Test Case Prioritization For Embedded Software

Elif Güşta ÖZER
Istanbul Technical University
Aselsan Inc.
Istanbul, Turkey
ozerel20@itu.edu.tr
egozer@aselsan.com.tr

Feza BUZLUCA
Computer Engineering Department
Istanbul Technical University
Istanbul, Turkey
buzluca@itu.edu.tr

ABSTRACT

Electronic devices used daily contain software, which may have errors due to human factors during coding. Testing is essential before release, especially as software complexity increases with diverse user needs. Testing new features separately and then in combination multiplies test cases. Rerunning all tests after each change is costly. The aim of this study is to develop a test case prioritization method to decrease the time to find software errors in embedded software systems. For this purpose, we extracted the basic features that characterize embedded software systems and tests that run on them. The proposed method calculates prioritization scores for test cases utilizing these characteristics. The test cases will then be arranged in a systematic manner according to their respective scores. This prioritization strategy is designed to minimize error detection time by promptly finding and resolving errors throughout the initial stages of the testing process. The proposed prioritization strategy was tested on an embedded software system, and it was evaluated using the metrics APFD (average percentage of faults detected) and APFDc (APFD with cost). The results indicate that the proposed method based on the attributes of software systems and related tests reduces the time required to find the majority of the errors.

CCS CONCEPTS

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

KEYWORDS

Test Case Prioritization, APFD, APFDc, Test Features, Embedded Software

ACM Reference Format:
Elif Güşta ÖZER and Feza BUZLUCA. 2024. Test Case Prioritization For Embedded Software. In *2024 13th International Conference on Software and Computer Applications (ICSCA 2024), February 01–03, 2024, Bali Island, Indonesia.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3651781.3651794

ICSCA 2024, February 01–03, 2024, Bali Island, Indonesia

ACM ISBN 979-8-4007-0832-9/24/02
https://doi.org/10.1145/3651781.3651794

1 INTRODUCTION

Embedded control systems are electronic systems designed for autonomous control, management, and monitoring of specific operations. They are usually employed in systems like cruise control, backup sensors, automotive navigation, aerospace, and defense systems. Embedded systems comprise hardware and software components, including microprocessors, sensors, input/output interfaces, and actuators. Embedded software runs on the microprocessors within these control systems, governing all connected systems. It receives sensor inputs to respond to changing environmental conditions, interpreting the data and executing desired operations, such as sending outputs to other devices or users.

Software systems typically consist of multiple functions that are not necessarily coded at the same time. While the core features are initially defined at the project's outset, there may be subsequent requests for new functions or changes to existing ones. These functions can be added incrementally, either at different times or by different software developers. Even when it is confirmed that the newly added or modified functions work correctly within the software, it is imperative to ensure that these changes do not introduce errors in other aspects of the software. This kind of testing is known as regression testing, which involves four primary methods: retesting all, test case minimization, test case selection, and test case prioritization[2].

This article presents a novel approach for prioritizing test cases for embedded software systems to detect numerous bugs in a short period of time. First process of prioritizing the test cases is extracting the general characteristics of embedded software and their associated test cases. Test cases for the given embedded software system are collected and used to create datasets containing extracted features. All analyses and assessments were performed based on these datasets. The features that can be utilized to score each test case are identified after the dataset development process is finished. In the selection of scores from extracted features for the dataset, features that are only used for description and are independent of the prioritization process, such as test name, etc. and features used for evaluation (e.g. test status, identified errors) are excluded. For the remaining features, the minimum and maximum value ranges that each feature could take are determined. Throughout the remainder of the article, these features will be called score features (SFs). Since the range of values for each SF differs, in calculating the overall test case score, each SF is individually normalized to produce scores between 0 and 1. While some of the SFs received a fixed score based on experimental findings, others were assigned by software or test engineers to assign values compatible with normalization. For those assigned features, flexibility is provided because of user knowledge about the parameters, messages, and scenarios

associated with the changing software in future versions, which can directly affect changes in the software. The main idea here is that before the test cases are run, test case prioritization is done with the help of SF values through the test case scoring process. In this experimental process, the most influential score features (SFs) are identified through the conducted experiments. Then, the values that give the best results for all score features at the same time are examined to see how much they contribute to the prioritization process in all versions. The evaluation of the results employed the use of APFD (average percentage of faults detected) and APFDc (average percentage of faults detected per cost) metrics. The execution time of test cases for each test is used to calculate the test cost value required for the APFDc metric. Additionally, the severity values for all identified errors are established by test engineers.

