Survey on Mutation-based Test Data Generation

Le Thi My Hanh, Nguyen Thanh Binh, Khuat Thanh Tung
The University of Danang, University of Science and Technology, Vietnam

Article Info

Article history:

Received Apr 27, 2015
Revised Jun 23, 2015
Accepted Jun 16, 2015

Keyword:

Constraint based testing
Dynamic symbolic execution
Mutation testing
Search-based test data generation
Test data generation

ABSTRACT

The critical activity of testing is the systematic selection of suitable test cases, which is able to reveal highly the faults. Therefore, mutation coverage is an effective criterion for generating test data. Since the test data generation process is very labor intensive, time-consuming and error-prone when done manually, the automation of this process is highly aspired. The researches about automatic test data generation contributed a set of tools, approaches, development and empirical results. This paperanalyzes and conducts a comprehensive survey on generating test data based on mutation. The paper also analyzes the trends in this field.

Corresponding Author:

Le Thi My Hanh,
Information Technology Faculty,
The Univeristy of Danang, University of Science and Technology
54Nguyen Luong Bang, Danang, 550000, Viet Nam.
Email: ltmhanh@dut.udn.vn

1. INTRODUCTION

Software testing is an important means used to assure the quality of software. However, testing is an expensive, tedious and time-consuming activity as it can consume more than 50% of the total cost of the software development [1]. Automatic test data generation is one of the approaches to improve the accuracy as well as to reduce the cost of testing activity. The effectiveness of test data is measured by the ability to reveal the undetected faults in software. However, complete testing is infeasible due to the vast input spaces involved and a lot of constraints on the test set. In order to overcome these limitations, criteria are used to provide a requirement for test data adequacy, and so give a measure for improving a test set. The compliance with a certain standard will generate adequate test sets and thus it is able to expose the faults existing in the program under test (PUT). Adequacy criteria are usually relevant to test coverage. This coverage is used to guide search process as well as assess the quality of the obtained test set. Coverage measures may be classified into two main categories: structure based and mutation-based coverage criteria.

The structure coverage specifies testing requirements in terms of the coverage of a particular set of elements in the program and includes control-flow and data-flow based criteria. However, these criteria do not focus on the cause of program's failures, while the mutation adequacy criterion does. In fact, mutation coverage provides a measure of test data effectiveness by showing that the tests can expose all possible simple faults of a program. De Millo *et al.*, [2] proposed mutation testing to provide a means of iteratively improving test data adequacy for PUT.

Based on the mutation adequacy criterion, fault induced variants of the program are executed with a test set to find out how many variants fail. The more that fail, the greater the adequacy of that test set. The tester's aim is to generate new test data that improves the adequacy of the existing tests. Mutation coverage subsumes many structural coverage criteria. It may detect faults that structural testing cannot discover [3].

There are some excellent surveys of test data generation techniques, such as [4], [5], [6], [7]. However, all of them used the structure coverage to generate test suites. This workconsiders the approaches using mutation analysis to identify a set of test data that maximizes the number of mutants killed. In other words, the paper targets to analyze and discuss an overview of test data generation methodswhich have been used in the mutation testing domain. Our research aims to answer two following questions:

+ **Q1**: What methodshave been used for test data generation using mutation analysis?

+ **Q2**: How can these test data generation methodsbe categorized?

The organization of the paper is as follows. Section 2 introduces the background of mutation-based test data generation. The particular methods will be presented in section 3. In section 4, some challenges and future trends in test data generation will be discussed. Section 5 is the conclusion and general assessment about methods.

2. BACKGROUND OF TEST DATA GENERATION BASED ON MUTATION ANALYSIS

Mutation analysis supports software testing by assessing the quality of test sets as well as creating the test data that can detect faults in program. In this testing process, faults are inserted into PUT. The resulting changed versions of the test program are called mutants. Each variant, or mutant, differs from the PUT by a small amount, such as a single lexeme, and is generated by a mutation operator. Mutation operators can be seen as representing common faults usually found in software. Thus, mutation operators are designed by basing on the most common faults committed by programmers when using a programming language. Figure 1 presents an example about the original program and its mutant program when applying the relational operator replacement operator.

