Encoding Test Cases using Execution Traces

Ziad A. Al-Sharif
Dept. of Software Engineering
Jordan University of Science and Technology
Irbid, Jordan
zasharif@just.edu.jo

Wafa F. Abdalrahman
Dept. of Computer Science
Yarmouk University
Irbid, Jordan
2016901062@ses.yu.edu.jo

Clinton L. Jeffery
Dept. of Computer Science and Engineering
New Mexico Tech
Socorro, NM 87801
jeffery@cs.nmt.edu

2021 12th International Conference on Information and Communication Systems (ICICS) | 978-1-6654-3351-8/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICICS52457.2021.9464569

***Abstract*—Test case minimization can be critical to meeting the release date of a software product. Identifying redundant test cases can help improve the quality of the test suite and speed up the testing process. Thus, there is a need to uniquely characterize test cases. This identification can support the test engineer to remove redundancy in the test suite and prioritize test cases that are highly affected by the most recent modification in source code. This paper proposes a test case encoding approach that allows engineers to facilitate execution traces to classify and identify their test cases. It will empower test engineers and allow them to minimize the time and cost of testing by reducing the number of test cases, especially in regression testing. Furthermore, it enhances the documentation of the testing process by providing a one-to-one mapping between test cases and their corresponding execution traces, each of which is a sequence of execution events triggered during the execution of the test case. The one-to-one mapping allows the approach to uniquely represent the control-flow and data-flow within the source code. This trace can be used as a signature for the test case. Whenever a modification occurred in the source code, the newly captured signatures are compared against the previous ones; any mismatch indicates that the test case has been affected by the modification. Repeating this process will help classify the test suite into four groups of test cases. This provides the ability to put the testing efforts where it is needed. Additionally, keeping a hashed value for each of the captured sequences simplifies the comparison and unifies the mapping between test cases and captured traces. It also allows detection of minor modifications in the traced events, and reduces the lengthy traces to a set of fixed size hashed values.**

***Index Terms*—Test Cases, Regression Testing, Execution Traces**

I. Introduction

Software bugs are costly [1]. Thus, software testing is significantly important, yet it requires time and resources [2]. Software testing is one of the means to ensure the quality of the software and reduce the risk induced by the potential bugs in software products [3]. The testing process can expose the existence of errors in the software [4]. A design phase is necessary to generate a good test suite, which is a set of test cases executed to verify a specific feature or functionality of the software application. When designed, the increase in the number of test cases will cause an increase in the testing time, cost, and effort [5]. Repeatedly, a successful pass of the test suite is required before every release whether it involves a minor update such as fixing a bug or incorporates a major modification that introduces a new feature.

Regression testing is a type of software testing that is used to ensure the continued functionality of previously-working features in modified software. Usually, it involves a full or partial selection of already executed test cases that are re-executed after the code change to confirm that existing functionalities are still working as intended [6], [7]. Thus, the goal of regression testing is to validate and evaluate whether the newly introduced code changes have side effects on the existing functionalities or not.

This paper introduces the idea of a test case encoding mechanism. The execution of all test cases are traced. A sequence of events called an execution trace is captured and hashed during the testing process, and then the hashed value is recorded along with its test case. An execution trace can be viewed as a sequence of (event code, location) tuples that are generated from the executed statements in the tested program; each for a specific test case. A sequence of these codes are captured during the execution of the test case. Whenever this test case is passed, the corresponding execution trace is hashed and saved in a database along with the test case itself. This infrastructure will provide the test engineer with the mechanism to identify whether any of these test cases is affected by the newly introduced code change. A modification in the source code will cause the sequence of event codes and values to be changed too. Hash functions are famous for their sensitivity to minor changes in the hashed source. Therefore, the captured execution trace is hashed and the resulting hash value is used instead of the lengthy sequence of traced events.

