Automatic Software Test Case Generation: An Analytical Classification Framework

Mohammad Reza Keyvanpour, Hajar Homayouni and Hossein Shirazee

1Computer engineering department, Alzahra University, Tehran, Iran
2Computer engineering department, Islamic Azad University, Qazvin Branch, Qazvin, Iran
keyvanpour@alzahra.ac.ir

Abstract
A challenging part of software testing entails the generation of test cases. A good test case should have the quality to cover more features of test objective. The techniques for automatic generation of test cases try to efficiently find a small set of cases that allow an adequacy criterion to be fulfilled, thus reducing the cost of software testing and resulting in more efficient testing of software products. In this paper we introduce an all-around classification framework for automatic test case generation approaches in terms of test type and Algorithm, and represent some test case evaluation approaches. Finally we illustrate a comparison between different existing techniques.

Keywords: software testing; test case generation; Automation, classification

1. Introduction
Software testing is the process of executing a program in order to find faults. Testing is a very important, though expensive phase in software development and maintenance; it has been estimated that software testing entails between 30 percent and 50 percent of software development [1]. A challenging part of this phase entails the generation of test cases. This generation is crucial to the success of the test because it is impossible to achieve a fully tested program given that the number of test cases needed for fully testing a software program is infinite, and a suitable design of test cases will be able to detect a great number of faults. For these reasons, the techniques for automatic generation of test cases try to efficiently find a small set of cases that allow an adequacy criterion to be fulfilled, thus reducing the cost of software testing and resulting in more efficient testing of software products. A test case is a set of tests performed in a sequence and related to a test objective, which will produce a number of tests comprising specific input values, observed output, expected output, and any other information needed for the test to run, such as environment prerequisites [2]. A good test case should have the quality to cover more features of test objective. The techniques for the automatic generation of test cases try to efficiently find a small set of cases that allow a given adequacy criterion to be fulfilled, thus contributing to a reduction in the cost of software testing [1].

There have been few efforts on representing an all-around classification, which covers all existing automatic test case generation approaches. In this paper we introduce a general
classification for automatic test case generation approaches in terms of test type and the Algorithm, and will define a comparison between these approaches.

The rest of the paper is organized as follows. Section 2 is an overview on classifications of automatic test case generation approaches. In Section 3 we read about the requirements, applications, and challenges of automating test case generation, Section 4 and 5 are classifications of techniques, section 6 is about test case evaluation approaches, and the last sections are conclusions and future work.

2. Overview

There has been a significant amount of work in automatic test case generation that attempts to increase the amount of observed behavior. Despite of these wide researches, there have been few efforts on representing an all-around classification, which covers all existing automatic test case generation approaches. A general classification for these techniques is presented in [3, 4, 5]. In [6] a classification framework for automatic test case generation methods is presented which is based on software development phase in which testing is applied. In [7] a classification of search-based automatic test case generation approaches is represented.

In this study, we introduce a comprehensive classification and evaluation of automatic test case generation techniques, which tries to cover all existing approaches.

3. Automatic Test Case Generation

Generally speaking there are different reasons to automate test case generation task in software testing. Some of the most important reasons are as follows.

Reducing the cost of software testing: During testing phase the cost can increase more than the expected value due to inappropriate test cases. These inappropriate test cases cause wastage of organizational resources as well as time. There is a need to minimize the cost for getting an acceptable product [8].

Reducing human errors: In order to find out how a test case is valid there is no definite mechanism. It basically depends on the testers understanding of the requirement. In this process there are lots of human errors and tester basic skill level taken into consideration. This leads to the inclusion of bugs in the system after testing. To overcome this problem, automatic test case generation phase should be considered [9].

Increasing software products quality: It is generally agreed that manual testing is becoming a bottleneck and is a frequent cause of project delays especially for large programs. Therefore, automatic test case design has become important to ensure the quality of present day large software products [10].

Reducing number of test cases: Generation efficient test cases are the essential passport for simplifying the test work and improving the test efficiency. The test work is inefficient because of the great number of the initial test cases, so some Automation algorithms are needed to optimize the test cases [11].
Covering all system requirements: Automating the test case generation process provides a means to ensure that test cases have been derived in a consistent and objective manner and that all system requirements have been covered [12].

The applications of automatic test case generation are as wide as software applications. In general, software systems can be classified to three main classes. Figure 1 shows this general classification and some examples for each group. Automating test case generation can be done for testing any of these groups, depending on their complexity and size.

