World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:18, No:8, 2024

On Determining the Most Effective Technique Available in Software Testing

Qasim Zafar, Matthew Anderson, Esteban Garcia, Steven Drager

***Abstract*—**Software failures can present an enormous detriment to people's lives and cost millions of dollars to repair when they are unexpectedly encountered in the wild. Despite a significant portion of the software development lifecycle and resources are dedicated to testing, software failures are a relatively frequent occurrence. Nevertheless, the evaluation of testing effectiveness remains at the forefront of ensuring high-quality software and software metrics play a critical role in providing valuable insights into quantifiable objectives to assess the level of assurance and confidence in the system. As the selection of appropriate metrics can be an arduous process, the goal of this paper is to shed light on the significance of software metrics by examining a range of testing techniques and metrics as well as identifying key areas for improvement. In doing so, this paper presents a method to compare the effectiveness of testing techniques with heterogeneous output metrics. Additionally, through this investigation, readers will gain a deeper understanding of how metrics can help to drive informed decision-making on delivering high-quality software and facilitate continuous improvement in testing practices.

***Keywords*—**Software testing, software metrics, testing effectiveness, black box testing, random testing, adaptive random testing, combinatorial testing, fuzz testing, equivalence partition, boundary value analysis, white box testings

I. Introduction

SOFTWARE testing is the primary means for determining a software's quality through assessing whether the software functions correctly and identifying issues that jeopardize its correct behavior. Testing is the most prominent form of Verification and Validation (V&V) - two facets that ensure a given software fulfills its intended purpose. The V&V process is ever present throughout all stages of the software development life cycle (SDLC). Verification addresses whether the software is built according to the design specifications at each development stage whereas validation evaluates whether the software satisfied its intended requirements successfully. Unlike other forms of V&V such as code inspections and formal methods (rigorous mathematical proof-like techniques which can be brittle due to unyielding assumptions and are very costly to maintain), software testing can be parallelized and automated – features that directly correlate to its wider appeal and ability to handle large and complex software.

Testing, nevertheless, does have substantial limitations as well. For one, testing is an incomplete process. A software system's behavior is governed by its input parameters or configurations and their interactions. For a system with n such configurations, the total state space of valid input combinations is a cartesian product of n sets where each set contains all possible values a given input can take. Mathematically this can be represented as:

$$S_n = V_1 \times V_2 \times ... \times V_n \quad (1)$$

where V_i is the set of values input i can take. For all but the most trivial of software, this state space (S_n) tends to be so large that it is infeasible to test exhaustively. Rather, testing is done on a sample of the input space commonly referred to as a test suite or test set which in turn is composed of test cases that are tuples with specific assigned values for each input configuration.

Therefore, regardless of how much testing is undertaken, testing cannot prove an absence of bugs [1]. Used interchangeably with errors at times, bugs refer to the actual manifestation of errors discovered during the testing phase of the SDLC whereas an error, as defined by ISO/IEC/IEEE 24765 [2], is any difference between a computed, observed, or measured value or condition from its true or theoretically correct value or condition. Fig. 1 visualizes the relationship between error and its related terms. Faults are synonymous with bugs and when they are executed in the code, a failure occurs. A defect can refer to either a fault or a failure and vulnerabilities are a subset of bugs that are more critical due to their potential exploitability from bad actors. The rest of this paper will use the term error to generalize between the different terms as error is still the source point regardless of when and how it is discovered in the SDLC.

As testing generally observes only a small sample of the astronomical state space, it is always possible an error exists amongst dormant test cases left untested. Also, it is possible that the error simply cannot be materialized within the testing environment. For example, an appointment scheduler system that only tests the current year and fails to account for leap years will yield errors after the next leap day potentially setting up appointments on days when the venue is closed. Similarly, any unsuspected unknown that could potentially alter or affect the software behavior will not be capturable within the limits of testing.

Secondly, with regards to limitations of testing, due to the lack of a unifying quantification of metrics for measuring the adequacy of a test, comparing different testing techniques can often seem like comparing apples and oranges due to the heterogeneity of metrics between the different techniques. A recent survey [3] conducted to determine which techniques were most prevalently utilized, identified a diverse range of techniques but failed to determine any outstanding winner or

Qasim Zafar is with USAF, USA (e-mail: qasim.zafar@us.af.mil).

favorites. Generally, a well-studied and understood problem gravitates towards a solution that utilizes the most efficient tactics available with little variation to achieve its goal. This paper seeks, first, to provide a comprehensive examination of prominent software testing techniques, including an analysis of their associated metrics, advantages and disadvantages, and secondly, to formulate a methodology for facilitating informed decision making and enhancing overall test effectiveness for determining the most efficient test technique at any given point in the testing process.