Consequently, hashing the execution trace – the sequence of event codes and values – brings simplification and integrity to the approach. First, it allows us to manage a fixed length of the hashed values regardless of the actual length of the captured execution trace; this is one of the interesting aspects about hash functions. For example, using the $MD5$ hashing algorithm allows us to generate, save, retrieve, and compare only 128 bits of hashed values regardless of the number of traced events, which are usually significantly larger than the length of this hashed value. Moreover, in our approach, selecting a hashing algorithm has nothing to do with its level of security but its performance in generating the hashed value and the size of this value. Second, managing and comparing these hashed values will be easier and often collision free, which means that each execution trace will provide a unique hashed value. Thus, any collision in the hashed values can be considered a redundancy in the test cases and allows us to safely remove one of them, leading to more efficient testing process.

The rest of this paper is organized as follows: Section II gives a background view about the execution trace and the used framework. Section III discusses some of the work related to this research paper. Section IV presents our proposed methodology. Section V provides a validation to our methodology demonstrated using a simple program. Section VI gives a further demonstration for the methodology using a sample set of test cases. Section VII shows how a test engineer can customize the execution trace based on its needs. Section VIII presents the limitations of our approach and explains some of the future works. Finally, Section IX concludes our research.

II. Background

The proposed approach in this paper employs program execution traces to improve the quality of software testing. It adds new insights into the testing process regardless of the used language or the host platform. However, this paper uses a research programming language called Unicon [8] and its Alamo [9] monitoring framework. Using Alamo does not limit this approach to Unicon nor prevent its applicability for other languages. The proposed approach is general and applicable during software testing regardless of the operating system or the programming language that is used. The only limitation is the availability of a sufficient monitoring or tracing mechanism.

Capturing the execution trace can be achieved by different means and/or tools [10], most of which depend heavily on the running platform and the language that is used. For example, $DTrace$, short for Dynamic Tracing, provides a system wide tracing capabilities that is applicable to all kinds of software. DTrace is not limited to user level applications, databases, and webservers; it is capable of instrumenting operating system kernels and device drivers. It is supported by different platforms including Solaris, Mac OS X and FreeBSD. On the other hand, mainstream programming languages such as Java, Python, and C provide their own tracing tools and packages.

Alamo is a lightweight architecture for monitoring; it provides the Unicon virtual machine with its monitoring facilities, and the capability for a program to load another program and execute it in a controlled environment. This facility is based on the thread model of execution monitoring, where a monitor program and its target program are in separate threads in a shared address space (*same process*). Alamo facilitates the ability to load a program, and sets it up with its own code, static data, stack, and heap, without linking symbols into the current program. Most importantly, Alamo's monitoring model ensures that there is no intrusion on the target program space, which is perfect for capturing the execution events triggered by the running test case.

Additionally, considering the millions of events that can be produced during the execution of a test case, Alamo's detailed VM instrumentation provides a set of 118 kinds of unique events. It provides an efficient filtering mechanism that allows the test engineer to specify the traced events (event codes and values) and then reduce the type and number of traced events. This masking allows the test engineer to customize the monitored events on the fly during the course of execution; any change on either of the two masks will immediately change the set of prospective events. This makes it ideal for experimentation.

III. Related Work

An execution trace gives a supplementary knowledge about the running software and its source code. Execution traces have been used for various reasons ranging from program debugging [11]–[14], visualization, to understanding parallel and distributed systems [15], [16], and of course software testing [17], [18]. Software testing incorporates activities across the entire software development life cycle (SDLC) and encompasses an intensive work of designing and running a large number of test cases (*test suite*). However, there are different methods to select, prioritize, and minimize test cases to improve quality and reduce the testing efforts [19]. Some researchers are focused on the test quality metrics [20]. For example, some metrics determine the coverage of the changed parts of code while others are depending on the number of fails in the past [21]. Others, such as Noor et al. are focused on test case prioritization, which ranks test cases within a test suite. They proposed a logistic regression model to predict the failing test cases in the current release based on a set of test quality metrics and named it the similarity-based metric [22].