Mutation-based test data generation proceeds by injecting syntactic mutations into a given program ***P***, generating from ***P*** a set ***M*** of mutants. The goal of this activity is to find a set of test-cases that kill each of the mutant $m \in \boldsymbol{M}$. This means that the test-case makes the mutant m produce outputs that differ from those of the original program ***P***. For the above example, test data (x = 5, y = 5) is able to detect the mutant because the output of the original program is *10* whereas the output of the mutant is *0*. However, test data (x = 5, y = 4) cannot detect this mutant due to the same output produced for both programs.

```
int foo (int x, int y)
{
int z = 0;
    if (x ≥ y)
        z = x + y;
    return z;
}
```

Original Program

```
int foo (int x, int y)
{
int z = 0;
    if (x > y)
        z = x + y;
    return z;
}
```

Mutant Program

Figure 1. A mutant example

In the case, there is no test data that are able to detect mutant because the mutant program is functionally equivalent to the original one and called equivalent mutant. A mutant is said to be equivalent if there is not such a test case able to differentiate between the output of the mutant and the output of the original program.

In mutation testing, each test data can detect some mutants in PUT. It is unlikely that one test will kill all mutants, requiring that, the result of test data generation based on mutation must incorporate a sufficient number of tests. It is expected that this test set can detect as many mutants as possible. The quality of test sets is evaluated through mutation score. It is the proportion of mutants killed out of all non-equivalent mutants that are generated from PUT by applying mutation operators. If this proportion is equal to 1, then test set is adequate and capable to detect the faults in PUT highly.

It is stated that mutation testing is an effective method to assess the quality of test data. However, most of studies in this field focus on reducing computational expense. There has been less work on applying mutation coverage in the context of test data generation. In this paper, we will analyze the test data generation techniques using mutation coverage to assess the quality of obtained test sets systematically. These methods are divided into some groups: random test date generation, constraint-based test data generation, enhanced control flow graph, dynamic symbolic execution, search-based test data generation and the hybrid techniques. The details of these methods will be presented in next section.

3. MUTATION-BASED TEST DATA GENERATION TECHNIQUES

After collecting the test data generation techniques based on mutation testing, we classify them into six categories: random test data generation (G1), constraint-based techniques (G2), enhanced control flow graph (G3), dynamic symbolic execution (G4), search-based test data generation (G5), andhybrid techniques (G6). These categories are derived based on the characteristics of each method. G1 generates test data randomly within the input space. G2 comprises the techniques using constraints on test data to kill mutants. G3 contains on the studies describing techniques relying on code coverage, analysis of the data flow and control flow to generate test data. G4 includes the approaches that incorporate execution of the program under test data with selection of paths from its control flow graph and solving the constraints on input data to kill mutant. G5 consists of techniques that apply the meta-heuristic algorithms to determine the best solution in a search space of a given problem. The approaches in G5 called the search-based techniques and the process of generating test data killing mutants is guided by fitness functions. G6 is the combination of G4 and G5 because it uses the meta-heuristic algorithms and DSE (dynamic symbolic execution) techniques to generate test sets.

3.1. Random Test Data Generation

Random Testing is one of the most fundamental and popular methods. It is simple in concept, easy to implement, and can be used on its own or as a component of many other testing methods. Random approaches generate test input vectors with elements randomly chosen from appropriate domains. Input vectors are generated until some identified criteria have been satisfied such as the maximal number of test cases. Random testing may be an effective means of gaining an adequate test set for simple programs. However, it may simply fail to generate appropriate data in any reasonable time-frame for complex software that has strict requirements of the data domain specification. For the vast data domain, it is difficult to randomly generate test data that can detect hard-to-kill mutants. For example in Figure 1, the mutant is only killed when *x* is equal to *y*. This is difficult to be achieved if random test generation in a vast data domain.

Some studies have been conducted to improve the limitation of random approaches. Adaptive random testing (ART) was proposed by Chen *et al.*[8] as an enhancement to pure random testing. Adaptive random testing seeks to distribute test cases more evenly within the input space. It is based on the intuition that for non-point types of failure patterns, an even spread of test cases is more likely to detect failures using fewer test cases than ordinary random testing. The fundamental idea behind ART is to reward diversity among sampled test cases. If a test case does not detect any failure, then in presence of contiguous faulty regions it would be better to sample test cases that are far from it. The distance among test cases in an input domain ***D*** depends on the type of PUT. In the case of numerical inputs, the Euclidean distance can be used. Chen's approach generates candidate inputs randomly, and at every step selects from them the one that is furthest away from the already used inputs. ART was initially introduced for numerical values and it calculates the distance between two such values using the Euclidean measure. The basic algorithm of ART is depicted in Figure 2. Results show that adaptive random testing does outperform ordinary random testing significantly (by up to as much as 50%) for the set of programs under study. However, the restrictions of this method are that it can generate only numeral test data, requires more memory and computation than purely random method.

```
Z = {}
Add random test case to Z and execute it
repeat until stopping criterion is satisfied
    sample set W of random test cases
    for eachw of these |W| test cases
        $w.minD = min(dist(w,z \in Z))$
    execute and add to Z the w with maximum minD
```