![Software Classification](image)

**Figure 1. Software Classification**

There are some challenges in automating test case generation, to be answered:

- What types of methods are there available at our disposal?
- What are the characteristics and underpinnings of different algorithms?
- How to determine which method is appropriate for what type of test case generation task?
- Which methods can be used to make headway in what aspect of the essential difficulties in test case generation?
- When to apply test case generation in software life cycle?
- What should be the inputs for a test case generation system?

4. Proposed Classification Framework in Terms of Test Type

There are different types of software testing approaches. Respecting to the fact that test case generation is an important phase in software testing, test case generation approaches directly depends on the type of software testing. Table 1 represents a classification for different automatic test case generation approaches in terms of test type.
Table 1. Classification Framework in Terms of Test Type

<table>
<thead>
<tr>
<th>Approach</th>
<th>Main Idea</th>
<th>Pros</th>
<th>Cons</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural-Based</td>
<td>In this approach test cases are generated using system source code or control flow graph of the program. The goal of structural testing is to cover a test adequate criteria</td>
<td>1. Comprehensible</td>
<td>1. Generating test cases is usually done very late in the software development cycle. 2. Errors discovered are very difficult and costly to correct, since changes affect large portions of the design, implementation, and testing procedure [13].</td>
<td>Automatic test case generation for functions in procedural programs (such as triangle classifier function [14], bubble sort function [14], etc), and automatic test case generation for some important classes in object oriented programs (such as container classes [4]).</td>
</tr>
<tr>
<td>Functional-Based</td>
<td>In this approach test cases are generated using system specifications. The goal of functional testing is to test the functionality of the software under test.</td>
<td>1. Testing can begin much earlier in the software design process. 2. Inconsistencies and ambiguities in specifications are found earlier by software testers. 3. Errors discovered are simply and not much costly to correct.</td>
<td>1. Specifications are complex and, therefore, difficult to understand. 2. Implementation should exactly correspond with specification. 3. There is a need for formal specification of the system, which is hard to represent an exact identification of system specification in real applications.</td>
<td>Automatic test case generation for (web-desktop) applications (such as banking system [15], employment system [16], etc).</td>
</tr>
<tr>
<td>Gray-Box-Based</td>
<td>In this approach both structural and functional information are used for generating test cases.</td>
<td>1. Takes the benefits of both previous approaches.</td>
<td>1. high complexity:</td>
<td>Automatic test case generation for detecting runtime errors called exceptional conditions in real time applications [7].</td>
</tr>
<tr>
<td>Non-Functional-Based</td>
<td>Non functional testing is testing of “how” the system works. Non functional testing may be performed at all test levels. The term non-functional testing describes the tests required to measure characteristics of systems and software that can be quantified on a varying scale.</td>
<td>1. Takes in to account other behaviors despite of logical behaviors of system.</td>
<td>1. is very complicated because it depends both on software and hardware features.</td>
<td>Automatic test case generation for execution testing of real time systems [7].</td>
</tr>
</tbody>
</table>

5. Proposed Classification Framework in Terms of Algorithm

Table 2 presents a classification framework for automatic test case generation approaches based on the kind of algorithm they used.
To illustrate, some examples of using mentioned algorithms are expressed as follows.

5.1. Random-based Automatic Test Case Generation

As it is mentioned before, this algorithm is mostly used to analyze the efficiency of other algorithms in comparison with it.

5.2. Search-based Automatic Test Case Generation

There are a variety of search algorithms to use for the purpose of test case generation. Figure 2 shows these search-based methods.
5.2.1. Automatic Test Case Generation using Hill Climbing: Hill climbing (HC) is a local search algorithm. It needs that a neighborhood $N$ of the current sequence $S_i$ is defined. The search will move to a new solution $S_j \in N$ only if $S_j$ is better. If there are no better solutions in $N$, a local optimum has been reached. In such a case, a restart of the algorithm from a new random solution can be done. Andrea Arcuri and Xin Yao [4] automatically generated test cases for structural testing of Java containers using Hill Climbing. A solution to the problem is a sequence $S_i$ of Function Calls (FCs) on an instance of the Container under Test (CuT). A Function Call (FC) can be seen as a triple:  

$$< \text{object reference}; \text{function name}; \text{input list}>.$$  

They used three types of operations on $S_i$ for generating $N$: Removal of a single FC from $S_i$, Insertion of a new FC in $S_i$, and Modification of the parameters of a FC. In general HC is a simple algorithm but may lead to a local optimum.

