Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 627–636
DOI: 10.15439/2023F9829
ISSN 2300-5963 ACSIS, Vol. 35

Current Trends in Automated Test Case Generation

Tomas Potuzak
0000-0002-8140-5178
Department of Computer Science and Engineering/ NTIS – New Technologies for the Information Society, European Center of Excellence, Faculty of Applied Sciences, University of West Bohemia
Univerzitni 8, 306 14 Plzen, Czech Republic
Email: tpotuzak@kiv.zcu.cz

Richard Lipka
0000-0002-9918-1299
NTIS – New Technologies for the Information Society, European Center of Excellence/Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia
Univerzitni 8, 306 14 Plzen, Czech Republic
Email: lipka@kiv.zcu.cz

***Abstract*—The testing is an integral part of the software development. At the same time, the manual creation of individu-al test cases is a lengthy and error-prone process. Hence, an intensive research on automated test generation methods is ongoing for more than twenty years. There are many vastly dif-ferent approaches, which can be considered automated test case generation. However, a common feature is the generation of the data for the test cases. Ultimately, the test data decide the prog-ram branching and can be used on any testing level, starting with the unit tests and ending with the tests focused on the behavior of the entire application. The test data are also mostly independent on any specific technology, such as programming language or paradigm. This paper is a survey of existing litera-ture of the last two decades that deals with test data generation or with tests based on it. This survey is not a systematic literature review and it does not try to answer specific scientific questions formulated in advance. Its purpose is to map and categorize the existing methods and to summarize their common features. Such a survey can be helpful for any teams developing their methods for test data generation as it can be a starting point for the exploration of related work.**

***Index terms*—Software testing, test case generation, test data generation, papers survey.**

I. Introduction

TESTING is an essential part of software development. At the same time, the manual creation of individual test cases is a lengthy and error-prone process. In many real-world projects, there is not enough time to ensure sufficient testing of the developed software product, which leads to its lowered quality. The programmers of the test cases can also miss some inputs, which leads to unexpected behavior of the software product. Hence, an intensive research of automated test generation methods is ongoing for more than twenty years, as can be seen, for example, in [1] or [2].

In the existing literature, automated test case generation is used or at least proposed on various testing levels. These levels include unit testing focused on the functionality of isolated features of the developed application (usually a method, procedure, or function), but also the regression and integration testing focused on the correct cooperation of the individual parts of the application. Automated test case gene-ration can also be used during the high-level testing of the functionality of the entire application and its adherence to the specified requirements. Automated test case generation is tempting and seems to be promising, as it should reduce the time the programmers spend on manual test case preparation. Nevertheless, there are several limitations.

First of all, it is difficult to automatically verify that the tested application or its part provides correct results. This would require generating the expected outputs for all the generated inputs, which is an inherently difficult task. Never-theless, this ability is crucial for the usage of the automated test case generations in real software projects. However, it should be noted that, in many cases, it is possible to detect the incorrect behavior of an application even without the known correct outputs. An obvious example is when the application crashes, but there can also be limitations of the outputs, which can be used for incorrectness checking (e.g., the calculated volume of a cube cannot be negative).

Another issue, which is often discussed (e.g., in [3]) is related to the combinatorial explosion. Consider a unit test of a method with several parameters where various combinations of the parameters should be considered. Even when the parameters can be grouped into several discrete classes, the number of all the possible combinations grows very fast with the growing number of parameters and classes. This problem is even more pronounced in higher-level tests, when multiple methods are executed during one higher-level functionality testing. Various settings and running environments of the tested application only worsen this problem. Hence, even in tools, which are used in real projects, such as EvoSuite or Randoop [4], the number of generated test cases can be very high, which leads to long running times. This partially limits the usability of automated testing. Nevertheless, the problem can be mitigated by employing efficient test case selection in

This work was supported by Institutional support for long-term strategic development of research organizations.

order to generate and run the test cases, which provide the highest expected code coverage and/or have the highest expected error detection rate. The increasing power of contemporary computers is also helpful, as running a huge number of tests is more and more feasible.

The last issue, we would like to mention, is the validation of the automated test-generating methods themselves. Several different approaches for the evaluation of the automated test-generating methods can be found in the existing literature. From the practical usability of the methods in real projects point of view, there are two most important questions – how realistic the methods are and how well they perform in finding different types of realistic errors.

There are many different approaches, which can be considered automated test case generation. However, a common feature is the generation of the data for the test cases. Ultimately, the test data decide the program branching and can be used on any testing level, from the unit tests to the tests focused on the behavior of the entire application. The test data are also mostly independent on any specific technology, such as programming language or paradigm.

This paper is a survey of existing literature of the last two decades that deals with automated test data generation or with tests based on it. This survey is not a systematic literature review and it does not try to answer specific scientific questions formulated in advance. Its purpose is to map and categorize the existing methods and to summarize their common features. Such a survey can be useful for any teams developing their methods for test data generation as it can be a starting point for the exploration of related work.

The remainder of this paper is structured as follows. Related surveys are discussed in Section II. The selection of the papers for this survey is described in Section III. The existing methods for test data generation are discussed in Section IV. Their common features and trends are described in Section V. Threats to validity are described in Section VI. The conclusions and the future work are in Section VII.

II. Related Work

There are multiple studies, which survey the existing testing approaches. Our survey is intended to complement them from the automated test data generation point of view.

A. Existing Methods Studies

Ref. [5] summarizes the methods for test generation based on control flow analysis, automatic random data generation, and program execution analysis and/or the methods designed to produce tests, which maximizes the code coverage. The majority of the methods described in this survey is designed to deal only with simple program constructions and are often based on the models of the program instead of real programs. This is quite understandable, since the survey is rather old (from 1999). Nevertheless methods based on the same principles repeat again and again in more modern papers, only the methods or at least the examples, on which the methods are demonstrated, are usually more complex. For example, a more recent orchestrated survey [6] is focused on adaptive random testing among other methods.

A thorough review in [7] focuses on the papers dealing with search-based test case generation. The review makes it obvious that there is a constant increase in the number of testing-related publications between 1995 and 2007. The main focus of the review is the quality of the verification of the test generation methods. It is concluded that there is a lack of a standardized rigorous method to perform, asses, and compare the individual methods. Moreover, in many papers, there is even not enough empirical data to perform any comparison. It is also pointed out that, while many methods can achieve relatively high code coverage, it is not clear, whether the tests covering the code are able to find errors in the code. Another survey focused on search-based test case generation can be found in [8].

The search-based testing with an emphasis on mutation-based methods is also the theme of the survey in [9]. The methods described in papers published between the year 1996 and 2014 are based on genetic algorithms, ant colony optimization, simulated annealing, or hill climbing. The survey discusses also the relations and development of the methods in multiple papers. There are several conclusions. One is that the above-mentioned meta-heuristics significantly reduce the number of generated test cases without negative effects on the code coverage. Another is that the automated test generation methods are not designed for the concurrency problems. The last conclusion is that the comparability of the automated methods is difficult, similarly to [7].

