Analysis of various Automated Test Cases Generation approaches

Mihaela Alexandra Bledea[a], Cezar Alexandru Bretan[a] and Tudor Boanca[a]

[a]*Babes-Bolyai University, Cluj-Napoca, Romania*

ARTICLE INFO

Keywords:
automation
testing
generation

ABSTRACT

This review explores the concept of automated test cases generation, looking at everything from the strategies used to the domains where it has applicability. We have analyzed research papers focusing on how automatic testing generation is done in the multiple fields and what benefits it brings. What we found is that by generating test cases automatically, the time required to identify all possible testing scenarios has been reduced and the coverage of the tests is increased in most scenarios. This review is valuable for anyone involved in software testing, highlighting multiple highly efficient approaches. Although there are numerous approaches employed, the genetic algorithms seem to have large applicability in the domain of testing.

1. Systematic literature review (SLR)

An approach known as a Systematic Literature Review (SLR) is a meticulous and rigorous methodology used to identify, evaluate, and synthesize current research in a specific field or topic. It involves conducting a systematic search and analysis of relevant literature, followed by a comprehensive evaluation and examination of selected studies based on predetermined criteria. The primary objective of an SLR is to provide a comprehensive and unbiased summary of the available information, enabling researchers to gain insights, identify patterns, and draw well-founded conclusions from the collective findings of multiple studies. By adhering to a well-defined methodology, an SLR reduces bias, ensures the ability to replicate the process, and enhances the overall reliability of the review.

2. Study design

To conduct an SLR on the topic of Automatic Test Cases in the general field, we adopted the methodology proposed by Kitchenham and Charters (2007). The guidelines we follow for conducting an SLR consist of three key phases, as described below:

- Planning Phase: This involves defining the research objectives, formulating research questions, establishing inclusion and exclusion criteria, devising a search strategy, and outlining the plan for data extraction.
- Conducting Phase: In this phase, we execute the search strategy, screen relevant studies based on the predetermined criteria, extract relevant data from the selected studies, and perform quality assessments.
- Reporting Phase: Here, we summarize the characteristics of the included studies, analyze the obtained results, address the research objectives and questions, and provide recommendations based on the findings.

By adhering to these guidelines, we ensure a systematic and comprehensive approach to conducting the SLR research process.

2.1. Review need identification

The aim is to provide an overview of existing research on Automatic Test Cases in the general field to acquire knowledge on its development. The motivation that lies behind is given by numerous factors:

- The inclination to discover the fields that benefit the most from the automated process of generating test cases.
- To gain insights about the approaches that yield the best results.

⋆ Automated Test Cases Generation

*Corresponding author

ORCID(s):

[1]

2.2. Research questions definition

The objective of the systematic literature review (SLR) is directed by the focus on two research questions.

RQ1 - What benefits certain approaches bring to the process of identifying bugs? - The aim of this question is to identify the most relevant and recurrent benefits that are brought by the approaches and that can make the testing process go more smoothly and easily.

RQ2 - What are the most researched fields in the domain of Automatic Generation of Test Cases? - The goal behind this question is to discover the fields that are the most beneficial when it comes to test cases generation and to acquire knowledge on how to take advantage of them.

2.3. Protocol definition

During the systematic literature review (SLR) process, it is crucial to establish the steps and criteria for analyzing each article. The following four steps are outlined for each article under study:

- Approach: Determine the approach or framework employed in the article.
- Methodology: Identify the specific methodology utilized in the article's research.
- Dataset: Describe the dataset or sources of data used in the article's study.
- Metrics and Obtained Results: Summarize the metrics used for evaluation and present the results obtained in the article.

By defining these steps for each article, a structured and comprehensive understanding of the SLR is achieved, enabling a thorough examination of the selected literature.

3. Conducting the SLR

3.1. Search and selection process

To determine the most significant papers for the questions we've stated at 2.2, the following steps have been taken, in accordance with the methodology outlined in Campeanu (2018).

3.1.1. Database search

The search operation was performed using the following databases: IEEE Explorer, SpringerLink, Arxiv, ResearchGate, ACM DL. The keywords used were: automation, generation, testing. With this method, 30 initial papers were selected.

3.1.2. Merging, and duplicates and impurity removal

From the initial papers, we didn't identify any explicit duplicates, but 2 papers were constructed on the top of other papers, bringing improvements, and no impurity removal was needed. All the remaining papers were of interest.