2 RELATED WORKS

Askarunisa et al. [2] propose two test case prioritization methods: coverage and cost-oriented. Coverage methods include loop, statement, branch, and condition-based coverage, while cost-oriented methods rely on test case cost, test case coverage, and fault severity values. The experiment involves creating test cases for basic software, introducing faults using mutation testing, and assigning weights based on real-time test case cost and fault severity. **Younghwan et al.** [5] employ historical data for a two-stage test case prioritization approach. The first stage involves statistical analysis of test history to identify failure patterns for each test case. The second stage, executed during regression testing, interrupts the process upon an error, initiating the reordering of test cases based on the analyzed data. **Ali et al.** [14] leverage swarm optimization for test case prioritization, aiming to minimize execution time and maximize fault detection. Three datasets are utilized, and results are compared among unordered, randomly ordered, and the proposed algorithm. **Zhang et al.** [7] present a prioritization algorithm aiming to detect the maximum number of faults early in the testing process. The algorithm initially orders all test cases by the number of faults, then selects the test case with the highest fault count as the first in the prioritized list. Subsequent test cases are reordered based on faults found in the selected test case, eliminating those covering already identified faults. This process continues until all test case faults are covered, ensuring efficient fault detection in the early stages of testing. **Thillaikarasi et al.** [15] introduce an algorithm for test case prioritization based on weight factors, including time, defect, requirement, and complexity factors. The weighted prioritization value (WPV) is calculated using these factors, and a total prioritization weight (WP) is derived. **Natarajan et al.** [12] introduce a weighted prioritization method utilizing coverage-based techniques, including statement, function, path, branch, and fault coverage. The experiment collects information for all test cases, and weights are calculated using a formula based on coverage information. **Kwon et at.** [9] propose a prioritization technique that first uses the regression test selection (RTS) technique to reduce the testing time and then uses the Bloom filter, which is a fast data structure for test case prioritization (TCP). **Bertolino et al.** [3] propose prioritization techniques based on ML and RL algorithms. Ten different ML algorithms are used to prioritize the test cases. The algorithms used are K-NN, Random Forest, LambdaMART, MART, RankBoost, RankNet, Coordinate ASCENT, Shallow Network, Multiplayer Perceptron and Random Forest (RL-RF). **Biswas et al.** [4] propose four fault-based algorithms for Test Case Prioritization (TCP). The optimized version of bug detection-based TCP prioritizes test cases. If some have the same fault, it prioritizes test cases that have another fault that has been detected by prioritized test cases before. The second, HDFDC, categorizes test cases based on fault detection capability. The third, Exp-TCP, prioritizes test cases by their expected bug detection potential. The fourth, Bayesian TCP, calculates and compares the average probabilities of detecting faults for each test case, selecting those with the highest probabilities for prioritization. **Ozawa et al.** [13] proposed an algorithm, which is a subclass of a model-based technique named operational profile-based test (OPBT). In the requirement phase of software, some information about the software is collected for OPBT. This information is used to increase the quality of test cases. The aim of the research is to find code metrics that affect TCP. **Luo et al.** [10] compare state-of-the-art dynamic TCP techniques with static, including call-graph, string-distance, and topic-model-based methods. The call-graph-based technique creates a call graph for each test case, prioritizing based on invoked methods. The string-distance-based technique assesses textual differences in test cases, utilizing test case codes. The topic-based technique prioritizes test cases by different functionalities, incorporating higher orders for various topics. The dynamic techniques involve the total strategy (genetic algorithms) and an additional strategy based on code coverage for test case prioritization. **Khalid et al.** [8]propose a weighted requirements method for test case prioritization, where weights are assigned based on customer preferences. The K-means algorithm is used to cluster test cases, and a priority percentage is calculated. The K-Medoids method then categorizes test cases into high, medium, and low priorities based on cost and priority percentage. Further division into sub-clusters is determined by time and complexity measures within the main clusters. **Ali et al.**[1] proposed an algorithm that uses component criticality (defined with a fuzzy inference system), test case criticality, and ant colony optimization. Using a fuzzy inference system, the criticality of software components is decided. Component code coverage information is used to determine the criticality of test cases. To prioritize the test cases, ant colony optimization is used with the test critically, the fault detection history, and execution time parameters.

There are numerous approaches to test-case prioritization that have been attempted in the literature. The methods mentioned are coverage[2, 12], historical data[5], swarm optimization[14], bloom filter[9], machine or reinforcement learning[3], model-based [13], ant colony optimization[1], and weighted based approaches[8, 12, 15]. The weighted-based approach is applied in this paper to test case prioritization. Features such as time factors, requirement factors (customer-assigned priority, implementation complexity, etc.), defect factors (defect occurrence and defect impact), and complexity factors (the total effort to execute one test case) are selected for weighting in the paper [15]. The study [8] utilizes customer preferences to assess the weight of the test cases and the study [12] uses a coverage-based methodology. When examining the research findings that used the weighting method, it is clear that features based on time limits for test preparation or execution, errors or

how they affect the software, the cost of running a single test, or user-based priorities are all used. Also, the code coverage approach or user requests were employed for certain weightings. As the software test cases are conducted as black box, it is not possible to acquire code coverage information. Therefore, the paper did not utilize the weighing procedure using the code coverage approach. This article identifies qualities that differ from those often used in other weighing articles, based on the operational principles of embedded software. The attributes encompass UsedSubsystemCount, Age, ErrorRate, and other similar characteristics. Section 4 offers a comprehensive description of every utilized function.

3 BACKGROUND

3.1 Testing Process

In this study, we consider automated software tests using a black-box test design. Black-box testing does not involve any knowledge of the code; the scenarios under which the software is expected to operate are known based on the input parameters and the expected output parameters. Automated testing, preferred for continuously evolving software, allows the same test cases to be reused for efficiency. Automated tests ensure consistent verification of parameters, minimize the risk of human error, and save time. When one of the test cases is run, a system is taken through all the necessary processes from scratch until it reaches the desired state.

In the case of constantly changing software, regression tests are performed to check the current state of the software. This makes testing significantly more expensive. In this paper, a test case prioritization technique is proposed to efficiently detect the maximum errors in minimal time. By executing the test cases in the specific sequence established by the prioritization method, it is possible to identify a greater number of problems within a shorter duration. Elbaum et al. [6] have described the test case prioritization problem as follows:

Problem Statement of Test Case Prioritization:Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers. **Problem:** Find T' ϵ PT such that ($\forall$ T") (T" ϵ PT) (T" $\neq$ T') [f (T') $\geq$ f (T")]

Here, PT represents the set of all possible prioritizations (orderings) of T; T" and T' are some of the test cases that exist at PT; and f is a function that, applied to any such ordering, yields an award value for that ordering. The definition assumes that higher award values are preferred over the lower ones.

3.2 Embedded Control Systems

Embedded control systems are defined as systems that enable the autonomous execution of control, management, and monitoring functions in embedded systems. These systems are responsible for communicating with predefined parameters and external factors with which they can interact to fulfill predetermined functions.