Often, researchers are focused on the reduction of test cases by providing techniques to minimize the time and cost of regression testing. For example, Pavi et al. [23] used program slicing techniques by concentrating on parts of the code in the program and removing other parts from the testing process as long as there is no impact on the results. This technique can show the control-flow of a program for each test case and specify which statements are invoked with the test case. This technique decreases the number of test cases, and consequently the time and cost of testing. On the other hand, other researchers reported that when we use a coverage base technique, the reduction average of test cases is huge. Shengwei et al [24] used algorithm that cover all reachable states in closed loop controller by concentrating on path coverage and used the tool KLEE to generates test cases. Their approach is based on the code functionality, using it to identify the test cases that cover all paths in program, and eliminate any test case that only covers sub paths that are already covered by some test cases. Nadeem et al. [25] proposed a novel test case reduction technique called TestFilter, which first calculates the number of occurrences of a particular test case where it covers different statement of the program under test. TestFilter produces a weight for all generated test cases, then selects the test cases with the higher weights, and marks all of their corresponding requirements as contented. They repeat this process until all requirements are satisfied. This technique reduces the number of test cases without any decrease in the number of faults found.

In summary, researchers have done a great effort in proposing and developing techniques to reduce the time and cost of software testing, and regression testing in particular, by reducing the number of test cases. However, the proposed methodology in this paper gives the test engineer more insights into the code executed by the test case. The methodology employs sequences of execution traces in a manner similar to the use of DNA in the field of bioinformatics. These sequences of traces are used to detect similarities between related test cases. This proposed approach will help software testing in general and regression testing in particular, in which the test engineer can be able to classify the test cases in a regression test suite into four major categories. This approach will improve the documentation process of the test cases, and eventually improve the quality of the test suites and the tested software in general.

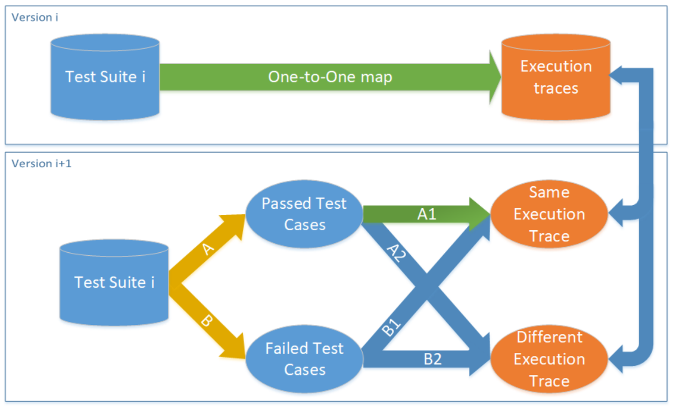


Fig. 1: Proposed Methodology: uses execution trace as a signature to identify a test case. Start from a passed $TestSuite_i$ that passed on $Release_i$. Therefore, before $Release_{i+1}$, it applies $TestSuite_i$ on the modified version and classify its test cases into four groups.

