Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing

Mitchell Olsthoorn
Delft University of Technology
Delft, The Netherlands
M.J.G.Olsthoorn@tudelft.nl

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
Arie.vanDeursen@tudelft.nl

Annibale Panichella
Delft University of Technology
Delft, The Netherlands
A.Panichella@tudelft.nl

ABSTRACT

Software testing is an important and time-consuming task that is often done manually. In the last decades, researchers have come up with techniques to generate input data (e.g., fuzzing) and automate the process of generating test cases (e.g., search-based testing). However, these techniques are known to have their own limitations: search-based testing does not generate highly-structured data; grammar-based fuzzing does not generate test case structures. To address these limitations, we combine these two techniques. By applying grammar-based mutations to the input data gathered by the search-based testing algorithm, it allows us to co-evolve both aspects of test case generation. We evaluate our approach, called G-EvoSuite, by performing an empirical study on 20 Java classes from the three most popular JSON parsers across multiple search budgets. Our results show that the proposed approach on average improves branch coverage for JSON related classes by 15 % (with a maximum increase of 50 %) without negatively impacting other classes.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Software testing and debugging**.

KEYWORDS

search-based software testing, test case generation, grammar-based fuzzing, unit testing

ACM Reference Format:
Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3418930

1 INTRODUCTION

Software testing is a critical activity for quality assurance and can take up to 50 % of developers' time [8]. Manually writing test cases that are meaningful and small in size is an expensive and error-prone task. With the ever-increasing complexity of modern applications, designing meaningful test cases with high coverage becomes harder each day. As a consequence, researchers have developed various techniques to automate the generation of test cases over the last decades [22]. Recent advances show that search-based approaches can achieve higher code coverage compared to manually written test cases [23, 28]. They can also detect unknown bugs [1, 3, 14] and have been successfully used in industry (e.g., [4, 7, 21]). Moreover, automatic test case generation significantly reduces the time needed for testing and debugging [31].

Search-based test case generation relies on evolutionary algorithms (EAs) to evolve an initial pool of randomly generated test cases, which include both the test structure and input data. While recent studies improved the effectiveness of EAs, automatic test case generation has limitations on creating highly-structured input data. Previous work shows that automatically generated inputs are usually unstructured and can be difficult to read and interpret [2, 11]. These limitations are critical when testing applications with highly-structured input data. Parsers are a typical example of such applications. With the move towards Application Programming Interfaces (APIs) and microservices, many systems nowadays heavily rely on parsers [5]. Common data formats for these APIs are JavaScript Object Notation (JSON) and Extensible Markup Language (XML) and are used to exchange data among different parts of applications. For this reason, properly testing these parsers is critical for application testing [17].

Grammar-based fuzzing is very effective in generating highly-structured input data based on a user-specified grammar [15, 33]. For this reason, fuzzing has been widely used for security and system testing [9, 16]. When applied to data formats, fuzzers can generate and manipulate well-formed input data. However, developers need to specify the entry points (for system testing) and manually create a test structure for each method under test.

In this paper, we address these limitations by combining the strength of grammar-based fuzzing and search-based test case generation with a focus on the JSON data format. More precisely, evolutionary algorithms create and evolve the test case structure (statement sequence) while grammar-based fuzzing is used to evolve parts of the input data. The fuzzer injects structured JSON inputs in the initial population of the EA with some probability and manipulates this data to maintain a well-formed JSON structure.

To assess the efficacy and feasibility of our idea, we implemented the grammar-based fuzzing approach in EvoSuite [13], a state-of-the-art test case generator for Java. We conducted an empirical study with 20 classes from the three most popular JSON parser libraries, namely `GSON`, `fastjson`, and `org.json`. In particular, we

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3418930

selected 16 classes that expect JSON input and 4 non-JSON related classes. We use the former group to assess whether our approach improves code coverage and use the latter to assess whether our approach is negatively impacting coverage for non-JSON related classes. We evaluate the performance (code coverage) for different search budgets (60 s, 120 s, and 180 s) to measure the effectiveness and efficiency over time.