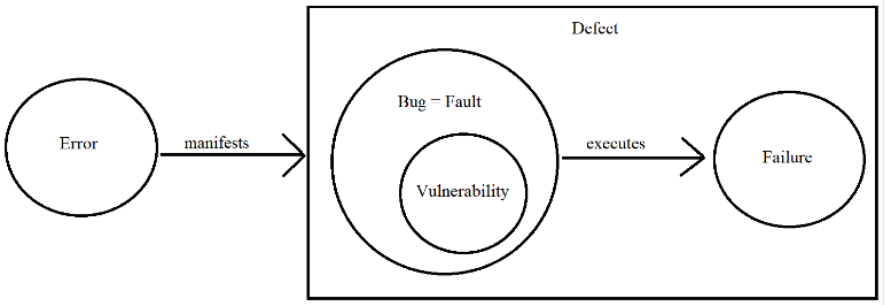

Fig. 1 Relation between error, bug, fault, failure, vulnerability and defect

The cost associated with fixing an error that is discovered late in the SDLC is exponentially greater than if it was discovered early. As development advances, more and more components are integrated together to form a cohesive piece of software and it becomes a daunting task to simply identify the root cause of the error. Also, as a software's lines of code (SLOC) grows, more changes may be needed to detangle the problem and issue a solution. Software testing accounts for about 50% of the development time and over 50% of the total cost [4]. These percentages increase even further for more critical software such as safety critical systems where human life may be impacted. As a result, extensive emphasis has been put forth towards shifting as much of testing to the left in the SDLC as possible (i.e. Agile methodologies) as well as utilizing system models such as digital twins to ensure correct-by-construction design to deter total cost and time investment.

To conform with Agile methodologies, different stages of testing are performed throughout the various stages of the development process. The most fundamental level, unit testing, involves testing components and modules of the software independently. This can be done as early as in the development or coding stage of the development lifecycle. The next stage, integration testing, combines multiple components together to form more complex components and verifies their interaction. The scope expands further in system testing which tests the system comprised of all components as a single entity. Acceptance test, the final test before deployment of the software, validates that all requirements of the software are met. Other tests like regression, smoke and stress test vary similarly either in terms of scale or the SDLC stage in which they are performed. The rest of this paper mentions testing in the more general sense, but the notion can be applied specifically to any of the different levels without loss of generality.

Given the infeasibility of exhaustive testing, testing frames into an optimization problem with an objective to maximize the confidence in the software assurance, namely that the software in question performs only its intended function and is free of vulnerabilities [5]. This problem is constrained by a tolerance cost which can depend on several factors such as budget and criticality. Different testing techniques utilize different methods to construct their test suites which are motivated by different empirical evidence. Overall, the techniques form two classifications which either construct tests based on input combinations or on the trajectory of the software code which observes the different paths that the software can undertake during execution from start to end. These classifications are referred to as black box and white box testing respectively. Generally, white box testing is utilized at the unit testing stage to ensure functional correctness of components independently until the system becomes more complex through continuous integration. Contrastingly, black box testing is applicable at every stage of software testing but is more strongly suited for the later levels due to its ability to abstract the varying layers of software and analyze at the system level.

II. Black Box Testing – Techniques and Analysis

Black box testing is a classification of testing in which the tester does not have knowledge about the inner workings of the software and is only able to interact with the software externally by providing a series of inputs. The inputs are then executed on the software resulting in some output. In older publications, this process was illustrated as a black box covering the software element and hence how it received its most widely recognized name. However, black box testing is also referred to as specification-based testing, behavioral testing, opaque testing, and closed testing. While an understanding of the underlying intricacies of the software is not needed, black box tests do require an executable artifact to test on - whether it be in the form of a source code or a software model, and therefore testing of this type cannot be performed until late into the development process. Similarly, the adequacy of these tests is inferred only from examining the results obtained from test suites.

Determining the functional correctness of the software system requires three items. The first as mentioned is the executable artifact required to run the tests. The second item is a test oracle which serves as a test "answer key" and is formally defined as any program, process, or body of data that specifies the expected outcome of a set of tests as applied to a test object

[6]. Thirdly, a mechanism is needed to compare the values between the two items. Given an oracle that is machine understandable, the entire process can be automated however oracle construction remains an active research problem within testing that is prominently focused on utilizing less-human centric methods to produce an automatable and scalable solution.

