ISSN 2278-3091
Volume 9 No.2, March - April 2020
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse183922020.pdf
https://doi.org/10.30534/ijatcse/2020/183922020

An Optimized and Unique Methodology for Software Test Case Automation Strategy

Dr. Balika J Chelliah[1], K. Shreya[2], Arka Raha[3], M. Sravya[4]
[1]Department of Computer Science and Engineering, SRM Institute of Science and Technology, Ramapuram, Chennai, India, balika888@gmail.com
[2]Department of Computer Science and Engineering, SRM Institute of Science and Technology, Ramapuram, Chennai, India, shreyakrish1308@gmail.com
[3]Department of Computer Science and Engineering, SRM Institute of Science and Technology, Ramapuram, Chennai, India, arkaraha96@gmail.com
[4]Department of Computer Science and Engineering, SRM Institute of Science and Technology, Ramapuram, Chennai, India, madithatisravya1998@gmail.com

ABSTRACT

Natural Language Processing is one of the major techniques in Artificial Intelligence which is used for processing, analysis and conversion of text from one language to another. It is used to process human language in a way machines can analyze. Test case generation is one of the most important parameters in determining the fitness value of the algorithms and to determine what test cases are accepted by the algorithms. The necessity to generate automatic test cases arises from the need to produce generations of test cases for complex algorithms. The main objectives of this paper include generating test cases automatically by the means of methodical practice and trail enhancement and defining test strategy for any software application which will not only reduce the long man hour needed to create the manual test cases but also help in searching the bugs and technical errors quickly, efficiently and accurately. The sole purpose of this paper is to automatically generate test cases on the basis of available test resources. In current scenario, the generation of most test cases is done manually. The manual generation of test cases is prone to human errors. The other test case generation systems do not rely upon NLP due to which ambiguous use cases often lead to incorrect generation of test cases. Due to the possibility in human error and having room for improvement in the other documented test case generation method for software application , the proposed system not only generates automated test cases according to the use cases but also makes use of ONTOLOGY and NLP UMTG TOOLKIT for overcoming ambiguous problem statements. In software development and testing, the main purpose is to produce high software with high quality output while optimizing the cost and the time needed to complete the application development. To achieve this goal, software teams will perform the test on their application before live production. For Test Automation, documentation plays a crucial role. To test the software output, functionality and quality, different kinds of documentation will be developed

1. Test Script
2. Test Case
3. Test Scenario

The proposed system ensures efficient test coverage where the key functionality will not be missed in the automatic test case generation. In this proposed system, test cases can be generated after a feature or a set of features are finished in the application development. The NLP not only helps in developing better and automated test cases but also gives maximum and efficient output in case of ambiguous statements.

Key words: Natural Language Processing (NLP), Test Automation, Test Case, test coverage, Use Case Modelling for System Tests Generation (UMTG).

1. INTRODUCTION

In Software Engineering, the only aim is to produce high quality output while optimizing the cost and the time needed to complete the application development. To achieve this goal, software teams will perform the test on their application before live production. Currently Test Cases are generated manually. Therefore, there is always a possible chance of human error. Plus, there is always a room for improvement in documenting the test cases of the software application. Test case writing is major activity and a time consuming process. Test cases are key components of the entire testing process. Test cases are ground-breaking antiquities that work as a decent source of truth for how a framework and a specific component of programming works. The proposed system involves automated test case generation using Natural Language Processing (NLP). NLP is one of the most

important techniques in AI which is used for analyzing, processing and conversion of text from one language to another. It is used to process human language in a way machine can analyze.

Our aim in this paper is to generate test cases according to the use case and with the help of Ontology, Decision tree algorithm using machine leaning and NLP UMTG toolkit for overcoming ambiguous problem statements. The proposed system believes improvements can be made in the existing state of research in software test case generation which is semantic-based. Ontology is a key element and should be also considered in the design of our framework. Ontology is an important element of the semantic web. With the help of ontology, the complex structure of natural language (i.e. user requirements) is reduced to a simpler format. This simpler format can be used as a use case in UMTG. By using use case specifications, Use Case Modelling for System Tests Generation (UMTG) generates the OCL constraints and the test inputs are generated from OCL constraints. Cohesion metric is used to check the correctness of code generated by UMTG.