The review in [10] focuses on the dynamic symbolic execution. There are twelve tools, which are compared based on various features, such as the number of publications dedicated to each tool, the utilized method for automated test generation, and the environment, in which the tool can be used. The ability of the tools to detect errors in the software is not among the investigated features. This feature is investigated in [11], which is focused on the methods utilizing aspect-oriented programming (namely Wrasp, Aspectra, Raspect, and EAT). One of the conclusions is that the structural evolutionary testing (EAT) shows the most promising results but at the cost of greater effort compared to random testing.

The short survey in [12] focuses on papers dealing with test data generation. It discusses various types of data generation from their architecture and usage points of view. The advantages and disadvantages of the methods as well as the best practices are discussed.

Although the majority of the surveys described above are focused on a technology or a set of technologies, there are also surveys focused on a specific type of software. An example is a systematic literature review [13], which deals with automated functional testing of mobile applications. Another example is a study [14], which discusses application of several different techniques for verification of flight software in Jet Propulsion Laboratory.

B. Practical Usability Studies

There are also studies, which focus on the usage of the automated testing methods in real projects, such as [15]. This study is not focused on published papers, but rather describes how the testing methods are used in real software projects and how the automated testing methods would improve the situation. The study has a bit darker tone than the studies mentioned in Section II.A as it points out that there is a lot of additional effort necessary when a promising method described in a paper should be used in a real industry project.

The study described in [16] is focused on the comparison of existing tools for automated test generation, such as Randoop, AutoTest, AnalitiX, Jtest, and so on. This study describes how the comparison of different methods for automated test generation should look like – precisely the aspect, which was mentioned as missing in [7] and [9]. In [16], a complex benchmark consisting of over 30 cases is described, which enables to empirically determine whether the automated test generation methods are able to uncover specified conditions. The results from this benchmark can be used for comparison of the methods. Although this benchmark is a good basis for the comparison of the automated test generation methods, it still utilizes synthetic cases, not real software [16].

An unorthodox practical study is the Java Unit Testing Tool Contest, which is held annually and its results are reported at various conferences (e.g., in [17] or [18]). The contest is intended for test generation tools designed for Java. Their ability to find errors in programs is tested using a benchmark consisting of real-life classes taken from various open-source GitHub projects. The contesting tools are evaluated based on the code coverage and mutation score [17], [18].

III. Survey Description

This paper is an intermediate result of our exploratory work to create a substance for a systematic literature review, which is the main aim of our current and future work (see Section VII). Although this intermediate result is only a (non-systematic) survey, the collection of the primary studies was performed in a rigorous manner described in following subsections, as the collected papers will also form part of the basis for our future systematic literature review.

A. Papers Searching

As the sources of the papers, we used the IEEE Xplore[1] library, which includes full texts of a large number of technology-related papers from both conferences and journals and the ScienceDirect[2] library, which includes papers from a large number of technology-related journals. Due to the institutional subscription, we have access to the majority of the full texts of the papers contained in both libraries, which is essential for the survey. Both libraries enable basic and advanced searching, but the available filters are quite different. For this reason, we used different settings for each library to obtain manageable numbers of relevant results. We made several attempts with various filters and search strings before we reached the final settings for both libraries.

The final search string for the IEEE Xplore library was "automated test data generating". It was used together with two filters. The year of publication had to be from 2000 to 2022 and the publication topic had to be "Program Testing". Using this setting, 461 results were obtained. The final string for the ScienceDirect was "automated test data generating program testing". It was used with three filters. Similarly to IEEE Xplore, the year of publication had to be from 2000 to 2022. Additionally, the subject area had to be "Computer Science" and the title of the paper had to contain "test data". Using this setting, 58 results were obtained. The searching in both databases was performed in April 2023.

B. Papers Filtering

From the search results, only the papers focused on the issues of automated test data generation for software testing, were selected. In first round, the selection was performed based on the titles. In second round, the selection was performed based on the abstracts, but only from the papers, which passed the first round. After the second round, there were 179 papers left (see Table I). The full texts were downloaded and investigated only for the 179 papers, which passed the second selection round. From these papers, some were eliminated from further processing, because, despite the promising title and abstract, the theme of the paper was outside the scope of this survey. Of the remaining papers, only 67 were included into the study, because they best represent the current trends in test data generation.

It should be noted that many of the obtained papers were already processed during our preliminary work with different search strings and filter settings in 2022. Hence, only the newest papers and papers not obtained previously due to different search settings of the libraries had to be processed. This enabled us to finish the paper in a relatively short time after the final search was performed.

C. Aims of the Survey

As this survey is intended to serve as a starting point for the exploration of related work for research teams dealing with automated test data generation, the aims of the survey can be summarized as follows:

- To categorize existing automated test data generation methods (see Section IV).
- To summarize and discuss common features of the methods (including their verification, implementation availability, testing level, and target platform) and observable trends (see Section V).

TABLE I SUMMARY OF THE NUMBERS OF SELECTED PAPERS

Library	Search results count	Selected papers count
IEEE	461	136
ScienceDirect	58	43

[1] https://ieeexplore.ieee.org
[2] https://www.sciencedirect.com

IV. Existing Methods in Literature

The categorization of the surveyed automated test data generation methods was performed based on the primary technology used for the test data generation. This categorization enables the readers to focus mainly on the papers related to the technology of their interest. It is also consistent with the existing surveys, as they are often focused on a relatively narrow set of technologies (see Section II). The papers of individual categories are discussed in following subsections.

A. Pseudorandom Generation-based Methods

The most basic approach, how to obtain test data, is to generate them using pseudorandom generators. Though the basic method can give relatively good results (e.g., code coverage) for number inputs, its usage for a more complex (and valid) data, such as specific strings or objects is difficult. Nevertheless, pseudorandom number generation is often combined with other approaches. In [19], the stochastic process models of the objects and their random initiation is used together with random method invocation.

In [20], the pseudorandom generating is combined with the constraint solving for the generation of test data for relational database schemas. The testing of object-relational mapping (ORM) based on the pseudorandom generation and formal models is described in [21]. In [22], data description using XML and regular expressions is used together with pseudorandom generating to generate invalid and atypical testing inputs for robustness testing.

B. Control-Flow-based Methods

The control-flow-based methods create control-flow graphs of the tested program using, for example, the static analysis. From these graphs, the tests are generated. A common aim is to achieve a high code coverage, which can be observed for example in [23], [24], or [25].

The method described in [24] is rather basic. It generates input data for the tested program in order to ensure the execution of all branches of the program. The number of generated test cases is limited by the elimination of already explored paths in the control-flow diagram. However, the method is limited to the numerical inputs only. A similar limitation can be also observed in [23].

The control-flow-based methods are often used for web applications. The method described in [25] is designed for the testing of the frontend of web-based applications. It analyzes the content and structure of the investigated website, creates the possible paths of the user, and generates the input testing data for the web forms in order to ensure path coverage. In [26], a method for the generation of test data for testing REST APIs is described. The connected control flow graphs are traversed in order to find patterns of variable usage to produce usable variable values. Another example of the usage for the web application can be found in [27].