Typically, these systems consist of microprocessors (controllers), sensors, actuators, input/output interfaces, and software. The working logic of an embedded control system is illustrated in Figure 1. The controller (microprocessors) receives data from sensors and processes this data using the controller algorithms embedded in its software. Based on the processed data, it makes a decision and

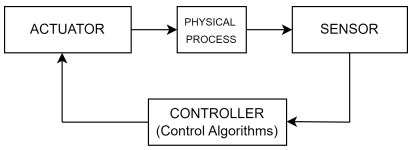


Figure 1: Basic Embedded Control System Diagram

sends the output to actuators. Actuators, in turn, perform predetermined actions based on the received data.

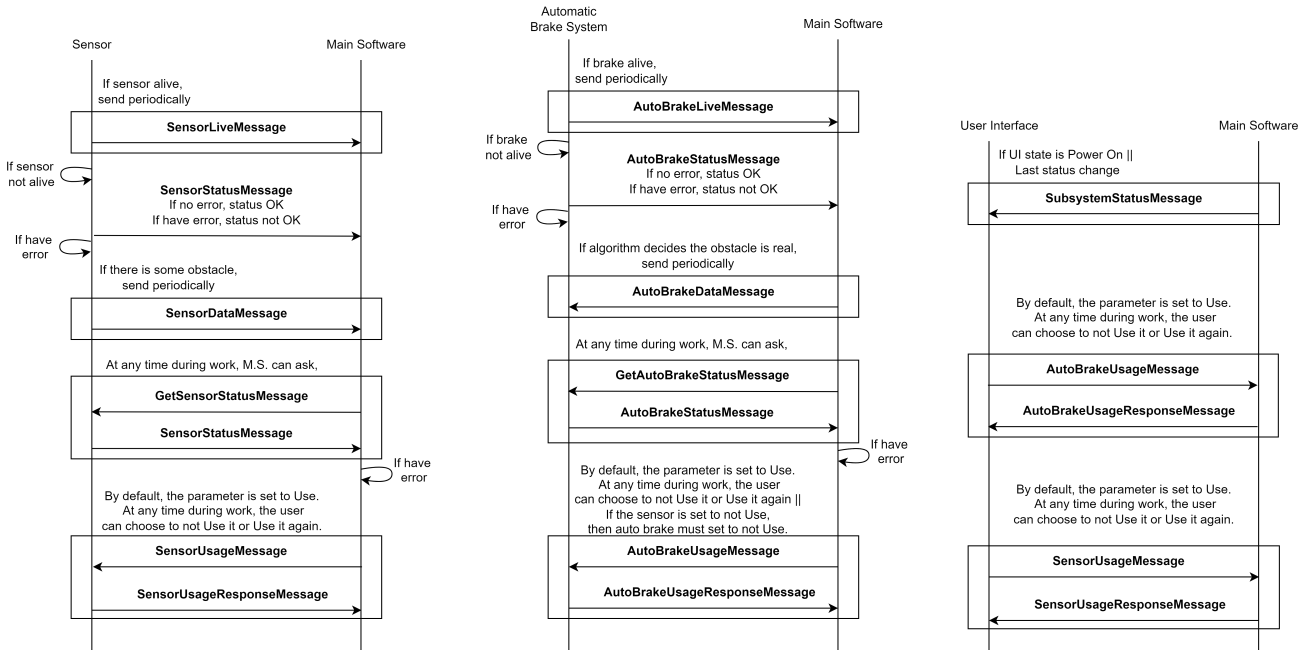
As an example of the mechanism of embedded control systems, Figure 2 provides a representation of basic messages and flows that can be used in an automatic braking system. There are four systems in the example presented in Figure 2. One is the main software that controls the other systems (controller). It decides whether to break or not by looking at sensor data. A sensor system is used to collect data from the outside of the car. An automatic braking system is used to brake the car when some obstacles are seen by sensors. The user interface is used to view the status of the sensor and braking systems and to select the use case of the sensor or auto-brake systems.

Embedded software refers to software that is capable of executing physical actions on hardware, operates in real time, and is subject to certain memory limitations. The embedded software have the ability to interface with several subsystems. It has established precise message structures with distinct boundaries that facilitate communication amongst the subsystems that interact with it. Given the real-time functionality of embedded software, effective handling of the timing of messages in software test cases is crucial. Embedded software mostly operates via a scheduler. An interrupt operation is executed whenever another function has to be performed at any given moment, allowing the execution of the desired function. On the other hand, other software like application software does not possess such prerequisites. Typically, application software engages in action-oriented tasks. For instance, when the user activates a button on an application, the program does the necessary function. The test cases designed for embedded software are tailored to meet the specific functionalities inherent to the program and are commonly employed in the field of embedded software. These features encompass the ability to manage the operational state of the hardware or software of the interconnected subsystems through ongoing message transmission.

For example, the main software is the system that is being tested. The message flow between the sensor and the main software is shown in Figure 2. First, the sensor system periodically sends live messages to the main software. Then the sensor system sends the sensor status data. If there's no error, it sends the parameter as OK, otherwise, it sends it as Not OK. In the presence of an obstacle, the sensor sends a SensorDataMessage to the main software periodically. The main software algorithm processes the sensor data and determines the presence of an obstacle. If an obstacle is detected, it sends an AutoBrakeDataMessage (Figure2b) to the automatic braking system, causing the car to slow down or stop.

3.3 Evaluation Metrics

APFD(Average Percentage Fault Detected). This metric can be used to determine how early the errors were discovered in the



(a) Sensor - Main Software **(b) Brake - Main Software** **(c) UI - Main Software**

Figure 2: Messaging Flow in an Exemplary Embedded Software for an Automatic Brake System

test cases. All test cases and faults are considered to have equal importance.

$$APFD = 1 - \frac{(Tf_1 + Tf_2 + \ldots + Tf_m)}{m \times n} + (\frac{1}{2 \times n}) \quad (1)$$

TFi is the position of the first test case in all test cases T, that exposes fault i, m is the total number of faults exposed under T and n is the total number of test cases in T [6].