2. BACKGROUND

In the existing system, test cases are generated manually. Test case generation techniques usually generate test cases from the following resources:

1. Requirement specification document
2. UML diagram
3. Source code

Test Case Generation Techniques has two processes:

1. Define
2. Design

In the Define Process, test engineers collect, evaluates and identify all required information and also the pre-requisites. In the Design Process, test engineers aims to construct, organize and produce all components in a series of tests, such as test series, test data, and the need for each test case. A test case consists of parts that depict information, activity and a normal reaction, in order to ensure efficient working of any element of the application. A test case specifies more about "HOW" to approve a specific target, which when approved will depict whether the normal conduct of the framework is fulfilled or not. One significant inspiration is to guarantee detectability among necessities and framework test cases. Therefore, the meaning of test cases is tedious and testing, particularly under time imperatives and when there are successive changes to necessities. In this unique situation, programmed test age decreases the expense of testing as well as helps ensure that test cases appropriately spread all necessities. This section elaborates the overall sequence of the proposed system and the purpose of the UMTG toolkit.

The testing process is categorized into static and dynamic testing. The performance indicators consist of software and hardware performance indicators. These is a need and possibility to improve the process of semantic-based software test case generation. Ontology is an essential element of the current approaches to the semantic representation of Unified Modelling Language models. The process sequence during and after the use of ontology in the proposed system, is as follows.

1. Conversion of conditions and requirements, which are complex, provided by the user, into simple use cases.
2. The generated use cases are utilized by the UMTG toolkit to generate test input data.

Software testing is the most important parameter to ensure that the software systems are coherent with their requirements. Complex procedures of testing are used to ensure that every requirement is covered by the test cases. The test input data must be generated using the following steps:

1. Identify of all the representative test execution scenarios from requirements
2. Determination of the runtime conditions that trigger these scenarios
3. Generation of input test data.

The generation of test cases is error-prone considering the fact that the requirement specifications are very large for any software. Moreover, the specifications are represented in natural language in most cases. This makes the process of test case generation highly expensive and non-feasible. We use Ontology to simplify the complex requirements and conditions, provided in natural language and to convert them into simple use-cases. The main aim of the UMTG approach is the generation of executable system test cases. UMTG ensures requirement coverage and also reduces the manual effort required to generate these test cases. In conventional systems, restrictions are imposed on the template of use case specifications. This drawback is eliminated in UMTG. Using recent advances, to generate test data, UMTG

1. Automatically identifies test scenarios
2. Generates formal constraints that identify conditions triggering the execution of the scenarios

A domain model and use case specifications for the system under test are produced. The generation of test cases is automated by UMTG based on these. UMTG also generate test cases which involves critical scenarios which was not done in conventional systems. Checking the correctness and efficiency of the test case generator plays an important role. In the next section, we have discussed the use of cohesion metric which test the efficiency of the code.

Software designers and developers have always aimed at producing high quality software systems. The parameters which influence the quality of software are complexity, coupling and cohesion. High cohesion is considered as an essential characteristic of a well-designed and well-structured

software system. Generally, cohesion metric is used for testing of modules within the software. We propose the use of cohesion metric to evaluate the test case generator program. It determines the degree to which the modules in the software test case generator program relate to each other. Modules with high cohesion are usually efficient, robust, reusable, reliable, and understandable. On the other hand, modules with low cohesion are difficult to understand, test, have maintenance issues, and lead to problems while reusing.

The different approaches to the test case generation process, test case prioritization, effect of testing on code quality, software [1]-[5] were studied to develop an efficient system. The various applications of the software testing processes [6]-[13] were also studied to understand the practical applications of testing. Comparison between existing models, approaches, surveys and analysis of the existing approaches [14]-[21] aided in developing the proposed system. The prioritization approaches in different types of testing [22]-[24] and the various test case generation methods [25]-[27] were thoroughly understood to develop an efficient system.