The control-flow-based methods are also quite often combined (among other technologies) with the pseudorandom generation of the input data. In [28], stochastic hill climbing is used for the finding the probabilistic distribution. This distribution is then used for the generation of the pseudorandom input testing data. The combination of control-flow diagrams and pseudorandom data generation can be found also in [29].

C. Specification-based Methods

The specification-based methods utilize a form of the specification of the investigated software to generate the test cases. This approach is tempting, as it should compare the actual behavior of the software with the expected behavior given by its specification. The existing methods utilize the UML models (e.g., in [30], [31], [32], or [33]), specification of use cases (e.g., in [31], [34], [35], or [36]), or contracts (e.g., in [37]). Program states description is utilized in [38].

In [30], tests of the entire system are generated from the UML use case and state diagrams. From these diagrams, a usage model is created, which is then used as the basis for the tests. In [32], the activity diagram, the sequence diagram, and the system testing graphs are used to create a combination graph, which is then explored using a modified Depth-First Search (DFS) to generate expected test cases. The contracts in [37] are used similarly to the use case diagrams in [30]. They are transformed into models describing the expected behavior of the investigated program. From this form, the executable test cases are created.

The method described in [35] utilizes textual use case specifications for the generation of acceptance tests. The method is based on natural language processing (NLP) and constraints solving. In [36], the use cases are used to generate a control flow graph and a NLP table, which are, in turn, used for test case generation. The method described in [39] is designed for process-driven applications. The method utilizes analysis of the application and the specification of tests to generate test codes.

D. Program Execution Analysis Methods

The methods based on the program execution analysis utilize the observation of the application behavior in order to generate test cases. There are two main approaches – the approaches based on the instrumentation and on the dynamic symbolic execution (also known as concolic testing).

First approach is based on instrumentation of the tested application in order to enable a simple observation of its behavior. Examples include wrappers around tested functions or methods (e.g., in [40]) or probes near important points of the program, such as control structures (e.g., in [41]), usage of augmented virtual machines (e.g., LLVM [42]), or usage of runtime instrumentation (e.g., in [43]).

Second approach is used for example in [44], [45], [46], [47], [48], or [49]. A dynamic symbolic execution is used in [45] to observe the behavior of the tested application. This observation is used for checking whether new randomly generated input data lead to better path coverage than already stored paths. In [48], the dynamic symbolic execution works with additional attributes enabling to check

the efficiency of the paths produced based on the random input data. It is also possible to check whether the expected boundary values described by the contracts are observed. In [50], the dynamic symbolic execution is used for the testing of C++ Qt Framework classes. A source code preprocessing phase is used to find constructors of Qt classes parameters. A similar approach is used in [51], but for C++ templates.

In [52], automated guided symbolic execution combined with constraint solving is used to avoid exploring useless paths in the program. The method is used for system vulnerability detection. In [44], preprocessing of enterprise applications to enable usage of existing symbolic execution tools for their testing is described. In [53], the tested program is transformed into a set of constraints, which are then solved using a symbolic reasoning engine. So, the approach resembles the dynamic symbolic execution. The evaluation of the CREST concolic testing tool's ability to find real-life errors in real embedded applications is described in [54]. In [43], a concolic test generation tool is combined with the automatic generation of test cases from a formal description of the program (e.g., database table definitions, process-flow diagrams, etc.).

E. Data-Description-based Methods

In some papers, the described methods are not focused on a program, but rather on the specification of the input testing data. This approach is quite common in relation to the increasing number of web-based applications and with the necessity to test their text-based APIs. The frequently used description formats include the Web Services Definition Language (WSDL) used for example in [55], [56], and [57] or the JavaScript Object Notation (JSON) used for example in [58]. XML Schema Definition (XSD) is used in [59].

An interesting comparison is described in [57] where a realistic WSDL-based data set is compared to a fully random data set. The conclusion is that the utilization of realistic data leads to a higher code coverage. In [53], a method for generating complex interconnected data from a WSDL specification is described. The method enables to generate both valid and invalid input data. In [60], a method for the preparation of the test data for web forms utilizes an ontology and types of the fields of the web form. In [61], existing data and rules for their converting were used for testing a data warehouse.

A quite different approach is used in [62]. It uses static analysis of existing tests for mining of literals, which can be suitable as input values in generated tests in a specific domain. Yet another different approach is described in [63]. There, the test cases are generated from inputs specification in natural language. Natural language processing (NLP) and key phrases detection are employed for this purpose.

F. Search-based Methods

A common aim of the search-based methods is to provide high code coverage with a relatively low number of generated test cases. These methods typically do not rely on the knowledge of the program structure, but rather employ various search meta-heuristics to find efficient input test data. Regardless of the utilized meta-heuristic, there must be a way to evaluate the solutions found by the heuristic. Hence, these methods are combined for example with models of the tested program behavior, such as the control flow [64] and event flow [65], or with the program instrumentation [66].

The commonly used meta-heuristics include genetic algorithms, which are employed, for example, in [67], [68], [69], [70], [71], or [72], ant colony optimization (e.g., in [73]), or particle swarm optimization (e.g., in [74] or [75]). A genetic algorithm is used for test data generation for unit testing of Java programs in [67]. In [76], a genetic algorithm is combined with grammar-based fuzzing to generate highly structured testing input data. In [77], a genetic algorithm is combined with random search and database instrumentation to generate test data for SQL queries testing. In [78], a genetic algorithm, an evolutionary algorithm, and an alternating variable method combined with an Object Constraint Language (OCL) description of constraints are investigated.

In [73], the ant colony optimization is employed to achieve higher branch coverage with a relatively small set of testing data. The method is based on the simulation of the pheromone path and is reported to provide better branch coverage than a standard genetic algorithm or particle swarm optimization. In [74], the particle swarm optimization is combined with formal specifications (written in SOFL) and mutation testing. Improved particle swarm optimization is also employed together with predicate functions and path similarity calculation in [75] for test case generation. An unspecified meta-heuristic is employed in [79] together with constraint solving of manually added constraints.

G. Machine-Learning-based Methods

The methods based on machine learning usually utilize artificial neural networks (ANNs) for the test data generation. In [80], a neural network is used for black-box testing of the graphical user interface (GUI) of Android applications. The input of the neural network is a set of screenshots of the tested application. In [81], generative adversarial networks are employed for automated test data generation. A neural network for test generation, which uses the execution trace of the program as an input, is employed in [82]. In [83], the dataset for the neural networks training for source code vulnerability detection is prepared using a mutation approach.

In [84], two approaches for test oracle generation are described. One is based on an artificial neural network and the second is based on data mining from decision trees. The advantages and limitations of both approaches are discussed. In [85], no artificial neural network is used. Instead, random forest, which is a generalization of tree-based classification, is employed for predictive mutation testing.

V. Common Features of Existing Methods

Regardless of the technology utilized by the methods described in Section IV, there are common features and issues of these methods discussed in following subsections.

A. Methods Verification

The lack of verification possibilities or of standard ways how to compare various methods is mentioned in several works (e.g., in [7] or [9]). Based on the investigated papers, it can be concluded that an objective comparison and assessment of the methods cannot be done by using the text of the papers only. Simply, there is not enough information and the provided examples and technologies are quite often vastly different. Some papers (e.g., [29]) contain only a very general description of the verification or testing of the proposed method. Some papers (e.g., [33]) contain no testing at all and focus solely on the description of the proposed method.

