A Comparative Study of Whole Issues and Challenges in Mutation Testing

Negar Nazem Bokaei
Department of Computer Engineering
Alzahra University
Tehran, Iran
Email: nnb029@gmail.com

Mohammad Reza Keyvanpour
Department of Computer Engineering
Alzahra University
Tehran, Iran
Email: keyvanpour@alzahra.ac.ir

***Abstract*— Mutation testing is a powerful and expensive method in software testing context. It is used as a test criterion to assess the quality of test suites or generate test suites which are able to kill mutants created by seeding artificial defects in the original program code. These test suites are high-quality tests with good ability to find probable faults in the program under test. Mutation testing has gained high attention recently through its superiority over other testing criteria in different aspects and is the main topic of researches in software testing area. As well as powerfulness of mutation testing, it has challenges and difficulties which are great barriers for industrial utilization. This paper provides a comprehensive classification of mutation testing issues and challenges. Furthermore, a comparative study is conducted on researches that made an effort to investigate issues and resolve challenges of mutation testing along with presenting a classification of these proposed approaches. This comparative structure suggests different research topics and makes a guideline for researchers interested in mutation testing context. This also can be used for the purpose of comparing the existing methods, selecting the best one and improving selected methods.**

Keywords—mutation testing; software engineering; mutation testing challenges

I. INTRODUCTION

Software testing is an important part of program development and an expensive activity. Mutation testing is a fault-based testing technique that seeds artificial defects into program code and makes faulty versions of the program under test, called mutants. These artificial faults represent mistakes that programmers may create [1]. Mutants are made by small syntactic alternations in the program code, then test suites are generated to detect most of these mutants by distinguishing their misbehavior from the original program. Detected mutants by tests are called killed mutants. The final test suite has a good ability to detect probable faults in the program under test and can be utilized for future uses.

Mutation testing is a powerful testing technique due to its underlying idea of simulating faults and their behavior. It is evidence that there is a tendency to use mutation testing in different contexts of software testing as it is more reliable in fault revealing. Mutation testing is growing to become a pervasive testing method as it subsumes other testing methods. The prominent property of mutation testing is falsifiability, which means failure in killing some mutants causes failure of detecting some faults, a property that not exists in other software testing techniques [2]. Despite this superiority of mutation testing, it is very expensive and time-consuming.

In this paper we collect, classify and introduce the main issues and challenges of mutation testing and the most recent resolving methods, based on the functionality of mutation testing procedure. This will clear that each proposed work would cover which part of mutation testing process. It also acts as a roadmap and helps active researchers in the context of mutation testing who are interested to make this testing technique as a useable pervasive technique in the industrial environment, to find and follow their interest and choose an appropriate issue to work on. This will help researchers to compare methods with each other, select the best one and improve it.

The remainder of the paper is organized as follow: Section 2 reviews the related work. The process of mutation testing is represented in Section 3. The proposed classification of mutation testing challenges and issues is presented in Section 4. Section 5 presents a classification of proposed resolving methods. In Section 6 a comparative structure is introduced that evaluate existing methods based on classified challenges. In the end, the Conclusion is presented in Section 7.

II. RELATED WORK

Mutation testing has become the subject of interest recently and reached its maturity phase by several proposed methods and enhancements that attempt to make it usable. Several types of researches have been conducted to develop various methods of cost reduction and automation in order to enhance different part of mutation testing system, test generation and mutant generation for instance. Despite this increasing popularity, just

978-1-7281-0872-8/19/$31.00 ©2019 IEEE

little reviews and survey works have been conducted in order to make a rich literature and roadmap for interested researchers.

Offutt and Untch [3] made a classification of proposed cost reduction techniques for mutation testing. They introduce “do fewer”, “do smarter” and “do faster” strategies. The “do fewer” approaches try to run fewer mutants without getting undesirable performance loss. The “do smarter” approaches attempt to reduce expenses by distributing computational costs over several machines, avoiding complete execution or saving state information between executions. The “do faster” approaches seek to generate and execute mutants as quickly as possible. Jia and Harman [1] provided a comprehensive survey of mutation testing by presenting some of the mutation testing challenges and the result of different development analysis proposed to resolve these problems. They classified the proposed method to cost reduction techniques, consisting of mutant reduction and execution cost reduction, and equivalent mutant detection techniques. Papadakis et. al. [2], presented a deep survey containing the latest advances about problems of mutation testing. They also mentioned some challenges and open problems of mutation testing.

There is actually not exists specific research that investigates all challenges and issues of mutation testing with a comprehensive classification and comparative structure of recent resolving methods.

III. Mutation Testing Process

The traditional process of mutation testing has expressed by Offutt and Untch [3]. Recently Papadakis et al. [2] have made an extension on it and presented a modern mutation testing process shown in figure 1.