IV. Methodology and Approach

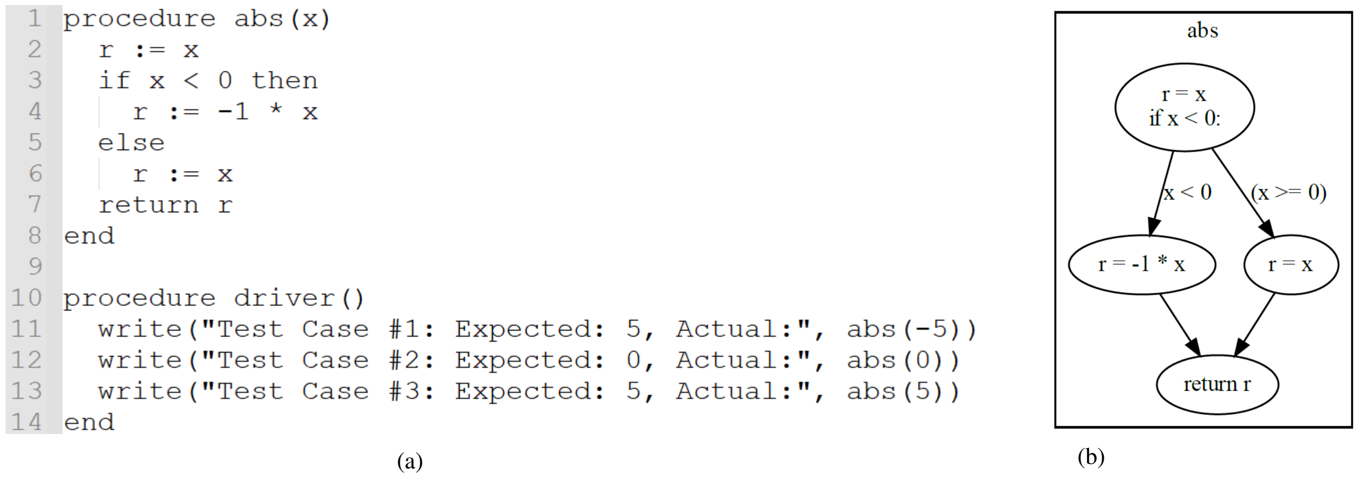
The goal of our methodology is to empower test engineers with a mechanism to improve the documentation and the quality of the used test cases. First, in a specific release (e.g. $release_i$), we need to be able to identify the redundant test cases; any two or more test cases that execute or cover the same sequence of execution traces are considered redundant. It is a waste of time and resources to run any two or more of the test cases if they share the same set of test requirements or execute the same test path; different test cases should follow distinctive and unique test paths, each of which should cover some of the not covered yet test requirements; regardless of the used testing criterion. Our proposed approach benefits from the execution trace, which is an ordered sequence of execution events that are reported from the execution of the test case (test input). Therefore, test cases that generate the same execution trace or simply the same hashed value of the ordered sequence of execution trace can be safely considered redundant. This similarity gives the test engineer the confidence in removing this redundancy.

Second, in-between releases, for example if we have two consecutive releases, say $release_i$ and $release_{i+1}$, then our approach can assist test engineers identify the set of test cases that are affected by the newly introduced modification in the source code; regardless whether it is a new feature or just a minor fix for a reported bug. Often, before a version of the software is released, it must pass all the test cases (a test suite). Usually, this test suite covers the set of test requirements imposed by one of the testing criteria. Then, before the next release ($release_{i+1}$), the testing team performs a regression test, or even a smoke test, which starts by running the test $suite_i$ on the beta version of the $release_{i+1}$. This test gives the test engineer the ability to find any of the broken functionalities. However, our proposed approach gives the test engineer more insight about these executed tests. It allows the testing team to identify and isolate the set of affected test cases. We break down the set of affected test cases into four different categories based on their execution trace and the comparisons between the old traces (in $release_i$) and the new traces from soon to become $release_{i+1}$, and for each test case. These categories are shown in Figure 1. All categories start with the passed test suite ($suite_i$); all test cases were passed in $release_i$, but in $version_{i+1}$ they fall into one of the following four categories:

- **A1**) This category includes all the test cases that are still considered passed test cases after the modification and still generate the same set of execution traces.
- **A2**) This category includes all the test cases that are still considered passed test cases after the modification but generate different set of execution traces and of course a different set of hashed values.
- **B1**) This category includes all the test cases that have just become failed test cases after the modification but still generate the same set of execution traces. These can result from parts of the source code (execution path) that were not modified but were affected by the side effects of a modification that happened somewhere else in the code.
- **B2**) This category includes all the test cases that have just become failed test cases after the modification and generate a different set of execution traces. This set of test cases formerly passed but now fail and generate a new set of execution traces.

V. Sample Program

Figure 2 presents a sample toy program that is used to demonstrate the basic idea of our proposed approach. This program finds the absolute value using the user defined $abs(x)$ function, which takes one numeric value as parameter and returns its absolute value. Some of the testing criteria may require the developer to provide test cases that cover a set of test requirements such as the set of nodes, set of edges, or set of edge-pairs, etc [26], [27]. Although this program consists of less than 10 lines of source code, the control flow graph shows that it consists of two distinct test paths, each of which generates a sequence of about 100 events during the execution trace; these fine-grain events include low level events such as the ones triggered by individual virtual machine instructions.



```
procedure abs(x)
  r := x
  if x < 0 then
    r := -1 * x
  else
    r := x
  return r
end

procedure driver()
  write("Test Case #1: Expected: 5, Actual:", abs(-5))
  write("Test Case #2: Expected: 0, Actual:", abs(0))
  write("Test Case #3: Expected: 5, Actual:", abs(5))
end
```

Fig. 2: Part (a) represents a simple program that performs the absolute value. Part (b) represents its corresponding control flow graph. It is clear that there are two different distinct paths in the graph.