Nevertheless, some papers provide means for assessing the quality of the described methods, which are "above average". For example, in [38], [49], [61], or [78], very thorough descriptions of the evaluation process of the proposed methods can be found. It is reported that the evaluation process includes tests performed on realistic programs with actual errors found by the methods. This is in contrast with the majority of the paper, in which the methods are often demonstrated on quite simplified examples (e.g., in [67] or [75]).

B. Implementation Availability

It would be beneficial if the implementations of the methods described in individual papers were available for download and further trials. If this is not possible, a complete data set with data supporting the quality of the described method would be also quite informative. However, from the investigated papers, the majority does not enable to perform a replication study without a reimplementation of the methods from the description in the paper. Of the 67 primary studies referred in this survey, there were only 15 studies with direct links to tools with implementation of the described methods.

From the available tools, 11 tools are provided in the form of GitHub repositories (see Table II) and the remaining 4 tools have dedicated websites. The website of the CREST [54] also contains a link to the GitHub repository along with a downloadable `.zip` file. The website of the CATG [43] contains downloadable `.jar` files. The method described in [72] is implemented in the DCRTT, which appears to be a commercial product, as we were unable to find direct download links on the website. Finally, the website of the SDG [79] contains downloadable `.zip` file. As of May 21 2023, all the links are functional. The available tools are summarized in Table II.

TABLE II DIRECTLY AVAILABLE TOOLS

Ref.	Tool name	Link
[35]	UMTG	https://sntsvv.github.io/UMTG
[41]	Ocelot	https://github.com/ocelab/ocelot
[54]	CREST	https://www.burn.im/crest/
[43]	CATG	https://morioh.com/p/bfdc4686b614
[62]	TestMiner	https://github.com/lucadt/testminer
[72]	DCRTT	https://www.gsse.biz/products/DCRTT
[77]	EvoSQL	https://github.com/SERG-Delft/evosql
[79]	SDG	https://people.svv.lu/tools/SDG
[20]	DOMINO	https://github.com/schemaanalyst/schemaanalyst
[22]	Data-Generators	https://github.com/simonpoulding/DataGenerators.jl
[21]	CYNTHIA	https://github.com/theosotr/cynthia
[80]	Deep GUI	https://github.com/Feri73/deep-gui
[82]	Agilkia	https://github.com/PHILAE-PROJECT/agilkia
[85]	PMT	https://github.com/sei-pku/PredictiveMutationTesting
[19]	SDgen	https://github.com/AussieGuy0/Sdgen

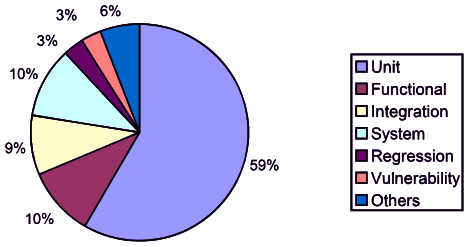

Fig. 1 Percentage of individual testing levels in primary studies

C. Testing Level

As it was stated in Section I, the automated test case generation methods exist for various testing levels. From the primary studies referred in this survey, the vast majority (specifically 39 papers) was focused on unit testing (see Fig. 1), for example [19], [23], [27], [59], or [69]. One of the possible reasons could be that the methods are often demonstrated on quite simple and/or short examples (see Section V.B). Short examples correspond well to unit tests, which usually deal with relatively short part of the source code with limited functionality.

As can be observed in Fig. 1, there were 6 testing levels, which were represented by more than one primary study (including the unit testing). There were papers focused on functional testing (7 papers, e.g., [20], [34], or [80]), integration testing (6 papers, e.g., [26] or [43]), system testing (7 papers, e.g., [25], [29], or [49]), regression testing (2 papers – [39] and [77]), and vulnerability testing (2 papers – [52] and [83]). There were also 4 other testing levels, each represented by a single primary study (4 papers, e.g., [22] or [64]). These papers/methods are grouped as "others" in Fig. 1 and 2.

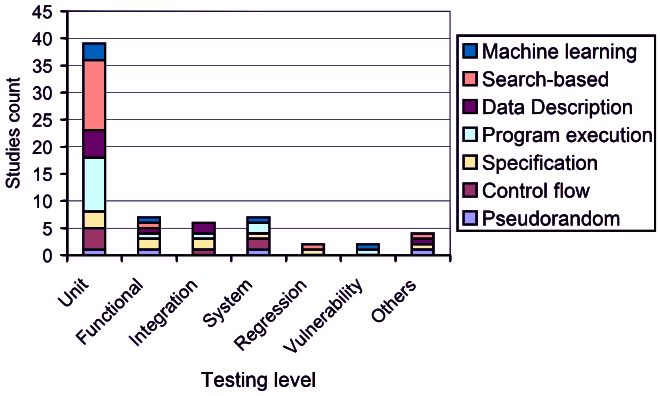

Fig. 2 Main utilized technologies for individual testing levels

In Fig. 2, the portions of the main utilized technologies of the methods for the individual testing levels are depicted. For the unit testing, there are comparatively high numbers of search-based methods (12 papers, e.g., [79], [81], or [84]) and program execution analysis methods (10 papers, e.g., [44], [48], or [51]). Together, they make up more than half of the primary studies focused on unit testing. For other testing levels, the methods are distributed relatively uniformly, but the numbers are too low to draw any further conclusions.

D. Target Platform

The methods described in primary studies are designed for a specific platform, for example for a specific programming language or a specific domain, such as web applications or databases. The methods can be also sufficiently general to be utilizable for multiple platforms. Such general methods usually do not use source code for the generations of the tests, but rather other forms of descriptions of the application, such as UML diagrams (e.g., [35] or [36]). There were 7 target platforms, which were represented by more than one primary study, including the generally utilizable methods (see Fig. 3). The generally utilizable methods also form just the largest group with 19 papers (e.g., [23] or [39]). The specific target platform with the largest number of papers was Java language (18 papers, e.g., [59] or [81]) followed by C/C++ languages (14 papers, e.g., [28] or [42]). Further groups include C#/.NET platform (2 papers – [48] and [56]), web applications (6 papers, e.g., [26] or [55]), databases (DB – 2 papers – [61] and [77]), and programmable logic controllers (PLCs – 2 papers – [46] and [47]). There were also 4 methods designed for other target platforms, each represented by a single primary study (4 papers, e.g., [64] or [74]). These papers/methods are grouped as “others” in Fig. 3 and 4.

In Fig. 4, the portions of the main utilized technologies of the methods for the individual target platforms are depicted. For the Java language, there are mostly search-based (6 papers, e.g., [76] or [79]) and then the machine-learning-based (3 papers – [80], [84], and [85]) and program execution analysis (3 papers – [43], [44], and [53]) methods. The program execution methods are prominent for the C/C++ programming languages (8 papers, e.g., [41] or [50]) and the data-description-based methods for the web applications (4 papers, e.g., [55] or [60]). The generally utilizable methods are mostly specification- (8 papers, e.g., [30] or [36]) and search-based (6 papers, e.g. [65] or [75]).