In general practice, the end user often performs the role of the oracle due to their high-level understanding of what the expected results should be based on the software requirements. As this process tediously identifies false positives and false negatives from test results, it can be an error-prone and time-consuming process due to human involvement. In this case, a false positive test result indicates an error that is unrealizable possibly due to dead code in the software that is unreachable under normal operation. This type of result can be more common among the so called "dirty" tests like fuzzing which will be discussed in depth later. On the other hand, false negative results are far less benign as they are test results that should have identified an error. Upon failing detection, primarily due to the incompleteness of testing as highlighted in the scheduler example, the errors lay dormant to potentially cause greater havoc during the operation and maintenance phase of the software.

If a testing process is unable to conjure a sufficient oracle, then functional correctness cannot be validated. Instead, the process devolves into checking for robustness only - whether the system hangs or crashes when it is presented with a given test case. Although less useful as a metric, this analysis can still be meaningful for software assurance, specifically for the discovery of vulnerabilities.

A. Random Testing

Random Testing (RT) is the simplest form of black box testing. It constructs test cases by randomly generating values for all inputs, executes them iteratively, and compares each output with the test oracle. The test quality is measured based on the total errors discovered proportional to the total number of runs. As the inputs are randomly generated and no record of previous executed tests is maintained to facilitate quick test execution, it can be easy to repeatedly test the same functional behavior across multiple tests, even generating the same test case although with significantly lower probability, and miss generating rare inputs that would have identified potential failures.

```
int component1 (int config1) {
    if (config1 == x) {
        # error-inducing code
    } else {
        # do something
    }
}
```

Fig. 2 An error that only occurs when the software executes a rare input

For example, we suppose a software component has the code structure illustrated in Fig. 2. If config1 can have hundreds of different values, then the probability of a test case exploring the *if* section of the *if* statement at random is extremely low that it would constitute as a rare input. Although a unit test is likely to pick up on this error, the issue is more pressing when the error occurs at an intersection of multiple input values from different components such that if the *if* condition was config1 == x && config2 == y && config3 == z instead, then the error would no longer be identifiable via unit testing and will have only a low probability of being discovered by random testing as well.

Over the years, advancing research has sought to improve upon random testing by managing two aspects of the testing process: limiting the degree of randomness of the input generation process or applying a reduction to the overwhelmingly large state space of test inputs.

B. Adaptive Random Testing

Adaptive Random Testing (ART) is a refinement of RT that reduces the former technique's degree of randomness. It maintains two sets: a candidate set and an executed set. The candidate set is a set of test cases generated randomly that determines the domain of test cases from which the next test case to execute is selected. The executed set is a set of all test cases that have been executed. For each iteration, ART generates a new candidate set, chooses one test case among them to execute and runs it, moving the test case to the executed set before it purges the candidate set. Fig. 3 provides a better detailed pseudocode implementation of ART.

```
candidate Set C , executed Set x = {},{}
generate initial test case t , execute t , add t to x

until reach stopping condition for iterations
    until reach stopping condition for populating candidate set
        generate new test case t
        if t is not in C , add t to C

    for each element i in C
        set i 's nearest neighbor distance value to inf
        for each element j in x
            d = fitness_function(i, j)
            if d < i 's nearest neighbor distance value, set them equal

    d_max = 0
    for each element i in C
        if i 's nearest neighbor distance value is less than d_max, set them equal
        t = i

    execute t , add t to x , reset C to {}
```

Fig. 3 Pseudocode for ART

ART uses a fitness function to calculate a "distance" measure between each member of the candidate set and each member of the executed set and associates the smallest value for each

member of the candidate set with any of the executed test case to represent the candidate test case's "distance" to its nearest neighbor. Then, the test case that has the highest distance value assigned to it is selected for execution. As ART is based on the empirical evidence that error-inducing inputs are often confined into contiguous clustered regions, the fitness function employs a greedy algorithmic approach to optimize its potential to explore a different region at every iteration by selecting the candidate that is furthest away from any neighbor. This form of testing is categorized as search-based software testing (SBST) which is another prominent area of research within software testing. The fitness function can utilize the distance function directly factoring the difference in values of each input equally or it can be fine-tuned to allow for specific inputs to have a greater or lesser influence on the distance measure and in turn identifying the cluster region to which the input belongs. For example, a health-based application that has both height and weight as inputs will need to normalize the input value units if it wants to ensure they have an equal impact on the calculations otherwise the greater range from the weight measurement will drown the influence from height. Furthermore, any input value that is not represented numerically will need to be converted to be used in the calculation.