APFDc(Average Percentage of Faults Detected per Cost). Different than APFD, this metric also considers test case costs and fault severities. In our study, the test execution time in milliseconds has been determined to be the cost value of test cases. Additionally, the test engineer provides severity values for each fault, and these weights are assigned to each fault based on how much of an impact the fault has on the user.

$$APFDc = \frac{\sum_{i=1}^{m} \left(fi \times \left(\sum_{j=TFi}^{n} t_j - \frac{1}{2} t_{TFi} \right) \right)}{\sum_{j=1}^{n} t_j \times \sum_{i=1}^{m} f_i} \quad (2)$$

Let T be a set of test cases that contains n test cases with cost t1, t2,……., tn. F is a set of m faults inside the T, and f1,f2…..,fm are the severity levels of those faults. Let TFi be the first test case in an ordering T' of T that reveals fault i [11].

Figure 3 illustrates the process of matching fault severity.

To illustrate the use of APFD and APFDc metrics, an example is provided in Table 1. The system in this example consists of 5 test cases and 4 faults. The cost values for each test and severity

```
{
    "Error": "Fault_35",
    "Severity": "1"
}
```

Figure 3: Representation of the fault and its severity for APFDc

Table 1: Example of APFD/APFDc

Faults(F)	Test Cases(T)					Severity
	T1	T2	T3	T4	T5	
F1	✓		✓		✓	1
F2			✓			2
F3		✓		✓		1
F4		✓	✓		✓	3
Cost(c)	2	3	3	4	2	

values for each fault are also displayed. The calculations, based on the formulas provided earlier, are as follows.

If test case running order is **T1 - T2 - T3 - T4 - T5;**

APFD = 1- ((1+3+2+2)/(5 × 4)) + (1/(2 × 5)) = **0.50**
APFDc = ((1 × (2+3+3+4+2 - ((1/2)× 1))) + (2 × (3+4+2 - ((1/2) × 3))) + (1 × (3+3+4+2 - ((1/2) × 2))) + 3 × (3+3+4+2 - ((1/2) × 2))))/((2+3+3+4+2) × (1+2+1+3)) 13.5 + 21 + 11 + 33 / (14 × 7) = **0.80**

If test case running order is **T3 - T2 - T1 - T4 - T5;**

APFD = 1- ((1+1+2+1)/(5×4)) + (1/(2×5)) = **0.65**
APFDc = ((1 × (2+3+3+4+2 - ((1/2)×1))) + (2 × (2+3+3+4+2 - ((1/2)×1))) + (1 × (3+3+4+2 - ((1/2)×2))) + 3 × (2+3+3+4+2 - ((1/2)×1))))
/ ((2+3+3+4+2)×(1+2+1+3)) = 13.5 + 26 + 11 + 40.5 / (14 * 7) = **0.92**

4 PROPOSED TECHNIQUE

When a new version of a software system arrives, new test cases are added, old ones are updated, or some scenarios are removed from the test cases to be run for that version. After these processes are completed, the test cases can be prioritized to find more errors in a short time before the test cases start running. In the method proposed in this article, the test case prioritization process is repeated before each new version. Figure 4 presents the general operation of the proposed prioritization method. The details of the modules are explained in the following sections.

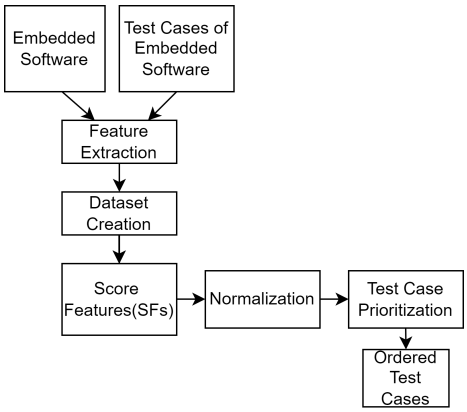


Figure 4: The General Structure of the Proposed Test Case Prioritization Technique

4.1 Collected Features

To calculate the scores of test cases, we use features obtained from the embedded software and associated test cases. Two types of features, i.e., static and historical features are used. Static features of test cases can be obtained without running them. Historical features are based on data obtained from test runs on previous versions of the embedded software test cases. Assuming the current version of the system under test is Vt, and the data collection window is w, we gather data from previous versions between Vt-w and Vt-1 to obtain historical features of the test cases. Static features are obtained from the version Vt. The extracted features are as follows: TestName, VersonOfSoftware, TestTime, ErrorRate, UsedSubsystemCount, SubsystemName, CountOfSubsystemScenario, ErrorCountOfSubsystemScenarios, CountOfScenarioChangeMade, totalSendMessageCount, TotalReceivedMessageCount, TotalParameterCountWhichContainError, MessageName, Reciever, Sender, IsPeriodic, CountOfParameter, CountOfListParameter, UsedModes, CountOfMessageRule, CountOfOtherSubsystemWhichMessageSend, FaultsFoundedInThisTestCaseFromAllFaults, TestResult. Since these characteristics are universal to embedded software systems, the proposed method is applicable to a variety of systems. All extracted features used to create the dataset are explained as follows:

- **TestName**:The name of the test.
- **VersionOfSoftware:** The version number of the software used in the dataset.
- **TestTime:** The execution time of the test case.
- **Age:** The test case age is determined by the version number it corresponds to. For instance, if a test case is used in all four versions (V04 to V07) of a project, the parameter is set to 1 in the dataset for V04, indicating its first use. In the subsequent version V05, this parameter is set to 2 for the same test case, and so on. It takes an incrementable value between 1 and 4 in the dataset, reflecting the number of software versions tested in this article.
- **ErrorRate:** This historical feature indicates the error rate of the test case. It is obtained from the previous versions of the software to be tested. Let the version under test be denoted as V07, and our previous test version is referred to as V06. To assign a parameter to a test in V07, an initial check is made to determine the existence of the test in the V06 version. If the test is not found in V06, its value is set to zero in V07. If the test exists in V06 and is deemed accurate, the value from the V06 dataset is transferred directly to V07. In cases where the test is present in V06 but is found to be faulty, its value in V06 is adjusted according to the scoring criteria, and the updated value is then written to the V07 dataset.
- **UsedSubsystemCount**: This is the subsystem count used in the test case.
- **Subsystems**: This parameter is repeated for each subsystem used in the test cases. It comprises four sub-parameters: SubsystemName, CountOfSubsystemScenario, ErrorCountOfSubsystemScenarios, and CountOfScenarioChangeMade. These parameters are version-dependent and have the same value for all test cases within a version. Each subsystem used in the test case has its set of these parameters, with values varying based on the subsystem name.
  SubsystemName is the name of a subsystem. This parameter can take values Sys1,Sys2,... etc.
  CountOfSubsystemScenario is determined by analyzing the software. Each subsystem has key scenarios, such as opening flows or actions triggered by specific messages. When a test case runs, it may utilize one or more scenarios. This parameter indicates the number of scenarios present in a version of test cases.
  ErrorCountOfSubsystemScenarios parameter shows count of the software-to-be tested scenarios have an error in this version of test cases.
  CountOfScenarioChangeMade parameter shows the number of changes made in this system for this version looking by the previous version of the software to be tested.
- **TotalSendMessageCount**: This parameter shows the number of messages sent in a single test case.
- **TotalReceivedMessageCount**: This parameter shows the number of messages received in a single test case.
  Note: Assume that the software that connects all systems is called the main software. The main software receives all messages from other subsystems. TotalReceivedMessageCount parameter is the count of the total messages sent from subsystems to the main software. The totalSendMessageCount parameter is the count of total messages sent from the main software to all other subsystems. Messages sent periodically are calculated as one since static features are collected from the test.

- **TotalParameterCountWhichContainError**: This parameter indicates the number of incorrect parameters. A value of 0 signifies no error, while -1 denotes no parametric error but errors in sub-scenarios. If the value is greater than zero, it represents the count of parametric errors.
- **FaultsFoundedInThisTestCaseFromAllFaults**: This parameter is employed to identify errors within a test case. It consists of two sub-parameters: ErrorID and ErrorStatus. ErrorID is used to categorize all errors, while ErrorStatus indicates whether an error corresponding to that ErrorID is present in this test case. Each test case includes values for both ErrorID and ErrorStatus for all errors. ErrorStatus can take values of 0 (no error) or 1 (error).
- **TestResult**: This parameter shows that a test case passes or fails. This parameter can take 0: Fail or 1: Pass.
- **Messages**: This section pertains to the messages contained in the test case, detailing which messages it includes and their parameters.
  MessageName:Message names, such as "FMsg46," are defined with an "Msg" tag. The alphabet tags (A, B, C, ... F) at the beginning designate the subsystem, providing a means to categorize messages. The number represents the message number within that subsystem.
  Receiver: The subsystem name of the receiver of this message.
  Sender: The subsystem name of the sender of this message.
  IsPeriodic: This parameter indicates whether the message is sent periodically. It has two options: Periodic (1) or Non-periodic (0).
  CountOfParameter: This parameter displays the total count of parameters in the message, excluding a list of them.
  CountOfListParameter: This parameter shows the total list parameter count of this message.
  UsedModes: This parameter shows the modes in which the message can be used. This mode belongs to the software to which the message is sent. When it comes to embedded software, it's not always desirable for certain functions to happen simultaneously with others. A few modes are defined in the software to make sure that these operations don't happen simultaneously. Although some messages are compatible with all modes, others are limited to specific modes of operation. This parameter is repeated as many times as messages can be sent in some mode.
  CountOfMessageRule: This parameter shows if the message has a sending rule. For example, some messages can send only one specific message comes. This parameter can take a value of 0, indicating no rule, or a value greater than 0 to specify the number of rules.
  CountOfOtherSubsystemWhichMessageSend: This parameter indicates the count of how many other subsystems receive information about a message sent from any subsystem to the main software. For instance, if Sys1 sends a message to the main software, and the main software forwards it to one or more of the other five systems, this parameter shows the count of those receiving the information.
- **Important Notes:** There are some rules about the dataset.
  1) "ErrorRate" parameters used in the first version Vt-w of the dataset, e.g., V04 are all set to 0.
  2) "Subsystems" subsections parameters except for parameter "SubsystemName" are about versions of the test cases, not about the test case.

The selection of some of the extracted features is driven by the messaging flow of embedded software, as depicted in Figure 2. For instance, the decision on obstacle detection relies on messages from sensors. A rule dictates that for the brakes to engage (AutoBrakeDataMessage), the sensor must send a SensorDataMessage. In case of a sensor error, a SensorUsageMessage is sent to the main software, triggering a SubsystemStatusMessage to the UI system, defining the CountOfMessageRule feature. Periodically sent messages are identified by the isPeriodic feature. Messages like SensorUsageMessage sent to other systems lead to the selection of CountOfOtherSubsystemWhichMessageSend. These features are essential for understanding the software's behavior and testing scenarios.