The test engineer can customize these possibly reported events through various techniques such as the Alamo's $eventmask$ and/or $valuemask$. An increase in the number of monitored events causes an increase in the size of the execution trace and vice versa. Additionally, it may induce some delay in the execution of the test case due to the implicit instrumentation that allows these execution events to be traced and reported. Thus, the test engineer needs to find the least amount of reported events that is enough to differentiate between the different test paths. The test engineer chooses how fine-grained the monitoring must be.

VI. Sample Test Cases with Execution Traces

The given sample program, $abs(x)$ function in Figure 2, can be tested based on various testing criteria [28]. For simplicity, we can easily think of three different test cases based on the input domain partitioning criterion [29], [30]; a positive value, a negative value, and a zero. For each of which, we can choose the level of execution trace required, run the test cases, capture the corresponding sequence of execution trace, find its hashed value (e.g. $MD5$), and save this hashed value along with the test case. Table I shows these three test cases and their corresponding execution traces; the monitored execution events are given in Table II, which shows only a dozen distinct kinds of run time events. A sequence of these events is good enough to distinguish a test case from another when the executed code is different. The Alamo framework provides over 118 kinds of execution events [9] that are available to the test engineer to choose from.

Therefore, looking only at one test case $abs(-5)$ and monitoring only the execution event that are presented in Table II, it shows that this simple test case can generate a sequence of events with 41 run time events. This demonstrates that even for simple programs, the sequence of execution events can be very long. Thus, hashing this sequence using one of the hashing algorithms, such as the $MD5$, can provide a distinct but shorter and fixed-length representation of the execution trace. Therefore, similar or redundant test cases would generate the same sequence of execution traces. This gives the test engineer the opportunity to optimize and remove redundancy in test cases. For example, this can be done simply by comparing the corresponding hashed values ($128bits$) obtained from applying the $MD5$ on the sequence of events triggered by test case.

VII. Custom Execution Traces

There is a direct relationship between the number of monitored events and the length of the sequence of the resulted execution trace. Thus, the test engineer needs to customize this length by changing the number of monitored events. Alamo provides a mechanism to select the set of monitored events for every test case. Moreover, the set of monitored events can be customized during the execution of the test case, not only using the $eventmask$ but also using another mask called the $valuemask$, which allows the monitored program to report only those events that are associated with the specified values. This opens more opportunities for further experimentation.

However, in order to optimize the encoding process of test cases and their corresponding execution speed, the test engineer needs to utilize the minimum number of execution events that can be used to distinguish test cases from each other. For example, simple events that directly relate to the executed statements (code coverage) are the location events such as E_Line and E_Loc that represents line number and column number respectively and many other procedure-based and function-based events that can be used to assist in various control flow coverage criteria [31]. Other events such as E_Assign for assignments, E_Dref for reading the value of the variable, both can be useful for a testing based on the data-flow coverage criteria; **defs** represents those statements that assign values to variables whereas **uses** represents those statements that use (access or read) variables' values [32].

Table III shows the execution trace for TC#3. The length of the execution trace jumped from 41 to 112 events due to monitoring all possible execution events including the

TABLE I: Sample execution traces for three different test cases applied on the sample program given in Figure 2. It shows three different test cases. It is clear that execution trace for TC 2 and 3 are the same, simply by looking at the hashed value.

