# More Effective Test Case Generation with Multiple Tribes of AI

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

## ABSTRACT

Software testing is a critical activity in the software development life cycle for quality assurance. Automated Test Case Generation (TCG) can assist developers by speeding up this process. It accomplishes this by evolving an initial set of randomly generated test cases over time to optimize for predefined coverage criteria. One of the key challenges for automated TCG approaches is navigating the large input space. Existing state-of-the-art TCG algorithms struggle with generating highly-structured input data and preserving patterns in test structures, among others. I hypothesize that combining multiple tribes of AI can improve the effectiveness and efficiency of automated TCG. To test this hypothesis, I propose using grammar-based fuzzing and machine learning to augment evolutionary algorithms for generating more structured input data and preserving promising patterns within test cases. Additionally, I propose to use behavioral modeling and interprocedural control dependency analysis to improve test effectiveness. Finally, I propose integrating these novel approaches into a testing framework to promote the adoption of automated TCG in industry.

## CCS CONCEPTS

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

## KEYWORDS

software testing, test case generation, search-based software testing, fuzzing, machine learning

**ACM Reference Format:**
Mitchell Olsthoorn. 2022. More Effective Test Case Generation with Multiple Tribes of AI. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3517066

## 1 RESEARCH PROBLEM AND HYPOTHESIS

Software testing is an important part of quality assurance. Manually writing test cases is a tedious and error-prone task that can take up to 50 % of developers' time [9]. Over the last decades, researchers have developed techniques for automating the process of generating these test cases [28]. Automated test case generation (TCG) significantly reduces the time needed for testing and debugging software [43]. Additionally, recent studies have shown that search-based approaches can achieve higher code coverage compared to manually written test cases [30, 36] and can identify unknown bugs [1, 3, 21]. Moreover, automated test case generation tools have been successfully used in industry (e.g., [4, 8, 27]).

Although automated TCG is becoming more common in large software companies, the widespread adoption of these techniques is lacking behind [10]. One of the key challenges in automated TCG is the size of the input space [13]. With the ever-increasing complexity of modern applications, generating test cases, which consist of input data, test structures, and assertions [46], that satisfy difficult constraints becomes harder each day. Moreover, studies have identified that to make these techniques practical for industry use, automated TCG should not only focus on test coverage but also on the quality of the test cases [20, 34]. An additional factor for the lack of adoption of TCG techniques in industry is the shortage of easy-to-use production-level tooling [14]. The goal of my dissertation is to improve automated test case generation to increase the adoption by developers of these techniques.

The current state-of-the-art TCG approaches use evolutionary algorithms (EAs) that evolve an initial set of randomly generated test cases over time to optimize a fitness function. One of the reasons why EAs are so effective might be because they mimic the process that developers use to create test cases. Developers copy, paste, and then either modify the values of a method call or replace it entirely [5]. Although these EAs could, in theory, generate any possible input data given enough time, this would, however, be inefficient for complex data [2, 17].

I hypothesize that *by combining multiple tribes of artificial intelligence (AI) to narrow down the search space, we can improve the effectiveness and efficiency of automated test case generation.* In my dissertation, I will test this hypothesis by focussing on the following research areas:

(1) **Generating Test Cases with Highly-structured Input Data (Section 2.1)**: Certain types of applications require highly-structured input data (*e.g.,* parsers). However, automated TCG has limitations on creating such data. Previous studies have shown that automatically generated input is usually unstructured and can be difficult to read and interpret [2, 17]. By combining EAs with grammar-based fuzzing, I plan to use the information from the grammar to limit the number of possible actions that can be performed on a subject under test (SUT) and improve the quality of the test data. Here, the aim is not to focus on how much of the grammar is covered but to use the grammar as guidance for the EAs.
(2) **Preserving Method Sequence Patterns in System-level TCG (Section 2.2)**: One of the limitations of the current related work is that while the state-of-the-art TCG approaches can successfully create promising sequences of methods, they do not directly recognize and preserve them when creating


*ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA*

ACM ISBN 978-1-4503-9223-5/22/05.
https://doi.org/10.1145/3510454.3517066

new test cases [50]. I argue that detecting and preserving promising patterns of method sequences improves the effectiveness and efficiency of the TCG process. Combining EAs with machine learning allows gaining more insight into the structure and relationship between the actions within the test cases. These insights can then be used to reduce the input space by prioritizing actions that match these patterns.

(3) **Creating a Hybrid Crossover Operator for Both Input-data and Method Sequences (Section 2.3)**: The encoding of a test case consists of both test data and method sequences [46]. Current state-of-the-art EAs for TCG use a crossover operator (*i.e.,* single-point) that swaps a group of method sequences between two test cases [12, 35]. These approaches, however, simply copy over the corresponding input data. I argue that a hybrid crossover operator that alters both the structure of the test cases as well as the input data can improve the test case diversity.