3. PROPOSED SYSTEM

We have studied current approaches to semantic representation of Unified Modelling Language models. The following sections give a detailed outline of the proposed system and system architecture.

3.1 Ontology

For understanding the domain ontology already have been recognized as a prerequisite for conceptualizing the domain and processing the information. The ontology is a powerful tool which is mainly used to gather and share knowledge while the explicit specifications of the domain are provided. In the recent years, the realm of machine learning has seen the development of ontology in explicit formal specification of the terms within the domain. The case studies and the knowledge about the related rules of software were extracted from the case library and knowledge base for inference and interpretation by the inference engine, so that judgment predicates can be extracted from the production rules .The biggest advantage in ontology is that it creates sub categories under the domains and each sub-category is categorized according to the specific requirements. Due to this, the process of extracting information by processing natural language becomes smoother and efficient. The process of categorizing the domain for processing of natural language is crucial for the generation test cases for the software.

Ontology has five uses:

1. Ontology helps to have an understanding of the information structure among the software agents.
2. It is used for reuse of domain knowledge.
3. Domain assumptions could be made explicit.
4. Operational knowledge can be separated from the domain knowledge.
5. It is used to analyze domain knowledge.

3.2 Generation of Test Cases using Ontology

The main purpose of ontology in this paper is to convert complicated and complex natural language into simpler format while removing all the ambiguity. The main aim of processing and converting natural language into simpler format is to create use case with simpler formation with extraction of all the information from the condition and requirements provided. Simpler use cases will create better test cases for the software. While developing a system based on Ontology, according to the implementation and its requirements, we must keep in mind a few goals. Firstly, it is open-ended. We should get facts about all types of domains. Secondly, with a large corpus, this extraction technology involves building a large knowledge base. That is, the task should be dominated by precision, and not recall. Thirdly, the results must not be extracted from only one specific text. Rather, they must be statistical aggregates collected from various sources. For example, Hearst (1992) analyzed the concept of learning an ontology of concept categories and subcategories from a large corpus. (In 1992, a large corpus meant 1000-page encyclopedia. Today a large corpus would be a 100-million-page Web corpus.) The work focused on general templates which have high precision and which are not tied to a specific domain; this means they almost always match; but they had low recall; that is, they do not always match. Figure 1 is an example of one of the most productive templates.

NP **such as** NP (, NP)* (,)? (**and** | **or**) NP)?.

Figure 1: Example of a high-precision template

Here the words specified in bold and the commas must appear literally in the text. The asterick indicates that the elements can be repeated zero or more times. The parentheses are for grouping. Question marks indicate optional elements. NP stands for a noun phrase and indicates a variable. This above mentioned template can be matched with the texts "subjects such as Science increase your knowledge" and "programming languages such as Java". From this, we can draw a conclusion that Science is a subject and Java is a programming language. Using key words like "which includes," "mainly," and "especially," other templates can be constructed. These templates will fail to match many relevant passages, like "Science is a subject." However, this is intentional. The "NP is a NP" template does not always indicate a subcategory relation. It often means something else. For example, consider the lines, "There is a cat" or "She is a too tall." When we have a large corpus, we can afford to ensure that we use

only those templates which have a high-precision. Many statements of the subcategory relation might be missed. However, it is definitely possible to find a paraphrase of the statement elsewhere in the corpus in a usable form. The extracted simplified information now can be use as user case in UMTG software.