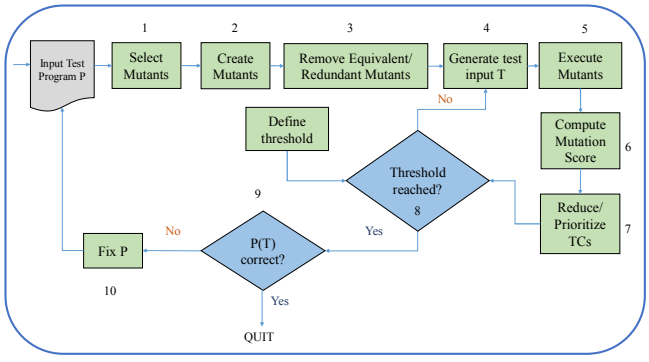

Fig. 1. Modern Mutation Testing Process

The modern mutation testing process is as follows: First, a set of mutation operators will be selected to apply to the program under test (step 1). A mutation operator is a rule for changing statements of a program in order to create mutants. By applying mutation operators, mutants will be created (step 2). Equivalent mutants, those which are syntactically different from the original program but semantically equivalent to it, and redundant mutants, mutants that are semantically different with the original program but are subsumed by others, are removed (step 3). In the next step, a set of test cases are produced to execute against mutants (step 4). Each of mutants is executed with produced test set and killed mutants are determined (step 5). Mutation score is calculated for that test set to determine how adequate it is (step 6). After that, test reduction in order to remove inappropriate test cases and test prioritization to order appropriate test cases first can be performed (step 7). Step 4 to 7 will be repeated until the desired mutation score is achieved (step 8). At the end of the process, the result of test execution is assessed and its behavior is asserted (step 9). If there are some faults in the program, they should be located and fixed (step 10). The process will be repeated until almost all faults in the program under test are found.

IV. Mutation Testing Issues and Challenges

The mutation testing process explained in the previous section can be summarized into three steps as below:

1. Mutant generation
2. Test generation and execution
3. Evaluation

Step two contains a partial loop that is repeated until the desired mutation score is achieved and an adequate test set found. The overall process continues until almost all of the program faults are found and fixed (main loop). We made a classification of challenges according to these three steps that explained in the following. This classification is shown in figure 2.

A. Challenges related to Mutant Generation Phase

As explained before, in this level mutants are created by applying mutation operators to the original program. This process has some challenges related to designing mutation operator, the large number of mutants created, stubborn and equivalent mutants. We made a brief explanation for each of them.

1) Mutation Operators

Mutation operator is an alteration rule. A small syntactic change is made in the original program under test by applying it. This change can be made by deleting, inserting or replacing operators of the program statements [1]. The main challenge in this phase is that sets of mutation operators should be designed for each programming language which the program wanted to be tested is written by. Therefore, a program could not be tested unless the mutation operators of its language have been designed and implemented before.

2) Large number of Mutants

By applying all of the mutation operators, mutants would be created each of them has a small change made by replacing,

inserting, deleting operators or statements in the program under test. So, a large number of mutant would be created each of them needs to be processed [4]. Some of these mutants are redundant, they are both syntactically and semantically different from each other, but can be killed by same test cases. Therefore, all of these mutants are not needed and they can skew mutation score as some of them can be killed by any test case [5]. Moreover, executing this large number of mutants is very time consuming and causes the mutation analysis to be a non-efficient and expensive test method to use [1]. It also affects the use of mutation testing for large-scale complex programs as a huge number of mutants would be created [6].

3) Equivalent and Stubborn Mutants

Equivalent mutant is a modified version of the program which has different syntaxes from the original version but semantically performs the same. This kind of mutants should be detected as they never killed and waste the test budget in order to find a test input that indicates the different behavior. Also, they have negative effects on calculating mutation score, as they cause a low score. Therefore, detecting equivalent mutants is important and is known as the equivalent mutant problem [7]. Equivalent mutant detection is an undecidable problem, meaning that no automated technique exists able to detect all the equivalent mutants [8]. Stubborn mutants (hard-to-kill) are mutants which can be killed, in other words, there exists a test case that can distinguish its misbehavior from the original version, but not have been found yet. This kind of mutants also consumes much time of the testing process [9].

B. Challenges related to Test Case Generation and Execution Phase

This level consists of three stages. First, mutation-based test cases are generated. These test cases should be able to kill mutants. Then all test cases with all mutants are executed and analyzed which is the most expensive task in mutation testing and should be optimized in some way [2]. At the end of the execution, test cases are evaluated by computing mutation score. Each of these phases contains some issues explained below.

1) Quality and Size of Test Suite

Test case generation, using mutation criterion or not, is an undecidable problem which has no algorithmic solution [10]. Mutation-based test generation is the process of producing adequate test cases which are able to kill mutants. One important challenge in mutation testing is generating test cases with high quality that kill most of the created mutants. Generating mutation-based test cases needs high effort and is time-consuming. Automation of test case generation process is a way to reduce costs of mutation testing, generating test cases that achieve high mutation score [11]. In the other hand generating high-quality mutation-based test cases (those which can kill a large number of mutants) is important because it is the only way showing the superiority of mutation criterion fault revelation ability compared to others [12]. In addition to generating a test suite with high quality, it is important that this test suite be minimal. The size of the test suite is a challenging issue as an oversized test set raises the costs and executing all of them consume much time.