Our preliminary results show that combining search-based testing with grammar-based fuzzing leads to higher code coverage for classes that parse and manipulate JSON without decreasing code coverage for non-JSON related classes (i.e., it has no side effect). On average, our approach achieves +15 % of branch coverage compared to standard EvoSuite (without fuzzing). In our experiment, the improvement on the branch coverage is up to 50 % for the class `JSONReader` from the `fastjson` project with a search budget of 180 seconds. This confirms the feasibility of our approach and the benefits of combining the strengths of different techniques that are often considered as alternatives rather than complementary solutions. While our approach is applied to the JSON data format, it can be extended and generalized to other data formats. We foresee further work in this line of research.

In summary, we make the following contributions: (i) a novel approach that combines grammar-based fuzzing and search-based software testing to maximize the code coverage in JSON parsers in a shorter amount of time; (ii) an empirical evaluation involving 3 major Java JSON projects that shows the effectiveness and efficiency of the proposed approach; (iii) We provide a full replication package including our code and results [24].

2 BACKGROUND AND RELATED WORK

In this section, we briefly describe the related work in the fields of test case generation and grammar-based fuzzing. We also describe the pros and cons of the two testing strategies.

Search-based Test Case Generation. Various search-based test case generation approaches have been proposed in the literature (e.g., [13, 22, 29]). These approaches rely on test adequacy criteria (e.g., branch coverage [22, 30]) and evolutionary algorithms (e.g., genetic algorithms [10, 22, 27]). Adequacy criteria are used to define search heuristics (or objectives) to optimize. For example, *approach level* and *branch distance* are well-known heuristics (or *objectives*) for *line* and *branch* coverage [22]. Evolutionary algorithms evolve test data or test cases and use the heuristics as guidance toward generating tests with high coverage and fault detection.

EvoSuite is a state-of-the-art test case/suite generation tool for Java. EvoSuite implements several evolutionary algorithms (AEs), such as monotonic genetic algorithms, local solvers, and many-objective algorithms [10]. EvoSuite can optimize multiple adequacy criteria simultaneously [26, 30]. We use EvoSuite as the starting point to implement our approach and also as the baseline in our empirical evaluation. Among the evolutionary algorithms available in EvoSuite, we choose the Dynamic Many-Objective Sorting Algorithm (DynaMOSA) [25]. DynaMOSA evolves test cases and optimizes multiple coverage targets (e.g., branches) simultaneously. We opted for DynaMOSA since recent studies showed its better effectiveness and efficiency compared to other EAs for testing [10, 27]. DynaMOSA uses a many-objective genetic algorithm that encodes test cases as *chromosomes*. Each chromosome is a sequence of statements (constructor, method invocation, primitive statements, and assignments) with variable length. Hence, the structure of the test cases evolves across the generations. The *single-point crossover* generates new test cases by recombining statements (genes) from the parent tests. The *uniform mutation* further modifies the offspring by adding, removing, or replacing statements. Lastly, DynaMOSA selects the fittest chromosomes using the *preference criterion*, the *non-dominance relation*, and the *crowding distance* [25].

Grammar-based fuzzing. Whitebox fuzzing is another method to automate software testing. Differently from test case generation, *fuzzing* focuses on generating test data (rather than test case structures), and it is very popular in security testing to find security vulnerabilities in software [12]. To fuzz, developers need to specify the entry points of the application under test. White-box fuzzing aims to generate test inputs that, when applied to specified entry points, allow satisfying/covers different program conditions [20]. Fuzzing can use different engines for the test data generation [20], such as symbolic execution [9], metaheuristics [32], grammars [15], and hybrid approaches [19].

Grammar-based fuzzing generates well-formed inputs by relying on a user-specified grammar [15, 34]. It creates random variants of well-specified inputs using the grammar derivative rules (hereafter called grammar-based mutations). This guarantees that the variants are still well-formed but diverse [20]. Typically, the grammars encode application-specific knowledge of the program under test [15]. As shown by previous work, grammar-based fuzzers are very effective in creating highly-structured inputs for applications like compilers and interpreters [15, 33].

Reasons for combining. On the one hand, search-based testing allows synthesizing the test case structure without requiring to specify the program entry points. It can evolve complex input data like `Objects` in Java, primitive data types, and strings. However, it is not effective in generating highly-structured input strings, such as the JSON data format. On the other hand, grammar-based fuzzing can effectively generate highly-structured input strings. However, it requires the user to specify both the entry points of the program under tests and the grammar. Besides, programs can have multiple entry points, not all requiring the same type of grammar. In our case, JSON parsers have some entry points (methods) that require data in JSON formats but also other entry points with different input types, such as complex Objects, or primitives.