3.3 UMTG

Use Case Modelling for System Tests Generation (UMTG) is a technique that uses the domain model and use case specifications to automatically generate test cases. Use case specification is a document that captures the requirements of a use case. UMTG uses the system's domain model to define the constraints which will be used to generate the test input data. Restricted Use Case Modelling (RCUM) introduces templates with keywords and also restricts rules to reduce ambiguity in the requirements and to permit the automated evaluation of use case specifications. By retaining precision, consistency and completeness in use case specifications, RCUM can extract behavioral information. From RCUM specifications, UMTG uses NLP to create Use Case Test Models (UCTMs).By capturing control flow defined in RCUM specification, UCTM allows the model-based identity of use case scenarios (Use case scenarios refer to the sequences of use case steps in model). UMTG consists of 3 model-based, coverage techniques to generate use case scenarios from the UCTMs: branch, def-use, and subtype coverages. In each use case specification a list of conditions such as pre, post and guard are extracted which enables UMTG to define the constraint that the test inputs must satisfy in order to cover a test scenario. Each extracted condition is translated into an OCL (Object Constraint Language) constraint which describes the conditions in the form of entity in domain model. UMTG exploits the abilities of NLP techniques such as Semantic Role Labeling (SRL) to generate OCL constraints. Then the generated constraints are used to automatically generate test input data.

Three assumptions that are used to generate OCL constraints are:
Assumption 1: The concepts that appears in the requirement specification are modeled as attributes or classes or associations in domain model
Assumption 2: (OCL constraint pattern): The conditions in the use case specifications are very simple and they captures information regarding the states of domain entities (That is, classes in the domain model).
Assumption 3 (SRL): To correctly generate an OCL constraint, an SRL toolset is used that identifies the semantic roles in a sentence.

Semantic Role Labeling (SRL) is a NLP technique, used to determine the various roles played by the words in a given sentence. The words are marked with various keywords (For example, A0, A1, A2, …. AN) to indicate their roles. A0 indicates the subject or the one performing the action, whereas A1 indicates the actor who/which is most directly affected by the performed action. The constraint generation process is depicted in Figure 2.

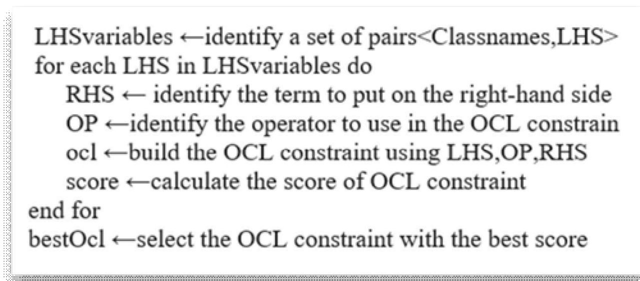

Figure 2: Constraint Generation Process

Identification of the Left-hand Side Variables: An algorithm is followed by the transformation rules(i.e., verb rules) which use the similarities in the strings between domain model elements' names such as classes, attributes, associations and the phrases in the use case step marked with the entity and support role to identify LHS-variable.

Identification of the Right-hand Side Terms: Based on the LHS-variable and based on the support roles that have not been used to select the LHS-variables, the RHS-terms are identified. An RHS-term can be a literal or a variable.

3.4 Use of Cohesion Metrics to check code Correctness

Cohesion determines to what degree the elements of a class or an object are related together. There are various object-oriented cohesion metrics which are based on the degree of similarity of methods. It has not been determined which of the above methods best measure cohesion. There are a wide range of suggested metrics. This is a huge challenge for software developers because it is very hard to make an informed choice. A non-cohesive class may need to be divided into smaller classes. For the following cohesion metrics, an assumption is made that methods are related if they function on the same class-level variables. On the other hand, they are unrelated if they work on variables on different levels. Methods in a cohesive class work with the same set of data. In case of a non-cohesive class, there are methods which work on different variables. Only one function is performed by a cohesive class. A class performing more than one function denotes a lack of cohesion which is undesirable. It is necessary for a class to split up if it performed a large number of unrelated functions. High cohesion is desirable for encapsulation. However, a limitation is that, for a class having high cohesion, the coupling between the class methods is very high, which denotes a high testing effort for that class. Low cohesion indicates high complexity and improper design. It also indicates a high possibility of errors. The class must be divided into two or more smaller classes.

There are two types of cohesions, namely TCC and LCC. TCC stands for Tight Class Cohesion and LCC stands for

Loose Class Cohesion. In order to determine the difference between good bad cohesions, TCC and LCC metrics are used, wherein large values indicate good cohesion and low values indicate bad cohesion. The idea of LCOM4 can be highly related to the TCC and LCC Metrics. Higher cohesiveness of the class can be associated to the higher values of LCC and TCC.