Overall, ART performs better than RT as it either finds more errors in the same number of tests run or an equal number of errors in a fewer number of runs [7]. However, it does have a relatively high overhead cost due to the extra computations it needs to perform to determine which test cases to run and the overhead increases linearly alongside the size of the executed set. Whereas an RT iteration would only ever have a relatively fixed runtime of some length f, ART's runtime composes of $f + ct$, where the ct term incurs from the scaling induced from the candidate set size c and executed set size t. Even in variations of the algorithm, where the candidate set size c might not be of a fixed length between iterations, c can be treated like a constant without loss of generality as the variance would not be substantial enough to significantly affect the calculation. Therefore, after an n number of test cases have been executed with both RT and ART, the runtime for the next test case will differ by a factor of x such that an extra $x - 1$ RT test cases can be run within the same interval that could effectively reduce and potentially challenge the net effectiveness gain from ART. Thus, due to the lack of a standard metric for comparison, many comparison metrics exhibit an innate level of bias making them difficult to compare on equal terms and setting.

C. Combinatorial Testing

Combinatorial testing (CT), unlike most other testing methods, focuses on managing the large test input space through reduction. Also referred to as combinatorial interaction testing (CIT) and high throughput testing (HTT), CT is based on the interaction rule backed by empirical evidence that most errors can be realized from an interaction of only a few different input parameters. A study investigating the distribution of errors with respect to the number of interactions between different input configurations concluded that up to 47% of total errors occur from just pairwise interactions, 19% can come from three-way interactions, an extra 7% can be discovered by considering four-way interactions and just about every error can be encapsulated through six-way interactions [8]. Thus, the problem of testing all valid input combinations can be reduced to obtaining a t-way coverage of the input values where t refers to the number of interactions being considered and signifies the strength of the coverage.

Combinatorial coverage is a metric that measures the completeness of a test suite in terms of how many of the total t-way interactions that are encapsulated in the test suite. Not only does 100% t-way coverage imply 100% $t - 1$-way coverage but it can also guarantee some percentage of $t + 1$-way coverage and this is considered as a strong criterion for comparing test suites. As constructing an optimal test suite with maximal combinatorial coverage in the fewest number of tests is a very difficult problem to solve, several tools like ACTS [9] are utilized to obtain a heuristic solution using greedy approaches to construct a near optimal covering array. A covering array is a test suite that provides 100% t-way test coverage in the form of a matrix where each row consists of inputs for a singular test case.

When combinatorial coverage is applied to test suites derived through other test techniques, most suites tend to exhibit relatively low t-way combinatorial coverage beyond pairwise testing. One study was conducted on a spacecraft software with 82 different input configurations using a test suite of 7,489 test cases to verify correct system behavior under a mix of normal operating conditions and faulty scenarios. The test discovered that although the pairwise coverage achieved was 94%, higher t-way coverage dropped to 83% for 3-way coverage, under 70% for 4-way and slightly over 50% for 5-way coverage [10]. As combinatorial testing is tuned to optimize combinatorial coverage, it is not surprising that other testing techniques would yield lower coverage. Another study [11] comparing the fault detection capability of CT, RT, and ART concluded that CT performed at least as good as RT and ART on 98% of test scenarios and the difference was most noticeable on software with relatively few errors that only surfaced from a small number of test cases. This aligns with the previous discussion that very specific errors are difficult to detect with random based techniques.

Still, other studies [12] conflict with the improved coverage and fault detection achieved with CT stating that the improvement is either minimal and not worth the computational effort to generate a covering array or that the results are indistinguishable altogether from other techniques like RT. These clashing conclusions are likely a result of the fact that a testing technique's effectiveness is determined by a multitude of factors and there may not be a one size fits all solution. For example, factors such as the software's size, complexity and budget, as well as the number of errors it has, including their density, how easy they are to discover, to even the software's use case which often determines the rigor of testing undertaken can all play an important role not only in determining the effectiveness of a specific testing technique but also in determining which technique is most suitable.

Nevertheless, there are still some key advantages of using

CT. For one, CT's fault detection effectiveness has a high degree of stability. Where stability refers to the discrepancy in results obtained over multiple consecutive executions of the testing technique generating a new test suite to execute each time, higher stability exudes more confidence on the test's effectiveness. Furthermore, as software development gravitates towards a less human-centric process, the behavior of the software system will no longer depend just on the written code like traditional software but rather more significantly on the input data. This may lessen the overall effectiveness of structural coverage testing techniques as even the coding process could be labeled as a black box and thus acquiring the source code or reasoning about the trajectory of the software may not be feasible. However, input-based tests like combinatorial testing will still be relevant [13] and may prove helpful at bridging the gap on understanding unexplainable results by allowing for better inferences to be made regarding the software behavior in light of results from testing.

