process involves several steps, including identifying and analyzing problems, creating a plan, implementing changes, testing, and releasing the updated software. These activities can lead to an increased size of test cases due to various factors. Such factors include the need to update or create new test cases when changes are made to the software, the impact of changes on the complexity of the software, creation of specialized test cases to address specific problems or bugs, and the need to consider all the different configurations and environments in which the software operates. These factors can result in a challenging testing process, making it more difficult to manage and execute test cases [1].

Therefore, test cases can become larger due to increasing software complexity, addition of new features, poor design and organization, and insufficient planning. Larger test cases can result in delays, increased costs, and difficulty in managing and executing software. Software testing teams can manage larger test cases by breaking them up into smaller units, using automated testing tools, and developing effective strategies for testing. Automated testing tools can streamline the testing process, making it more efficient to execute larger test cases, and help identify defects quickly and accurately [2].

The growth of test cases in size can lead to various challenges in software testing. These include increased time and resource requirements for execution, difficulties in maintenance and updating, increased complexity, the presence of redundant test cases, and an increased risk of defects. These challenges can impact the quality and reliability of the software being tested, making it important for testing teams to develop strategies for managing and mitigating them [3].

To solve this problem, test case selection (TCS), minimization (TCM), and prioritization (TCP) are effectively developed. TCS identifies and selects the most important and relevant test cases for a particular testing scenario. This can be achieved by prioritizing adequate test coverage (TC) based on factors such as risk, functionality, and criticality [4]. By selecting the most important test cases, testing teams can reduce the number of required test cases, test cases required, while still ensuring adequate TC. Furthermore, TCM removes redundant or unnecessary test cases. This can be achieved by analyzing the test suite (T) and detecting test cases that cover the same functionality or requirements as other test cases. By removing redundant test cases, testing teams can reduce the overall size

*Corresponding author.
Email address: thacha.l@icdi.cmu.ac.th
doi: 10.14456/easr.2024.26

of the T, while still ensuring that all critical functionality is tested [5]. Furthermore, TCP prioritizes the most critical and high-risk areas of the software to focus testing efforts. This approach involves evaluating the risks associated with the software and prioritizing testing efforts based on the level of risk associated with different areas of the software. By prioritizing testing in this way, testing teams can reduce the number of test cases required while still ensuring that the most important areas of the software are thoroughly tested [6].

By prioritizing testing based on the risk and impact of potential defects, testing teams can focus their efforts on the most critical test cases. This means that they can reduce the number of test cases needed to achieve the desired level of TC. A software development team may have identified 100 test cases that need to be executed to achieve a desired level of test coverage for a new feature they are developing [7]. However, they realize that some of these test cases may not be necessary or are redundant. By prioritizing the most critical test cases, they can achieve the same level of *TC* with fewer tests. After a careful analysis, the team may decide that they can achieve the same level of *TC* with only 70 test cases, by focusing on the most critical requirements and functionalities. Through a reduction of the number of *tc* needed, the team can save time and resources while still ensuring that the new feature is thoroughly tested before release [8].

By reducing the number of test cases, testing teams can reduce time and conserve resources, as well as improving their testing efficiency. Additionally, by prioritizing the most critical test cases, testing teams can identify high-risk defects early in the testing process, which can prevent issues from escalating and becoming more costly to address later in the software development cycle [9].

This paper focuses on *TCP*, which is the process of ordering test cases based on their relative importance or urgency. The goal of TCP is to ensure that the most critical *tc* are executed first to identify and address high-risk issues as early as possible [10]. There are several techniques that can be used for TCP. These include (i) risk-based prioritization, a technique that involves assessing the risk associated with each test case and prioritizing test cases based on their potential impact on the system. Also, (ii) requirements-based prioritization involves prioritizing test cases based on their coverage of the system's requirements, with higher priority given to test cases that cover critical or high-priority requirements. Additionally (iii), business value-based prioritization, it includes giving initial emphasis to test cases based on the business importance of the system functionality that they cover. Furthermore (iv), time-based prioritization, prioritization emphasizes test cases based on their deadline or scheduled release date. Finally, (v) dependency-based prioritization, which concerns emphasizing test cases based on their dependencies on other test cases or system components. However, risk-based prioritization and requirements-based prioritization are two methods used to emphasize test cases and that have different advantages and disadvantages depending on the specific context and objectives of the testing process [11].

This paper proposes a combined risk-based and requirements-based prioritization scheme for optimizing testing resources and assuring that both critical risks and functional requirements are thoroughly tested. Ultimately, the choice of prioritization method will depend on the specific context and objectives of the testing process.

2. Materials and methods

2.1 Seven programs used in the experimental studies

Table 1 presents specifications for C-language systems under test (SUTs). The following is an explanation of the columns contained in this table. The 'Name' column displays the name of the object that a particular SUT represents. The 'LOC' column shows the total lines of code (LOC) in each source code file, including comments. The 'No. of test cases ' column indicates the number of available test cases for the program. This number is obtained from fault matrices within the SUT. The 'No. of Fault' column shows the number of faults present in the SUT. In a multisession program, this number represents the sum of faults found in each version. For space fault objects, each faulted version of the SUT contains only one fault, so the number of faults is equal to the number of faulted versions [12].