4.2 The Dataset

Figure 5 shows an example of a single test case of the dataset obtained using collected features. To provide a basic example of the dataset in Figure 5 the Subsystems, FaultsFoundedInThisTestCaseFromAllFaults, and UsedModes sections have been shortened. These parts normally repeat as many times as the number of subsystems, errors, and modes in the test cases. The dataset can be accessed at the following link: https://github.com/elifgusta/TestCasePrioritizationForEmbeddedSoftware

4.3 Score Features and Calculation of Total Test Score

In the prioritization process, we assign a score to each test case using the extracted score features (SFs). The SFs were selected by considering the following properties of the collected dataset. First, the features that are not relevant to prioritization, such as Test Name, Test Version, MessageName, TotalParameterCountWhichContainError and Test Result are excluded. Secondly, the features that are used only in evaluation metrics, such as TestTime, and FaultsFoundedInThisTestCaseFromAllFaults were eliminated. Finally, the remaining features that would be suitable for determining the effectiveness of the test cases were selected. These SFs are Age, UsedSubsystemCount, SubsystemName, CountOfSubsystemScenario, ErrorCountOfSubsystemScenarios, CountOfScenarioChangeMade, TotalSendMessageCount, TotalReceivedMessageCount, Receiver, Sender, IsPeriodic, CountOfParameter, CountOfListParameter, CountOfMessageRule, CountOfOtherSubsystemWhichMessageSend, UsedModes. In the method to be applied in the article, to perform the prioritization process before running the test cases, the determined features must be selected from the static properties of the test cases or must be determined based on data before the version under test. Since the ErrorRate and Age features do not comply with the static properties of such test cases, the data of these features were used by looking at the historical data.

Since the range of values for each SFs differs, they must be normalized prior to their use in a single equation that computes

```
[
  {
    "Test_1": [
      {
        "TestName": "Test_1",
        "VersionOfSoftware": "V07",
        "TestTime": "24s,35ms",
        "Age": "3",
        "ErrorRate": "0",
        "UsedSubsyatemCount": "1",
        "Subsystems": [
          {
            "SubsystemName": "Sys_2",
            "CountOfSubsystemScenario": "17",
            "ErrorCountOfSubsystemScenarios": "0",
            "CountOfScenarioChangeMade": "0"
          }
        ],
        "TotalSendMessageCount": "12",
        "TotalReceivedMessageCount": "1",
        "TotalParameterCountWhichContainError": "0",
        "FaultsFoundedInThisTestCaseFromAllFaults": [
          {
            "ErrorID": "Fault_50",
            "ErrorStatus": "0"
          }
        ],
        "TestResult": "0",
        "Messages": [
          {
            "MessageName": "F_Msg_46",
            "Receiver": "Sys_2",
            "Sender": "main",
            "IsPeriodic": "0",
            "CountOfParameter": "1",
            "CountOfListParameter": "0",
            "UsedModes": [
              {
                "Mode": "Mode_2"
              }
            ],
            "CountOfMessageRule": "0",
            "CountOfOtherSubsystemWhichMessageSend": "0"
          }
        ]
      }
    ]
  }
]
```


Figure 5: Simple Representation of Dataset for a Single Test Case

the test case's score. We normalized the values of each SF between 0 and 1 in two different ways, depending on their effects on the effectiveness of the test case. If a characteristic with a high value implies that the relevant test case will be more successful, the maximum value of the feature is converted to 1, while the minimum value is represented as 0. In the opposite case, where low values affect the test case positively, the minimum value of the feature is converted to 1, while the maximum value is represented as 0. Figure 6 presents two exemplary normalizations of the features Age and UsedModes. The columns on the left side under each feature represent the real values obtained from the dataset. The columns on the right side contain the corresponding normalized values. The main purpose here is to observe whether prioritizing the test cases by features selected from the dataset by giving them different scores can help shorten the time to find errors. Points are given by increasing or decreasing them in equal proportions at certain intervals only to create the difference between test cases, and the effects of the values determined here on other versions are observed. In these value assignments, attention is paid to whether the left real values side taken from the dataset of SFs were numerical or categorical data.

In the given example, Age has a numerical value, UsedModes has a categorical value. For each range of real values, particular normalized values are assigned in the context of numerical parameters. When it comes to numerical data, real data is always sorted from the smallest to the biggest value. The assignments between 0 and 1 on the right normalized data side of the SFs are set to increase in value at equal intervals according to the number of left real value sides of the SFs.


```
"Age": {
    "1": "1",
    "2": "0.66",
    "3": "0.33",
    "4": "0"
},

"UsedModes": {
    "Mode_1": "1",
    "Mode_2": "0.8324",
    "Mode_3": "0.6658",
    "Mode_4": "0.4992",
    "Mode_5": "0.3332",
    "Mode_6": "0.1666",
    "Mode_7": "0"
},
```


Figure 6: Exemplary Normalizations of The Features

Due to the inability to sort categorical data numerically, a single value is assigned to each category instead of allocating unique normalized values to distinct ranges. Hence, the MaxToOne or MaxToZero approaches are inapplicable in this context. The test or software engineer is responsible for inputting the normalized values of categorical features such as SubsystemName, Receiver, Sender, and UsedModes. Figure 7 shows how the user-defined normalized values can be done.


```
"SubsystemName": {
    "Sys_1": "0",
    "Sys_2": "0.20",
    "Sys_3": "0.40",
    "Sys_4": "1",
    "Sys_5": "1",
    "Sys_6": "1"
},
```

```
"SubsystemName": {
    "Sys_1": "0",
    "Sys_2": "0.20",
    "Sys_3": "1",
    "Sys_4": "1",
    "Sys_5": "1",
    "Sys_6": "1"
},
```


(a) Sys4 and Sys5 **(b) Sys3, Sys4 and Sys5**

Figure 7: Example of User Defined Score Values For SubsystemName

In Equation 3, n is a count of score features to be used in the prioritization process, and c_i denotes the weighted value assigned to each feature. and "pscore" is a score that is given for each SF.

$$TestCaseScore = \sum_{i=1}^{n} (c_i \times pscore_i) \tag{3}$$

Some SFs are repeated in the form of a list of as many values as are in the test case. These features are located in Subsystems, Messages, and UsedModes under Messages. Since the number of these varies for each test case, the value of test cases with a larger number of list elements was getting higher score values in their calculations. To reduce this effect, after adding up the scores of all elements in the list belonging to that one SF, the result is divided

by the number of elements and taken as the result to be added to the overall score for that SF. In this article, the first contribution of SFs to test case prioritization one by one is examined, and then the effects of score features when used all at the same time are examined. One drawback of this approach is that the sorting process cannot begin until all test cases for the version being evaluated are ready. Moreover, several aspects are derived from historical data. In order to leverage the impact of those features on the sorting, it is important to own data from the prior version of the version that is being evaluated.