D.Equivalence Partitioning and Boundary Value Analysis

If input configurations are continuous or unbounded, the valid input state space becomes infinite. Equivalence Partitioning (EP) is a technique that can transform a state space into regions, called equivalence classes, where each class is a grouping of inputs that output the same system behavior. EP is guided by the principle that if a test case resolves successfully, then other test cases from the same equivalence class would also fail to produce an error. Therefore, testing multiple test cases belonging to the same equivalence class becomes inefficient as the software will treat them in the same manner. This allows for a reduction on the state space by requiring a minimum of only a single test case from each equivalence class to be performed. This reduction process can be visualized in Fig. 4.

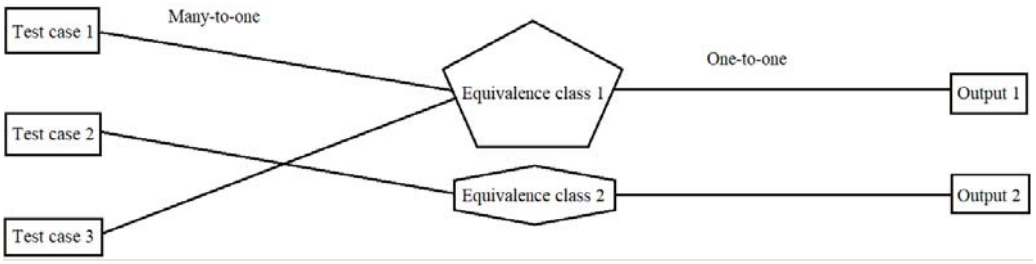

Fig. 4 Reducing the input state space for testing by categorizing test cases into equivalence classes

As all equivalence classes define the complete system behavior, the relation between an equivalence class and a distinct output result is an injective mapping. Thus, testing in terms of equivalence classes bounds the state space by the number of distinct output results – the sum of total distinct errors that can be encountered and the total sum of distinct correct behaviors of the system. This is upper bounded by the total input state space but more importantly is finite even if the input space is not. However, lacking perfect knowledge as a black box technique, this technique relies on heuristics to carve out each partition and is as much a form of art as it is science as two very different partitioning representations could be equivalent if they discover the same errors.

Boundary Value Analysis (BVA), generally used in conjunction with EP, evaluates inputs at both sides of each boundary formed via EP to not only check for software correctness but also ensures that the partition regions have been constructed accurately. As programs fail some equivalence class condition and transition to a neighboring class at the boundary of each partition, the test cases constructed near the boundary regions maximize the probability of discovering errors [14] and thus are more critical to test.

E. Fuzzing

Fuzz testing or fuzzing has generally been utilized as a black box testing technique although in recent years it has also seen emergence as a grey box technique utilizing structural coverage criterion to generate tests. Conceptualized by Barton Miller [15], a professor at the University of Wisconsin, when he encountered several crashes while attempting to run command line programs over a modem connection due to electrical interference during a severe thunderstorm, fuzzing performs negative testing through sending anomalous data through the lens of invalid, malformed, and unexpected inputs to investigate the non-functional properties of a software such as security and reliability. Fuzzing has been utilized as a powerful tool for security-critical software due to its ability to detect crashes, timeouts, memory leaks, assert violations as well as prevent zero-day attacks by gauging security risks before a malicious adversary may be able to exploit them.

Starting with an input seed (a starting test case), the technique continually modifies the previous set of inputs using a mutation scheme to generate all future test cases. This process can be as simple as applying an increment, flipping a bit, truncating and/or repeating some of the bit of the input values but can also utilize more popular and complex genetic algorithms like particle swarm and ant colony optimization to mutate some of the inputs. As fuzzing allows for invalid inputs to be considered, the input state space is once again infinite as the set of invalid inputs is unbounded in cardinality. Assuming inputs that cause a vulnerability are very specific in terms of test cases to constitute as a "rare input", it will take a very long amount to time to discover these vulnerabilities if exploration from test cases is random. However, while some of the simpler implementations of fuzzing can indeed be random, those utilizing more complex mutations such as genetic algorithms can be more deliberate at deviating from previously executed

test cases. Similarly, the white box implementation of fuzzing leverages symbolic execution to ensure each test case traverses a different execution path of the code altogether. As such the white box implementation tends to be more efficient than the black box implementation on code coverage as black box mutation schemes tend to be unguided and randomized.

Due to its conceptual simplicity and low barrier to deployment, fuzzing has been very successful at discovering vulnerabilities in real world software such as Heartbleed and is a requirement by Microsoft for their products.