Figure 2. Algorithm of ART

3.2. Constraint Based Test Data Generation

Constraint Based Testing (CBT) was the first test case generation method used for mutation testing. It was proposed by DeMillo and Offutt [9], based on the idea of control flow analysis, symbolic execution, constraint on mutants and program execution in order to generate test data automatically. According to the authors, a mutant should be killed if it satisfies three conditions known as reachability, necessity and sufficiency. The reachability condition expresses that the mutated statement must be reached by test data. If test suites cannot execute the mutated statement, then tests have no chance of killing the introduced mutant. The necessity condition requires the execution of the mutated statement to result in an error in the program's state. This means that the execution outcome of the original and the mutated statements must be different. Otherwise, the syntactical equality of the rest of the original and mutant program versions will never create the different computations and will thus never result in visible output differences. The sufficiency condition states that the incorrect state must propagate to the last statement and create at least one different output between original and mutated program. The CBT method has been implemented in a tool called *Godzilla* in order to test the Fortran programs and has been integrated with the *Mothra* [10] mutation testing toolset. CBT has empirically been shown to be an effective approach; however, it has certain drawbacks with respect to the symbolic evaluation when solving the handling of arrays, loops, non-linear expressions and the path explosion problem.

To overcome these difficulties, Offut *et al.* proposed Dynamic Domain Reduction (DDR) method which was presented in [11]. This approach still retains reachability, necessity and sufficiency conditions but improves how to handle these conditions. This method generates tests by basing on the reduction of the input spaces of the variables involved. The DDR approach takes in account the test data generation problem as a dynamic path based problem. It uses the program control flow graph and bases on a search heuristic method over the input variables domain to generate test data. When it reaches a branch point in the path, the variables used within that branch predicate have their domains reduced in accordance with the execution of the desired branch path. If there is a choice of how to reduce the domain, a search process is made from the subsequent path in order not to make an inappropriate selection and restrict subsequent branch outputs. Upon reaching the target node, the remaining values in each variable's domain represent the values that will cause execution of the desired path, *i.e.* for mutation testing where the mutated statement is the target node; the remaining domains represent test values that satisfy the reachability constraint. In some cases though, a domain will be empty indicating that the procedure failed either because the path is infeasible or it was too difficult to find value to execute it. Offutt *et al.* [11] concluded that DDR "*is less likely to fail to find a test case when a test case exists, and that implementations can be more efficient*". Compared to the CBT approach, this procedure would seem more favorable.

3.3. Enhanced Control Flow Graph

This approach transforms the problem of generating test data killing mutants to a covering branches alternative. Thus, effective heuristics applied for branch testing can be extended to mutants too. The most popular methods for branch testing are those that select specific path sets to generate the sought test data. As with all path generation methods their major deficiency is the generation of infeasible paths, this problem is also inherited when employing path generation for performing mutation testing too. In [12], Papadakis and Malevris proposed an effective path selection strategy in order to achieve adequate coverage. The presented strategy targets on generating mutation adequate test sets in an effective way. This is achieved by a strategy that reduces the effects of infeasible paths and equivalent mutants. The authors built an enhanced graph by adding a special type of vertex for each mutant. Every mutant related vertex is connected with its original corresponding node and represents a special mutant related necessity constraint [12]. Treating each mutant as a branch helps focus on specific mutants and select candidate paths in order to generate data aiming at killing the specified mutants. Figure 3 demonstrates the construction of the enhanced control flow graph (the original control flow graph on the left and the enhanced one including three mutants on the right).

The main idea of the proposed approach is to reduce the problem of fulfillment of the necessity conditions of mutants to that of covering branches. This is done by transforming the required necessity constraints into branch predicates and representing them in the program control flow graph. The innovation of this approach is the use of an enhanced control flow graph and its respective branch predicates in order to proceed with the symbolic evaluation and test data generation phase. The benefit of the approach is the incorporation of mutation based criteria with existing path based test data generation methods inheriting all their merits. The details of this method are presented in [12].

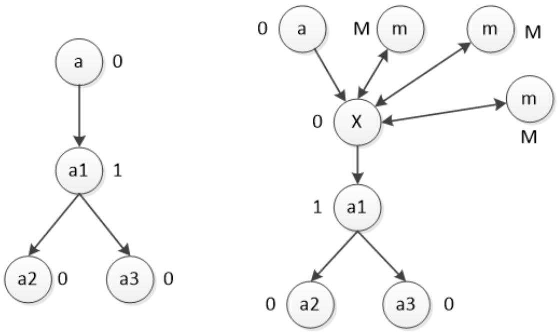

Figure 3.The Enhanced Control Flow Graph [12]

Enhanced control flow graph can be considered as a supplement to the theory of test data generation methods based on mutation coverage. It transforms the test data generation problem into path selection problem in graph. However, for a large program and there are many generated mutants, this method will face up to the limitation that are the vast number of nodes in the graph and the path explosion problem. These reduce the effectiveness of the method. These restrictions need to be further figured out to improve the effectiveness and efficiency of the solution.