(4) **Generating Complex Objects during TCG from Behavioral Models (Section 2.4)**: One of the challenges of TCG approaches is the generation of complex objects with logical (*i.e.,* not random) sequences of method calls. By combining automated TCG with model inference techniques, I can build a model based on behavioural patterns. This model can then be used to seed the EA. Here, the aim is not to focus on all possible actions that can be performed on the subject but on the actions that are on the critical path.

(5) **Improving Search Guidance for Explicit Contracts and Declarative Input Validation Rules (Section 2.5)**: Most software makes use of conditional checks to make sure that the input to a method is valid or preconditions have been met (*e.g.,* `@NotNull` in Java). However, since these language constructs often are not treated as first-class citizens during testing, they cause a partial flat landscape preventing the fitness functions from effectively guiding the EA. I plan to improve the search guidance for these design by contract constructs by restoring the fitness gradient.

(6) **Improving the Effectiveness of Regression testing (Section 2.6)**: One use case of automated TCG is for regression testing, where the generated tests are utilized to ensure that code changes do not impact the functionality of the application. However, running the entire test suite quickly becomes unfeasible for very large systems. State-of-the-art approaches use test case selection (TCS) to select a subset of the test suite to run. I plan to optimize these approaches for this use case to make them more effective.

(7) **Improving the tooling (Section 2.7)**: One of the reasons for the lack of adoption of TCG techniques in industry is the absence of production-level tooling [14]. I plan to create a new testing framework that will integrate the different research focus areas. Additionally, I aim to make the framework modular so that it eases the adoption of different languages.

## 2 CONTRIBUTIONS AND RESULT SO FAR

This section describes each contribution in the context of the related work, how I plan to evaluate that contribution, and presents the results I have obtained so far.

### 2.1 Generating Test Cases with Highly-structured Input Data

Automated test case generation has limitations on creating highly-structured input data. Previous studies have shown that automatically generated input is usually unstructured and can be difficult to read and interpret [2, 17]. Grammar-based fuzzing, on the other hand, is very effective in generating highly-structured input data based on a user-specified grammar [22, 51]. For this reason, fuzzing has been widely used for security and system testing [11, 23]. When applied to data formats, fuzzers can generate and manipulate well-formed input data using grammar derivative rules. However, since grammar-based fuzzing only generates input data, developers need to specify the entry points (for system testing), manually write the method calls, and come up with their own assertions.

To address these limitations, I proposed a novel approach [33] that combines the strength of grammar-based fuzzing and EAs with a focus on the JSON data format. At the initialization phase, I make use of grammar-based fuzzing to inject structured JSON input data with some probability. This allows the EA to discover if the SUT can make use of this structured data. Then, the EA creates and evolves the test case structures (*i.e.,* the sequence of method challenges) using both randomly generated and injected data. The injected input data are evolved separately using grammar-based fuzzing, which mutates the input using grammar derivative rules.

I performed an empirical study [33] to assess the efficacy and feasibility of this approach, which I implemented in EvoSuite [19], a state-of-the-art TCG tool for Java. This study was conducted on 20 classes from the three most popular Java JSON parsers, namely the `GSON`, `fastjson`, and `org.json`. To assess whether the proposed approach negatively impacts the performance of classes that don't make use of highly-structured input, I evaluated both JSON and non-JSON related classes. Since the approach makes use of randomized algorithms, I repeat the experiment multiple times and make use of statistical analysis. Specifically, I use the unpaired Wilcoxon rank-sum test [16] for the statistical significance and the Vargha-Delaney statistic [47] for the effect size. In the study, I answer the following research questions:

**RQ1.1** *To what extend does grammar-based fuzzing improve the effectiveness of test case generation in evosuite?*

On average, the proposed approach achieves +15 % of branch coverage compared to the baseline (EvoSuite without fuzzing). The largest improvement that was observed in the study was +50 % of branch coverage for one of the classes in the benchmark.

**RQ1.2** *What is the effectiveness of combining grammar-based fuzzing and search-based testing over different search budgets?*

When comparing the performance of the proposed approach to the baseline over time, the study showed that the delta difference does not substantially decrease and in most cases even increases with a larger search budget. The study also showed that injecting JSON strings in the initial population is not sufficient by itself to reach a higher coverage.

> Combining EAs 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.