Table 1 Specifications for C-language Systems Under Test (SUTs)

Name	LOC	No. of test cases	No. of Faults
printtokens	726	4113	7
printtokens2	570	4115	10
replace	564	5542	32
schedule	3412	2650	9
schedule2	5374	2710	10
tcas	173	1608	41
totinfo	565	1052	23

2.1.1 Basic knowledge of test case prioritization (TCP)

TCP is a crucial strategy employed in software testing to optimize the order in which test cases are executed. It is a method of ensuring that the most critical defects are identified and resolved early in the software development life cycle, which ultimately reduces the overall cost of testing and time-to-market [13]. In this article, we will discuss the importance of TCP and its benefits in various software testing environments. The prioritization process is an essential aspect of TCP. It involves evaluating each test case and assigning a priority level based on various risk factors. These factors can include the likelihood of detecting a defect, the severity of the potential defects, and the impact on the software or end-users. Based on these risk factors, priority levels are assigned to each test case, which then helps determine the order in which the test cases will be executed [14]. One of the significant advantages of TCP is that it ensures that the most critical defects are identified and resolved early in the software development cycle. By prioritizing high-risk test cases, software testers can focus their efforts on the most critical areas of the software and identify serious defects that may impact the software's functionality. By doing so, it is possible to address these defects early, which can ultimately reduce the testing costs and time-to-market [15].

Another benefit of TCP is that it is particularly useful in agile or continuous testing environments, where there are time and resource constraints. In such an environment, it is essential to identify and address defects quickly and efficiently, as delays can have a significant impact on overall project timelines. TCP helps in such scenarios by enabling identification of critical defects early, thus enabling software testers to address these defects before they become more significant problems [16].

Furthermore, TCP is also useful in complex software systems that have multiple interdependent components. In such systems, early identification of critical defects is crucial, as any defect in one component can have a cascading effect on other components, resulting in more significant issues at later stages. By prioritizing high-risk test cases in such systems, software testers can focus their efforts on the most critical areas, ensuring that impactful defects are identified and resolved before they can impact other components [17].

Therefore, TCP is a useful strategy in software testing that can help reduce testing costs and time-to-market while ensuring that critical defects are identified and resolved early. By evaluating each test case and assigning a priority level based on various risk factors, software testers can focus their efforts on the most critical areas of the software, enabling them to identify and address major defects before they can cause more significant problems. Additionally, TCP is particularly useful in agile or continuous testing environments, where time and resource constraints demand quick and efficient defect identification and resolution. Ultimately, TCP is a valuable tool in any software testing process, enabling software testers to improve the overall quality and reliability of the software they develop [18].

2.1.2 Risk-based prioritization (R_1)

Risk-based prioritization is a valuable approach that organizations use to determine which tasks or projects require focus based on their level of risk. By prioritizing those that pose the greatest potential impact on the organization's goals or objectives, organizations can maximize their resources, mitigate risks, and demonstrate due diligence to stakeholders. By allocating resources to the areas that require the most attention, organizations can address potential problems before they become major issues, reducing the likelihood of negative consequences. Also, prioritizing tasks or projects based on their level of risk can help organizations make informed decisions and prioritize actions that align with their overall strategy [19]. There are six steps for reducing test cases by using risk-based prioritization.

```
Step 1: Identify and evaluate risks associated with the software system.
risks = identifyAndEvaluateRisks(softwareSystem)
Step 2: Prioritize testing efforts based on the level of risk.
testingPriorities = prioritizeTestingEfforts(risks)
Step 3: Determine test cases covering the highest-risk areas.
testCases = generateTestCases(softwareSystem)
highRiskTestCases = prioritizeTestCases(testCases, testingPriorities)
Step 4: Focus testing efforts on the highest-priority test cases.
executeTestCases(highRiskTestCases)
Step 5: Eliminate redundant or unnecessary test cases.
filteredTestCases = eliminateRedundantTestCases(testCases, highRiskTestCases)
Step 6: Continuously monitor and update risk assessment and testing priorities.
while testingProcessOngoing:
    updateRiskAndPriorities(risks, testingPriorities)
# Additional testing steps as needed
```

This approach offers a comprehensive and adaptive framework for software testing, spanning these six strategic steps to ensure a thorough and efficient examination of a software system. Initially, the process is initiated by carefully identifying and evaluating risks associated with the software. This foundational step serves as a bedrock for subsequent testing decisions, offering insights into potential pitfalls and vulnerabilities. Building upon risk assessment, the algorithm progresses to prioritize testing efforts based on the perceived level of risk in different areas of the software. This step aims to strategically allocate resources to high-risk areas, ensuring that testing efforts align with the critical aspects of the system. The subsequent step involves determination of test cases that specifically target the highest-risk areas, emphasizing precision in the testing strategy. The execution phase follows, with the algorithm directing its focus towards the highest-priority test cases. This intentional concentration on critical functionalities ensures a robust examination of the most vital components of the software. Notably, the pseudocode incorporates a step to optimize testing resources by eliminating redundancy. Redundant or unnecessary test cases that overlap with higher-priority areas are identified and discarded, streamlining the testing process while maintaining comprehensive coverage. A pivotal feature of the pseudocode lies in its emphasis on adaptability. Continuous monitoring and updating of risk assessments and testing priorities occurs throughout the testing process. This iterative approach allows the algorithm to dynamically respond to changes, emerging risks, or new insights that may arise during the testing lifecycle. This adaptability ensures that the testing strategy remains aligned with the evolving nature of the software system, fostering resilience and responsiveness in the face of dynamic development environments. In essence, the pseudocode encapsulates a holistic and nuanced approach to software testing, combining risk-aware prioritization, resource optimization, and ongoing refinement for comprehensive and effective testing.