2) Execution Cost

Execution cost is related to the process of executing all mutants by test cases at each partial iteration and consists of fault seeding time, compiling time and testing time which specify test failing or passing. As the number of mutants and test cases are usually high, the mutant execution process is very time consuming and expensive [1] [13]. Another problem is related to determining the termination of mutant execution time with a test case, which is an undecidable problem [2]. When a mutant execution time exceeds, it is considered to have an infinitive loop and will affect the total execution time [14]. So, identifying these infinitive loops that occur frequently is important.

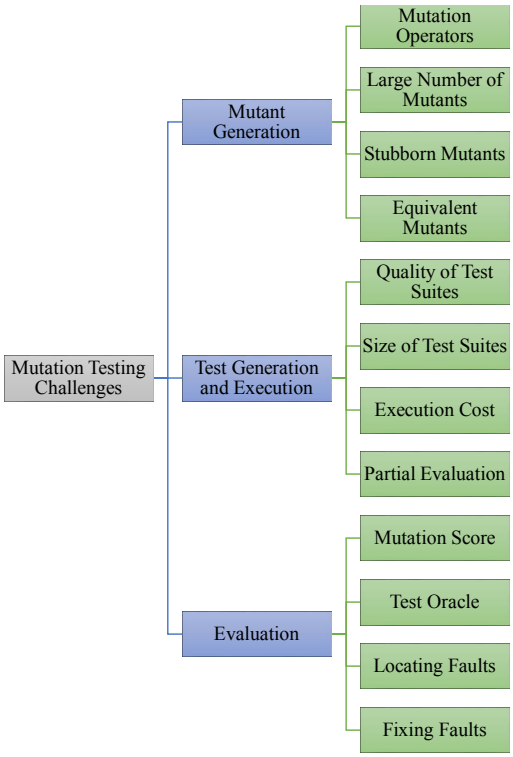

Fig. 2. Classification of Mutation Testing Challenges and Issues

3) Partial Evaluation

At the end of each iteration of generating test cases and executing them with mutants, an evaluation is needed to calculate the mutation score for those test cases. We call this evaluation, a partial evaluation that is for assessing test cases quality at each iteration of improving test cases, different from the overall evaluation that is about testing and evaluating the program under test. In this inner loop of the process, test cases are generated and improved until reaching a defined threshold.

After executing mutants with test cases the produced output should compare to the expected result determining whether the mutants have been killed. This assessing is performed in form of an oracle created based on killing conditions according to the type of program output. In non-deterministic systems, defining the oracle which models the behavior of the output based on the killing conditions is a challenging problem [2].

C. Challenges related to Evaluation Phase

In this phase, an adequate test suite with desired mutation score is achieved. The program under test is asserted, actual faults are located and repaired.

1) Mutation Score

Mutation score is a termination threshold for the test generation and execution loop of mutation testing indicating the adequacy of test cases and reflecting the level of confidence of testing for developers. Quantifying the relation between mutation score and fault revelation is a challenging problem known as the confidence inspired by mutation score proposed in [2].

2) Test Oracle Problem

One important factor for fault revealing is test oracle which compares the actual result with the desired [15]. A mutation testing system should be equipped by appropriate test oracles for future uses. After generating test cases and reaching the desired mutation score, the behavior of the program under test should be asserted and compared with the expected behavior. This oracle is used to check the expected output of effective test cases with the purpose of detecting real faults and repairing them. If the program does not behave as expected, it contains actual faults. This task is costly and usually performed manually that is a challenging problem called test oracle problem. The automation of test oracle is a solution to reduce these costs, which is investigated in two dimensions: writing test oracle and evaluating test outcomes [16]. This problem has not gained much attention in testing context specifically mutation testing.

3) Debugging

If the program contains actual faults, the main attempt is finding the position of faults in the program code, the costlier part of the debugging process, and then repairing them automatically which are important challenging issues. Mutation-based fault localization is the most effective kind of fault localization as the location of mutants that are killed, represents realistic faults and aids identifying faulty statements in the program [17]. It is important to fix the faults after detecting and locating. This process can be automated in mutation testing tools. Automated fault repair is an important part of mutation testing system which repairs discovered faults in the program and is a solution for manual time-consuming fault fixing [18].

V. Approach Classification

Many types of researches have been conducted in the context of mutation testing in order to cover its different issues. Generally, researches in the context of mutation testing can be categorized into three groups. Automation techniques which proposed to reduce human interventions, cost reduction techniques that are introduced to reduce the high expenses of mutation testing and analytical techniques that attempt to prove the fundamental assumption of mutation testing and its superiority. In this section, we present a classification of proposed approaches that try to resolve mutation testing challenges. In figure 3 this classification is presented.