3.1.3. Application of the selection criteria

On the remaining 28 papers, the following criteria was applied:

- Studies presenting what data was leveraged in the validation process, in an explicit way.
- Studies provided appropriate information on the topic of interest
- No studies were selected that date before 2007.

3.2. Data extraction

After applying the filters mentioned in 3.1.3, the total number of remaining papers is 10. These papers are listed in table 1. The extracted data focuses on the approach and the used methods, dataset and results. It is described in section 3.3.

Id	Citation	Title, Authors	Year
P1	Irawan, Hendradjaya and Sunindyo (2016)	Test case generation method for go language	2016
P2	Martin-Lopez, Segura and Ruiz-Cortés (2021)	RESTest: automated black-box testing of RESTful web APIs	2021
P3	Hong, Zeng, Wang, Zhang and Wang (2010)	An approach of automated test cases generation in database stored procedure testing	2010
P4	Stallenberg, Olsthoorn and Panichella (2021)	Improving test case generation for REST APIs through hierarchical clustering	2021
P5	Meyer, Ciupa, Leitner and Liu (2007)	Automatic testing of object-oriented software	2007
P6	Amalfitano, Fasolino, Tramontana, De Carmine and Memon (2012)	Using GUI ripping for automated testing of Android applications	2012
P7	Li, Ghosh and Rajan (2011)	KLOVER: A symbolic execution and automatic test generation tool for C++ programs	2011
P8	de Moura, Charao, Lima and de Oliveira Stein (2017)	Test case generation from BPMN models for automated testing of Web-based BPM applications	2017
P9	Du, Pan, Ao, Alexander and Fan (2019)	Automatic test case generation and optimization based on mutation testing	2019
P10	Lukasczyk and Fraser (2022)	Pynguin: Automated unit test generation for python	2022

Table 1
Selected papers

3.3. Data synthesis

3.3.1. Test case generation method for Go language

The research paper introduced by Irawan et al. (2016), proposes an approach for generating test cases with code coverage adequacy in Go language using a Genetic Algorithm (GA). The developed tool, called Goceng, consists of four main components: an analyzer, test data generator, test code generator, and executor. The analyzer collects information from the source code, such as function declarations, imports, types, and constants. The test data generator generates input values based on the collected information, and the test code generator produces the test code including the invocation of the function under test. The executor runs the generated test code using the go test tool to collect coverage profiles. The researchers evaluated the approach on eight different Go source codes and found that it facilitated the process of unit testing by reducing the effort and time required for test case generation. In what concerns the experiments, they performed 10 times generation at maximum crossover duration time (5 minutes). The same process was repeated for the other 7 source codes. The paper discusses the design and implementation of Goceng, as well as the experimental results demonstrating its applicability in achieving full code coverage. However, it also mentions some weaknesses, such as the difficulty in determining which paths will be covered by the generated test cases. Overall, the approach shows promise in automating test data generation and code coverage in Go language, with potential for further enhancements and applications.

3.3.2. RESTest: automated black-box testing of RESTful web APIs

Martin-Lopez et al. Martin-Lopez et al. (2021) introduce in their research paper RESTest, a black-box testing framework for RESTful web APIs. The framework supports the generation of test cases using various techniques, including fuzzing and constraint-based testing. RESTest takes the API specification in the OpenAPI Specification format as input and generates abstract test cases that can be transformed into executable test cases for specific testing frameworks and programming languages. The paper presents the architecture and workflow of RESTest, describing its components such as test data generators, test case generators, test case mutators, test writers, and test runners. The framework was evaluated through offline and online testing scenarios. In offline testing, RESTest outperformed random approaches in detecting failures by leveraging constraint-based testing. In online testing, RESTest continuously tested multiple APIs, generating thousands of test cases and uncovering failures related to server errors, client errors, and inconsistencies between the API specification and implementation. The paper highlights the effectiveness and

potential of RESTest in testing real-world APIs and suggests future work on expanding the framework with additional test data generators and generation strategies.