The size of the test suite can be reduced while still guaranteeing that the most important areas of the software are thoroughly tested. This method can save time and conserve resources, while improving the overall quality and reliability of the software [20]. While risk-based prioritization is a valuable approach for many organizations, there are some potential drawbacks to consider. One of the main challenges is that it can be difficult to accurately assess and quantify risks. This is especially true for complex systems or situations where there are many interacting variables [21]. Additionally, risk-based prioritization may not always align with an organization's values or long-term goals. In some cases, there may be projects or tasks that are deemed low-risk but are still important for achieving strategic objectives. Finally, risk-based prioritization can sometimes create a "risk averse" culture where employees focus primarily on avoiding potential problems rather than pursuing new opportunities. Despite these challenges, many organizations find that the benefits of risk-based prioritization outweigh its drawbacks, and use it as a valuable tool for managing their resources and achieving their goals [22, 23].

2.1.3 Requirements-based prioritization (R_2)

Requirements-based prioritization is an approach to project management that focuses on ranking tasks or projects based on their importance to meeting specific requirements or objectives. There are several reasons why organizations use requirements-based

prioritization. First, it helps ensure that the most critical requirements are met first, which is essential for the success of a project. By prioritizing tasks based on their impact on meeting specific requirements or objectives, organizations can ensure that they are making the best use of their resources and time. Additionally, it helps keep the project team focused on the most important issues, which can be especially helpful in complex projects where there are many different tasks and objectives. Finally, it can help ensure that stakeholders are satisfied with the project outcomes by ensuring that their most important requirements are met first. By aligning project tasks and priorities with stakeholder needs and requirements, organizations can increase the likelihood of project success and ultimately achieve their goals [24]. The following five steps for reducing test cases test cases by using requirements-based prioritization are listed.

```
Step 1: Identify the software requirements
softwareRequirements = identifySoftwareRequirements()
Step 2: Prioritize the requirements
prioritizedRequirements = prioritizeRequirements(softwareRequirements)
Step 3: Identify key functions and features
keyFunctionsAndFeatures = identifyKeyFunctionsAndFeatures()
Step 4: Create test cases
testCases = createTestCases(keyFunctionsAndFeatures)
Step 5: Verify requirements
verifyRequirements(testCases)
```

In this pseudocode, a systematic approach to software development and testing is outlined in five key steps. The first step involves the identification of software requirements, a crucial foundation for the entire development process. The identifySoftwareRequirements function is a placeholder for the actual process of gathering and documenting the software's functional and non-functional specifications. Following identification of requirements, the second step involves prioritizing them based on their significance to project objectives. The prioritized requirements function represents the process of assigning importance levels to different requirements, ensuring that development efforts focus on the most critical aspects first. In the third step, the pseudocode suggests identifying key functions and features. The identifyKeyFunctionsAndFeatures function is a placeholder for the process of identifying essential functionalities and features crucial for the software's success. The fourth step involves creating test cases based on the identified key functions and features. The createTestCases function signifies generation of test scenarios and inputs that validate the software's intended behavior, ensuring comprehensive testing coverage. Finally, the fifth step involves verifying the identified requirements through execution of test cases. The verifyRequirements function represents the process of systematically assessing whether the implemented software meets the specified requirements, ensuring the desired functionality and minimizing the risk of defects. Together, these pseudocode steps provide a structured framework for software development, emphasizing requirements identification, prioritization, key feature identification, comprehensive test case creation, and rigorous verification.

Below is an example of how to determine the percentage of test coverage using prioritization based on project requirements. If a software development project has 100 test cases and 50 requirements associated with those test cases, each requirement is assigned a priority score based on its importance to meeting project objectives, ranging from 1 to 5 (with 5 being the highest priority). The test team prioritizes testing based on these requirement priority scores. After reviewing the requirements and their priority scores, the test team focuses their efforts on the requirements with the highest priority scores. In this example, they may identify 20 requirements with a priority score of 5, 15 requirements with a priority score of 4, 10 requirements with a priority score of 3, and 5 requirements with a priority score of 2.

They then determine the number of test cases associated with each requirement priority level. For example, the 20 requirements with a priority score of 5 have a total of 60 test cases associated with them, the 15 requirements with a priority score of 4 have a total of 30 associated test cases, the 10 requirements with a priority score of 3 have a total of 5 test cases, and the 5 requirements with a priority score of 2 are covered by a total of 5 test cases. Finally, the total number of test cases covered by the prioritized testing approach is calculated. In this case, the total number of test cases associated with the prioritized requirements is 100 (60 + 30 + 5 + 5). The percent test coverage achieved through the requirement-based prioritization approach can be calculated as follows:

$$\%TC = \frac{T_c}{T'} \times 100\% \quad (1)$$

T_c = the total number of test cases covered and T' = the total number of test cases.