These approaches and their containing methods are introduced in bellow. Due to the diversity of proposed methods, most popular and recent techniques are mentioned and explained. It is important to note that test case generation, test oracles and debugging consisting fault localization and repair, are general problems in each testing system since they are human intensive. They also are challenging issues in mutation testing system. Several techniques proposed to resolve these challenges but we just mention mutation-based methods introduced for these problems as mutation testing issues, since mutation criterion is the most powerful in software testing context.

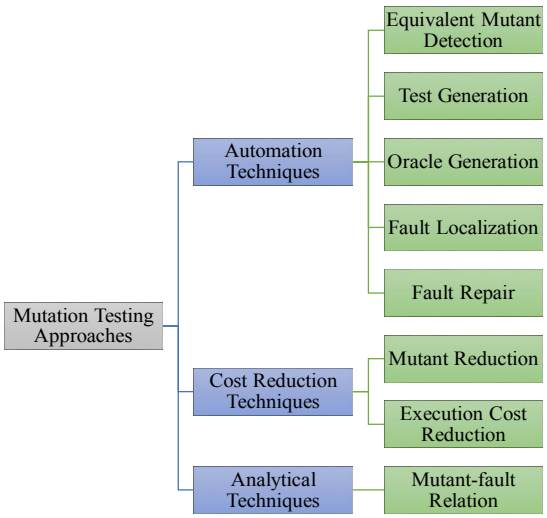

Fig. 3. Classification of Mutation Testing Approaches

A. Automation Techniques

These methods attempt to automate some challenging and human-intensive part of mutation testing consisting equivalent

mutant detection, test case generation, test oracle generation, fault localization, and fault repair.

1) Equivalent Mutant Detection

Several techniques have been proposed in order to automate process of detecting the equivalent mutants. *Compiler optimization* is one of these techniques that detects equivalent mutants by compiling each of the mutants and comparing their machine code with that of the original program [19]. Another attempt is using *symbolic execution* to identify non-equivalent mutants by executing mutants symbolically with symbolic inputs. The underlying idea is that showing non-equivalency of mutants is much easier than proving the equivalency of them [20]. In [21], *Data flow analysis* has been used for developing a tool, named MEDIC (Mutants' Equivalence DIsCovery), to identify equivalent and stubborn mutants. MEDIC uses a set of data follow patterns previously proposed, that their existence in the source code of the program results in the creation of equivalent mutants. The set of problematic data follow pattern, identify the location of the source code having potential of equivalent mutant generation. Another way avoids creating equivalent mutants by detecting *behavior-affecting part* of program code [22].

2) Test Case Generation

As test case generation is the main part of mutation testing, many techniques have been implemented in order to produce test cases able to find most of the mutants. The most popular and effective method is *search-based test generation* that uses search algorithm in order to find high quality test cases among all possible test cases. *Hill Climbing algorithm* is used to find appropriate test cases that strongly kill mutants [23]. *Genetic algorithm* is used to generate whole test suite that have good ability in killing mutants, while trying to maintain the number of test cases as small as possible [24]. DynaMOSA is a search based test generation technique, recently proposed. This technique tries to solve test generation problem in multi-objective form by dynamic selecting targets. a mutation-based fitness function is introduced to guide the mutation-based test generation problem [25]. There are also other methods like *constraint-based* [26] and *concolic execution* techniques [27] [28] used in test case generation problem.

3) Test Oracle Generation

In [29], a method is proposed to improve test oracles by using test case generation to identify false positive and mutation analysis to identify false negative. Another method is proposed to create effective oracle data, the set of variables that expected values are defined for, by using mutation analysis to decide which variable include in oracle data. Underlying idea is that oracle data which are probable to reveal fault of mutants will also be probable to show actual faults in the program under test [30]. A method has been proposed to generate oracles for JavaScript applications based on *state difference detection* using mutation analysis to detect state disagreements of application elements by comparing collected execution traces of original applications and mutants [31]. Another work was conducted to generate oracle assertions, which is related to the partial evaluation for assessing the behavior of each mutant. This method uses an observer to record all necessary information and assertions are created by analyzing differences of execution traces generated by an observer [32].

4) Fault Localization

Among fault localization techniques, mutation-based fault localization is more powerful than others. Two approaches are proposed in this context. *Metallaxis* [17], implies that mutants killed by failing tests have a connection with program faults as they provide an indication to the faulty location of the program. These faults can be localized by mutants that are coupled with. *MUSE* [33] localizes faults based on mutants which turns the failing test cases to passing. This method attempts to find faulty statements by utilizing different properties of mutants which are made by mutating a faulty statement and those made by mutating correct statement.