## 2.2 Preserving Method Sequence Patterns in System-level TCG

EvoMaster is the state-of-the-art test case generation tool for Java REST API testing. A test case, in this context, is a sequence of API requests (*i.e.,* HTTP requests) on specific resources [6, 7]. REpresentational State Transfer (REST) APIs deal with states. Each individual request changes the state of the API, and therefore, its execution result depends on the state of the application (*i.e.,* the previously executed requests). This creates patterns of HTTP requests that depend on each other. While the state-of-the-art algorithms can successfully create promising sequences of HTTP requests, they do not directly recognize and preserve them when creating new test cases [50]. I argue that detecting and preserving patterns of HTTP requests, hereafter referred to as *linkage structures*, improves the effectiveness of the test case generation process.

I proposed a novel approach [44], named LT-MOSA that uses Agglomerative Hierarchical Clustering (AHC) to infer these linkage structures from automatically generated test cases in the context of REST API testing. Specifically, AHC generates a *Linkage Tree* (LT) model from the test cases that are the closest to reach uncovered test targets (*i.e.,* lines and branches). This model is used by the genetic operators to determine which sequences of HTTP requests should not be broken up and should be replicated in new tests.

To evaluate the feasibility and effectiveness of this approach, I implemented this approach within EvoMaster. I performed an empirical study [44] with 7 real-world benchmark web/enterprise applications from the EvoMaster Benchmark (EMB) dataset. The study compares the proposed approach against the two state-of-the-art algorithms for system-level test generations implemented in EvoMaster, namely Many Independent Objective (MIO) and Many Objective Search Algorithm (MOSA). In the study, I answer the following research questions:

**RQ2.1** *How does LT-MOSA perform compared to the state-of-the-art approaches with regard to code coverage?*

The results show that LT-MOSA covers significantly more test targets (*i.e.,* lines and branches) in 4 and 5 out of the 7 applications compared to MIO and MOSA, respectively. On average, the approach covered 11.7 % more test targets than MIO (with a max improvement of 66.5 %) and 8.5 % more than MOSA (with a max improvement of 37.5 %).

**RQ2.2** *How effective is LT-MOSA compared to the state-of-the-art approaches in detecting real-faults?*

LT-MOSA can detect, on average, 27 and 18 unique real-faults that were not detected by MIO and MOSA, respectively.

**RQ2.3** *How effective is LT-MOSA at covering test targets over time compared to the state-of-the-art approaches?*

LT-MOSA achieves higher AUC values than the baselines. Meaning, it covers more targets in less time. For the largest application (OCVN), LT-MOSA takes half of the time to reach the same coverage compared to MOSA, while MIO never reaches this coverage.

> Inferring and preserving linkage structures in REST APIs achieves significantly higher code coverage and fault-detection capability compared to the state-of-the-art approaches (*i.e.,* MIO and MOSA).

## 2.3 Creating a Hybrid Variational Operator for Both Input-data and Method Sequences

Over the years, related work has used three types of encoding schemata to represent test cases for search algorithms, namely data-level, test case-level, and test suite-level. These schemata typically implement genetic operators at the same level as the encoding. For example, the crossover operator at the data-level exchanges data between two input vectors [29]. The test case-level crossover exchanges statements between two parent test cases [46]. Lastly, the test suite-level crossover swaps test cases within two test suites [19]. Recent studies have shown that the test case-level schema combined with many-objective search is the most effective at generating test cases with high coverage [12, 35]. The current many-objective approaches use the single-point crossover to recombine groups of statements within test cases. Test cases consist of both test structures (method sequences) and test data [46]. Hence, the crossover operator only changes the test structure and simply copies over the corresponding input data. Therefore, input data has to be altered by the mutation operator, usually with a small probability.

I argue that better test case diversity can be obtained by designing a crossover operator that alters both the structure of the test cases as well as the input data by creating new data that is in the neighborhood of the parents' data. To validate this hypothesis, I propose a new operator, called Hybrid Multi-level Crossover (*HMX*) [31], that combines different crossover operators on multiple levels. I implemented this hybrid operator within EvoSuite [19].

To evaluate the effectiveness of *HMX*, I performed an empirical study [31] where I compare it with the single-point crossover used in EvoSuite, *w.r.t.* structural coverage and fault detection capability. To this aim, I have built a benchmark with 116 classes from the Apache Commons and Lucene Stemmer projects, which include classes for numerical operations and string manipulation. In the study, I answer the following research questions:

**RQ3.1** *To what extent does HMX improve structural coverage compared to the single-point crossover?*

The study shows that *HMX* achieves higher structural coverage for ~30 % of the classes in the benchmark. On average, *HMX*, covered 6.4 % and 7.2 % more branches and lines than the baseline, respectively (with a max improvement of 19.1 % and 19.4 %).