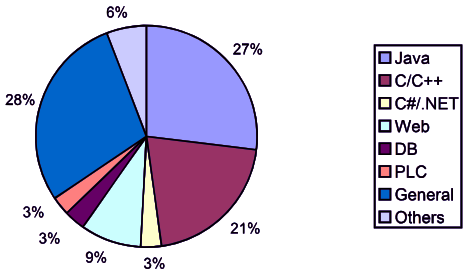

Fig. 3 Percentage of individual target platforms in primary studies

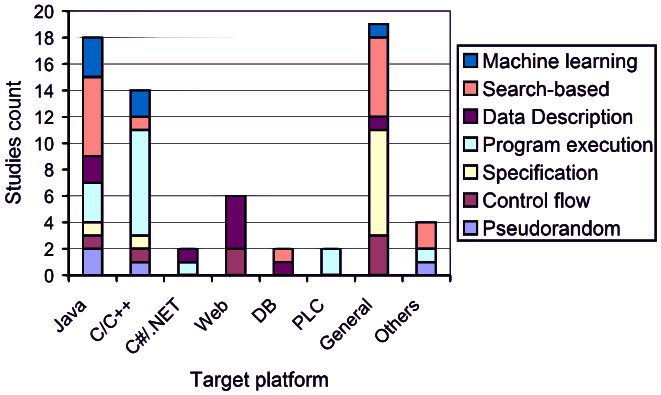

Fig. 4 Main utilized technologies for individual target platforms

E. Observable Trends

Since the time period of the analyzed primary studies is more than two decades (2000 to 2022), there are a few observable trends. Two technologies, which exist for a relatively long time, but are practically used only recently for the test case generation, are natural language processing (e.g., [35] or [36]) and artificial neural networks (e.g., [80] or [81]). Of the primary studies referred in this survey, the oldest study is from 2021 and 2020 for the NLP and the ANNs, respectively. This can be attributed to the relatively recent but significant progress in these fields leading to the practical usability of both technologies.

Another observable trend is the slight increase in the number of studies with direct links to the tools implementing the proposed methods (see Fig. 5). As can be observed in Fig. 5, studies with 11 of 15 available tools were published in 2017 and later. From the primary studies referred in this survey, there was no available tool before 2007.

VI. Threats to Validity

As pointed out in Section I, this survey is not a systematic literature review and does not attempt to answer specific research questions formulated in advance. It also does not attempt to exhaustively list all papers related to the test case or test data generation. Hence, there are papers, which would fit the theme of this survey, but we did not include them. There are several possible reasons:

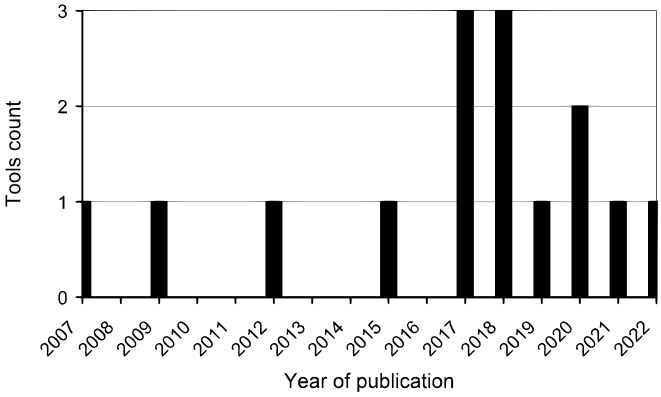

Fig. 5 Number of available tools in individual years

1. The paper was not discovered in the libraries, because it did not pass the utilized filters (see Section III.A).
2. The paper was not present in the two utilized libraries, but may be present in others.
3. The paper was discovered and its full text was read, but because of the similarity to other papers (in the sense of used techniques and/or their combinations), it was not included into the survey.

For the reasons described above, the reader should have in mind that this survey is not exhaustive in any sense, but tries to summarize the approaches and technologies currently in use in the field of automated test data generation.

VII. Conclusion and Future Work

In this paper, the existing literature that deals with test data generation or with tests based on test data generation was summarized. The commonly used approaches were discussed and their common issues and features were described including a few observable trends.

The collected primary studies, which this (non-systematic) survey summarizes, will be used as part of the basis for our future systematic literature review that will cover the theme of this survey, but will add specific research questions and formalization of the entire review process.

Another branch of our current and future work is the creation of a benchmark for the test data generation methods. Such a benchmark would allow us to objectively compare the ability of the methods to find known realistic errors. For this purpose, we are currently developing the Testing Applications Generator (TAG) [86]. This tool is intended to generate applications with selected introduced errors of various types. It enables to introduce errors on the method level meaning that each method can have several different implementations with various introduced errors. The resulting generated application is a general Java application with few limitations and with a structure of the entire project (not only source codes, but also libraries, additional files, and folder structure). The common types of errors should be also obtained during our future research. The tool will be used to create a set of several applications (with several versions each) with multiple introduced errors. This set will serve as the benchmark for automated test generation methods.

References

[1] N. Gupta, A. P. Mathur, and M. L. Soffa, “Generating test data for branch coverage,” in Proceedings ASE 2000 - Fifteenth IEEE International Conference on Automated Software Engineering, Grenoble, September 2000, https://doi.org/10.1109/ASE.2000.873666

[2] P. Fröhlich and J. Link, “Automated Test Case Generation from Dynamic Models,” in ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming, Cannes, June 2000, pp. 472-491, https://doi.org/10.1007/3-540-45102-1_23

[3] B. S. Ahmed, K. Z. Zamli, W. Afzal, and M. Bures, “Constrained Interaction Testing: A Systematic Literature Study,” in IEEE Access, vol. 5, 2017, https://doi.org/10.1109/ACCESS. 2017.2771562

[4] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, “An industrial evaluation of unit test generation: Finding real faults in a financial application,” in Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), Buenos Aires, May 2017, pp. 263–272, https://doi.org/10.1109/ICSE-SEIP.2017.27

[5] J. Edvardsson, “A Survey on Automatic Test Data Generation,” in Proceedings of the Second Conference on Computer Science and Engineering, Linköping, October 1999, pp. 21–28.

[6] 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,” in The Journal of Systems and Software, vol. 86, no. 8, 2013, pp. 1978-2001, https://doi.org/10.1016/j.jss.2013.02.061

[7] 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 Trans. Softw. Eng., vol. 36, no. 6, 2009, pp. 742–762, https://doi.org/10.1109/TSE.2009.52

[8] P. McMinn, “Search-based software test data generation: a survey,” in Softw. Test. Verif. Reliab., vol. 14, no. 2, 2004, pp. 105–156, https://doi.org/10.1002/stvr.294

[9] R. Jeevarathinam and A. S. Thanamani, “A survey on mutation testing methods, fault classifications and automatic test cases generation,” in J. Sci. Ind. Res., vol. 70, no. 2, 2011, pp. 113–117.