5) Fault Repair

One method to repair faults based on mutation testing is reversing the process of seeding artificial faults in order to repair actual faults. To do so, suspicious statements found by fault localization are mutated until a possible fix be found. By applying a mutation operator at the faulty location, a mutant would be created that is a candidate repair. Despite the requirement in the test execution phase that all mutants should fail at list one test to be killed, in debugging phase it is required that one mutant pass all test to be accepted as a repair [34] [35] [36].

B. Cost Reduction Techniques

As mentioned before, mutation testing is an expensive and time-consuming technique. In order to reduce these costs and expenses many techniques have been proposed which have focused on reducing the number of generated mutants and reducing the efforts during execution of them.

1) Mutant Reduction

In order to reduce the huge number of produced mutants, many techniques have been proposed. Mutant reduction is a critical task should be performed to reduce the cost of mutation testing and improve its scalability in order to test more complex programs. As a result, the execution time of mutation testing would decrease [37]. The main attempt of mutant reduction techniques is reducing the huge number of produced mutants keeping the same effectiveness of fault detection ability. One method to reduce the huge number of mutants is selecting them. Mutants can be selected randomly (*mutant sampling*) or based on some strategies like choosing a small set of mutation operators (*selective mutation*) or even single operator [38] to produce a representative subset of mutants [39]. In *Higher Order Mutant* method, mutation operators applied multiple times to produce higher order of mutants instead of applying just once. This method causes a reduction of mutants as each higher order mutant represents multiple first

order mutants and leads to reducing test effort [40]. Some recent work is conducted with respect to the program structure. Constructing a new program by inserting mutant branches and analyzing the *dominance relation* between them to detect non-dominated mutant is the basis of the work conducted by [41]. The non-dominated mutant branches correspond to a reduced set of mutants. Another method proposed by [6] predict the *mutant utility* in terms of equivalence, triviality, and dominance based on the programs abstract syntax tree. In this way equivalent, trivial and redundant mutants would be detected and it will help to reduce the mutants by removing these types of mutants. The *path-aware mutant reduction* technique [42], select mutants based on their depth in the program structure and diversity of their behavior. The mutants which are in deeper locations and are as diverse as possible in terms of their behavior give higher priorities. In [43], mutation operators are classified into two ranking, one based on their degree of redundancy and another based on the quality of tests they help to produce. These rankings are manipulated in *selective mutation.*

2) Execution Cost Reduction

Even though some mutant reduction techniques are performed in the first step to reduce costs, some optimization techniques should be performed during the execution to optimize mutant execution process by reducing the number of executions or computational overhead [44]. *Mutant schema* generation is a well-known technique that embeds all mutants in on metaprogram, thus the compilation time of mutants would be reduced since just one metaprogram needs to be compiled [1].

More recent researches focused on collecting information at the run-time in order to filter redundant executions. *Infected-based* technique (weak mutation) indicate that a test case is executed on a mutant only if it leads to a different execution state of that mutant. These mutants can be *partitioned* based on their infected state and so test cases execute against just a representative mutant to avoid redundant execution [44]. *MUSIC*, mutant schema improved with extra code, is a technique to filter mutant execution by removing uncovered mutants and no infected mutants and identify mutants with infinite loops [14]. *Split-stream* technique execute common parts of all mutants together in one process until mutated statements are reached, then the execution stream of the program splits into a set of processes [13] [45]. Another attempt is performed to enhance the split-stream technique. In addition to removeing redundant executions before mutated statements, also redundant execution after these points can be omitted. By *clustering* mutants according to their equivalence classes modulo the current state, mutants are clustered in equivalence classes. Mutants in the same class have the same states after executing the mutated statement so the redundant execution after that can be removed [45]. *ComMT* is a proposed method which combines state infection and data compression technique. It compresses the execution by selecting a subset of mutants and a subset of test cases based on the similarity of mutants and test cases [46]. The underlying idea of *Metamutation* technique is raised from the mutant schema technique. Every mutable program element is replaced with a metamutation function. The difference is that each of metamutation function is activated in a virtual machine instead of having native code implementation [13]. *Test case minimization and prioritization* is an orthogonal way to reduce the execution cost of mutation testing by reducing the number of test execution. Test prioritization reorders the tests in a way that mutant killing tests run earlier and this leads to a reduction of test execution time. Test minimization reduces the number of test cases without any loss in the quality of tests [47] [48] [49].

C. Analytical Techniques

Some techniques have been proposed with aim of proving fundamental assumptions about mutation testing according to some logical analysis and empirical evaluation.

1) Mutant-fault Relation

Several researchers have made an effort to evaluate the effectiveness of mutation testing in line with the challenging issue of its fault revelation ability. To do so, researchers trie to measure the level of confidence inspired by mutation scores which implies that how much the mutation score correlates to the actual faults rates or mutants to real faults.