3.4. Dynamic Symbolic Execution

Symbolic execution [13] is an advanced program analysis technique in which inputs and other variables assume symbolic rather than particular values and output is a mathematical expression of these symbols. The symbolic evaluation process of a program consists of assigning symbolic values to variables in order to deduce an abstract algebraic representation of the program’s computations and representation. This technique is based on the selection of paths from its control flow graph and the computation of symbolic states. The symbolic state of a path forms a mapping from input variables to symbolic values and a set of constraints called path conditions over those symbolic values. Path conditions represent a set of constraints called symbolic expressions that form the computations performed over the selected path. Solving the path conditions results in test data which if input to the selected path, this will be executed. If the path condition has no solution the path is infeasible. Despite of the capabilities of symbolic execution, there are several problems associated with its application: path selection and the evaluation of loops, module calls, arrays and constraint solving. The problem of path selection and the evaluation of loops is that it will cause path explosion. When the size of program increases, the size of constraint expressions obtained may become very large. These are the limitations of this approach and it is difficult to apply it for a large program.

Dynamic Symbolic Execution (DSE) is a more recent innovation that overcomes many of the limitations of traditional symbolic execution. This method allows automatic generation of test inputs that achieve high code coverage. DSE executes the program under test for some given test inputs (ones generated randomly), and at the same time performs symbolic execution in parallel to collect symbolic constraints obtained from predicates in branch statements along the execution traces. During test generation, DSE is performed iteratively on the PUT to increase code coverage. Initially, DSE randomly chooses one test input from the input domain. Next, in the each iteration, after running each test input, DSE collects the path condition of the execution trace, and uses a search strategy to flip a branching node in the path. Flipping a branching node in a path constructs a new path that shares the prefix to the node with the old path, but then deviates and takes a different path. Whether such a flipped path is feasible is checked by building a constraint system. If a constraint solver can determine that the constraint system is satisfied within the available resources, DSE generates a new test input that will execute along the flipped path and achieve additional code coverage. In this way, DSE is able to generate a set of test inputs that achieve high code coverage. Using DSE, non-linear path constraints are simplified by the instantiation of concrete runtime values, harvested from program execution.

The authors in [14] proposed the application of DSE for generating test data based on mutation testing. After generating mutants for the program under test, they will generate corresponding weak-mutant-killing constraints for each mutant. For example, *op1>op2* has a corresponding mutated expression *op1≥op2* then condition to kill this mutant is ((*op1>op2*) && !(*op1≥op2*)) || (!(*op1>op2*) && (*op1≥op2*)). Next, all the generated constraints are inserted into proper positions of the original program to form a meta-program. After transforming the original program under test into a meta-program containing all mutant-killing constraints, the DSE engine is used to generate test inputs for the meta-program [10]. The authors built the *PexMutator* tool in order to support for generating test data for C# programs. Their preliminary experimental study shows that *PexMutator* is able to strongly kill more than 80% of all the mutants for the studied subjects. However, this method faces up to the problem is to introduce as many constraints as mutants of the program under test into the corresponding meta-program, making it expensive for the DSE engine to generate test inputs for a large number of branches. One of the ways to overcome this problem is investigate techniques that generate constraints that enable to weakly kill multiple mutants, thus reducing the number of total generated constraints while keeping the same effectiveness.

Papadakis *et al.*, [15] used mutant schemata in conjunction with DSE to generate test data based on mutation analysis to be satisfied three conditions named reachability, necessity and sufficiency. An automated framework for testing *Java* programs according to mutation was also proposed by authors. First of all, the schematic meta-program versions of the program under test are generated. Next, the mutant schemata generation component produces a static structure of the call and control flow graphs and a list of the introduced mutants with their respective program statements. These two artifacts are then passed to the test generation module. This module iteratively selects a mutant as a target, performs DSE on the schematic meta-program and produces some test cases. These test cases are then passed to the test executor which determines their execution path, the mutants that the test cases can infect and the mutants that are killed. After that the process continues with the next iteration. Finally, after reaching a predefined number of iterations or time limit the process ends and reports the produced test cases and the achieved mutation score. The proposed approach was experimented on 5 programs with up to 500 lines of code and the obtained average mutation score is 63%. This approach faces up to some limitations when applying for the large PUT. Because many mutants are generated, the computational cost is expensive when specifying the feasible path in PUT as well as solving the constraint conditions. An improved method to overcome these restrictions is to save the constraints checked to avoid solving the same constraints many times. In order to solve the problem of path explosion and equivalent mutants, it is possible to use the appropriate fitness evaluations in conjunction with the effective heuristics.