3 APPROACH

Our approach, called G-EvoSuite, aims to combine the strengths of search-based test case generation and grammar-based fuzzing. We use EvoSuite as the test case generator tool, and we implemented a JSON fuzzer, i.e., fuzzer based on JSON grammar. The fuzzer is built on top of SNODGE [1], a mutation engine for JSON strings. Our approach uses EvoSuite to create and evolve the test case structures and the JSON fuzzer to generate highly-structured input strings when needed. To implement our approach, we modified DynaMOSA in EvoSuite (see Section 2). In the following paragraphs, we explain the changes we introduced in DynaMOSA to incorporate the grammar-fuzzer.

[1] https://github.com/npryce/snodge

Initialization. To start the evolutional process, DynaMOSA creates an initial genetic pool of test cases. The initialization routine is designed to generate a well-distributed set of tests that call different methods of the target class. Each test is created in DynaMOSA randomly by adding method calls to the class under test. Before inserting each method call m, EvoSuite also instantiates an object of the class containing m and generates proper input parameters, such as other objects or primitives. The number of method calls to insert in an initial test case is randomly chosen. Therefore, EvoSuite creates different initial tests with different structures (method sequence). The input data is either generated at random or selected from the literals (constants) that statically appear in the class under test (*constant pool*).

Our approach modifies the initialization phase by using well-formed JSON strings generated with the fuzzer as test data. Injecting JSON strings in every method call with string inputs is not effective because not all methods under tests (or not all input parameters of the same method) require JSON inputs. Therefore, we inject JSON strings only into a portion of the initial population. Given a population $P = \{T_1, \ldots, T_N\}$ of size N, we randomly select test cases from P and inject them with JSON data. Given an initial test T to modify, its string inputs have a probability of mutating equal to $p = 1/k$, where k is the number of input strings in T.

Selection. In each generation, the fittest test cases (structure + data) are selected using *tournament selection*. These test cases are ranked on different code coverage criteria (Line, Branch, Exception, Weak Mutation) using the *preference sorting* algorithm [25]. If test cases with JSON data are ranked first, they will be selected for reproduction (i.e., to create new tests) and will survive in the next generations. If not, the genetic characteristics of the tests with the JSON files will not be transmitted to the next generations. In this way, the portion of test cases that are created using the fuzzer changes across the generations depending on whether they are useful to improve coverage or not.

Grammar-based mutation. To introduce variation in the genetic pool, DynaMOSA makes use of mutations. The use of mutations makes it less likely for the algorithm to reach a local optimum. In our approach, we extend the *uniform mutation* in DynaMOSA, which adds, removes, or inserts new statements in each newly generated test T. At the end of the uniform mutation, we inspect all input data in T, identify string inputs, and use JSON parsers to check whether they are well-formed JSON strings. If valid JSON strings are found, we mutate them using the grammar-based fuzzer.

A well-formed JSON string is a sequence of ⟨`key`, `value`⟩ pairs. `Keys` must be strings, and `values` must be one of the following JSON data types: string, number, object, array, boolean, or null. Based on this structure, we define five different mutation operators:

- *Adding new* ⟨`key`, `value`⟩ *pairs*: it adds a ⟨`key`, `value`⟩ pair at the root level of the JSON structure. The `key` is generated using the *constant pool* in EvoSuite. The `value` is randomly generated and can be any of the JSON data types.
- *Adding JSON objects*: it adds a new JSON object as a value to an existing ⟨`key`, `value`⟩ pair in a random position.
- *Removing* ⟨`key`, `value`⟩ *pairs*: it randomly removes a ⟨`key`, `value`⟩ pair in the JSON structure. Which element is removed is randomly selected.
- *Modifying* ⟨`key`, `value`⟩ *pairs*: This mutation randomly selects a ⟨`key`, `value`⟩ pair from the JSON structure, and mutates either the `key` or the `value`. The replacing element is proportionately divided across all JSON primitives. The array and dictionary primitives are replaced as is. The other primitives are sourced from the *constant pool* of EvoSuite.
- *Reordering* ⟨`key`, `value`⟩ *pairs*: it randomly shuffles the ⟨`key`, `value`⟩ pairs inside the JSON structure. The pair to be reordered is selected randomly. The new location is also selected randomly.