A Visible Method is the one that is not private and has an interface implementing it and such are the methods which are used for TCC and LCC. If any two methods, such as a and b, are able to access the same class-level variable, or if their call trees start at a and b then such methods are said to be related. All the methods, including the private methods are considered for the call trees. A call branch needs to stop being followed when there is a call going outside the class. Directly connected methods are those which are connected in the above manner. An indirectly connected method is one wherein the two given methods have a connection through other methods. Example: A, B and C are three methods wherein, the connections made from A to B to Care direct , that is AB are directly connected and so are BC. However, the connection to C from A is indirect. The TCC and LCC definitions are given in Figure 3.

NP = maximum number of possible connections
= N * (N-1) / 2 where N is the number of methods
NDC = number of direct connections (number of edges in the connection graph)
NID = number of indirect connections
Tight class cohesion TCC = NDC/NP
Loose class cohesion LCC = (NDC+NIC)/NP
TCC is in the range 0..1.
LCC is in the range 0..1. TCC<=LCC.

Figure 3: TCC and LCC definitions

Higher TCC and LCC would lead to an increase in the cohesiveness of the class .The non-cohesive classes are those where the value of TCC and LCC are lesser than 0.5 .While, quite cohesive classes are those where the value of LCC is equivalent to 0.8 and when the values of TCC and LCC are equivalent to 1, then this is considered as a maximally cohesive class which means that all the methods present in it are connected. LCC is an indication of the overall connectedness, which depends upon the number of methods and how they are grouped together. When the value of LCC is equivalent to 1, it is an indication that all the methods in the class are connected, either directly or indirectly. This is known as the cohesive case. Whereas, when the value of LCC is lesser than 1,it is an indication that there may be two or more unconnected method groups, which might be because the methods might be having access to totally different variables. The class is further known as “ not cohesive “. Further, such classes need to be reviewed in order to check why they are not cohesive. The non-cohesive case is the one where the value of LCC is equivalent to 0. TCC is an indication of the "connection density".

Moreover, when all the existing connections are direct, then the values of TCC and LCC evaluate to a value lesser than 1. Furthermore, the connection density is said to be lower when the value of TCC is lesser than that of LCC. In the example that follows, it is being proven that all methods are not directly connected with each other. Let A and B be the two points which are to be connected through variable x and B and C which are to be connected through y. Therefore, A and C are indirectly connected through B, even though they do not share a variable. Thus, the non-cohesive classes are those in which the methods are totally unconnected and TCC and LCC are equivalent to 0.

The overall system architecture of the proposed system is depicted in Figure 4.