Test Cases	**Sample Sequences of Execution Trace**	**Hashed Value (MD5)**
TC#1: $abs(-5)$	E_Pcall:10,E_Line:10,E_Loc:10,E_Line:11,E_Loc:11,E_Loc:11,E_Deref:11, E_Pcall:1,E_Line:1,E_Loc:1,E_Line:2,E_Loc:2,E_Deref:2,E_Assign:2, E_Value:2,E_Syntax:3,E_Line:3,E_Loc:3,E_Loc:3,E_Deref:3,E_Line:4, E_Loc:4,E_Loc:4,E_Deref:4,E_Loc:4,E_Assign:4,E_Value:4,E_Line:3, E_Loc:3,E_Line:7,E_Loc:7,E_Pret:7,E_Deref:7,E_Line:11,E_Loc:11, E_Deref:11,E_Fcall:11,E_Fret:11,E_Line:12,E_Loc:12,E_Pfail:12	6F41EA76233B48F9ACAE4BF2ACEA3D35
TC#2: $abs(0)$	E_Pcall:10,E_Line:10,E_Loc:10,E_Line:11,E_Loc:11,E_Deref:11,E_Pcall:1, E_Line:1,E_Loc:1,E_Line:2,E_Loc:2,E_Deref:2,E_Assign:2,E_Value:2, E_Syntax:3,E_Line:3,E_Loc:3,E_Loc:3,E_Deref:3,E_Line:4,E_Loc:4, E_Line:6,E_Loc:6,E_Deref:3,E_Assign:3,E_Value:3,E_Line:3,E_Loc:3, E_Line:7,E_Loc:7,E_Pret:7,E_Deref:7,E_Line:11,E_Loc:11,E_Deref:11, E_Fcall:11,E_Fret:11,E_Line:12,E_Loc:12,E_Pfail:12	EB7628161CB787B644B6F205E6D55288
TC#3: $abs(5)$	E_Pcall:10,E_Line:10,E_Loc:10,E_Line:11,E_Loc:11,E_Deref:11,E_Pcall:1, E_Line:1,E_Loc:1,E_Line:2,E_Loc:2,E_Deref:2,E_Assign:2,E_Value:2, E_Syntax:3,E_Line:3,E_Loc:3,E_Loc:3,E_Deref:3,E_Line:4,E_Loc:4, E_Line:6,E_Loc:6,E_Deref:3,E_Assign:3,E_Value:3,E_Line:3,E_Loc:3, E_Line:7,E_Loc:7,E_Pret:7,E_Deref:7,E_Line:11,E_Loc:11,E_Deref:11, E_Fcall:11,E_Fret:11,E_Line:12,E_Loc:12,E_Pfail:12	EB7628161CB787B644B6F205E6D55288

TABLE II: Execution events that are used during the trace of test cases: 1, 2 and 3 given in Table I.

Event Code	**Triggered during the execution when a/an:**
E_Assign	assignment operation is started
E_Value	value is assigned
E_Deref	variable is dereferenced (*has been read*)
E_Line	line is changed (*line in the source code*)
E_Loc	location is changed (*column in the source code*)
E_Pcall	procedure is called
E_Pret	procedure is returned
E_Pfail	procedure is failed
E_Fcall	built-in function is called
E_Fret	built-in function is returned
E_Ffail	built in function is failed
E_Syntax	major syntactic structure is changed (*e.g. loops*)

events internal to the virtual machine. The use of the hash function unifies this process and allows simple management and comparisons between the execution of the test cases.

VIII. Future Work

For future work, we are planning to evaluate our approach during the testing and validation of real life applications; during which the approach will be used for testing a software product within a development environment. This evaluation will give us more insights into the amount of overhead that might be imposed by capturing the execution traces and evaluating whether it is worth the effort; getting actual feedback from software testers would put our approach to real life test.

Additionally, execution traces can be analyzed by different means such as finding or measuring the similarities between these test cases; various algorithms or even machine learning techniques can be used. For example, finding a similarity measure can be used to categorize these test cases into groups and prioritize them. This can be beneficial to the testing process; especially during the regression testing. For example, we may start our testing process with these test cases that are least common, which allows for a wider coverage of the source code and gradually go forward to those test cases that are the most common based on the similarity measure.