In this example, the requirement-based prioritization approach achieves 100% test coverage, as all 100 test cases are associated with the prioritized requirements. By following these steps, the testing team can reduce the size of the test suite while ensuring that the most important areas of the software are thoroughly tested. Although requirements-based prioritization is a valuable approach for many organizations, there are some potential drawbacks. Specifically, it can be difficult to determine which requirements are truly critical to the success of a project. In some cases, stakeholders may have conflicting requirements or may not fully understand the implications of their requests, which can make it challenging to effectively prioritize tasks. Moreover, requirement-based prioritization may not always align with an organization's overall strategy or long-term goals. In some cases, there may be tasks or projects that are deemed less important in terms of meeting specific requirements but are still critical for achieving broader organizational objectives [25]. Finally, requirements-based prioritization can sometimes result in a narrow focus on meeting specific requirements rather than considering the entire project. This can result in missed opportunities or overlooked risks that impact project success. Despite these challenges, many organizations find that requirement-based prioritization is a valuable tool for managing their resources and achieving their goals when used appropriately [26].

2.1.4 Proposed model (R_3)

The objective of the model proposed in the current study is to effectively prioritize project tasks and associated requirements to ensure successful project completion within the given timeline and budget. This involves creating a comprehensive list of tasks and requirements, assessing each task's risk score based on its potential impact on project success, and assigning each requirement a priority

score based on its importance in meeting project objectives. The prioritization process involves evaluating tasks based on their risk score, with higher-risk tasks receiving greater priority, and further prioritizing tasks within each risk-based priority group based on their associated requirements priority score. This approach ensures that the most critical tasks and requirements are addressed first, minimizing the potential impact of project risks and delays. Eventually, the goal is to start with the highest-priority task and continue working on tasks in the order of priority until all requirements have been met. This ensures that the project stays on track and is aligned with stakeholder needs and goals, ultimately resulting in a successful project outcome. The algorithm of the proposed model is explained as follows:

```
Step 1: Create a list of project tasks and associated requirements
projectTasks = ['Task A', 'Task B', 'Task C']
taskRequirements = {'Task A': ['Req1', 'Req2'], 'Task B': ['Req2', 'Req3'], 'Task C': ['Req1', 'Req3']}
Step 2: Assign each task a risk score based on its potential impact
riskScores = {task: calculateRiskScore(requirements) for task, requirements in taskRequirements.items()}
Step 3: Assign each requirement a priority score based on its importance
priorityScores = {req: calculatePriorityScore(req) for reqList in taskRequirements.values() for req in reqList}
Step 4: Prioritize tasks based on their risk scores
prioritizedTasks = sorted(projectTasks, key=lambda task: riskScores.get(task, 0), reverse=True)
Step 5: Within each risk-based priority group, further prioritize tasks based on requirement priority scores
finalTaskPriorityOrder = sorted(prioritizedTasks, key=lambda task: max(priorityScores.get(req, 0) for req in
taskRequirements[task]), reverse=True)
Step 6: Begin working on the highest-priority task and continue until all requirements are met
for task in finalTaskPriorityOrder:
    executeTask(task)
```

Application of the proposed model to a list of diverse test cases involves a strategic approach aimed at optimizing resources, managing risks, and ensuring successful validation of critical functionalities. A detailed analysis of each program within this comprehensive testing framework is necessary. The first program, "printtokens," with its 4113 test cases, is the initial focus. This program undergoes a rigorous risk assessment where each test case's impact on project success is examined. Concurrently, individual testing requirements within the "printtokens" suite are assigned priority scores based on their significance to project objectives. This dual evaluation process guides prioritization of test cases, ensuring that those with higher risk scores and requirement priority levels take precedence in the testing sequence. Moving to "printtokens2," which includes 4115 test cases, a similar thorough evaluation occurs. The risk assessment considers factors such as the complexity of the code and the potential criticality of functionalities. Likewise, the prioritization of requirements influences the testing order, resulting in a nuanced sequence that aligns with the overarching project objectives. The interplay between risk scores and priority levels ensures that testing efforts are not only comprehensive but are also strategically directed toward the most impactful areas of the software. The "replace" program, featuring a substantial 5542 test cases, emerges as a focal point due to its potential high risk. This high number of test cases indicates complexity and necessitates a granular analysis of its project impact. Each test case is evaluated and specific functionalities, such as intricate replacement algorithms, may be identified as high-risk. Consequently, the risk scores influence the prioritization of "replace" in the testing sequence, emphasizing the need for early and thorough validation of its critical components. "Schedule" and "schedule2," comprising 2650 and 2710 test cases, respectively, undergo a complete risk and priority evaluation process. The sheer number of test cases highlights the significance of these programs in the overall testing. Beyond the quantitative aspects, the focus shifts to the specific functionalities associated with scheduling within these programs. For example, scheduling algorithms might be identified as high-risk components, warranting careful attention in the testing sequence. The risk and priority assessments, therefore, contribute to a nuanced prioritization within the broader risk-based categories, ensuring that testing efforts align with critical project objectives.