3.3.3. An Approach of Automated Test Cases Generation in Database Stored Procedure Testing

This paper introduced by Hong et al. (2010) proposes an approach that combines white-box testing with black-box testing for testing database stored procedures. The approach utilizes path coverage to select a set of testing paths and generates test cases using program slices. The authors constructed a constraint system for test data generation by extracting and replacing predicates in the program slices. They used Z-path coverage testing and depth-first search algorithm to generate test paths for the stored procedures. The constraint system was solved using a genetic algorithm to obtain test data that meets the constraints. The experiments conducted showed that the approach can efficiently generate test cases and achieve path coverage testing for database stored procedures. In the provided excerpt, the results of the experiments are briefly mentioned. The system takes input in the form of information about the database stored procedure to be tested and the desired number of test cases to be generated. The output includes formatted test case files, database states, and system execution logs. The experiments conducted on MySQL stored procedure testing involved generating automated test cases for 80% of simple stored procedures (up to 20 lines without complex logic). After running the tests using the generated test cases, it was found that 70% of the test cases were executable and provided coverage for general path testing. This approach demonstrated improved efficiency and increased test coverage compared to manual test case generation. However, the experiments also identified some limitations. The approach failed to cover the entire paths within loop bodies, requiring additional manual work to add test data. Additionally, compatibility issues with other testing tools, such as integrating with test data generated by existing tools, were observed. The conclusion acknowledges the need for further research and development in automated test case generation for database stored procedure testing, including addressing challenges related to loop bodies, system and user functions, dynamic test case generation, and more.

3.3.4. AndroidRipper: Using GUI Ripping for Automated Testing of Android Applications

AndroidRipper is presented in paper Stallenberg et al. (2021) as an automated technique that tests Android apps via their Graphical User Interface (GUI). It uses ripping to automatically and systematically traverse the app's user interface, generating and executing test cases as new events are encountered. AndroidRipper dynamically analyses the application's GUI with the aim of obtaining sequences of events fireable through the GUI widgets. Each sequence provides an executable test case. During its operation, AndroidRipper maintains a state machine model of the GUI, which is called a GUI Tree. The tool has been tested for an open-source application named "WordPress for Android", and the obtained results show that the generated GUI-based test cases are able to detect severe, previously unknown, faults in the underlying code, and the structured exploration outperforms a random approach.

3.3.5. AutoTest: Automatic testing of object-oriented software

AutoTest was proposed as automatic testing of object-oriented software approach, in paper Meyer et al. (2007). AutoTest permits completely automated unit testing, applied to classes, the only input required being the list of classes to be tested. Testing a class means, for AutoTest, testing a number of routine calls on instances of the class. Firstly, the issue of generating objects is considered. The general strategy, when an object of type T is needed — as target of a call, but also possibly as argument to a routine — is the following, where some steps involve pseudo-random choices based on preset frequencies. Secondly, many of these routine calls require arguments. These can be either object references or basic value. For objects, the strategy is to get an object from the pool, or create a new object. For basic values, the current strategy is to select randomly from a set of values preset for each type. Lastly, test oracles decide whether a test case has passed or failed. Contracts state what conditions the software must meet at certain points of the execution and they can be evaluated at runtime. They include the pre- and the post- conditions of the routines, and the invariants of a class. AutoTest's strategy must be, to minimize occurrences of direct precondition violations, and to maximize the likelihood of precondition and invariant violation. All violations matching these cases will be logged, together with details about the context: exact contract clause being violated, full stack trace. The authors have put libraries like EiffelBase and Gobo under test using their proposed tool. A significant proportion of the bugs currently found by AutoTest have to do with "void" issues: failure of the code to protect itself against feature calls on void (null) targets.

3.3.6. Improving Test Case Generation for REST APIs Through Hierarchical Clustering

This paper Amalfitano et al. (2012) proposes a novel approach called LT-MOSA (Linking Tree + Many Objective Sorting Algorithm) that utilizes Agglomerative Hierarchical Clustering (AHC) to create a linkage tree model. This

model captures, replicates, and preserves patterns in new test cases. The authors evaluate LT-MOSA by conducting an empirical study on seven real-world benchmark applications (e.g. CatWatch, ProxyPrint), focusing on branch coverage and real-fault detection capability. The results indicate that LT-MOSA achieves a statistically significant increase in test target coverage (specifically lines and branches) compared to MIO (Many Independent Objective) and MOSA algorithms in four and five out of the seven applications, respectively. These findings demonstrate the improved effectiveness of LT-MOSA in terms of test coverage and real-fault detection in the context of REST API testing.