In [50], the *correlation between mutant detection and real fault detection (MDFD)* is investigated and the research tries to show that the ability of test suites to detect mutants (mutation score) correlates with their ability to reveal actual faults. This research also attempts to prove that real faults are coupled with mutants. The results of the experiment represent that most of the real faults are coupled with mutants and also there is a correlation between mutant detection and real fault detection. There is an attempt which implements a methodology to evaluate test suites for different testing criteria by using faulty versions of the program instead of assuming clean (fix) programs. Also, the *correlation between mutation score and real fault detection (MSFD)* is investigated that measures the real fault detection ability of test suites. The results indicate that strong mutation testing has the most ability of fault revelation and also there is a strong relation between fault revelation and higher mutation score levels, that means fault revelation considerably improves only at high score levels [12]. This shows the real fault revelation rate of test suites since the mutation score is a measurement for test suites fault revelation ability. Another research has investigated the *correlation between mutation score and real fault detection* by considering *test suite size* (MSFD-TSize). The results show that higher mutation score results in better fault detection. There is a dependency between mutation score and test suite size and it affects the correlations. The fault detection is influenced with both mutation score and test suite size and correlations become weak by decoupling mutation score from test suite size [51].

VI. Evaluation

In this section, some researches addressing different kinds of mutation testing challenges are mentioned in table I. The methods and an evaluation of these methods are presented based on classified challenges. Since there are various proposed methods to resolve the vast number of mutation testing challenges, just more recent and popular methods are mentioned. The challenges that the specific research work directly focused on resolving them are marked by "*". As some challenges are dependent on each other, resolving one challenge may implicitly lead to alleviate the difficulty of other challenges. These are also marked.

VII. Conclusion

Mutation testing is a powerful testing technique but has some challenges which prevent it to become a pervasive practical testing method. The main attempt to do this is the automation of different part of the mutation analysis system and reducing its costs using different techniques. In this paper, a comprehensive classification of important challenges and issues in the mutation testing context have been proposed and introduced techniques to resolve these challenges have been evaluated. This will help to eliminate limitations of these proposed techniques, create more powerful methods in the future and also make potential topics for future researches in mutation testing context by the facility of comparing and selecting best methods that the proposed structure provides.

TABLE I. Evaluation of recent mutation testing approaches based on classified issues and challenges

Challenge Classification		Research Work	Used Technique	Year	Mutant Generation				Test Generation and Execution				Evaluation			
					Mutation Operators	Number of Mutants	Stubborn Mutants	Equivalent Mutants	Quality of Test Suite	Size of Test Suite	Execution Cost	Partial Evaluation	Mutation Score	Test Oracle	Locating Fault	Fixing Fault
Automation Techniques	Test Generation	[24]	Genetic algorithm	2015					*	*	*	*				
		[23]	Hill climbing algorithm	2016					*							
		[25]	DynaMOSA	2018					*							
	Equivalent Mutant Detection	[21]	Data flow analysis	2015			*	*								
		[22]	Behavior-affecting portion choosing	2015	*	*		*								
		[20]	Symbolic execution	2016				*					*			
		[19]	Compiler optimization	2018		*		*								
	Fault Localization	[52]	Metallaxis	2014		*									*	
		[33]	MUSE	2014											*	
		[17]	Metallaxis	2015											*	
	Fault Repair	[35]	Mutation analysis	2014											*	*
		[36]	Search-based and mutation analysis	2015											*	*

TABLE I. (Continued.)

Challenge Classification		Research Work	Used Technique	Year	Mutant Generation				Test Generation and Execution				Evaluation			
					Mutation Operators	Number of Mutants	Stubborn Mutants	Equivalent Mutants	Quality of Test Suite	Size of Test Suite	Execution Cost	Partial Evaluation	Mutation Score	Test Oracle	Locating Fault	Fixing Fault
Automation Techniques	Oracle Generation	[32]	Execution trace difference detection	2012					*			*				
		[30]	Variable effectiveness ranking	2015										*		
		[31]	State differences extraction	2015										*		
		[29]	Mutation analysis and test case generation	2016										*		
Cost Reduction Techniques	Mutant Reduction	[9]	Selective mutation	2014		*									*	
		[37]	Mutant sampling	2015		*	*	*								
		[53]	Ranking	2015		*	*	*								
		[54] [55]	Higher order mutants	2014 2016		*										
		[6]	Path-aware	2017		*		*			*					
		[41]	Dominance relation detection	2017		*					*					
		[42]	Mutant utility prediction	2017		*		*								
		[43]	Selective mutation	2017	*	*			*							
	Execution Cost Reduction	[13]	Split-stream execution and metamutation	2016		*					*					
		[45]	Equivalence modulo state clustering	2017				*			*					
		[46]	ComMT	2017							*					
		[48]	Test case reduction and prioritization	2018			*		*	*	*					
Analytical Techniques	Mutant-fault Relation	[50]	MDFD	2014									*			
		[12]	MSFD	2017									*			
		[51]	MSFD-TSize	2018						*			*			

REFERENCES

[1] Y. Jia, M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering,* vol. 37, no. 5, pp. 649 - 678, 2011.