The "tcas" program, with 1608 test cases, demands a nuanced analysis of its potential risks and priority requirements. This program may involve critical functionalities related to traffic collision avoidance systems, necessitating a thorough examination of its impact on overall project success. The risk scores and prioritization of requirements guide the positioning of "tcas" in the testing sequence, ensuring that testing efforts are tailored to efficiently address high-impact areas. Last, the "totinfo" program, with 1052 test cases, undergoes similar scrutiny. Risk assessment and prioritization of requirements contribute to the systematic testing plan, ensuring that the testing sequence is aligned with the critical aspects of the software. This comprehensive approach guarantees that the testing efforts are not only thorough but also strategic, addressing potential risks and prioritizing requirements based on their significance to project objectives. Each program is scrutinized for potential risks, and prioritization of requirements guides the testing sequence. This strategic alignment ensures that testing efforts are directed towards critical functionalities, maximizing the likelihood of project success. The nuanced interplay between risk scores and priority levels enhances the efficiency and effectiveness of the testing process, contributing to the overall quality and reliability of the software.

3. Results

Table 2 presents specifications for a set of seven programs along with the number of test cases needed to test each SUT. The table also shows the number of final test cases that were selected for each SUT from three different rounds of TCS, represented by R_1, R_2, and R_3. Each row of the table represents a different SUT, identified by its name. The first column lists the name of each SUT. The second column gives the total number of test cases required to test each SUT, regardless of which TCP round was used. The remaining columns (R_1, R_2, and R_3) show the number of final test cases that were selected from each round of TCP. For example, the first row of the table shows that the SUT named "printtokens" requires a total of 4113 test cases to validate all its functionality. From round 1 of TCP, 4054 test cases were selected as final test cases for this SUT. From round 2, the number of final test cases decreased slightly to 4040. Finally, from round 3, the number of final test cases decreased further to 4024. This table can be useful for assessing the effectiveness of different rounds of TCP and identifying the minimum number of test cases required to adequately test each SUT. It can also help to track the progress of testing over time and identify potential areas of improvement for the testing process [26]. The formula for percent size reduction is as follows:

$$\%SR = 100\% \times \left(\frac{T' - T_s'}{T'}\right) \quad (2)$$

where the initial size of the test suite (T') is the total number of test cases in the original test suite before TCP. The final size of the test suite (T_s') refers to the total number of test cases in the reduced test suite after TCP. The resulting value represents the percentage reduction in the size of T' achieved through the TCP process. Additionally, it presents information about the degree of risk found in each program when applying R_1. These test case metrics are crucial in software testing and quality assurance. A high number of initial test cases compared to the final ones may indicate that some cases were refined, eliminated, or modified during the testing process. Reduction in test cases could be due to identifying and addressing issues, ensuring that the final version is more robust and reliable. The process begins by calling the identify and evaluate risk function, which analyzes each feature in the software system. Risks are identified for features deemed critical or complex, providing a detailed assessment. The prioritize testing efforts function then categorizes these risks based on severity levels, creating a prioritized list to guide testing. Subsequently, the algorithm generates test cases through the generate test cases function, encompassing various scenarios for each feature. The prioritize test cases function evaluates the coverage of identified risks for each test case, creating a sorted list based on prioritization. The focus shifts to high-priority test cases using the focus testing efforts function, ensuring that testing resources are concentrated on critical areas. During the ongoing testing process, the algorithm executes high-priority test cases through the execute test cases function. Simultaneously, the update risk assessment and testing priorities function is employed to adapt to evolving conditions, revising risk assessments and testing priorities. This holistic approach facilitates comprehensive risk identification, prioritization, and ongoing management, contributing to the overall quality and reliability of the software system. The risk found in each program is calculated by subtracting the number of final test cases from the total number of test cases. This difference represents the potential vulnerabilities, bugs, or problems discovered and addressed during testing. Lower values indicate more successful testing processes with fewer identified risks. Effective software testing is essential for ensuring the reliability and functionality of programs in real-world scenarios. Analyzing the difference in the number of test cases provides insights into the testing efficiency and the level of risk mitigation achieved during the software development life cycle. Table 3 shows three different reduction percentages for each SUT, representing the reduction achieved by the first, second, and third studies, respectively. The columns show the percentage reduction in size achieved by each study. For example, the first row of the table shows that the SUT named "printtokens" achieved a 1.43% reduction in size through R_1, 1.77% reduction through R_2, and a 2.16% reduction through R_3. Similarly, for the SUT named "Schedule2", R_3 achieved the largest reduction in size, 4.02%, while R_1 and R_2 achieved reductions of 2.10% and 3.28%, respectively. A larger %SR implies that prioritization process was more effective in reducing the number of test cases required to achieve adequate testing coverage. This information can be used to evaluate the effectiveness of different TCP strategies and to optimize the testing process to reduce the overall size and complexity of the SUTs. The formula for calculating the percent faultless for TCP can be found in [27, 28]:

$$\%\mathrm{F} = 100\% \times \left(\frac{\mathrm{F} - \mathrm{T}'}{\mathrm{T}'}\right) \quad (3)$$