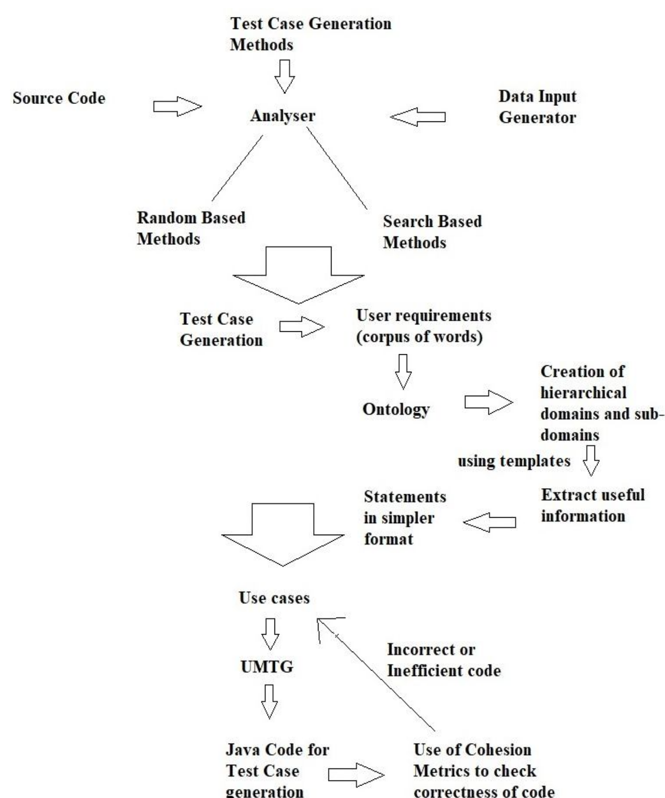

Figure 4: System Architecture

4. CONCLUSION

This paper sets up an automatic test case generation system based on knowledge base, case study library, solution space tree and testing weight technique. In this system, test cases are generated from the large number of rules which are established for the tested program. The knowledge base ensures the reliability of the tests. The case study library ensures the reusability for testing. The traversal inference

technique of the solution space tree ensures test adequacy. Hence, the testing weight technique can apply the skills, experience and knowledge of the test engineer for the generation of test cases. This enhances the efficacy of the test cases. Moreover, the use of simple use cases generated using ontology and the processing of these use cases in the UMTG toolkit to generate the test cases leads to the generation of a much more efficient set of input test data.

5. FUTURE WORK

Our future studies will focus on combining other technologies of Artificial Intelligence such as Artificial Neural Network (ANN), Genetic Algorithm (GA) and Simulated Annealing Algorithm (SAA).

REFERENCES

1. J. F. Chen, L. L. Zhu, T. Y. Chen, D. Towey, F. C. Kuo, R. B. Huang and Y. C. Guo. **Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering**, Journal of Systems and Software, 2018, 135: 107-125 https://doi.org/10.1016/j.jss.2017.09.031
2. T. H. Chen, S. W. Thomas, H. Hemmati, M. Nagappan and A. E. Hassan. **An Empirical Study on the Effect of Testing on Code Quality Using Topic Models: A Case Study on Software Development Systems**, IEEE Transactions on Reliability, 2017, 66(3): 806824
3. Cipriani, P. Citton, M. Romano and S. Fabbi. **Testing two opensource photogrammetry software as a tool to digitally preserve and objectively communicate significant geological data: the Agolla case study (Umbria-Marche Apennines)**, Italian Journal of Geosciences, 2016, 135(2): 199-209 https://doi.org/10.3301/IJG.2015.21
4. E. Engstrom and P. Runeson. **Test overlay in an emerging software product line - An industrial case study**, Information and Software Technology, 2013, 55(3): 581-594
5. J. R. Geringer, C. Y. Tuan and P. D. Lindsey. **Assessment of Software for Blast Loading and Structural Response Analysis Using a Lightweight Steel-Joist Roof as a Test Case**, Journal of Performance of Constructed Facilities, 2013, 27(2): 144-154
6. S. Ibaraki, S. Tsujimoto, Y. Nagai, Y. Sakai, S. Morimoto and Y. Miyazaki. **A pyramid-shaped machining test to identify rotary axis error motions on five-axis machine tools: software development and a case study**, International Journal of Advanced Manufacturing Technology, 2018, 94(1-4): 227-237 https://doi.org/10.1007/s00170-017-0906-9
7. J. Itkonen and M. V. Mantyla. **Are test cases needed? Replicated comparison between exploratory and test-case-based software testing**, Empirical Software Engineering, 2014, 19(2): 303-342
8. P. Janczarek and J. Sosnowski. **Investigating software testing and maintenance reports: Case study,** Information and Software Technology, 2015, 58: 272-288 https://doi.org/10.1016/j.infsof.2014.06.015
9. E. Jee, D. Shin, S. Cha, J. S. Lee and D. H. Bae. **Automated test case generation for FBD programs implementing reactor protection system software,** Software Testing Verification & Reliability, 2014, 24(8): 608-628
10. AminShokravi, H. Eskandar, A. M. Derakhsh, H. N. Rad and A. Ghanadi. **The potential application of particle swarm optimization algorithm for forecasting the air overpressure induced by mine blasting**, Engineering with Computers, 2018, 34(2): 277-285 https://doi.org/10.1007/s00366-017-0539-5
11. M. H. Esfe, O. Mahian, M. H. Hajmohammad and S. Wongwises. **Design of a heat exchanger working with organic nanofluids using multi-objective particle swarm optimization algorithm and response surface method**, International Journal of Heat and Mass Transfer, 2018, 119:922-930
12. S. S. Gill, R. Buyya, I. Chana, M. Singh and A. Abraham. **BULLET: Particle Swarm Optimization Based Scheduling Technique for Provisioned Cloud Resources**, Journal of Network and Systems Management, 2018, 26(2): 361-400
13. L. Guneshwor, T. I. Eldho and A. V. Kumar. **Identification of Groundwater Contamination Sources Using Meshfree RPCM Simulation and Particle Swarm Optimization**, Water Resources Management, 2018, 32(4): 1517-1538
14. S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. **An orchestrated survey of methodologies for automated software test case generation**, Journal of Systems and Software, vol. 86, no. 8, pp. 1978–2001, Aug. 2013. https://doi.org/10.1016/j.jss.2013.02.061
15. É. F. de Souza, R. D. A. Falbo, and N. L. Vijaykumar. **Knowledge management initiatives in software testing: A mapping study**, Information and Software Technology, vol. 57, pp. 378– 391, 2014.
16. J. J. Gutiérrez, M. J. Escalona, and M. Mejías. **A Model-Driven Approach for Functional Test Case Generation**, Journal of Systems and Software, vol. 109, pp. 214–228, 2015.
17. S. Paydar and M. Kahani. **A semantic web enabled approach to reuse functional requirements models in web engineering**, Automated Software Engineering, vol. 22, no. 2, pp. 241– 288, 2014. https://doi.org/10.1007/s10515-014-0144-4
18. M. Soltani et al. **Search-Based Crash Reproduction and Its Impact on Debugging**, IEEE Transactions on Software Engineering, 2018.
19. Panichella et al. **A large scale empirical comparison of state-of-the-art search-based test case generators**,