5 EXPERIMENTS AND RESULTS

In this study, the software that is used is an embedded system designed to operate on the microprocessor featured in embedded control systems, functioning as a controller. Its primary role is to acquire data from connected sensors and relay this information to specific subsystems responsible for executing predefined functions. The software utilized in this study can accommodate up to six subsystems, comprising the user interface, sensors, and actuators. Notably, the simultaneous connection of all six subsystems is not obligatory. For the experiments to be carried out in this article, datasets were prepared for four versions of the used software. The count of test cases and fault numbers in these versions are shown in Table 2. At the beginning of our experiments, we determined

Table 2: Count of Test Case and Total Faults

	Test Case Count	Fault Count
V04	72	15
V05	159	21
V06	157	9
V07	439	30

the effect of each SF on the effectiveness of the test cases. First, we normalized a feature by converting its maximum value to 1, and we calculated test case scores using only that feature. Scores for other features were counted as zero. As a result, test cases were prioritized using a single feature. Secondly, we normalized the same feature, converted its maximum value to 0, and prioritized test cases again. By comparing the results, we determined how an SF affects the success of the prioritization. We repeated this analysis for each SF. Figure 8 shows an example of how to adjust the maximum value to 0 (MaxToZero) and to 1 (MaxToOne) for an SF.

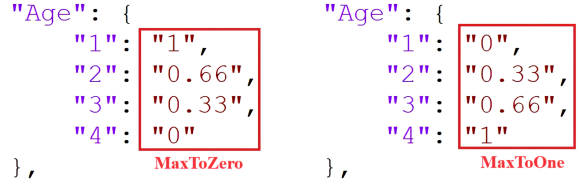


Figure 8: Example of Assigning MaxToZero and MaxToOne Normalized Values

Table 3 shows the best results of APFD and APFDc scores obtained for each SF using the V07 dataset. Thus, using features and scoring to help with prioritization, the impact of the ideal scores established for V07 on other versions was also observed with these outcomes. After this analysis, the following features are normalized using the MaxToOne technique because their larger values affect the priority of the test cases positively: CountOfSubsystemScenario, CountOfScenarioChangeMade, TotalSendMessageCount, TotalReceivedMessageCount, CountOfParameter, CountOfListParameter, and CountOfOtherSubsystemWhichMessageSend. On the other hand, the following features are normalized using the MaxToZero technique because their smaller values affect the priority of the test cases positively: Age, ErrorRate, IsPeriodic, CountOfMessageRule. The second method employs normalized score values, specifically

Table 3: Results of APFD and APFDc of V07 for each SF

		Normalized Value Order	APFD	APFDc
Age	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToZero	**0.51602**	**0.53393**
ErrorRate	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToZero	**0.39840**	**0.41764**
UsedSubsystem Count	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToOne	**0.61966**	**0.60919**
Subsystem Name	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	UserDefined	**0.64555**	**0.58657**
CountOf SubsystemScenario	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToOne	**0.60994**	**0.55383**
CountOfScenario ChangeMade	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToOne	**0.30311**	**0.32295**
TotalSend MessageCount	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToOne	**0.67722**	**0.61683**
TotalReceived MessageCount	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToOne	**0.65641**	**0.58378**
Receiver	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	UserDefined	**0.57858**	**0.62208**
Sender	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	UserDefined	**0.71548**	**0.68024**
IsPeriodic	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToZero	**0.54274**	**0.61547**
CountOf Parameter	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToOne	**0.60387**	**0.60022**
CountOfList Parameter	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToOne	**0.65930**	**0.64871**
UsedModes	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	UserDefined	**0.64244**	**0.60842**
CountOf MessageRule	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToZero	**0.52171**	**0.56975**
CountOfOther Subsystem Which MessageSend	Not Sorted	MaxToZeroAndOne	0.30311	0.32295
	Sorted	MaxToOne	**0.56085**	**0.61412**

from the MaxToZero or MaxToOne options of V07, which yields optimal results. This method aims to assess the impact of simultaneously utilizing all features for test case prioritization in the software under test. Additionally, it aims to evaluate the influence

of applying normalized values from one version to all other versions. Table 4 presents prioritized test case results for all versions using all features to calculate test case scores. The version of the software for which we determined the effect of each feature was software version V07. The APFD and APFDc values of the test cases of V07 are 0.78 and 0.72, respectively, which are calculated after the MaxToOne, MaxToZero, or UserDefined normalized values that give the best results for each feature are determined and test cases sorted with their score calculation.

The optimal MaxToZero or MaxToOne normalizations for each feature are chosen based on the evaluates' results, and these scores are then utilized to prioritize all other software versions. The objective is to assess the effectiveness of applying a fixed normalization value to each feature in the ranking of test cases. The trials demonstrate that the proposed method can obtain success rates ranging from 52 percent to 78 percent across all versions of the score calculation. This result is achieved by utilizing all normalization values that yield favorable sorting results for each feature simultaneously.