The five operators can be applied to a test T with equal probability. If the test case T contains multiple JSON strings, each string has a probability of being replaced equal to $p = 1/k$, where k is the number of well-formed JSON strings in T.

4 EMPIRICAL STUDY

This section details the empirical study we conducted to assess the performance of our approach (hereafter G-EvoSuite) compared to standard test case generation (EvoSuite). Our empirical evaluation is steered by the following research questions:

RQ1 *To what extend does grammar-based fuzzing improve the effectiveness of test case generation in EvoSuite?*

RQ2 *What is the effectiveness of combining grammar-based fuzzing and search-based testing over different search budgets?*

For our empirical study, we selected a total of 20 classes from the three most popular Java JSON parsers. These parsers are the GSON library from Google, fastjson from Alibaba, and the org.json standard library. 16 classes are related to JSON data. This was determined based on class name and by manually inspecting the individual classes. The remaining four classes (indicated with an asterisk in Table 1) are used to assess whether our approach negatively impacts classes not related to parsing JSON.

Search budget. To assess the effectiveness of our approach over different search budgets, we selected three commonly-used values: 60 seconds, 120 seconds, and 180 seconds [3, 18, 23].

Parameter setting. For this study, we have chosen to adopt the default search algorithm parameter values set by EvoSuite. Previous studies have shown that although parameter tuning has an impact on the performance of the search algorithm, the default parameters provide a reasonable and acceptable result [6]. The parameters used for both the EvoSuite and G-EvoSuite approaches are: population size of 50 test cases; single-point crossover with a probability of 0.75; mutation with a probability of $1/n$, where n is the number of statements in the test case; and tournament selection, the default selection operator in EvoSuite.

Statistical analysis. Since both approaches used in the study are randomized, we can expect a fair amount of variation in the results. To mitigate this, ever experiment has been repeated 20 times so an average can be taken. To determine if the results are statistically significant, we use the unpaired Wilcoxon test with a threshold of 0.05. This is a non-parametric statistical test that determines if two data distributions (coverage values by the two approaches) are significantly different. We combine this with the Vargha-Delaney statistic to measure the effect size, which determines how large the difference between the two approaches is.

Table 1: Median branch coverage achieved by our approach (G-EvoSuite) and the baseline (EvoSuite) over 20 independent runs. We report the *p*-values produced by the Wilcoxon test together with the Vargha-Delaney statistics ($\hat{A}_{12}$). For the effect size, we use the labels S, M, and L to denote *small*, *medium*, and *large* effect size. $\hat{A}_{12} > 0.50$ indicates a positive effect size.