3.5. Search-Based Test Data Generation

Static approaches generating mutation-based test data rely on solving the entire set of constraints, whereas dynamic test data generation methods detect individual faults based only on actual executions of program or design. Search based test data generation is one of the dynamic approaches. It models the testing task as a search problem that is guided by a fitness function. Then, problem is solved by applying meta-heuristic techniques like genetic algorithms, simulated annealing, clonal selection algorithm or Tabu search to optimize this function. However, whatever may be the search techniques employed fitness function or cost function plays a major role to guide and seek input test data that achieve the highest mutation score.

Bottaci [16] was the first to suggest using search-based software engineering to kill mutants. Bottaci proposed a fitness function for genetic algorithms based on the constraints defined by DeMillo and Offutt in order to generate mutation-based test data. However, this proposed approach remained unimplemented and unevaluated until the subsequent for work of Ayari *et al.,* [17]. They used Bottaci's proposal to define and implement a fitness function that measures how close a test case is to kill a mutant. They used this fitness in conjunction with the ant colony optimization (ACO) algorithm for automatic test data generation for *Java* programs. Two *Java* programs were used to assess the effectiveness of the ant colony algorithm. The obtained results indicate that ACO approach performed significantly better than genetic algorithm and Hill Climbing in terms of attained mutation score as well as computational cost. However, the fitness function only implements the reachability component. The necessity and sufficiency components are still not solved.

Another meta-heuristic algorithm is usually used to support for test data generation that is genetic algorithm (GA). This algorithm is probabilistic search algorithm inspired from biology. The GA is an iterative process to find the best solution in the population of solutions through many generations. Algorithm is started with a set of solutions (test data) is randomly generated called population. Then, the individuals in the population are evaluated based on their fitness function. Better individuals are having more chances to be selected as parents in order to reproduce better offspring by using crossover and mutation mechanisms. The algorithm continues iterating until the pre-defined stopping condition is met such as the maximal number of generations, all mutants are killed or pre-defined mutation score is reached. After a number of generations, the algorithm will return the best individual of population which kills the most mutants. Baudry *et al.,* [18] applied the GA to improve the randomly generated test sets according to mutation for C# programs. They used a population of 12 individuals consisting of 4 tests each. Using a 2% mutation rate over 200 iterations, the mutation score reaches a peak of 80%, with an average of 65-70%. In [19], we proposed to apply the GA in order to generate test data for *Simulink* models. We also conducted the experiments and evaluated the results on 5*Simulink* models and obtained the average mutation score of 85.7% within 20 genetic generations. Both studies, each individual is a set of test data and the fitness calculations are performed by basing on the achieved mutation score. The limitations of this representation are to use much memory and increase the execution time. Moreover, the fitness function does not guide the search method by quantitatively measuring the closeness of killing specific mutants in the each genetic generation. Fraser *et al.,* [20] proposed an effective fitness function for the GA based on branch distance and approach level. The approach level describes how far a test case is from the target in the control flow graph when it has deviated from the anticipated course. This is usually measured as the number of unsatisfied control dependencies between the point of deviation and the target, and the value of approach level will be 0 if all control dependent branches are reached [20]. The branch distance estimates how far the branch at which execution diverged from reaching the mutation is from evaluating to the necessary outcome. The authors applied the proposed approach to a set of 10 open source *Java* libraries and obtained the average mutation score of 72%.

In [18], Baudry also proposed a new algorithm to overcome the limitations of the GA in order to enhance the quality of test data. That was Bacteriological algorithm (BA). In this algorithm, an individual is an atomic unit - it cannot be divided. BA maintains a memory set consisting of the best individual(s) from each generation. As an atomic unit, individuals cannot be mated and so variation stems purely from the reproduction and mutation process. A memory set is also maintained, where new individuals are added if their fitness exceeds some threshold. Individual fitness is based on a narrowing search space, with calculations based on what is left to optimize. For example, applied to mutation testing, fitness is calculated by basing on how many mutants a test kills that the memory set does not. If this fitness exceeds a threshold, that individual is reproduced and placed into the memory. Results reported that using a BA generates a memory set with a mutation score of 96% in just 30 generations. An average mutation score of 96% was obtained by executing only 46375 mutants, compared with the GA attaining an average mutation score of 85% in 480000 mutant executions. For BA, the initial population consisted of 30 tests and a memory threshold of 20% (*i.e.* a test had to kill over 20% of remaining mutants to be added to the memory set). Comparisons were also made with a GA approach; however it is difficult to ascertain the fairness of the experiments. For example, the GA consisted of 12 members of 4 tests each, equating to 48 tests, where as the BA used between 3 and 10 tests for its initial main population.