[10] T. Chen, X. S. Zhang, S. Z. Guo, H. Y. Li, and Y. Wu, “State of the art: Dynamic symbolic execution for automated test generation,” in Futur. Gener. Comput. Syst., vol. 29, no. 7, 2013, pp. 1758–1773, https://doi.org/10.1016/j.future.2012.02.006

[11] R. M. Parizi, A. A. A. Ghani, R. Abdullah, and R. Atan, “Empirical evaluation of the fault detection effectiveness and test effort efficiency of the automated AOP testing approaches,” in Inf. Softw. Technol., vol. 53, no. 10, 2011, https://doi.org/10.1016/j.infsof. 2011.05.004

[12] S. Popić, B. Pavković, I. Velikić, and N. Teslić, “Data generators: a short survey of techniques and use cases with focus on testing,” in 2019 IEEE 9th International Conference on Consumer Electronics (ICCE-Berlin), Berlin, September 2019, https://doi.org/10.1109/ICCE-Berlin47944.2019.8966202

[13] P. Tramontana, D. Amalfitano, N. Amatucci, and A. R. Fasolino, “Automated functional testing of mobile applications: a systematic mapping study,” in Software Quality Journal, vol. 27, 2019, pp. 149–201, https://doi.org/10.1007/s11219-018-9418-6

[14] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu, “Establishing flight software reliability: testing, model checking, constraint-solving, monitoring and learning,” in Annals of Mathematics and Artificial Intelligence, vol. 70, 2014, pp. 315–349, https://doi.org/10.1007/s10472-014-9408-8

[15] M. Bures, “Automated testing in the Czech Republic: the current situation and issues,” in Proc. 15th Int. Conf. Comput. Syst. Technol., June 2014, pp. 294–301, https://doi.org/10.1145/2659532.2659605

[16] S. J. Galler and B. K. Aichernig, “Survey on test data generation tools: An evaluation of white- and gray-box testing tools for C#, C++, Eiffel, and Java,” in Int. J. Softw. Tools Technol. Transf., vol. 16, no. 6, 2014, pp. 727–751, https://doi.org/10.1007/s10009-013-0272-3

[17] U. R. Molina, F. Kifetew, and A. Panichella, “Java Unit Testing Tool Competition: Sixth round,” in SBST '18: Proceedings of the 11th International Workshop on Search-Based Software Testing, May 2018, pp. 22–29, https://doi.org/10.1145/3194718.3194728

[18] X. Devroey, S. Panichella, and A. Gambi, “Java Unit Testing Tool Competition: Eighth Round,” in Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, June 2020, pp. 545–548, https://doi.org/10.1145/ 3387940.3392265

[19] Y. Zheng, Y. Ma, and J. Xue, “Automated large-scale simulation test-data generation for object-oriented software systems,” in Proceedings of the 1st International Symposium on Data, Privacy, and E-Commerce (ISDPE 2007), Chengdu, November 2007, pp. 74-79, https://doi.org/10.1109/ISDPE.2007.104

[20] A. Alsharif, G. M. Kapfhammer, and P. McMinn, “DOMINO: Fast and Effective Test Data Generation for Relational Database Schemas,” in 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), Västeras, April 2018, pp. 12– 22, https://doi.org/10.1109/ICST.2018.00012

[21] T. Sotiropoulos; S. Chaliasos, V. Atlidakis, D. Mitropoulos, and D. Spinellis, “Data-Oriented Differential Testing of Object-Relational Mapping Systems,” in 2021 IEEE/ACM 43rd International Conferen-

ce on Software Engineering (ICSE), Madrid, May 2021, pp. 1535–1547, https://doi.org/10.1109/ICSE43902.2021.00137

[22] S. Poulding and R. Feldt, "Generating Controllably Invalid and Atypical Inputs for Robustness Testing," in Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Tokyo, March 2017, https://doi.org/10.1109/ICSTW.2017.21

[23] N. T. Sy and Y. Deville, "Automatic test data generation for programs with integer and float variables," in Proc. 16th Annu. Int. Conf. Autom. Softw. Eng. (ASE 2001), San Diego, November 2001, pp. 13–21, https://doi.org/10.1109/ASE.2001.989786

[24] N. Gupta, A. P. Mathur, and M. L. Soffa, "Generating test data for branch coverage," in Proc. ASE 2000 15th IEEE Int. Conf. Autom. Softw. Eng., Grenoble, September 2000, pp. 219–227, https://doi.org/10.1109/ASE.2000.873666

[25] H. Huang, W.-T. Tsai, R. Paul, and Y. Chen, "Automated model checking and testing for composite Web services," in Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), Seattle, May 2005, pp. 300–307, https://doi.org/10.1109/ISORC.2005.16

[26] D. T. Thu, L. D. Quang, D. A. Nguyen, and P. N. Hung, "A Method of Automated Mock Data Generation for RESTful API Testing," in Proceedings - 2022 RIVF International Conference on Computing and Communication Technologies (RIVF 2022), Ho Chi Minh City, December 2022, https://doi.org/10.1109/RIVF55975.2022.10013835

[27] D. T. Thu, D. A. Nguyen, P. N. Hung, "Automated Test Data Generation for Typescript Web Applications," in Proceedings – International Conference on Knowledge and Systems Engineering, Bangkok, November 2021, https://doi.org/10.1109/KSE53942.2021. 9648782

[28] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," in IEEE Trans. Softw. Eng., vol. 36, no. 6, 2010, pp. 763–777, https://doi.org/10.1109/TSE. 2010.24

[29] J. Alava, T. M. King, and P. J. Clarke, "Automatic validation of java page flows using model-based coverage criteria," in Proc. - Int. Comput. Softw. Appl. Conf., Chicaco, September 2006, pp. 439–446, https://doi.org/10.1109/COMPSAC.2006.32

[30] M. Riebisch, I. Philippow, and M. Götze, "UML-Based Statistical Test Case Generation," in LNCS 2591, 2003, pp. 394–411, https://doi.org/ 10.1007/3-540-36557-5_28

[31] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong, "Test Case automate Generation from UML Sequence diagram and OCL expression," in Proc. - 2007 Int. Conf. Comput. Intell. Secur., Harbin, December 2007, pp. 1048–1052, https://doi.org/10.1109/CIS.2007.150

[32] Meiliana, I. Septian, R. S. Alianto, Daniel, and F. L. Gaol, "Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm," in Procedia Computer Science, vol. 116, 2017, pp. 629–637, https://dx.doi.org/10.1016/j.procs.2017.10.029

[33] Y. Zheng, J. Xue, and Y. Zhu, "ISDGen: An automated simulation data generation tool for object-oriented information systems," in 2008 Asia Simul. Conf. - 7th Int. Conf. Syst. Simul. Sci. Comput., Beijing, October 2008, https://doi.org/10.1109/ASC-ICSC.2008. 4675401

[34] M. Zhang, T. Yue, S. Ali, H. Zhang, and J. Wu, "A Systematic Approach to Automatically Derive Test Cases from Use Cases Specified in Restricted Natural Languages," in LNCS, vol. 8769, 2014, pp. 142–157, https://doi.org/10.1007/978-3-319-11743-0_10