IX. Conclusion

Most of the software solutions live for a long time, during which they undergo various changes and many evolution. Any modification in the existing software mandates a regression testing that would secure the trust in these software products. For example, recently introduced features should not break the already existing ones. Unfortunately, this is not always the case, which requires a thorough regression testing to pass; meaning that all test cases should be passed before the new release. Focusing on the most affected test cases is a challenging problem, which often requires classifying and ranking test cases. This paper utilized the execution traces as a signature for the test case, it compares new signatures with the previously captured signatures; this approach assists test engineers pinpointing and isolating these test cases that execute parts of the newly introduced/modified code due to the most recent modification. Additionally, sequences of execution traces can be captured for various test cases during the testing process and it can be incorporated as part of the automatic testing process.

Moreover, our approach benefits from the nature of the hashed values such as $MD5$ and $SHA1$, which allows the detection of any small modification in the execution trace. This proposed methodology gives test engineers the ability to isolate touched test cases. Consequently, handling hashed values will provide a fixed length for the execution trace and simplify the management of test cases and their signatures.

TABLE III: A sequence of execution trace for the TC#3 presented in Table I. It is clear how the number of events jumped from 41 to 112. This motivates the customization of the reported events and consequently the length of the resulted trace.

A	E_Intcall:0,E_Cstack:0,E_Opcode:0,E_Opcode:0,E_Stack:0,E_Pcall:10,E_Line:10,E_Loc:10,E_Opcode:10,E_Opcode:10,E_Operand:10,E_Opcode:10,E_Opcode:10,E_Opcode:10,E_Opcode:10,E_Opcode:10,E_Opcode:10,E_Literal:11,E_Line:11,E_Loc:11,E_Opcode:11,E_Opcode:11,E_Stack:11,E_Deref:11,E_Pcall:1,E_Line:1,E_Loc:1,E_Opcode:1,E_Opcode:1,E_Operand:1,E_Opcode:1,E_Opcode:1,E_Opcode:1,E_Opcode:1,E_Line:2,E_Loc:2,E_Opcode:2,E_Ocall:2,E_Deref:2,E_Assign:2,E_Value:2,E_Oret:2,E_Opcode:2,E_Opcode:2,E_Opcode:2,E_Syntax:3,E_Operand:3,E_Line:3,E_Loc:3,E_Opcode:3,E_Opcode:3,E_Operand:3,E_Opcode:3,E_Opcode:3,E_Opcode:3,E_Opcode:3,E_Literal:3,E_Loc:3,E_Opcode:3,E_Ocall:3,E_Deref:3,E_Ofail:3,E_Line:4,E_Loc:4,E_Opcode:4,E_Opcode:4,E_Opcode:4,E_Opcode:4,E_Line:6,E_Loc:6,E_Opcode:3,E_Ocall:3,E_Deref:3,E_Assign:3,E_Value:3,E_Oret:3,E_Line:3,E_Loc:3,E_Opcode:3,E_Opcode:3,E_Opcode:3,E_Operand:3,E_Opcode:3,E_Opcode:3,E_Operand:3,E_Opcode:3,E_Opcode:3,E_Line:7,E_Loc:7,E_Opcode:7,E_Pret:7,E_Deref:7,E_Line:11,E_Loc:11,E_Opcode:11,E_Opcode:11,E_Stack:11,E_Deref:11,E_Ecall:11,E_Fcall:11,E_Aconv:11,E_Tconv:11,E_Sconv:11,E_Fret:11,E_Opcode:11,E_Opcode:12,E_Line:12,E_Loc:12,E_Opcode:12,E_Pfail:12,E_Opcode:0,E_Exit:0
B	C2B3BCD2817C54E4C4334B997E97AB11

References

[1] W. E. Wong, X. Li, and P. A. Laplante, "Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures," *Journal of Systems and Software*, vol. 133, pp. 68–94, 2017.

[2] J. Lee, S. Kang, and D. Lee, "Survey on software testing practices," *IET software*, vol. 6, no. 3, pp. 275–282, 2012.

[3] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1580–1596.

[4] K. Shimari, T. Ishio, T. Kanda, and K. Inoue, "Near-omniscient debugging for java using size-limited execution trace," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 398–401.