Another evolutionary approach that is very promising to support automatic test data generation is clonal selection algorithm (CSA) – a subfield in artificial immune system. When applying this method in the context of mutation testing, an antigen is an introduced mutant. An antibody is a test data which is generated to kill mutants. The affinity of an antibody (test data) is measured by mutation score. Antibody evolution occurs through the process of clonal selection, guided by the affinity values. The high affinity antibodies generate more clones than low affinity ones, but mutate less. This process aims at refining antibodies to kill as many mutants as possible. The best antibodies will be added to a memory set for saving and return to testers when testing process finishes. May [21]adopted this method to evolve test data for *Fortran* programs basing on mutation testing. In [22], we also proposed to apply CSA with some modifications to generate test data for Simulink models.

Tao Xie *et al.,* [23] defined a cost function from fault simulation traces to real values in HDL programs. This cost function for directing search heuristics has been defined on the test input space. By mapping a pair of fault simulation traces onto the Control and Data Flow Graph (CDFG) structure, the authors were able to analyze quantitatively how far a fault effect has been propagated through both the control and data flows. The macro propagation distance and the local propagation cost together form a complete solution to the necessity and sufficiency sub-problems of fault detection. One major limitation of the work is to still lack an automation tool for the construction of CDFG as well as test data generation.

Another work was proposed by Zhan and Clark [24] in which they generated test data in the context of mutation testing for *Matlab/Simulink* models by using simulated annealing and random testing. The method randomly generates a large set of test-data, detects the mutant-killing ability, and then minimizes the test set whilst retaining its overall mutant-killing ability. For mutants that cannot be killed by the random test set, the method provides an effective means of automatically generating individual test data for fulfilling individual mutant-killing aims. In this way, a targeted test data generation is utilized to complement the random test generation in order to achieve mutation adequacy.

3.6. Hybrid Techniques

As presented above, meta-heuristic search and dynamic symbolic execution technique have emerged as two successful approaches to automatically generate test data that achieve high mutation coverage. Both approaches have their advantages, but they also have specific drawback. Search-based testing scales well and can handle any code and test criterion but it gets stuck in local optima and degrades when the search landscape offers no guidance. DSE based testing exploits the efficiency of modern constraint solvers which are not dependent on search heuristics, but there are limits to both scalability and the types of constraints that can be handled. In [25], Harman *et al.* introduced a hybrid DSE and search-based software testing approach to generate strongly adequate test data to kill first and higher order mutants. The authors implemented their approach in a tool called SHOM. The proposed approach was evaluated on 17 subject programs, including 7 real world programs (four from two different closed source industrial systems and three for which source code is publicly available). They reported the results of an empirical evaluation of SHOM's efficiency and effectiveness for strong first order mutation adequacy. The results show that SHOM can kill up to 38% of the first order mutants left unkilled using reachability and infection, which in turn kills up to 36% of the mutants left unkilled using reachability alone. They also reported the results of a further empirical study of SHOM's efficiency and effectiveness for strong second order mutation adequacy. The results showed that SHOM can kill up to 48% of the second order mutants left unkilled using reachability and infection, which in turn kills up to 41% of the mutants left unkilled using reachability alone.

Thispresented several techniques used to generate test data based on mutation testing. All approaches are promising to formulate high quality test suites. Some well-known techniques and related papersare briefly presented in Table 1.

Table 1. The typical test data generation approaches

Author [Ref]	Year	Technique	Subjects Studied	Subject Language	Average mutation score
DeMillo and Offut [9]	1991	Constraint-based test data generation	5 small programs	Fortran	98%
Offut *et al.* [11]	1999	Dynamic Domain Reduction	12 small programs	Fortran	Not given
Chen *et al.* [8]	2004	Adaptive Random Testing	12 small programs	C++	Not given
Baudry *et al.* [18]	2005	Search-based test data generation using genetic algorithm	An example in Eiffel Library	C#	85%
Baudry *et al.* [18]	2005	Search-based test data generation using Bacteriological algorithm	An example in Eiffel Library	C#	96%
Ayari *et al.* [17]	2007	Search-based test data generation using ant colony algorithm	2 small programs	Java	88%
Papadakis *et al.* [12]	2009	Enhanced Control Flow Graph	8 small programs	Java	90.2%
Zhang *et al.* [14]	2010	Dynamic Symbolic Execution	5 small programs	C#	90%
Papadakis *et al.* [15]	2010	Dynamic Symbolic Execution	5 tiny examples + plus 3 small Siemens suite examples	C	63%
Frazer *et al.* [20]	2010	Search-based test data generation using genetic algorithm	2 non-trivial examples: Commons Math &JodaTime	Java	72%
Harman *et al.* [25]	2011	Combine Dynamic Symbolic Execution and Search-based testing	7 real world programs and 10 small programs	C	71%
Hanh *et al.* [19]	2014	Search-based test data generation using genetic algorithm	5 Simulink models	Simulink	85.7%
Hanh *et al.* [22]	2014	Search-based test data generation using the clonal selection algorithm	2 Simulink models	Simulink	88.1%