5.2.2. Automatic Test Case Generation using Genetic Algorithm: Genetic algorithms [4] represent a class of adaptive search techniques and procedures based on the process of natural genetics and Darwin's principle of the survival of the fittest. Genetic algorithm searching mechanism starts with a set of solution called a population. One solution in the population is called a chromosome. The search proceeds for a number of generations, for each generation the fitter solutions (based on the fitness function) will be selected to form a new population. During the cycle, there are three main operators namely reproduction, crossover and mutation. The cycle will repeat for a number of generations until certain termination criteria are met.

In [14] a method of generating test cases for structural testing based on genetic algorithms to cover multiple target paths in one run is presented. First, the problem of generating test cases is formulated as a multi-objective optimization problem in which the number of objectives decreases along with generation of test cases. Then, test cases are generated using genetic algorithms incorporating with domain knowledge.

In [19] a Constraint-based Genetic Algorithm technique is used to generate optimized test cases from UML Activity diagram and Collaboration diagrams. [11] Is about test cases generated for software security based on GA. The method selects the most favorite test cases in order to discover the vulnerabilities in the software. This method can reduce the test time and improve the test efficiency to a certain extent. The probability of selection $P_i$ is decided according to the fitness value $f_i$ of the test case $d_i$ in this algorithm.
\[ P_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \]  

(1)

Where \( n \) is the number of the test cases in the group and,

\[ \sum_{i=1}^{n} P_i = 1 \]  

(2)

In [18] some new categories of genetic codes are applied in some problem optimizations for the generation of reliable software test cases based on the specification of the software. Figure 3 shows the GA algorithm used.

**Figure 3. GA Algorithm**

The simulation shows that the proposed GAs with the specification can find solutions with better quality in shorter time. All in all, genetic algorithm is the most favorite algorithm used in test case generation task till now, as a result of leading to acceptable test suits.

5.2.3. **Automatic Test Case Generation using Genetic Programming:** genetic programming (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. It is a specialization of genetic algorithms (GA) where each individual is a computer program. It is a machine learning technique used to optimize a population of computer programs
according to a fitness landscape determined by a program's ability to perform a given computational task.

[20] Presents a search-based approach to automatically generating test cases for object-oriented software. It relies on a tree-based representation of method call sequences. Strongly-typed genetic programming is employed to generate method call trees which respect the call dependences among the methods. A new kind of distance-based fitness function applied that accounts for runtime exceptions. The approach proved successful and produced test cases leading to full branch coverage for four test objects completely automatically.

5.2.4. Automatic Test Case Generation using Memetic Algorithm: The Memetic algorithms (MAs) are metaheuristics that use both global and local search (e.g., a GA with a HC). It is inspired by the cultural evolution. A meme is a unit of imitation in cultural transmission. The idea is to mimic the process of the evolution of these memes. From an optimization point of view, a MA can be described as a population based metaheuristic in which, whenever an offspring is generated, a local search is applied to it until it reaches a local optimum.

The MA used in [4] for testing of object-oriented containers is fairly simple. It is built on GA, and the only difference is that at each generation on each individual a Hill Climbing is applied until a local optimum is reached. The cost of applying those local searches is high, hence the population size and the total number of generations is lower than in the GA.

5.2.5. Automatic Test Case Generation using Tabu Search: Tabu Search (TS) is a metaheuristic search technique based on the premise that, in order to qualify as intelligent, problem solving must incorporate adaptive memory and responsive exploration. Thus, the algorithm of Tabu Search is based on that of the next k neighbors, while maintaining a Tabu list (memory) of visited neighbors that are forbidden. The Tabu Search algorithm has a number of parameters that have to be chosen on the basis of the problem to be solved: the objective function (fitness function) which has to measure the cost of a solution, an appropriate Candidate list strategy (to try to choose good neighbor candidates that goes beyond a local optimum without exploiting the examination of elements in the neighborhood) and, it is also necessary to define short-term memory and long-term memory and their respective strategies. Short-term memory stores the recent moves of the search as Tabu. Long-term memory, on the other hand, stores the frequency with which a move occurs in order to penalize frequently visited moves that diversify the search. TSGen [21] is a Tabu search metaheuristic algorithm for the automatic generation of structural software tests. The goal that TSGen has to achieve is that of generating the tests that obtain the maximum branch coverage for the program under test. Figure 4 shows the scheme of this tool. [22] Created an efficient testing technique that combines Tabu Search with Korel's chaining approach. The technique automatically generates test data in order to obtain branch coverage in software testing.
5.2.6. Automatic test case generation using Simulated Annealing: Simulated Annealing is similar in principle to Hill Climbing. However, by allowing for a probabilistic acceptance of poorer solutions, Simulated Annealing allows for less restricted movement around the search space. The probability of acceptance $P$ of an inferior solution changes as the search progresses, and is calculated as

$$P = e^{-\frac{\delta}{t}}$$  \hspace{1cm} (3)

Where $\delta$ is the difference in objective value between the current solution and the neighboring inferior solution being considered, and $t$ is a control parameter known as the temperature. The temperature is cooled according to a cooling schedule. Initially the temperature is high, in order to allow free movement around the search space, and so that dependency on the starting solution is lost. As the search progresses, the temperature decreases. However, if cooling is too rapid, not enough of the search space will be explored, and the chances of the search becoming stuck in local optima are increased.