[2] M. Papadakisa, M. Kintisa, J. Zhangc, Y. Jiab, Y. L. Traona, M. Harmanb, "Mutation Testing Advances: An Analysis and Survey," in *Advances in Computers*, 2017.

[3] A. J. Offutt, R. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation Testing for New Century*, W. E. Wong, Ed., San Jose, California, USA, 2001, p. 45–55.

[4] T. T. Chekam, M. Papadakis,T. F. Bissyande´,Y. L. Traon, "Selecting Fault Revealing Mutants," in *2018 International Conference on Software Engineering*, 2018.

[5] B. Kurtz, P. Ammann, J. Offutt, M. Kurtz, "Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Chicago, 2016.

[6] C. Sun, F. Xue, H. Liu, X. Zhang, "A path-aware approach to mutant reduction in mutation testing," *Information and Software Technology,* vol. 81, p. 65–81, 2017.

[7] L. Madeyski, W. Orzeszyna, R. Torkar, M. Jo□zala, "Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation," *IEEE Transactions on Software Engineering,* vol. 40, no. 1, pp. 23 - 42, 2014.

[8] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, M. Harman, "Detecting Trivial Mutant Equivalences via Compiler Optimisations," *IEEE Transactions on Software Engineering,* vol. 44, no. 4, pp. 308 - 333, 2018.

[9] X. Yao, M. Harman, Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *36th International Conference on Software Engineering*, Hyderabad, India, 2014.

[10] P. McMinn, "Search-based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability,* vol. 14, no. 2, pp. 105-156, 2004.

[11] F. C. Souza, M. Papadakis, V. H. S. Durelli, M. E. Delamaro,, "Test Data Generation Techniques for Mutation Testing: A Systematic Mapping," in *Proceedings of the 11th ESELAW*, 2014.

[12] T. T. Chekam, M. Papadakis, Y. L. Traon, M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*, Buenos Aires, Argentina, 2017.

[13] S. Tokumoto, H. Yoshida, K. Sakamoto, S. Honiden, "Muvm: Higher order mutation analysis virtual machine for C," in *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, Chicago, IL, USA, 2016.

[14] P. R. Mateo, M. P. Usaola, "Reducing mutation costs through uncovered mutants," *Software Testing, Verification and Reliability,* vol. 25, no. 5-7, p. 464–489, 2015.

[15] N. Li, J. Offutt, "Test Oracle Strategies for Model-Based Testing," *IEEE Transactions on Software Engineering,* vol. 43, no. 4, pp. 372 - 395, 2017.

[16] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering,* vol. 41, no. 5, pp. 507 - 525, 2015.

[17] M. Papadakis, Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *software testing, verification and reliability,* vol. 25, no. 5-7, p. 605–628, 2015.

[18] V. Debroy, W. E. Wong, "Combining mutation and fault localization for automated program debugging," *Journal of Systems and Software,* vol. 90, pp. 45-60, 2014.

[19] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering,* vol. 44, no. 4, pp. 308-333, 2018.

[20] D. Holling, S. Banescu, M. Probst, A. Petrovska, A. Pretschner, "Nequivack: Assessing mutation score confidence," in *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops*, Chicago, IL, USA, 2016.

[21] M. Kintis, N. Malevris, "MEDIC: A static analysis framework for equivalent mutant identification," *Information & Software Technology,* vol. 68, pp. 1-17, 2015.

[22] S. Mirshokraie, A. Mesbah, K. Pattabiraman, "Guided mutation testing for javascript web applications," *IEEE Trans. Software Eng.,* vol. 41, no. 5, p. 429–444, 2015.

[23] F. C. M. Souza, M. Papadakis, Y. L. Traon, M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST@ICSE 2016*, Austin, Texas, USA, 2016.

[24] G. Fraser, A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering,* vol. 20, no. 3, p. 783–812, 2015.

[25] A. Panichella, F. M. Kifetew , P. Tonella, "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets," *IEEE Transactions on Software Engineering,* vol. 44, no. 2, pp. 122 - 158, 2018.

[26] M. Papadakis, N. Malevris, "Mutation based test case generation via a path selection strategy," *Information & Software Technology ,* vol. 54, no. 9, p. 915–932, 2012.

[27] M. Papadakis, N. Malevris, M. Kallia, "Towards automating the generation of mutation tests," in *The 5th Workshop on Automation of Software Test, AST 2010*, Cape Town, South Africa, 2010.

[28] A. Sabbaghia, H. R. Kananb, M. R. Keyvanpour, "FSCT: A new fuzzy search strategy in concolic testing," *Information and Software Technology,* vol. 107, pp. 137-158, 2019.

[29] G. Jahangirova, D. Clark, M. Harman, P. Tonella , "Test oracle assessment and improvement," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Saarbru¨cken, Germany, 2016.

[30] G. Gay, M. Staats, M. W. Whalen, M. P. E. Heimdahl, "Automated oracle data selection support," *IEEE Trans. Software Eng.,* vol. 41, no. 11, p. 1119– 1137, 2015.