4. FUTURE TRENDS

It would not be possible to conclude this survey without spending a little time discussing the possible future trends of test data generation based on mutation analysis. It can see that there are four important directions for research: the scalability of the current techniques for large-scale software project, the study for high quality higher order mutants to reduce the size of test set, elimination of equivalent mutants before generating test data, and automatic adjustment of configuration parameters for the meta-heuristic algorithms in search-based test data generation.Solving these concerns will improve significantly the effectiveness of techniques and the quality of test set.

The approaches presented in this paper were experimented on small programs. It is expected to conduct studies about the scalability of the current techniques for large-scale industry software. Recent work has tended to focus on more elaborate forms of mutation than on the relatively simply faults which have been previously considered. There is an interest in the semantic effects of mutation, rather than the syntactic achievement of a mutation [26]. It is desired to generate higher order mutation and resemble real faults. In the future, therefore, studies will focus on generating test data to kill higher order mutation as well as reduce the size of test suites. The more mutants killed by a test set, the better the measured adequacy of the test set. This will enhance the quality of test data and decrease time spent on software testing process.

One of the most challenges in test data generation using mutation testing is the detection of equivalent mutants. It wastes time to generate test data but we never find out test that is able to kill these mutants. In the future, mutation testing methods should seek to avoid initial creation or elimination of equivalent mutants before generating tests. This work will reduce the time wasted on generating test suites and improve the mutation coverage of test data.

It can also be seen that search-based test data generation techniques depend on setting parameters. These parameters are drawn from experiments and fixed in the searching process. They affect to the effectiveness of meta-heuristic algorithms and obtained test suites. It is expected that, in the future, there will be more work concentrating on self-tuningof parameters to seek high quality test data. The information from previous generations can be used to choose the appropriate parameters for the current generation

5. CONCLUSION

This paper presents a comprehensive survey of the most prominent techniques in automatic test data generation based on mutation. These techniques include random test data generation, constraint-based test data generation, enhanced control flow graph, dynamic symbolic execution, search-based techniques and the hybrid approaches.

The results of the survey indicate that there are quite a lot researches in the field of generating test data based on mutation testing, and most of them are positive. In the work that we deployed about the use of meta-heuristic algorithms to search for test data that is able to kill many mutants, the initial results are promising.

For the test data generation techniques based on constraint and dynamic symbolic execution, the obtained tests can kill mutants with a high proportion. However, they can encounter the path explosion problem when handling large programs and designs. For search-based methods, test data are optimized through generations rely on evaluating the cost function. In the study of the other authors as well as our work, the cost function has not guided for test data generation towards the higher mutation score yet. Thus, in the future, the cost function may be improved by guiding the process of test data generation that orientates the specific mutants in each genetic generation.

REFERENCES

[1] B. Beizer, *Software Testing Techniques*, 2nd ed.: Thomason Computer Press, 1990.

[2] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for Practicing for Programmer", *IEEE computer*, no. 11, pp. 34-41, 1978.

[3] A.J. Offutt and J.M. Voas, "Subsumption of Condition Coverage Techniques by Mutation Testing", *George Mason University and Reliable Software Technologies Corp.*, Technical Report ISSE-TR-96-01, 1996.

[4] Saswat Anand *et al.*, "An Orchestrated Survey on Automated Software Test Case Generation", *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978-2001, 2013.

[5] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation", in *IEEE Transactions on Software Engineering*, 2010, pp. 742 - 762.

[6] M. Harman, "Automated test data generation using search based software engineering", in *2nd Workshop on Automation of Software Test (AST 07) at the 29th International Conference on Software Engineering*, USA, 2007, p. 2.

[7] P. McMinn, "Search-based software test data generation: A survey", *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105-156, 2004.

[8] T.Y. Chen, H. Leung, and I.K. Mark, "Adaptive Random Testing, Advances in Computer Science - ASIAN 2004: Higher-Level Decision Making," in *9th Asian Computing Science Conference*, pp. 320-329, 2004.