Table 4: Results for All Versions Software with Normalized Values Of V07

		APFD	APFDc
V04	Not Sorted	0.37301	0.35088
	Sorted	**0.61111**	**0.56028**
V05	Not Sorted	0.51886	0.467233
	Sorted	**0.77493**	**0.73745**
V06	Not Sorted	0.29263	0.33286
	Sorted	**0.52264**	**0.51910**
V07	Not Sorted	0.30311	0.32295
	Sorted	**0.78010**	**0.72343**

Looking at the results obtained in related articles Natarajan et al. [12] use a weighted TCP method based on code coverage and APFD was used as evaluation metric. The paper uses multiple code coverage approaches. The results of each coverage range from 74% to 81.82%. Additionally, Askarunisa et al. [2] prioritized test cases using different code coverage methods. That paper used APDF and APFDc evaluation metrics and it was found APFD metrics results in the range of 57.14 to 67.7 percent. In the same study, APFDc was in the range of 53.8 to 84.9 percent. Results show that some coverage strategies perform better than the ones used in this paper, while others perform worse. Ritika et al.[7] examined the proposed technique in three ways. Unsorted, randomly sorted, and algorithm-sorted. After comparing the three methods indicated in their study, the proposed method generated better results than the unsorted and randomly sorted methods. The findings show that APFD had a 75% success rate and APFDc 85.64 percent. After reviewing prior research, this paper's APFD results were similar to those of other studies. More improvements are needed to improve APFDc results.

6 CONCLUSION AND FUTURE WORK

This article presents a method to prioritize embedded software test cases for faster error detection before executing each version. General features found in embedded software and their software tests were extracted for the prioritization process. With the acquired features, software tests of embedded software were used to create datasets. Features that could be utilized for scoring and prioritization were identified from the test-derived datasets. First, it was shown how each determined score feature affected the test case prioritization procedure. Subsequently, the contributions of each score feature to the prioritization are displayed once they are processed at the same time. The obtained results show that each of the determined score features helps to find more errors in a shorter amount of time. Simultaneously, it has been noted that combining these features yields better outcomes than utilizing them separately. The features of the software and the tests themselves can be utilized to help prioritize embedded software test cases.

As a future work, more static features from the current version and additional historical data from previous iterations can be extracted to improve the performance of the prioritization model. Every feature in this study is given the same weight. Future research may assign particular weights to different features. Machine learning techniques can be used to prioritize the test cases using extracted features.

REFERENCES

[1] Sadia Ali and Yaser Hafeez. 2019. Enabling Test Case Prioritization For Component Based Software Development. 105–1054. https://doi.org/10.1109/FIT47737.2019.00029

[2] Ms Askarunisa, Ms Shanmugapriya, and N. Ramaraj. 2009. Cost and Coverage Metrics for Measuring the Effectiveness of Test Case Prioritization Techniques. (07 2009).

[3] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1–12.

[4] Sourav Biswas, Aman Bansal, Pabitra Mitra, and Rajib Mall. 2023. Fault-Based Regression Test Case Prioritization. *IEEE Transactions on Reliability* 72, 3 (2023), 1176–1190. https://doi.org/10.1109/TR.2022.3205483

[5] Younghwan Cho, Jeongho Kim, and Eunseok Lee. 2016. History-Based Test Case Prioritization for Failure Information. 385–388. https://doi.org/10.1109/APSEC.2016.066

[6] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2002. Test Case Prioritization: A Family of Empirical Studies. *IEEE Trans. Softw. Eng.* 28, 2 (feb 2002), 159–182. https://doi.org/10.1109/32.988497

[7] Ritika Jain and Neha Agarwal. 2013. Additional Fault Detection Test Case Prioritization. *B V I C A M's International Journal of Information Technology* 5, 2 (Jul 2013), 623–629. https://www.proquest.com/scholarly-journals/additional-fault-detection-test-case/docview/1566944629/se-2

[8] Zumar Khalid and Usman Qamar. 2019. Weight and Cluster Based Test Case Prioritization Technique. https://doi.org/10.1109/IEMCON.2019.8936202

[9] Jung-Hyun Kwon and In-Young Ko. 2017. Cost-Effective Regression Testing Using Bloom Filters in Continuous Integration Development Environments. 160–168. https://doi.org/10.1109/APSEC.2017.22

[10] Qi Luo, Kevin Moran, Lingming Zhang, and Denys Poshyvanyk. 2019. How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects. *IEEE Transactions on Software Engineering* 45, 11 (2019), 1054–1080. https://doi.org/10.1109/TSE.2018.2822270

[11] Alexey G. Malishevsky, Joseph R. Ruthru, Gregg Rothermel, and Sebastian G. Elbaum. 2006. Cost-cognizant Test Case Prioritization. https://api.semanticscholar.org/CorpusID:16237327

[12] Prakash Natarajan and T.R. Rangaswamy. 2013. Weighted method for coverage based test case prioritization. *Journal of Theoretical and Applied Information Technology* 56 (01 2013), 235–243.

[13] Masataka Ozawa, Tadashi Dohi, and Hiroyuki Okamura. 2018. How Do Software Metrics Affect Test Case Prioritization?. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 01. 245–250. https://doi.org/10.1109/COMPSAC.2018.00038

[14] Ali Samad, Hairulnizam Mahdin, Rafaqat Kazmi, Rosziati Ibrahim, and Zirawani Baharum. 2021. Multiobjective Test Case Prioritization Using Test Case Effectiveness: Multicriteria Scoring Method. *Scientific Programming* 2021 (06 2021), 1–13. https://doi.org/10.1155/2021/9988987

[15] Thillaikarasi and Dr. K. Seetharaman. 2013. A Test Case Prioritization Method with Weight Factors in Regression Testing Based on Measurement Metrics. *International Journal of Advanced Research in Computer Science and Software Engineering* 3 (12 2013), 390–396.