[35] C. Wang, F. Pastore, A. Goknil, and L. C. Briand, "Automatic Generation of Acceptance Test Cases from Use Case Specifications: An NLP-Based Approach," in IEEE Trans. on Softw. Eng., vol. 48, no. 2, 2022, https://doi.org/10.1109/TSE. 2020.2998503

[36] M. Lafi, T. Alrawashed, and A. M. Hammad, "Automated Test Cases Generation from Requirements Specification," in 2021 International Conference on Information Technology, Amman, July 2021, https://doi.org/10.1109/ICIT52682.2021.9491761

[37] D. Xu, W. Xu, M. Tu, N. Shen, W. Chu, and C. H. Chang, "Automated Integration Testing Using Logical Contracts," in IEEE Trans. Reliab., vol. 65, no. 3, 2016, pp. 1205–1222, https://doi.org/10.1109/TR.2015.2494685

[38] O. N. Timo and G. Langelier, "Test Data Generation for Cyclic Executives with CBMC and Frama-C: A Case Study," in Electron. Notes Theor. Comput. Sci., vol. 320, 2016, pp. 35–51, https://doi.org/10.1016/j.entcs.2016.01.004

[39] K. Schneid, L. Stapper, S. Thone, and H. Kuchen, "Automated Regression Tests: A No-Code Approach for BPMN-based Process-Driven Applications," in 2021 IEEE 25th International Enterprise Distributed Object Computing Conferenc (EDOC), Gold Coast, October 2021, https://doi.org/10.1109/EDOC52215.2021.00014

[40] C. Fetzer and Z. Xiao, "An automated approach to increasing the robustness of C libraries," in Proc. 2002 Int. Conf. Dependable Syst. Networks, Washington D.C., June 2002, pp. 155–164, https://doi.org/10.1109/DSN.2002.1028896

[41] S. Scalabrino, M. Guerra, G. Grano, A. De Lucia, R. Oliveto, D. D. Nucci, and H. C. Gall, "Ocelot: A search-based test-data generation tool for C," in ASE 2018 - Proceedings of the 33rd ACM/IEEE Int. Conf. on Autom. Softw. Eng., Montpellier, September 2018, pp. 868-871, https://doi.org/10.1145/ 3238147.3240477

[42] H. Riener and G. Fey, "FAuST: A framework for formal verification, automated debugging, and software test generation," in LNCS, vol. 7385, 2012, https://doi.org/10.1007/978-3-642-31759-0_17

[43] H. Tanno, X. Zhang, T. Hoshino, and K. Sen, "TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications," in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, May 2015, pp. 717–720, https://doi.org/10.1109/ICSE.2015.231

[44] H. Ohbayashi, H. Kanuka, and C. Okamoto, "A Preprocessing Method of Test Input Generation by Symbolic Execution for Enterprise Application," in 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, December 2018, https://doi.org/10.1109/APSEC.2018.00104

[45] T. Su et al., "Automated Coverage-Driven Test Data Generation Using Dynamic Symbolic Execution," in 2014 Eighth Int. Conf. Softw. Secur. Reliab., San Francisco, June 2014, pp. 98–107, https://doi.org/10.1109/SERE.2014.23

[46] L. Hao, J. Shi, T. Su, and Y. Huang, "Automated Test Generation for IEC 61131-3 ST Programs via Dynamic Symbolic Execution," in 2019 International Symposium on Theoretical Aspects of Software Engineering (TASE), Guilin, July 2019, https://doi.org/10.1109/TASE.2019.00004

[47] W. He, J. Shi, T. Su, Z. Lu, L. Hao, and Y. Huang, "Automated test generation for IEC 61131-3 ST programs via dynamic symbolic execution," in Science of Computer Programming, vol. 206, 2021, https://doi.org/10.1016/j.scico.2021.102608

[48] K. Jamrozik, G. Fraser, N. Tillman, and J. De Halleux, "Generating test suites with augmented dynamic symbolic execution," in LNCS, vol. 7942, 2013, pp. 152–167, https://doi.org/10.1007/978-3-642-38916-0_9

[49] B. Chen, Z. Yang, L. Lei, K. Cong, and F. Xie, "Automated Bug Detection and Replay for COTS Linux Kernel Modules with Concolic Execution," in 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London (Canada), February 2020, https://10.1109/SANER48275.2020.9054797

[50] T. A. Bui, L. N. Tung, H. V. Tran, and P. N. Hung, "A Method for Automated Test Data Generation for Units using Classes of Qt Framework in C++ Projects," in 2022 RIVF International Conference on Computing and Communication Technologies (RIVF), Ho Chi Minh City, December 2022, https://10.1109/RIVF55975.2022.10013869

[51] M. H. Do, L. N. Tung, H. V. Tran, and P. N. Hung, "An Automated Test Data Generation Method for Templates of C++ Projects," in 2022 14th International Conference on Knowledge and Systems Engineering (KSE), Nha Trang, October 2022, https://doi.org/10.1109/ KSE56063.2022.9953626

[52] T. Liu, Z. Wang, Y. Zhang, Z. Liu, B. Fang, and Z. Pang, "Automated Vulnerability Discovery System Based on Hybrid Execution," in 2022 7th IEEE International Conference on Data Science in Cyberspace (DSC), Guilin, July 2022, pp. 234-241, https://doi.org/10.1109/DSC55868.2022.00038

[53] K. Li, C. Reichenbach, Y. Smaragdakis, Y. Diao, and C. Csallner, "SEDGE: Symbolic example data generation for dataflow programs," in 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, November 2013, https://doi.org/10.1109/ASE.2013.6693083

[54] M. Kim, Y. Kim, and Y. Jang, "Industrial application of concolic testing on embedded software: Case studies," in 2012 IEEE Fifth Inte-

rnational Conference on Software Testing, Verification and Validation, Montreal, April 2012, https://doi.org/10.1109/ICST.2012.119

[55] C. Ma, C. Du, T. Zhang, F. Hu, and X. Cai, "WSDL-Based Automated Test Data Generation for Web Service," in 2008 Int. Conf. Comput. Sci. Softw. Eng., Wuhan, December 2008, pp. 731–737, https://doi.org/10.1109/CSSE.2008.790

[56] W. Krenn and B. K. Aichernig, "Test Case Generation by Contract Mutation in Spec#," in Electron. Notes Theor. Comput. Sci., vol. 253, no. 2, 2009, pp. 71–86, https://doi.org/10.1016/j.entcs.2009.09.052

[57] M. Bozkurt and M. Harman, "Automatically generating realistic test input from web services," in Proc. - 6th IEEE Int. Symp. Serv. Syst. Eng., Irvine, December 2011, pp. 13–24, https://doi.org/10.1109/SOSE.2011.6139088

[58] A. Arcuri, "RESTful API Automated Test Case Generation," in 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, July 2017, pp. 9–20, https://doi.org/10.1109/QRS.2017.11

[59] N. Havrikov, A. Gambi, A. Zeller, A. Arcuri, and J. P. Galeotti, "Generating unit tests with structured system interactions," in 2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST), Buenos Aires, May 2017, pp. 30–33, https://doi.org/10.1109/AST.2017.2