[31] S. Mirshokraie, A. Mesbah, K. Pattabiraman, "SEFT: automated javascript unit test generation," in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST*, Graz, Austria, 2015.

[32] G. Fraser, A. Zelle, "Mutation-Driven Generation of Unit Tests and Oracles," *IEEE Trans. Software Eng.,* vol. 38, no. 2, pp. 287-292, 2012.

[33] S. Moon, Y. Kim, M. Kim, S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Seventh IEEE International Conference on Software Testing, Verification and Validation*, Cleveland, Ohio, USA, 2014.

[34] W. Weimer, Z. P. Fry, S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *28th IEEE/ACM International Conference on Automated Software*

Engineering, Silicon Valley, CA, USA, 2013.

[35] V. Debroy, W. E. Wong, "Combining mutation and fault localization for automated program debugging," *Journal of Systems and Software,* vol. 90, p. 45–60, 2014.

[36] S. H. Tan, A. Roychoudhury, "relifix: Automated repair of software regressions," in *37th IEEE/ACM International Conference on Software Engineering*, Florence, Italy, 2015.

[37] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, A. Groce, "How Hard Does Mutation Analysis Have to Be, Anyway?," in *26th IEEE International Symposium on Software Reliability Engineering*, Gaithersbury, MD, USA, 2015.

[38] M. E. Delamaro, L. Deng, V. H. S. Durelli, N. Li, J. Offutt, " Experimental evaluation of SDL and one-op mutation for C," in *Seventh IEEE International Conference on Software Testing, Verification and Validation*, Cleveland, Ohio, USA, 2014.

[39] J. Zhang, "Scalability studies on selective mutation testing," in *37th IEEE/ACM International Conference on Software Engineering*, Florence, Italy, 2015.

[40] M. Polo, M. Piattini, I. G. R. de Guzm´an, "Decreasing the cost of mutation testing with second-order mutants," *Software Testing, Verification and Reliability,* vol. 19, no. 2, p. 111–131, 2009.

[41] D. Gong, G. Zhang, X. Yao, F. Meng, "Mutant reduction based on dominance relation for weak mutation testing," *Information & Software Technology,* vol. 81, p. 82–96, 2017.

[42] R. Just, B. Kurtz, P. Ammann, "Inferring mutant utility from program context," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Santa Barbara, CA, USA, 2017.

[43] P. Delgado-Prez, S. Segura, I. Medina-Bulo, "Assessment of c++ object- oriented mutation operators: A selective mutation approach," *Software Testing, Verification and Reliability,* vol. 27, no. 4-5, 2017.

[44] R. Just, M. D. Ernst, G. Fraser, "Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States," in *International Symposium on Software Testing and Analysis, ISSTA '14*, San Jose, CA, USA, 2014.

[45] B. Wang, Y. Xiong, Y. Shi, L. Zhang, D. Hao, "Faster mutation analysis via equivalence modulo states," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Santa Barbara, CA, USA, 2017.

[46] Q. Zhu, A. Panichella, A. Zaidman, "Speeding-up mutation testing via data compression and state infection," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Tokyo, Japan, 2017.

[47] L. Zhang, D. Marinov, S. Khurshid, " Faster mutation testing inspired by test prioritization and reduction," in *International Symposium on Software Testing and Analysis, ISSTA '13*, Lugano, Switzerland, 2013.

[48] L. Gonzalez-Hernandez, B. Lindström, J. Offutt, S. F. Andler, P. Potena, M. Bohlin, "Using Mutant Stubbornness to Create Minimal and Prioritized Test Sets," in *2018 IEEE International Conference on Software Quality, Reliability and Security*, Lisbon, Portugal, 2018.

[49] N. Mottaghi , M. R. Keyvanpour, "Test suite reduction using data mining techniques: A review article," in *2017 International Symposium on Computer Science and Software Engineering Conference (CSSE)*, Shiraz, Iran, 2017.

[50] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Hong Kong, China, 2014.

[51] M. Papadakis, D. Shin, S. Yoo, D. H. Bae, "Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults," in *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, Gothenburg, Sweden, 2018.

[52] M. Papadakis, Y. L. Traon, " Effective fault localization via mutation analysis: a selective mutation approach," in *Symposium on Applied Computing*, Gyeongju, Republic of Korea, 2014.

[53] A. S. Namin, X. Xue, O. Rosas, P. Sharma, "Muranker: a mutant ranking tool," *Software Testing, Verification and Reliability,* vol. 25, no. 5-7, p. 572–604, 2015.

[54] A. Parsai, A. Murgia, S. Demeyer, "A model to estimate first-order mutation coverage from higher-order mutation coverage," in *IEEE International Conference on Software Quality, Reliability and Security*, Vienna, Austria, 2016.

[55] L. Madeyski, W. Orzeszyna, R. Torkar, M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Trans. Software Eng.,* vol. 40, no. 1, p. 23–42, 2014.