ID	Class Under Test	60 s				120 s				180 s			
		EvoSuite	G-EvoSuite	*p*-value	$\hat{A}_{12}$	EvoSuite	G-EvoSuite	*p*-value	$\hat{A}_{12}$	EvoSuite	G-EvoSuite	*p*-value	$\hat{A}_{12}$
1	fastjson.JSON	0.76	0.74	0.12	S (0.35)	0.81	0.81	0.51	— (0.56)	0.82	0.83	0.01	M (0.73)
2	fastjson.JSONArray	0.77	0.76	0.35	S (0.41)	0.83	0.82	0.53	— (0.44)	0.82	0.84	0.17	S (0.62)
3	fastjson.JSONObject	0.49	0.49	0.94	— (0.51)	0.51	0.52	0.26	S (0.53)	0.52	0.53	0.14	S (0.62)
4	fastjson.JSONPath	0.36	0.36	0.92	— (0.48)	0.41	0.41	0.06	M (0.30)	0.42	0.44	0.09	M (0.67)
5	fastjson.JSONReader	0.22	0.22	0.07	M (0.67)	0.23	0.70	<0.01	L (0.89)	0.23	0.73	<0.01	L (0.83)
6	fastjson.JSONValidator	0.52	0.75	<0.01	L (1.00)	0.58	0.83	<0.01	L (1.00)	0.59	0.84	<0.01	L (1.00)
7	fastjson.DefaultJSONParser	0.28	0.50	<0.01	L (1.00)	0.33	0.58	<0.01	L (1.00)	0.36	0.61	<0.01	L (1.00)
8	fastjson.JSONReaderScanner	0.72	0.72	0.82	— (0.48)	0.75	0.76	0.72	— (0.53)	0.77	0.78	0.02	M (0.70)
9	fastjson.JSONScanner	0.31	0.36	<0.01	L (0.90)	0.34	0.44	<0.01	L (0.95)	0.35	0.44	<0.01	L (0.98)
10*	gson.Gson	0.77	0.79	0.02	M (0.72)	0.81	0.81	0.55	— (0.44)	0.81	0.82	0.30	S (0.59)
11	gson.JsonTreeReader	0.88	0.89	0.76	— (0.53)	0.90	0.90	0.54	— (0.44)	0.90	0.91	0.37	S (0.43)
12	gson.JsonTreeWriter	0.91	0.91	1.00	— (0.50)	0.91	0.91	0.60	— (0.52)	0.91	0.91	0.77	— (0.49)
13*	gson.LinkedHashTreeMap	0.43	0.43	0.71	— (0.47)	0.50	0.47	0.20	S (0.38)	0.50	0.51	0.60	— (0.54)
14	gson.JsonReader	0.68	0.74	<0.01	L (0.98)	0.72	0.78	<0.01	L (1.00)	0.73	0.80	<0.01	L (0.97)
15	gson.JsonWriter	0.90	0.90	0.95	— (0.49)	0.91	0.91	0.77	— (0.47)	0.91	0.91	0.70	— (0.47)
16	json.JSONArray	0.74	0.77	0.03	M (0.70)	0.78	0.82	0.01	M (0.73)	0.80	0.81	0.15	S (0.62)
17	json.JSONObject	0.66	0.69	0.02	M (0.72)	0.74	0.77	<0.01	L (0.86)	0.75	0.78	<0.01	L (0.89)
18	json.JSONTokener	0.78	0.82	0.17	S (0.63)	0.83	0.88	0.25	S (0.60)	0.89	0.91	<0.01	L (0.75)
19*	json.XML	0.75	0.76	0.82	— (0.52)	0.77	0.77	0.77	— (0.47)	0.77	0.78	0.12	S (0.63)
20*	json.XMLTokener	0.99	0.99	0.70	— (0.54)	0.99	0.99	0.87	— (0.52)	0.99	0.99	0.15	S (0.61)

4.1 Results

Table 1 summarizes the results of the comparison between EvoSuite and G-EvoSuite. The table is divided into the three different search budgets used for the empirical evaluation. For each search budget, we show the median branch coverage for the baseline, EvoSuite, and G-EvoSuite, the statistical significance produced by the Wilcoxon test, and the effect size with the Vargha-Delaney statistic. In the table, we denote the classes with a negligible effect size with "—", and highlight results that are statistically significant with a gray color. Next, we discuss the results for each search budget separately.

For 60 seconds, our approach achieves significantly higher coverage than EvoSuite in seven out of 20 classes. The effect size is large in four cases and medium in the other three cases. The average improvement in branch coverage with the G-EvoSuite approach is 9.02 %. The class with the least improvement is `gson.Gson` with an average improvement of 1.77 %. The class with the most improvement is `JSONValidator` (ID=6) with an average improvement of 23.35 % which corresponds to 46 additionally covered branches.

For 120 seconds, seven out of 20 classes show a significant improvement with our approach. The effect size is large in six cases and medium in only one case. The average improvement in branch coverage is 17.1 %. The class with the least improvement is `JSONArray` (ID=16) with an average increase of 3.20 %. The most improved class is `JSONReader` (ID=5) with an average increase of 47.83 % resulting in 49 branches being covered additionally.

Lastly, for 180 seconds, our approach significantly outperforms EvoSuite in nine out of 20 classes. The effect size is large in seven cases and medium in two cases. The average improvement in branch coverage is 13.6 %, with a minimum of +1 % for `JSONReaderScanner` and a maximum of 50.87 % (+52 branches) for the class `JSONReader` (ID=5). In terms of the number of covered branches, the biggest improvement (+166 branches) can be observed for `DefaultJSONParser`.

It is worth to notice that in none of the classes, we observed a decrease in branch coverage when using G-EvoSuite. This shows that our approach improves the overall effectiveness of test case generation in EvoSuite without negatively impacting coverage of non-JSON related classes (**RQ1**).