III. White Box Testing Techniques and Analysis

On the opposite spectrum, white box testing also referred to as structure-based testing, clear box testing, glass box testing, and open box testing among others requires knowledge over the inner working of the software. Due to this reason, this type of testing is usually performed by the developer while the former black box testing can be outsourced to another party without concern for loss of proprietary or confidential information. White box testing can begin as early as post design phase of the SDLC and is more focused towards testing the internal structure, code, and implementation of the software even to the extent of algorithm testing. Generally, white box testing tends to be more exhaustive and guided compared to black box testing, although it does also come with the caveat of being more time consuming as well as requiring greater expertise of the tester. White box tests assess the structural coverage by breaking the software into measurable building blocks in the form of statements, branches and conditions. These criteria are defined ahead.

Statement coverage measures the total statements that are executed by a test suite as a proportion of the total statements in the software program. It tries to ensure that each statement is executed at least once throughout the test suite, providing some level of basic coverage on the codebase. However, failing that, it can highlight areas which have not been tested to identify potential dead code that may be unreachable. A significant disadvantage of statement coverage is that it is the weakest form of coverage that is still widely used. It does not check for correctness but only that the statements are executed and does not guarantee comprehensive testing such that executing only one path of a condition is sufficient and it does not need to cover all possible scenarios.

Next, branch coverage, also referred to as decision coverage, measures how many of the total branches – edges that emerge as a result of diverging paths created from conditional statements, are traversed. Similarly, path coverage looks at all potential full paths that the software can partake from start to end. Although path and branch coverage seem very similar, path coverage supersedes branch coverage and thus is a more rigorous coverage metric. Likewise, branch coverage also supersedes statement coverage. Conditional coverage testing seeks to test all the conditional expressions for all possible outcomes at least once.

Modified Condition/Decision Coverage (MC/DC) testing goes one step further combining branch and condition coverage and strengthens it further by requiring test cases to illustrate that each input can independently affect the outcome. Per DO-178C guidance, MC/DC testing is a requirement for avionics software. Generally, it requires a minimum of $n + 1$ test cases, where n denotes the number of unique inputs to the software. Fig. 5 highlights the differences between the coverages as applied to the accompanying code structure.

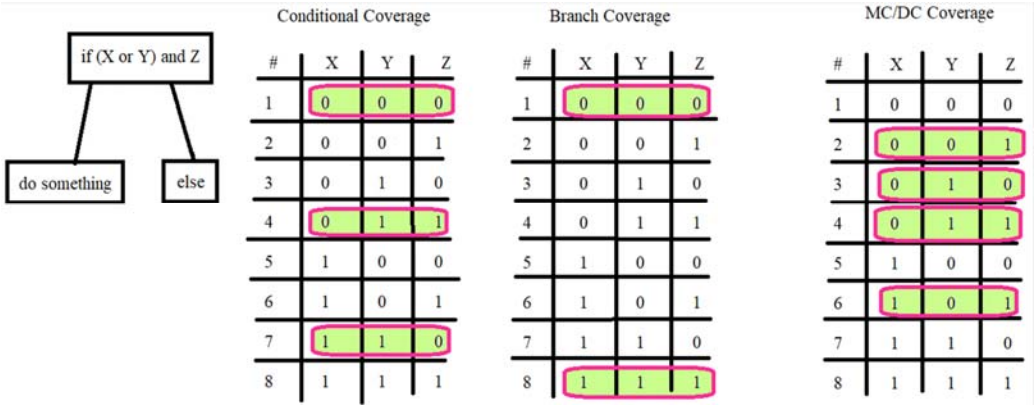

Fig. 5 Minimal 100% coverage test suites for conditional, branch and MC/DC coverage

For conditional coverage testing, X, Y and Z need to evaluate to true and false at least once each. The first test case evaluates both X and Y to false and thus Z is not evaluated. The second test (4th row) evaluates X to false again, but Y to true and Z to true. The third test evaluates X to true and Z to false. Together, the three tests cover at least one instance of X, Y and Z being true and false. Branch coverage testing on the other hand only aims to explore the diverging branches from the if condition at least once and that can be achieved by multiple test case pairs of which one is shown in Fig. 5.

For MC/DC testing, a different pair of tests show the independence of X, Y and Z (test 2 and 6 for X, test 2 and 4 for Y, and test 3 and 4 for Z) while also adhering to branch and conditional coverage criteria. Between the respective pairs, only one evaluation is changed such that only one bit differs between each pair in the representation. This showcases the influence a single input has on the system behavior as it evaluates to both true and false while everything else remains

consistent.