[5] T. Hynninen, J. Kasurinen, A. Knutas, and O. Taipale, "Software testing: Survey of the industry practices," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2018, pp. 1449–1454.

[6] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

[7] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 821–830.

[8] C. Jeffery, S. Mohamed, J. Al Gharaibeh, R. Pereda, and R. Parlett, *Programming with Unicon*. GNU Free Documentation License, 2018.

[9] C. L. Jeffery, *Program monitoring and visualization: an exploratory approach*. Springer Science & Business Media, 2012.

[10] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry, "Advanced and efficient execution trace management for executable domain-specific modeling languages," *Software & Systems Modeling*, vol. 18, no. 1, pp. 385–421, 2019.

[11] Z. AL-Sharif and C. Jeffery, "An agent-oriented source-level debugger on top of a monitoring framework," in *2009 Sixth International Conference on Information Technology: New Generations*, 2009, pp. 241–247.

[12] Z. Al-Sharif and C. Jeffery, "An extensible source-level debugger," in *Proceedings of the 2009 ACM symposium on Applied Computing*, 2009, pp. 543–544.

[13] Z. A. Al-Sharif, C. L. Jeffery, and M. H. Said, "Debugging with dynamic temporal assertions," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 2014, pp. 257–262.

[14] Z. Al-Sharif and C. L. Jeffery, "Language support for event-based debugging." in *SEKE*, 2009, pp. 392–399.

[15] M. Babaei, M. Bagherzadeh, and J. Dingel, "Efficient reordering and replay of execution traces of distributed reactive systems in the context of model-driven development," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 285–296.

[16] R. Uhrie, C. Chakrabarti, and J. Brunhaver, "Automated parallel kernel extraction from dynamic application traces," *arXiv preprint arXiv:2001.09995*, 2020.

[17] A. C. Paiva, A. Restivo, and S. Almeida, "Test case generation based on mutations over user execution traces," *Software Quality Journal*, vol. 28, no. 3, pp. 1173–1186, 2020.

[18] M. Utting, B. Legeard, F. Dadeau, F. Tamagnan, and F. Bouquet, "Identifying and generating missing tests using machine learning on execution traces," in *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 2020, pp. 83–90.

[19] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

[20] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Towards a catalogue of software quality metrics for infrastructure code," *Journal of Systems and Software*, p. 110726, 2020.

[21] B. Miranda and A. Bertolino, "Testing relative to usage scope: Revisiting software coverage criteria," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 3, pp. 1–24, 2020.

[22] T. B. Noor and H. Hemmati, "Studying test case failure prediction for test case prioritization," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2017, pp. 2–11.

[23] P. Saraswat, A. Singhal, and A. Bansal, "A review of test case prioritization and optimization techniques," *Software Engineering*, pp. 507–516, 2019.

[24] S. Xu, H. Miao, and H. Gao, "Test suite reduction using weighted set covering techniques," in *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE, 2012, pp. 307–312.

[25] A. Nadeem, A. Awais *et al.*, "Testfilter: a statement-coverage based test case reduction technique," in *2006 IEEE International Multitopic Conference*. IEEE, 2006, pp. 275–280.

[26] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[27] U. Kanewala, J. M. Bieman, and A. Ben-Hur, "Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels," *Software testing, verification and reliability*, vol. 26, no. 3, pp. 245–269, 2016.

[28] B. Miranda and A. Bertolino, "Testing relative to usage scope: Revisiting software coverage criteria," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 3, pp. 1–24, 2020.

[29] A. Bertolino, B. Miranda, R. Pietrantuono, and S. Russo, "Adaptive coverage and operational profile-based testing for reliability improvement," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 541–551.

[30] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia, "A survey on adaptive random testing," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[31] A. Rahmani, J. L. Min, and A. Maspupah, "An evaluation of code coverage adequacy in automatic testing using control flow graph visualization," in *2020 IEEE 10th Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. IEEE, 2020, pp. 239–244.

[32] H. Hemmati, "How effective are code coverage criteria?" in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 151–156.