Information and Software Technology, vol. 104, 2018, pp.236– 256.

20. J. Campos et al. **An empirical evaluation of evolutionary algorithms for unit test suite generation**, Information and Software Technology, vol. 104, 2018, pp. 207–235.
https://doi.org/10.1016/j.infsof.2018.08.010
21. J. Ferrer et al. **Evolutionary algorithms for the multi-objective test data generation problem**, Software: Practice and Experience, vol. 42, issue 11, 2012, pp. 1331–1362,
22. Priyanka Paygude, Shashank Joshi, Debnath Bhattacharyya, Tai-hoon Kim. **Comparative analysis of Test Case Prioritization Approaches in Regression Testing**, International Journal of Advanced Trends in Computer Science and Engineering, Volume 8, No.4, 2019, pp. 1260-1267.
https://doi.org/10.30534/ijatcse/2019/36842019
23. Omdev Dahiya, Kamna Solanki, Sandeep dalal. **Comparative Analysis of Regression Test Case Prioritization Techniques**, International Journal of Advanced Trends in Computer Science and Engineering, Volume 8, No.4, 2019, pp. 1521-1531.
https://doi.org/10.30534/ijatcse/2019/74842019
24. P. Huang et al. **Performance Regression Testing Target Prioritization via Performance Risk Analysis**, in *Proc. The 36th International Conference on Software Engineering (ICSE)*, 2014.
25. L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. **Link: Exploiting the Web of Data to Generate Test Inputs**, in *Proc. 2014 International Symposium on Software Testing and Analysis (ISSTA),* 2014, pp. 373–384.
https://doi.org/10.1145/2610384.2610397
26. Arcuri et al. **Private API access and functional mocking in automated unit test generation**, in *Proc. IEEE international conference on software testing*, verification and validation (ICST), 2017.
27. K. Lakhotia et al. **A multi-objective approach to search-based test data generation**, in *Proc. The 9th International Conference on Genetic an Evolutionary Computation (GECCO)*, 2007.
https://doi.org/10.1145/1276958.1277175