where "number of faultless test cases (F)" refers to the number of test cases that have no faults or defects in the SUT. The percent faultless rate is a measure of the effectiveness of the TCP process in detecting faults and defects. A higher percent faultless rate indicates a more effective process. In Table 4, the program listed are “printokens”, “printokens2”, “replace”, “schedule”, “schedule2”, “tcas” and “totinfo”. This table shows that the number of faults reported varies with the different prioritization techniques. The programs with the highest number of faults reported are replace and tcas, with both components having more than 28 faults reported by each approach. The programs with the lowest number of faults reported are printokens and printokens2, with both components having less than 10 faults. The number of faults reported decreases as we move from R_1 to R_3. Table 5 provides information on the %F for several programs, as reported by R_1, R_2, and R_3. The table shows that the %F varies with the different programs and methods. The program with the highest %F are “printokens” and “printokens2”, with all approaches reporting rates above 99.7%. The lowest %FR are "tcas" and “totinfo", with "tcas" having the lowest rate of all program at 97.51% reported by R_1, and “totinfo” having %F values ranging from 97.91% to 98.57% across all approaches. In general, the %F values reported by all comparative studies are high for all programs. However, there are some variations across the approaches, with R_3 consistently reporting the highest percent faultless rates for the seven programs.

Table 2 Number of final test cases by three studies

Name	No. of test cases	No. of final test cases (R_1)	No. of final test cases (R_2)	No. of final test cases (R_3)
printtokens	4113	4054	4040	4024
printtokens2	4115	4051	4039	4020
replace	5542	5484	5475	5450
schedule	2650	2600	2589	2575
schedule2	2710	2653	2621	2601
tcas	1608	1564	1550	1533
totinfo	1052	1040	1029	1019

Table 3 Percent size reduction by three studies

Name	%SR by R_1	%SR by R_2	%SR by R_3
printtokens	1.43	1.77	2.16
printtokens2	1.56	1.85	2.31
replace	1.05	1.21	1.66
schedule	1.89	2.30	2.83
schedule2	2.10	3.28	4.02
tcas	2.74	3.61	4.66
totinfo	1.14	2.19	3.14

Table 4 Number of faults

Name	No. of faults by R_1	No. of faults by R_1	No. of faults by R_1
printtokens	6	5	4
printtokens2	9	8	6
replace	31	28	24
schedule	8	7	5
schedule2	9	7	5
tcas	40	35	28
totinfo	22	18	15

Table 5 Percent of faultless cases

Name	%F by R_1	%F by R_2	%F by R_3
printtokens	99.85	99.88	99.90
printtokens2	99.78	99.81	99.85
replace	99.44	99.49	99.57
schedule	99.70	99.74	99.81
schedule2	99.67	99.74	99.82
tcas	97.51	97.82	98.26
totinfo	97.91	98.29	98.57

4. Discussion

Based on the above tables, we can compare the capabilities of R_1, R_2, and R_3 in terms of final test cases , size reduction (%SR), faults, and faultless rate (%F) for the different programs. Final test cases, R_1 has the fewest final test cases among the three studies. For most of the programs (except for "printtokens2"), R_1 has the lowest number of final test cases compared to R_2 and R_3. Accordingly, R_2 generally has slightly fewer final test cases than R_3 for most programs. R_3 has the highest %SR for all programs compared to R_1 and R_2. However, R_2 has a higher %SR than R_1 for all programs, except for “tcas” where R_1 has a slightly higher percentage. R_1 has the highest number of faults for all programs compared to R_2 and R_3. However, R_2 has fewer faults than R_3 for all programs, except for “tcas” where R_3 has fewer faults than R_2. Observing %F values, R_3 has the highest value for all programs compared to R_1 and R_2. Also, R_2 generally has a higher percent faultless rate than R_1, except for “totinfo” where R_1 has a slightly higher %F. Overall, based on the provided tables, R_3 has the best performance among the three studies in terms of %SR, faults, and %F. However, R_2 has slightly better performance than R_3 in terms of the number of final test cases for most programs. R_1 generally has the poorest performance among the three studies in all aspects.

Software testing is an essential part of the software development process as it helps to identify and eliminate errors and defects in software. However, testing can be time-consuming and expensive, and developers are constantly seeking ways to optimize their testing processes. One approach to reducing the cost and time required for testing is to use automated techniques such as TCP. It is a technique that aims to reduce the number of test cases required to adequately test a piece of software while maintaining the same level of test coverage. This technique can be applied to both manual and automated testing and it has been shown to be effective in reducing the cost and time required for testing. In this context, the provided tables compare the capability of three approaches, R_1, R_2, and R_3, in terms of final test cases, size reduction, faults, and faultless rate for the different programs. The tables provide insights into how effective each study is in reducing the number of test cases required, detecting faults, and improving faultless rates.

Regarding the number of final test cases, R_1 has the fewest final test cases among the three approaches. This suggests that R_1 may be less effective in reducing the number of test cases required compared to R_2 and R_3. However, it is noteworthy that for most of the programs (except for "printtokens2"), R_1 has the lowest number of final test cases compared to R_2 and R_3. This could mean that R_1 is more effective for certain types of programs or that R_2 and R_3 have more stringent TCP criteria. R_3 has the highest %SR for all programs compared to R_1 and R_2. However, R_2 has a higher %SR than R_1 for all programs, except for “tcas” where R_1 has a slightly higher percentage. R_1 has the highest number of faults for all programs compared to R_2 and R_3. This suggests that R_1 may be less effective in detecting faults compared to the other approaches. However, R_2 has fewer faults than R_3 for all programs, except for “tcas” where R_3 has fewer faults than R_2. R_3 has the highest %F value for all programs compared to R_1 and R_2. Also, R_2 generally has a higher %F value than R_1, except for “totinfo” where R_1 has a slightly higher %F.