The work of Tracey and co-authors [23, 24] applies Simulated Annealing to structural test data generation, in the hope of overcoming some of the problems associated with the application of local search. In this work, test case can be generated for specific paths, or for specific statements or branches. In order to apply Simulated Annealing, a neighborhood structure has to be defined for the various different input variable types. The objective function is simply the branch distance of the required branch when control are diverges away from the intended path, or away from the target structure down a critical branch. In [25] a new automatic test case generation approach is presented based on Length-N coverage criterion through genetic simulated annealing algorithm. Firstly, a new test coverage criterion called Length-N is proposed to aim at the feasible of path coverage criteria. To enhance the generated efficiency, some improvements are made on genetic simulated annealing algorithm and use the results to generate test data. Lastly, experiment results show the method has a better effect.
5.2.7. Automatic Test Case Generation using Scatter Search: Scatter Search is an evolutionary method that works on a population of solutions of the problem to be solved, which are stored in a set of solutions called the Reference Set.

The solutions in this set are combined in order to obtain new ones, trying to continually generate better solutions, according to quality and diversity criteria. [26] Presents an approach based on the metaheuristic technique Scatter Search for the automatic test case generation of BPEL business processes using a transition-pair coverage criterion. The test case generator is called TCSS-LS-for-BPEL and it combines a diversity property with a local search. The diversity property is used to extend the search of test cases in order to reach different transitions of the business process. The local search is used to intensify the search when the diversification has problems to find test cases that cover transitions that have not been covered yet. Figure 5 shows the algorithm schema.

![Figure 5. Scatter Search Schema [26]](image)

5.3. Data Mining-based Automatic Test Case Generation

[16] Presents a novel methodology for identifying important test cases automatically using neural networks. These test cases involve input attributes which contribute to the value of an output and hence are significant. The reduction in the number of test cases is attributed to identifying input-output relationships. A ranked list of features and equivalence classes for input attributes of a given code are the main outcomes of this methodology. Reducing the number of test cases results directly in the saving of software testing resources. The proposed testing methodology is essentially a black-box testing approach employing a NN. The focal idea is to perform I-O analysis on the given piece of code. A trained neural network is pruned and rules are extracted from the pruned network. The significant inputs for a particular output are identified. The test cases involving unrelated attributes can be eliminated from a combinatorial set of test cases for a particular output.
6. Classification of Test Case Generation Evaluation Approaches

There are different ways to evaluate generated test cases; some of these approaches which are mostly used in mentioned automatic test case generation approaches will be described.

6.1. Number of Faults Detected (mutation testing): The effectiveness of test cases can be evaluated using a fault injection technique called mutation analysis. Mutation testing is a process by which faults are injected into the system to verify the efficiency of the test cases. Mutation-based analysis is a fault based testing strategy that starts with a program to be tested and makes numerous small syntactic changes into the original program. Program with injected faults are inserted and tested in the following manner. One faulty version of program is created at a time and run against all test cases one by one until either fault is revealed or all test cases are executed. A fault is considered to be revealed if the output of faulty version of program is different from the original program on same input. If a test case set is capable of causing behavioral differences between original program and mutant, mutant is considered as killed by test. The product of mutation analysis is a measure called mutation score, which indicated the percentage of mutants killed by a test set [17, 27].

\[
\text{Mutation score} = \frac{\text{No. of faults detected}}{\text{No. of faults injected}}
\]

6.2. Coverage Criteria: One critical task in software testing is to generate test case to satisfy given adequacy criteria. Given a coverage criterion, the problem of generating test case is to search for a set of test cases that lead to the highest coverage when given as input to the software under test.