When looking at how the two approaches perform over time, we can see that the delta between EvoSuite and G-EvoSuite does not substantially decrease, and in most cases even increases. For example, the `JSONReader` (ID=5) class shows that the delta of the branch coverage goes from 0 % at 60 s to 50 % at 180 s. This shows that just injecting JSON strings in the initial population is not sufficient to reach a higher coverage. Otherwise, we would have observed a large difference already at the 60 s search budget. Therefore, for our benchmark, the benefit of combining search-based testing and grammar-based fuzzing increases with time (**RQ2**).

5 CONCLUSIONS AND FUTURE WORK

In this paper, we have combined search-based testing with grammar based-fuzzing to achieve higher code coverage for programs with highly-structured inputs. We implemented our approach in EvoSuite and evaluated it on a benchmark with 20 Java classes. Our results show that G-EvoSuite significantly improves code coverage independently of the search budget.

In future work, we plan to improve our grammar-based fuzzer and extend it to more data formats. Our current approach makes use of grammar-based mutation operators that are specific to the data format of the target application, in this case JSON. These operators only work on valid input and therefore limit the output to also be valid. Investigating mutation operators for invalid input is part of our future agenda. Next to JSON, the XML data format is commonly used for APIs and it is similarly hard to test. We plan to extend our approach to include mutators for other data formats.

A further next step is to look into using machine learning to infer the data format accepted by an application. Data format specific mutators can then be created based on this model without requiring pre-defined mutators for all possible data formats.

REFERENCES

[1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2018. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 143–154.

[2] Sheeva Afshan, Phil McMinn, and Mark Stevenson. 2013. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 352–361.

[3] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 263–272.

[4] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying search based software engineering with Sapienz at Facebook. In *International Symposium on Search Based Software Engineering*. Springer, 3–45.

[5] Andrea Arcuri. 2019. RESTful API automated test case generation with Evomaster. *ACM Transactions on Software Engineering and Methodology* 28, 1 (2019), 1–37.

[6] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18, 3 (2013), 594–623.

[7] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 63–74.

[8] Frederick Brooks. 1975. *The mythical man-month.* Addison-Wesley.

[9] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 1–38.

[10] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.

[11] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 107–118.

[12] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of {TLS} Implementations. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 193–206.

[13] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM Press, 416–419.

[14] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering* 20, 3 (2015), 611–639.

[15] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 206–215.

[16] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 445–458.

[17] Sadeeq Jan, Cu D Nguyen, and Lionel Briand. 2015. Known xml vulnerabilities are still a threat to popular parsers and open source systems. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 233–241.

[18] Fitsum Kifetew, Xavier Devroey, and Urko Rueda. 2019. Java unit testing tool competition-seventh round. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 15–20.

[19] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. 2017. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering* 22, 2 (2017), 928–961.

[20] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[21] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2016. Automated test suite generation for time-continuous simulink models. In *proceedings of the 38th International Conference on Software Engineering*. 595–606.

[22] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[23] Urko Rueda Molina, Fitsum Kifetew, and Annibale Panichella. 2018. Java unit testing tool competition-sixth round. In *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 22–29.

[24] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. *Replication package of "Generating Highly- structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing"*. https://doi.org/10.5281/zenodo.4001744

[25] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[26] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Incremental control dependency frontier exploration for many-criteria test case generation. In *International Symposium on Search Based Software Engineering*. Springer, 309–324.

[27] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology* 104 (2018), 236–256.

[28] Annibale Panichella and Urko Rueda Molina. 2017. Java unit testing tool competition - Fifth round. *Proceedings - 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing, SBST 2017* (2017), 32–38.

[29] IS Wishnu B Prasetya. 2013. T3, a combinator-based random testing tool for java: benchmarking. In *International Workshop on Future Internet Testing*. Springer, 101–110.

[30] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*. Springer, 93–108.

[31] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. 2018. Search-Based Crash Reproduction and Its Impact on Debugging. *IEEE Transactions on Software Engineering* (2018), 1–1.

[32] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*. Springer, 581–601.

[33] Jingbo Yan, Yuqing Zhang, and Dingning Yang. 2013. Structurized grammar-based fuzz testing for programs with highly structured inputs. *Security and Communication Networks* 6, 11 (2013), 1319–1330.

[34] Hyunguk Yoo and Taeshik Shon. 2016. Grammar-based adaptive fuzzing: Evaluation on SCADA modbus protocol. In *2016 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 557–563.