Based on the provided tables, R_3 seems to have the best performance among the three approaches in terms of %SR, faults, and %F. However, R_2 has slightly better performance than R_3 in terms of the number of final test cases for most programs. R_1 generally has the poorest performance among the three approaches in all aspects. It is notable that the effectiveness of TCP techniques may vary depending on the type of software being tested and the specific testing requirements. Therefore, it is essential to evaluate the effectiveness of different TCP techniques on a case-by-case basis. In Table 6, we provide a comparison and contrast the three prioritization models with regards to their %SR. Risk-based prioritization (R_1) is a model where risks associated with a project or system are evaluated and prioritized based on the likelihood and potential impact of the risk. This model may result in a size reduction of 1.05-2.74%, as a result of prioritizing and addressing high-risk areas. Requirement-based prioritization (R_2), however, is a model where requirements or features of a system are prioritized based on their importance or criticality to the overall function or purpose of the system. This model may result in a size reduction of 1.21-3.61%, indicating that the final product or system may be reduced in size by a slightly higher percentage than risk-based prioritization. The proposed model (R_3), which is not further specified, is said to result in a size reduction of 1.66-4.66%. However, based on the information provided, R_3 may result in a greater size reduction than either the R_1 or R_2 models. The three models prioritize different factors and may result in different levels of size reduction in the final product or system. It is important to carefully consider the specific goals and requirements of a project when choosing a prioritization model. Furthermore, R_1, R_2, and R_3 are three approaches used to select tasks or activities. One factor that can be compared is the percentage of faultless outcomes achieved through each approach. R_2 involves identifying and prioritizing tasks or activities based on their level

of risk. The percentage of faultless outcomes achieved through this approach typically ranges from 97.9 99.85%, indicating that while this approach can be effective, there is still some error. R_2 involves prioritizing tasks or activities based on their importance in meeting specific requirements or goals. The percentage of faultless outcomes achieved through this approach typically ranges from 97.8 99.88%, indicating that it is effective in achieving high-quality outcomes, with a relatively low risk of error. R_3 may incorporate aspects of both R_1 and R_2, as well as other approaches, to achieve even higher quality outcomes. The percentage of faultless outcomes achieved through the proposed model typically ranges from 98.26 99.90%, indicating that it can be a highly effective approach for achieving faultless outcomes. All three approaches have the potential to achieve high quality outcomes, but the percentage of faultless outcomes achieved varies. R_2 tends to have a higher percentage of faultless outcomes than R_1, and the R_3 model may achieve even higher quality outcomes by incorporating multiple approaches.

Table 6 Comparison of the current study and previous studies using prioritization techniques.

Evaluation	R_1	R_2	R_3
% Size Reduction	1.05-2.74	1.21-3.61	1.66-4.66
% Faultless	97.91-99.85	97.82-99.88	98.26-99.90

5. Conclusions

The growth of test cases during software maintenance can lead to problems such as longer execution times, increased costs, difficulty in maintenance and updates, and increased complexity. Larger test cases may also include redundant tests and miss defects during testing, so it is important to reduce their size while ensuring adequate testing coverage. Risk-based prioritization was developed and focused on critical and high-risk areas of the software to reduce test case size. However, this approach may not cover all functional requirements and necessitates a comprehensive understanding of the software system's architecture and vulnerabilities. Requirements-based prioritization, however, ensures that all functional requirements are thoroughly tested and helps identify areas that require further testing based on clear criteria. However, this approach may not address all potential risks and vulnerabilities or prioritize critical areas of the software. Then, combining both risk-based and requirements-based prioritization can be an alternative for managing risks and requirements, while optimizing testing efforts by identifying critical and high-risk areas using risk-based prioritization. Then, test cases can be emphasized test cases within those areas based on functional and non-functional requirements using requirements-based prioritization. Accordingly, the proposed model shows a higher percentage of reduce test sizes and identified faults, which is an improvement from the test case prioritization technique.

6. Acknowledgements

This research was supported by the CMU Junior Research Fellowship Program Chiang Mai University, Thailand.

7. References

[1] MacIver DR, Donaldson AF. Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper). The 34th European Conference on Object-Oriented Programming (ECOOP 2020); 2020 Nov 15-17; Berlin, Germany. Germany: Schloss Dagstuhl; 2020. p. 1-27.

[2] Lawanna A. An effective test case selection for software testing improvement. 2015 International Computer Science and Engineering Conference (ICSEC); 2015 Nov 23; Chiang Mai, Thailand. USA: IEEE; 2015. p. 1-6.

[3] Srisura B, Lawanna A. False test case selection: improvement of regression testing approach. The 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON); 2016 Jun 28 – Jul 1; Chiang Mai, Thailand. USA: IEEE; 2016. p. 1-6.

[4] Prado Lima JA, Vergilio SR. Test case prioritization in continuous integration environments: a systematic mapping study. Inf Softw Technol. 2020;121:106268.

[5] Bajaj A, Sangwan OP, Abraham A. Improved novel bat algorithm for test case prioritization and minimization. Soft Comput. 2022;26(22):12393-419.