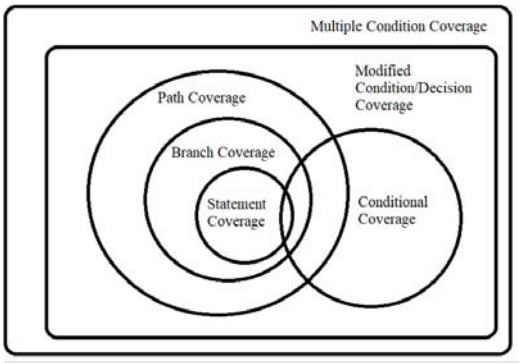

Fig. 6 Varying strengths of coverage

Lastly, Multiple Condition Coverage (MCC) is a combinatorial white box testing technique that requires testing all value combinations of conditions. It is more rigorous than the other white box testing techniques and would require testing all 8 rows of the table in Fig. 5. More generally it requires a minimum of 2^n test cases where n would be the total number of conditions. Due to the combinatorial aspect with no bounds unlike the t-way coverage of combinatorial testing, the required number of test cases can be quite large. Due to this, MCC has yet to gain more widespread acceptance, but it highly recommended in the European standardization, EN 50128, for railway systems with high safety integrity requirements. For completeness, the levels of coverage are visualized in Fig. 6.

Although code coverage metrics are useful, 100% coverage with any criteria is seldom achieved. Statement coverage, as the most used coverage technique is feasible for 100% coverage (discounting any unreachable code), yet most test suites only reach upwards of 85% coverage. Similarly other types of coverage like branch coverage seek to achieve coverage in the range of 70-90% [16]. This is partly due to the pareto principle as applied to software testing, which claims that 80% of the testing effort can be expended with only 20% of the cost and further gains have significant diminishing returns.

IV. Further Analysis

Table I provides a snapshot of all the testing techniques that have been discussed in this paper along with their metrics and where they are most suitable.

As many metrics are technique specific, it is difficult to utilize them for a direct comparison. This is even more apparent when trying to compare a black box and a white box technique as all structural coverage criterion like statements ran or branches traversed cannot be measured with black box testing techniques. However, the total number of errors and runtime are metrics that can be universally measured between all techniques. The previously mentioned study [11] comparing three black box techniques simply used errors found as its defining metric. However, this study can be done only in hindsight at the conclusion of testing and does not help provide guidance to the testing process itself. In a more practical setting, efficient testing is an online problem where a decision needs to be made on which test should be run for most efficient discovery of errors under imperfect information.

TABLE I
Comparative Analysis of Software Testing Techniques

Black Box	Metric	Best Suited for
Random Testing	Faults discovered/ time	Quick execution, faults are easy to discover
Adaptive Random Testing	Faults discovered/ number of test cases ran	Exploratory, faults are clustered
Combinatorial Testing	T-way coverage achieved, faults discovered, t+1-way coverage achieved	Identifying errors related to interactions between inputs
Equivalence Partitioning & Boundary Value Analysis	Faults discovered	Easy to draft partitions
Fuzz Testing	Faults discovered, false positive & negative ratios	Discovering vulnerabilities (security focused)
White Box	**Metric**	**Best Suited for**
Statement Coverage	Number of statements executed at least once	Reachability analysis
Branch Coverage	Number of branches explored at least once	Code coverage
Path Coverage	Number of full execution paths traversed at least once	Discovering logic errors
Condition Coverage	Number of conditional outcomes explored at least once	Examining behavior of conditions
Modified Condition/Decision Coverage	Branch coverage + condition coverage + showing each input can independently affect system output	Safety critical software
Multiple Condition Coverage	Testing all value combinations arisen from conditions	Safety critical software

We suppose a tester wants to know which technique between A or B is more effective at finding errors for a specific software. A testing technique A is more effective than technique B if it either discovers more errors in a shorter or equal amount of time than the runtime of B or if it discovers an equal number of errors in a strictly shorter amount of time. However, as many testing techniques utilize some degree of randomness or lack an established order in which to execute test cases, it is highly probable that while a technique A is more efficient than technique B at some time t, the situation can be reversed later at another time $t + i$. Therefore, time needs to be an important factor in the metric to capture the constrained nature of the problem otherwise exhaustive testing would be the best testing technique. As more errors are discovered the longer a test is run, the discovery rate of errors for each technique is universally measurable and comparable as a rate of change: de/dt.

At first, each of the rates will need to be initialized by running each of the testing technique with a sample number of tests. Afterwards, the most efficient technique is the one with the highest discovery rate and as more tests are run with that

technique, the rate continues to change. Once the rate for the technique being utilized is no longer the greatest, the testing technique should be switched to the one with the new greatest discovery rate and so forth.

Over a prolonged period, the rate of error discovery converges to a concave behavior exhibiting diminishing returns from testing. With the total set of errors that can be encountered being finite and as more and more errors get discovered over time, the remaining errors will get fewer, impacting the discovery rate in the same fashion.