3.3.7. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs

Li et al. Li et al. (2011) proposes KLOVER as an automatic generation tool for C++ programs. KLOVER builds on top of the symbolic execution engine KLEE, which is capable of handling sequential C programs. The tool extends KLEE to handle C++ features and addresses various challenges specific to C++ programs. It introduces an extended symbolic virtual machine, library optimization for C and C++, object-level execution and reasoning, interfacing with efficient solvers, and semi-automatic unit and component testing. The execution generates statistics information, sanity check results, and concrete test cases that can be replayed in a real setting. KLOVER models machine execution states, employs source code definitions for building solvers, and incorporates specific solvers for common data structures like strings. KLOVER has been used on real-life industrial applications developed at Fujitsu as well as publicly available C++ programs. In experiments, KLOVER has shown promising results in terms of achieving high structural coverage and reducing manual testing efforts. It has been able to achieve higher coverage with a small number of test cases compared to manual testing. The tool has been successful in detecting bugs and revealing issues in various C++ applications, including cryptographic hash functions, tree structures, regular expressions, and URI analysis. The tool was ran on industrial applications developed at Fujitsu, which suggests the use of proprietary software developed by the company. Additionally, publicly available C++ programs from sources like SourceForge were used to test the tool's effectiveness on widely used algorithms such as SHA-1, AVL tree balancing, regular expressions, and URI analysis.

3.3.8. Test Case Generation from BPMN Models for Automated Testing of Web-Based BPM Applications

The approach proposed in the paper de Moura et al. (2017) aims to assist in the generation of test cases for automated functional testing of web applications built with the support of Business Process Management Systems (BPMS). The focus is on BPMN models, which represent the business processes. The approach involves two main steps: flow table generation and test code generation. A tool is designed and implemented to analyze BPMN files exported by BPMS tools. The tool traverses the XML representation of the BPMN file, analyzing the Sequence Flow elements to identify the execution paths and possible flows in the process. The tool generates an Excel file containing a table that represents all the possible scenarios/process flows. Using the flow table, another tool is developed to automate the generation of test scripts' initial code for functional testing. The generated code is intended to be used with Selenium, a web application testing tool, and Cucumber-JVM, a tool for behavior-driven development. The tool helps in generating the features file and steps file required for test scenarios. The paper mentions preliminary results, but the specific details are not provided in the excerpt. It states that the results of analyzing BPMN models from two popular open-source BPMS tools, Bonita BPMS and Activiti, corroborate the problems raised by other authors regarding difficulties in performing automated load and functional tests in web-based BPM applications. The results also indicate that the effort associated with automated tests may vary depending on the BPMS and its underlying technologies. The article refers to the evaluation of processes built with different BPMS tools, such as Hardware Retailer, Nobel Prize - Nobel Assembly, Incident Management (Whole Collab), Order Fulfillment, Pizza - Pizza Customer, and Travel Booking. These processes likely serve as examples or case studies for evaluating the effectiveness of the approach.

3.3.9. Automatic Test Case Generation and Optimization Based on Mutation Testing

The article Du et al. (2019) proposes an approach for automatic test case generation and optimization based on mutation testing. The approach involves a mutation operator selection strategy to reduce the number of mutants. Five mutation operators are selected from the Mujava tool. A test case generation method is then applied, combining mutation testing with a genetic algorithm. The test cases are optimized using crossover, insertion, change, and deletion operators. The proposed approach is compared with other algorithms and tools. The experimental results show that the proposed method outperforms other approaches in terms of coverage and mutation scores. The coverage comparison indicates that Evosuite performs better than Randoop and the proposed method (MUGA). However, the test case optimization algorithm in the proposed method successfully reduces the length of test cases compared to Evosuite. The mutation scores comparison reveals that MUGA performs well, while Randoop, which does not consider mutation

Article	Approach	Tools	Case study
Irawan et al. (2016)	Genetic Algorithm (GA)	Goceng	Academic
Martin-Lopez et al. (2021)	Fuzzing and constraint-based testing	RESTest	Academic
Hong et al. (2010)	Z-path coverage testing and depth-first search algorithm	-	Academic
Stallenberg et al. (2021)	GUI tree ripping	AndroidRipper	Academic
Meyer et al. (2007)	Min-Max Algorithm	AutoTest	Academic
Amalfitano et al. (2012)	Agglomerative Hierarchical Clustering (AHC)	EvoMaster	Academic
Li et al. (2011)	Builds on Execution engine KLEE	KLOVER	Real life
de Moura et al. (2017)	Flow table generation and test code generation	-	Academic
Du et al. (2019)	Genetic algorithm (GA)	MUGA	Academic
Lukasczyk and Fraser (2022)	DynaMOSA, MIO, MOSA	Pynguin	Academic

Table 2
Approach Comparison Table

testing, generates test cases that are less effective in killing mutants. Evosuite, using the MOSA algorithm, achieves intermediate results. The article mentions using a Java project in SF100 for the experiments. The dataset consists of 10 projects labeled C1 to C10.