[6] Bajaj A, Abraham A, Ratnoo S, Gabralla LA. Test case prioritization, selection, and reduction using improved quantum-behaved particle swarm optimization. Sensors. 2022;22(12):4374.

[7] Pan R, Bagherzadeh M, Ghaleb TA, Briand L. Test case selection and prioritization using machine learning: a systematic literature review. Empir Software Eng. 2022;27(2):29.

[8] Demir ZC, Emrah Amrahov Ş. Dominating set-based test prioritization algorithms for regression testing. Soft Comput. 2022;26(17):8203-20.

[9] Ahmed FS, Majeed A, Khan TA, Bhatti SN. Value-based cost-cognizant test case prioritization for regression testing. PLoS One. 2022;17(5):e0264972.

[10] Dahiya O, Solanki K, Rishi R, Dalal S, Dhankhar A, Singh J. Comparative performance evaluation of TFC-SVM approach for regression test case prioritization. In: Goyal V, Gupta M, Mirjalili S, Trivedi A, editors. Proceedings of International Conference on Communication and Artificial Intelligence. Lecture Notes in Networks and Systems, vol 435. Singapore: Springer; 2022. p. 229-38.

[11] Chen Z, Chen J, Wang W, Zhou J, Wang M, Chen X, et al. Exploring better black-box test case prioritization via log analysis. ACM Trans Softw Eng Methodol. 2023;32(3):1-32.

[12] Min JL, Rajabi N, Rahmani A. Comprehensive study of SIR: leading SUT repository for software testing. J Phys: Conf Ser. 2021;1869(1):012072.

[13] do Prado Lima JA, Vergilio SR. An evaluation of ranking-to-learn approaches for test case prioritization in continuous integration. J Softw Eng Res Dev. 2023;11(1):1-20.

[14] Thomas SW, Hemmati H, Hassan AE, Blostein D. Static test case prioritization using topic models. Empir Software Eng. 2014;19:182-212.

[15] Fang C, Chen Z, Wu K, Zhao Z. Similarity-based test case prioritization using ordered sequences of program entities. Software Qual J. 2014;22:335-61.
[16] Yadav DK, Dutta S. Test case prioritization based on early fault detection technique. Recent Adv Comput Sci Commun. 2021;14(1):302-16.
[17] Saraswat P, Singhal A, Bansal A. A review of test case prioritization and optimization techniques. In: Hoda M, Chauhan N, Quadri S, Srivastava P, editors. Software Engineering. Advances in Intelligent Systems and Computing, vol. 731. Singapore: Springer; 2019. p. 507-16.
[18] Xing Y, Wang X, Shen Q. Test case prioritization based on artificial fish school algorithm. Comput Commun. 2021;180:295-302.
[19] Paul Friedman K, Gagne M, Loo LH, Karamertzanis P, Netzeva T, Sobanski T, et al. Utility of in vitro bioactivity as a lower bound estimate of in vivo adverse effect levels and in risk-based prioritization. Toxicol Sci. 2020;173(1):202-25.
[20] Been F, Kruve A, Vughs D, Meekel N, Reus A, Zwartsen A, et al. Risk-based prioritization of suspects detected in riverine water using complementary chromatographic techniques. Water Res. 2021;204:117612.
[21] Santhiapillai FP, Chandima Ratnayake RM. Risk-based prioritization method for planning and allocation of resources in public sector. TQM J. 2022;34(4):829-44.
[22] Wan Q, Miao X, Wang C, Dinçer H, Yüksel S. A hybrid decision support system with golden cut and bipolar q-ROFSs for evaluating the risk-based strategic priorities of fintech lending for clean energy projects. Financ Innov. 2023;9(1):1-25.
[23] Singh A, Singhrova A, Bhatia R, Rattan D. A systematic literature review on test case prioritization techniques. In: Hooda S, Sood VM, Singh Y, Dalal S, Sood M, editors. Agile Software Development: Trends, Challenges and Applications. Beverly: Scrivener Publishing; 2023. p. 101-59.
[24] Freeda RA, Rajendran PS. An overview of efficient regression testing prioritization techniques based on genetic algorithm. In: Dutta P, Chakrabarti S, Bhattacharya A, Dutta S, Shahnaz C, editors. Emerging Technologies in Data Mining and Information Security. Lecture Notes in Networks and Systems, vol 490. Singapore: Springer; 2023. p. 383-90.
[25] Guerra-García C, Nikiforova A, Jiménez S, Perez-Gonzalez HG, Ramírez-Torres M, Ontañon-García L. ISO/IEC 25012-based methodology for managing data quality requirements in the development of information systems: Towards Data Quality by Design. Data Knowl Eng. 2023;145:102152.
[26] Khaleel SI, Anan R. A review paper: optimal test cases for regression testing using artificial intelligent techniques. Int J Electr Comput Eng. 2023;13(2):1803-16.
[27] Aung NL, Lawanna A. A competence-based deletion model for the improvement of case-based maintenance in case-based reasoning. Appl Sci Eng Prog. 2021;14(1):3-12.
[28] Lakshminarayana P, Suresh Kumar TV. Automatic generation and optimization of test case using hybrid cuckoo search and bee colony algorithm. J Intell Syst. 2021;30(1):59-72.