As an example Branch coverage [14] means the percentage of no of edges/branches of the control flow graph covered by the test suit. The branch coverage factor for any test suit, T here is denoted as \( R(T) \). To evaluate this, firstly control flow graph of the program is created. Then, for each test case of a test suit, evaluate the set of branches that has been covered. Take the union of all the evaluated set of branches. Count the number of elements of the resultant set (let it be \( n \)). At the last, Branch coverage is evaluated as:

\[
R(T)=\frac{n}{e}
\]

Where \( e \) is the total number of edges of control flow graph. Some other examples are path coverage, condition coverage, and statement coverage explained in [28, 29, 30] respectively.

6.3. Input/output Analysis: Having a large set of expected input /outputs of a program under test, a good test set can be a set of test cases that finds the most mismatch between real inputs and outputs of a program.

6.4. Data Mining Approaches: It might be useful to have a mechanism that is able to learn, based on past history, which test cases are likely to yield more failures versus those that are not likely to uncover any. In fact the fault exposure capability of newly generated test cases can be predicted. Some features of test cases are extracted such as test case length and various coverage metrics relevant to the testing strategy used, and levels of severity of faults detected by those test cases can be found out [31].
8. Discussions

Figure 6 shows automatic test case generation system architecture, in terms of test type. The results of comparing different mentioned techniques are presented in table 3 and table 4. Table 3 shows a comparison between approaches in terms of test type, and Table 4 shows a comparison between approaches in terms of algorithm.

![Figure 6. Automatic Test Case Generation System Architecture](image)

<table>
<thead>
<tr>
<th>Approach</th>
<th>Input</th>
<th>Applicable for real-time systems</th>
<th>Complexity</th>
<th>Evaluation criteria of generated test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural-Based</td>
<td>Code/CFG</td>
<td>No</td>
<td>low</td>
<td>Coverage Criteria</td>
</tr>
<tr>
<td>Functional-Based</td>
<td>Specification</td>
<td>No</td>
<td>high</td>
<td>Input Output analysis</td>
</tr>
<tr>
<td>Gray-Box-Based</td>
<td>Code/Specification/CFG</td>
<td>Yes</td>
<td>high</td>
<td>Input Output analysis and coverage criteria</td>
</tr>
<tr>
<td>Non-Functional-Based</td>
<td>Software and hardware features</td>
<td>Yes</td>
<td>high</td>
<td>Depends of problem (e.g. execution time)</td>
</tr>
</tbody>
</table>
Table 4. Automatic Test Case Generation Approaches in Terms of Algorithm

<table>
<thead>
<tr>
<th>Approach</th>
<th>Input</th>
<th>Applicable for real-time systems</th>
<th>Complexity</th>
<th>Evaluation criteria of generated test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random-Based</td>
<td>Depends on Test Type</td>
<td>No</td>
<td>low</td>
<td>Number of detected faults</td>
</tr>
<tr>
<td>Search-Based</td>
<td>Depends on Test Type</td>
<td>Depends on Test type</td>
<td>high</td>
<td>Depends on Test Type.</td>
</tr>
<tr>
<td>Data mining-Based</td>
<td>Input/outputs of program</td>
<td>Depends on Test type</td>
<td>high</td>
<td>Depends on Test Type.</td>
</tr>
</tbody>
</table>

7. Conclusion

In this paper we represented two different classification frameworks for the existing automatic test case generation approaches, and also have a brief look at each one. We described how to evaluate generated test cases, and introduce a classification of evaluation approaches. The results show that different approaches should be selected based on types of applications, features of software we want to test, technique’s complexity, and other features.

Although there have been lots of researches on automatic test case generation problem, but for real world systems more researches are still needed.

Acknowledgements

This work is supported by Alzahra university of Tehran, computer engineering department. The authors are grateful to anonymous referees of this paper for their constructive comments.

References

Authors

Mohammad Reza Keyvanpour
Mohammad Reza Keyvanpour is an Assistant Professor at Alzahra University, Tehran, Iran. He received his B.S. in Software Engineering from Iran University of Science & Technology, Tehran, Iran. He received his M.S. and Ph.D. in Software Engineering from Tarbiat Modares University, Tehran, Iran. His research interests include software testing.

Hajar Homayouni
Hajar Homayouni received his B.S. in Software Engineering from University Of Kashan, Isfahan, Iran. Currently, she is pursuing M.S. in Software Engineering at Alzahra University, Tehran, Iran. His research interests include software testing, test case generation, and machine learning.

Hossein Shirazee
Hossein Shirazee received his B.S. in Software Engineering from University Of Kashan, Isfahan, Iran. Currently, he is pursuing M.S. in Software Engineering at Islamic Azad University, Qazvin Branch. His research interests include software engineering, software testing, Service Oriented Architecture, and cloud computing.