[9] R.A. Demillo and A.J. Offutt, "Constraint-Based Automatic Test Data Generation", *IEEE Transaction Software Engineering*, vol. 17, pp. 900-910, 1991.

[10] R.A. DeMillo, D.S. Guindi, W.M. McCracken, A.J. Offutt and K.N. King, "An extended overview of the Mothra software testing environment", in *Proceedings of Symposium on Software Testing, Analysis and Verification*, 1988, pp. 142-151.

[11] A.J. Offutt, Z. Jin and J. Pan, "The dynamic domain reduction approach to test data generation", *Software: Practice and Experience*, vol. 29, no. 2, pp. 167-193, 1999.

[12] M. Papadakis and N. Malevris, "An Effective Path Selection Strategy for Mutation Testing", in *Proceedings of Asia Pacific Software Engineering Conference*, 2009, pp. 422-429.

[13] J.C. King, "Symbolic execution and program testing", *Communications of the ACM*, vol. 19, no. 7, pp. 38-94, 1976.

[14] Lingming Zhang, Tao Xie, Lu Zhang, N. Tillmann, J. de Halleux, Hong Mei, "Test generation via Dynamic Symbolic Execution for mutation testing", in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, Timisoara, 2010, pp. 1-10.

[15] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution", in *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10)*, USA, 2010, pp. 121-130.

[16] Leonardo Bottaci, "A Genetic Algorithm Fitness Function for Mutation Testing", in *Proceedings of the Software Engineering using Metaheuristic inovative Algorithms workshop*, April 2001, pp. 3-7.

[17] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony", in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, London, England, 2007, pp. 1074-1081.

[18] B. Baudry, F. Fleurey, J.M. Jézéquel, and Y.L. Traon, "Genes and Bacteria for Automatic Test Cases Optimization in the NET Environment", in *Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002, pp. 195-206.

[19] L.T.M. Hanh, K.T. Tung and N.T. Binh, "Mutation-based Test Data Generation for Simulink Models using Genetic Algorithm and Simulated Annealing", *International Journal of Computer and Information Technology*, vol. 3, no. 4, pp. 763-771, July 2014.

[20] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles", in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*, Trento, Italia, 2011, pp. 147-158.

[21] P.S. May, "Test Data Generation: Two Evolutionary Approaches to Mutation Testing", *The University of Kent*, PhD Thesis, 2007.

[22] Le Thi My Hanh, Nguyen Thanh Binh and Khuat Thanh Tung, "A Novel Test Data Generation Approach Based Upon Mutation Testing by Using Artificial Immune System for Simulink models", in *The Sixth International Conference on Knowledge and Systems Engineering*, Hanoi, Vietnam, 2014.

[23] Tao Xie, W. Mueller and F. Letombe, "HDL-Mutation Based Simulation Data Generation by Propagation Guided Search", in *14th Euromicro Conference on Digital System Design*, Oulu, 2011, pp. 608-615.

[24] Y. Zhan and J.A. Clark, "Search-based Mutation Testing for Simulink Models", *ACM Digital Library*, pp. 1061-1068, 2005.

[25] Mark Harman, Yue Jia and William B. Langdon, "Strong Higher Order Mutation-Based Test Data Generation", in *Proceedings of the 19th ACM SIGSOFT symposium*, New York, USA, 2011, pp. 212-222.

[26] Yue Jia and Mark Harman, "An Analysis and Survey of the Development of Mutation Testing", *Crest Centre, King's college London, Technical Report TR-09-06*, 2011.

BIOGRAPHIES OF AUTHORS

Le Thi My Hanh gained M. Sc from the University of Danang in Computer Science in 2004. She is currently a PhD student of the Information Technology Faculty, University of Science and Technology, Danang, Vietnam. Her research interest is about software testing and more generally application of heuristic techniques to problems in software engineering.

Nguyen Thanh Binh graduated from The University of Danang, University of Science and Technology in 1997, he got a PhD degree in Computer Science from Grenoble Institute of Technology (France) in 2004. He is currently associate professor of the Information Technology Faculty, The University of Danang, University of Science and Technology (Vietnam). He is dean of Information Technology Faculty at The University of Danang, University of Science and Technology since 2010.
He directs a research team since 2009. His research interests include software testability, software testing and software quality.

Khuat Thanh Tung completed the B.E. degree in Software Engineering from University of Science and Technology, Danang, Vietnam, in 2014. Currently, he is participating in the research team at DATIC Laboratory, The University of Danang, University of Science and Technology, Vietnam. His research interests include software engineering, software testing, and AI in software engineering.