Although, there is no clear stopping condition for this method, just like there is no stopping condition for most testing techniques, several conditions can be constructed such as in the form of a time limit or ensuring all rates of error discovery are within some tolerable amount, ϵ. Overall, the stopping condition will largely depend on several distinct factors such as usage and budget of the software in question.

Overall, the analysis of the state-of-the-practice on testing techniques reveals that software metrics leave a lot to be desired on evaluating the overall effectiveness of software testing. The first shortcoming lies in the lack of comparable metrics. While various metrics exist, they often differ in definitions, measurement scales, or units, as seen with ART and RT as well as the various white box testing techniques, that it is quite a challenge to establish a common ground for comparison and draw any meaningful conclusion that will aid in making informed decisions. Addressing this issue will likely require stronger standardization of metrics with broad applicability and adaptability like the error discovery rate to enable fair and accurate comparisons.

Another critique on current metrics is that while good metrics need to be easy to measure, they seem to overlook the qualitative attributes that define software quality. Many existing metrics such as code coverage and faults detected focus only on measurable aspects and do not quite holistically capture the essence of crucial qualitative aspects such as useability and reliability of the software. To address this limitation, a concrete mapping that connects qualitative attributes to quantifiable metrics needs to be established, and it is something we seek to investigate further.

V. Conclusion

In conclusion, determining the most efficient technique available in software testing requires a comprehensive analysis of several intricate factors, including the specific test objectives, the nature of the system under test, and the available resources. This paper examined and compared different black box and white box testing techniques, along with their associated metrics, to evaluate their efficiency in achieving thorough test coverage. Although there is no single technique that can universally address all contexts, the efficiency of testing techniques can be evaluated by considering the error discovery rate. By measuring the number of errors detected per unit of testing effort, stakeholders can gain insights into the relative efficiency and efficacy of various testing techniques to make informed decisions for optimizing their testing efforts and enhancing the quality and reliability of the software involved.

References

[1] W. Dijkstra, "Notes on Structured Programming," Technological University Eindhoven T.H. Report 70-WSK-03, Second edition, April 1970.

[2] "ISO/IEC/IEEE International Standard - Systems and software engineering--Vocabulary," in ISO/IEC/IEEE 24765:2017(E), vol., no., pp.1-541.

[3] V. Vukovic, J. Djurkovic, M. Sakal, & L. Rakovic, "An Empirical Investigation of Software Testing Methods and Techniques in the Province of Vojvodina," Tehnicki Vjesnik-Technical Gazette, 2020.

[4] B.W. Boehm, "The High Cost of Software", in Practical Strategies for Developing Large Software Systems, E. Horowitz (editor), Addison-Wesley, Reading, MA, 1975.

[5] DoD Instruction 5200.44: "Protection of Mission Critical Functions to Achieve Trusted Systems and Networks (TSN)," Nov 5, 2012.

[6] B. Beizer, *Software Testing Techniques.* London: International Thompson Computer Press, 1990.

[7] T. Y. Chen, F. Kuo, H. Liu, & W. C. Wong, "Does Adaptive Random Testing Deliver a Higher Confidence than Random Testing?", the Eighth International Conference on Quality Software, 2008.

[8] D. R. Kuhn and D. R. Wallace, "Software fault interactions and implications for software testing," IEEE Trans. Softw. Eng., vol. 30, no. 6, pp. 418–421, Jun. 2004.

[9] Y. Lei, R. N. Kacker, & D. R. Kuhn, "ACTS: A combinatorial test generation tool" IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), 2013.

[10] D. R. Kuhn, R. N. Kacker and Y. Lei. "Combinatorial coverage as an aspect of test quality," 2015.

[11] H. Wu, C. Nie, J. Petke, Y. Jia and M. Harman, "An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing" in IEEE Transactions on Software Engineering, vol. 46, no. 03, pp. 302-320, 2020.

[12] L. S. Ghandehari, J. Czerwonka, Y. Lei, S. Shafiee, R. Kacker, and R. Kuhn, "An empirical comparison of combinatorial and random testing," in Proc. IEEE Int. Conf. Softw. Testing Verification Validation Workshops, 2014, pp. 68–77.

[13] R. Kuhn, M. S. Raunak and R. Kacker, "Combinatorial Coverage for Assured Autonomy," IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Charlotte, NC, USA, 2022, pp. 357-358.

[14] Y. Singh, *Software Testing*. Cambridge University Press, 2012.

[15] A. Takanen, J. D. DeMott, & C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*, 2018.

[16] J. W. Hollén, P. S. Zacarias. "Exploring Code Coverage in Software Testing and its Correlation with Software Quality; A Systematic Literature Review," 2015.