3.3.10. Pynguin: Automated unit test generation for python

This article Lukasczyk and Fraser (2022) proposes an approach that uses the Pynguin tool. Pynguin is a flexible and comprehensive test-generation framework for Python, designed to create regression tests that achieve extensive code coverage. By taking a Python module as input, Pynguin conducts an analysis to extract relevant information, including declared classes, functions, and methods. During the construction of a test case, Pynguin selects a specific function or method from the module under examination. Subsequently, Pynguin employs a backward approach to fulfill the requirements of the selected function's parameters. To generate test inputs, users have the option to choose from a variety of well-established algorithms, such as DynaMOSA, MIO, MOSA, random, Whole Suite, and Whole Suite with archive. These algorithms can be used individually or in combination to optimize different forms of coverage. After assessing the fitness of generated test cases, Pynguin proceeds with the subsequent iteration of the test-generation algorithm. An inherent advantage of this approach is that it implicitly minimizes the number of assertions within the resulting test cases. In order to evaluate Pynguin's effectiveness, we conducted a small experiment for this paper, employing the aforementioned algorithms for test generation. The results clearly demonstrate that the search-based techniques outperform the random algorithm. Among the search-based algorithms, there are only marginal differences, with DynaMOSA achieving the highest coverage values and Whole Suite exhibiting the lowest coverage values. These findings indicate the robustness and effectiveness of Pynguin in generating regression tests that offer significant code coverage.

4. Results

RQ1 - What benefits certain approaches bring to the process of identifying bugs?

RA1 - Based on the studied approaches, we identified multiple benefits. The Goceng tool reduces the effort and time required for test case generation. RESTest helps uncover issues related to server errors, client errors and inconsistencies between API specification and implementation. The approach proposed for testing database stored procedures reduces the manual test case generation effort. AndroidRipper has been effective in detecting severe and most important, previously unknown faults in the underlying code. KLOVER is said to reduce manual testing efforts. LT-MOSA improves test coverage and real-fault detection compared to other algorithms. Pynguin outperforms random test generation.

RQ2 - What are the most researched fields in the domain of Automatic Generation of Test Cases?

RA2 - Based on the covered studies, we cannot narrow down the search on a singular universal field in Automatic Generation of Test Cases. This conclusion is based on the multitude of software engineering fields identified in the presented papers that can benefit from the studied domain.

References

Amalfitano, D., Fasolino, A.R., Tramontana, P., De Carmine, S., Memon, A.M., 2012. Using gui ripping for automated testing of android applications, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 258–261.

Campeanu, G., 2018. Gpu support for component-based development of embedded systems.

Du, Y., Pan, Y., Ao, H., Alexander, N.O., Fan, Y., 2019. Automatic test case generation and optimization based on mutation testing, in: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE. pp. 522–523.

Hong, M., Zeng, Q., Wang, Z., Zhang, Y., Wang, H., 2010. An approach of automated test cases generation in database stored procedure testing, in: 2010 Second International Workshop on Education Technology and Computer Science, IEEE. pp. 533–537.

Irawan, E.W., Hendradjaya, B., Sunindyo, W.D., 2016. Test case generation method for go language, in: 2016 International Conference on Data and Software Engineering (ICoDSE), IEEE. pp. 1–5.

Kitchenham, B., Charters, S., 2007. Guidelines for performing systematic literature reviews in software engineering 2.

Li, G., Ghosh, I., Rajan, S.P., 2011. Klover: A symbolic execution and automatic test generation tool for c++ programs, in: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23, Springer. pp. 609–615.

Lukasczyk, S., Fraser, G., 2022. Pynguin: Automated unit test generation for python, in: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, pp. 168–172.

Martin-Lopez, A., Segura, S., Ruiz-Cortés, A., 2021. Restest: automated black-box testing of restful web apis, in: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 682–685.

Meyer, B., Ciupa, I., Leitner, A., Liu, L.L., 2007. Automatic testing of object-oriented software, in: SOFSEM 2007: Theory and Practice of Computer Science: 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007. Proceedings 33, Springer. pp. 114–129.

de Moura, J.L., Charao, A.S., Lima, J.C.D., de Oliveira Stein, B., 2017. Test case generation from bpmn models for automated testing of web-based bpm applications, in: 2017 17th International Conference on Computational Science and Its Applications (ICCSA), IEEE. pp. 1–7.

Stallenberg, D., Olsthoorn, M., Panichella, A., 2021. Improving test case generation for rest apis through hierarchical clustering, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE. pp. 117–128.