[60] S. Hanna and H. Jaber, "An Approach for Web Applications Test Data Generation Based on Analyzing Client Side User Input Fields," in 2019 2nd International Conference on new Trends in Computing Sciences (ICTCS), Amman, October 2019, https://doi.org/10.1109/ICTCS.2019.8923098

[61] H. M. Sneed and K. Erdoes, "Testing big data (Assuring the quality of large databases)," in 2015 IEEE Eighth Int. Conf. Softw. Testing, Verif. Valid. Work., Graz, April 2015, pp. 1–6, https://doi.org/10.1109/ICSTW.2015.7107424

[62] L. D. Toffola, C. A. Staicu, and M. Pradel, "Saying 'Hi!' is not enough: Mining inputs for effective test generation," in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, October 2017, https://doi.org/10.1109/ASE.2017.8115617

[63] T. Li, X. Lu, and H. Xu, "Automated Test Case Generation from Input Specification in Natural Language," in 2022 IEEE International Symposium on Software Reliability Engineering Workshops, Charlotte, October 2022, https://doi.org/10.1109/ISSREW55968.2022.00076

[64] T. Shu, Z. Ding, M. Chen, and J. Xia, "A heuristic transition executability analysis method for generating EFSM-specified protocol test sequences," in Information Sciences, vol. 370–371, 2016, pp. 63–78, https://doi.org/10.1016/j.ins.2016.07.059

[65] A. Rauf, S. Anwar, M. A. Jaffer, and A. A. Shahid, "Automated GUI test coverage analysis using GA," in 7th Int. Conf. Inf. Technol. New Gener., Las Vegas, April 2010, pp. 1057–1062, https://doi.org/10.1109/ ITNG.2010.95

[66] S. Khor and P. Grogono, "Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically," in 19th Int. Conf. Autom. Softw. Eng., Linz, September 2004, pp. 346–349, https://doi.org/10.1109/ASE.2004.1342761

[67] Z. J. Rashid and M. Fatih Adak, "Test Data Generation for Dynamic Unit Test in Java Language using Genetic Algorithm," in 6th International Conference on Computer Science and Engineering (UBMK), Ankara, September 2021, https://doi.org/10.1109/UBMK52708.2021.9558953

[68] E. Diaz, J. Tuya, and R. Blanco, "Automated software testing using a metaheuristic technique based on Tabu search," in 18th IEEE Int. Conf. Autom. Softw. Eng., Montreal, October 2003, pp. 310–313, https://doi.org/10.1109/ASE.2003.1240327

[69] J. Khandelwal and P. Tomar, "Approach for automated test data generation for path testing in aspect-oriented programs using genetic algorithm," in Int. Conf. Comput. Com. Autom., Greater Noida, May 2015, pp. 854–858, https://doi.org/10.1109/CCAA.2015.7148494

[70] B. L. Li, Z. S. Li, J. Y. Zhang, and J. R. Sun, "An Automated Test Case Generation Approach by Genetic Simulated Annealing Algorithm," in Third Int. Conf. Nat. Comput., Haikou, August 2007, pp. 106–111, https://doi.org/10.1109/ICNC.2007.187

[71] Z. J. Rashid and M. F. Adak, "Test Data Generation for Dynamic Unit Test in Java Language using Genetic Algorithm," in 2021 6th International Conference on Computer Science and Engineering (UBMK), Ankara, September 2021, http://dx.doi.org/10.1109/UBMK52708. 2021.9558953

[72] R. Gerlich and C. R. Prause, "Optimizing the Parameters of an Evolutionary Algorithm for Fuzzing and Test Data Generation," in 2020 IEEE 13th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto, October 2020, https://doi.org/10.1109/ICSTW50294.2020.00061

[73] H. Sharifipour, M. Shakeri, and H. Haghighi, "Structural test data generation using a memetic ant colony optimization based on evolution strategies," in Swarm Evol. Comput., vol. 40, 2018, pp. 76-91, https://doi.org/10.1016/j.swevo.2017.12.009

[74] R. J. Cajica; R. E. G. Torres, and P. M. Álvarez, "Automatic Generation of Test Cases from Formal Specifications using Mutation Testing," in 18th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE), Mexico City, November 2021, http://dx.doi.org/10.1109/CCE53527.2021.9633118

[75] H. Cui, L. Chen, B. Zhu, and H. Kuang, "An efficient automated test data generation method," in 2010 International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), Changsha, March 2010, https://doi.org/10.1109/ICMTMA.2010.556

[76] M. Olsthoorn, A. van Deursen, and A. Panichella, "Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing," in ASE '20: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, December 2020, pp. 1224-1228, http://dx.doi.org/10.1145/3324884.3418930

[77] J. Castelein, M. Aniche, M. Soltani, A. Panichella, and A. Van Deursen, "Search-based test data generation for SQL queries," in Proceedings of the 40th International Conference on Software Engineering, Gothenburg, May 2018, pp. 1220–1230, https://doi.org/10.1145/3180155.3180202

[78] S. Ali, M. Zohaib Iqbal, A. Arcuri, and L. C. Briand, "Generating test data from OCL constraints with search techniques," in IEEE Transactions on Software Engineering, vol. 39, no. 10, 2013, pp. 1376–1402, https://doi.org/10.1109/TSE.2013.17

[79] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Synthetic data generation for statistical testing," in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, October 2017, https://doi.org/10.1109/ASE.2017.8115698

[80] F. Y. B. Daragh and S. Malek, "Deep GUI: Black-box GUI Input Generation with Deep Learning," in 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, November 2021, pp. 905–916, https://doi.org/10.1109/ASE51524.2021.9678778

[81] X. Guo, H. Okamura, and T. Dohi , "Automated Software Test Data Generation With Generative Adversarial Networks," in IEEE Access, vol. 10, 2022, https://doi.org/10.1109/ACCESS. 2022.3153347

[82] M. Utting, B. Legeard,F. Dadeau, F. Tamagnan, and F. Bouquet, "Identifying and Generating Missing Tests using Machine Learning on Execution Traces," in 2020 IEEE International Conference On Artificial Intelligence Testing (AITest), Oxford, August 2020, https://doi.org/10.1109/AITEST49225.2020.00020

[83] K. Cheng, G. Du, T. Wu, L. Chen, and G. Shi, "Automated Vulnerable Codes Mutation through Deep Learning for Variability Detection," in 2022 International Joint Conference on Neural Networks (IJCNN), Padua, July 2022, https://doi.org/10.1109/IJCNN55064.2022.9892444

[84] Vineeta, A. Singhal, and A. Bansal, "Generation of test oracles using neural network and decision tree model," in 2014 5th Int. Conf. - Conflu. Next Gener. Inf. Technol. Summit, Noida, September 2014, pp. 313–318, https://doi.org/10.1109/CONFLUENCE.2014.6949311

[85] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia,and L. Zhang, "Predictive Mutation Testing," in IEEE Transactions on Software Engineering, vol. 45, no. 9, 2019, pp. 898–918, https://doi.org/10.1109/TSE.2018.2809496

[86] T. Potuzak and R. Lipka, "Generation of Benchmark of Software Testing Methods for Java with Realistic Introduced Errors" in FedCSIS 2023 communication papers, September 2023, to be published