**RQ3.2** *How does HMX impact the fault-detection capability of the generated tests?*

*HMX* improved the fault detection capability in ~25 % of the classes with an average improvement of 3.9 % (max. 14 %) and 2.1 % (max. 12.1 %) for weak and strong mutation, respectively.

> *HMX* significantly improves the structural coverage and fault detection capability of the generated test cases compared to the standard crossover operator used in EvoSuite (*i.e.,* single-point).

## 2.4 Generating Complex Objects during TCG from Behavioural Models

Seeding consists of injecting additional information (*e.g.,* manually-written test suites) for use in the search process to make it more

effective [39]. A recent study proposed a method to infer a behavioral model from the usage patterns of applications in the context of crash reproduction [18]. I plan to build a model based on behavioral patterns that can be used to seed the search process. I will evaluate my proposed approach using the following research question:

**RQ4.1** *To what extend does the complex objection generation improve the effectiveness of TCG?*

## 2.5 Improving Search Guidance for Explicit Contracts and Declarative Input Validation Rules

Popular Java frameworks make use of annotations (*e.g.,* `@NotNull`) to check for the validity of input parameters [26]. In Solidity, a *require* function exists that is used for authority and validity checks to protect smart contracts against invalid requests [24]. These constructs halt the execution of a method or program when their conditions are not met. Since these constructs are not part of the control flow of the method they are applied to, the EA has no guidance on how to satisfy the condition within them. This creates partially flat fitness landscapes, forcing the EA to resort back to random testing. I plan to restore the fitness gradient in the landscape by using interprocedural control dependency analysis to determine how these constructs influence the execution of the method under test at runtime and provide this information to the EA. I will evaluate my proposed approach using the following research question:

**RQ5.1** *To what extend does interprocedural control dependency analysis improve the effectiveness of TCG for design by contract constructs?*

## 2.6 Improving the Effectiveness of Regression testing

Regression testing aims to assess that changes to the production code do not affect the behavior of unchanged parts [55]. Running the entire test suite within a DevOps pipeline quickly becomes unfeasible for very large systems [48]. Various techniques were developed to reduce the cost of regression testing [15, 40, 41]. Multi-objective Evolutionary Algorithms (`MOEAs`), and `NSGA-II` in particular, have been successfully used in the literature to produce Pareto efficient subsets of the test suites *w.r.t.* different testing criteria [52, 53, 55]. Test case selection is inherently a multi-objective problem since developers want to maximize the coverage of the selected subset but minimize the cost of running it. `MOEAs` that rely on Pareto ranking and problem decompositions have been shown to achieve good performance also compared to greedy algorithms and local solvers [53]. One limitation for classic `MOEAs` (including `NSGA-II`) is that new solutions are generated using fully randomized recombination (crossover) operators [45, 49]. This could destroy potential promising patterns that can be created by `MOEAs`. While *linkage learning* has been shown to be effective for single-objective numerical problems [38, 45, 49], we argue that it can also have huge potential for multi-objective test case selection.

To address these limitations, I proposed a novel approach [32], named `L2-NSGA`, a variant of `NSGA-II` that integrates key elements of *linkage learning* for the test case selection problem. In particular, `L2-NSGA` uses Agglomerative Hierarchical Clustering (AHC) to identify linkage structures in the non-dominated solutions produced by `NSGA-II` in every other generation. Then, `L2-NSGA` uses a novel crossover operator that stochastically selects and replicates some of the inferred structures into new individuals.

I performed an empirical study [32] on four software systems with multiple versions and regression faults. The study compares the quality and fault detection capability of the solutions produced by `L2-NSGA` against `NSGA-II`, which is the most widely-employed MOEA in the regression testing literature (*e.g.,* [37, 42, 52, 54]). In the study, I answer the following research questions:

**RQ6.1** *To what extent does* `L2-NSGA` *produce better Pareto efficient solutions compared to* `NSGA-II`?

The study shows that the sub-test suites produced by `L2-NSGA` achieve higher coverage while incurring lower execution costs than the baseline, as measured by +23 % increase in hypervolume for `bash` v3.

**RQ6.2** *What is the cost-effectiveness of the solution produced by* `L2-NSGA` *vs.* `NSGA-II`?

The solutions created by `L2-NSGA` detect more regression faults than the solutions produced by `NSGA-II`, as measured by +18 % increase in fault-detection for `bash` v3.

> `L2-NSGA` produces better trade-offs between cost and coverage than its predecessor `NSGA-II` (the baseline), which is widely used in the literature.

## 2.7 Improving the tooling

Two existing state-of-the-art TCG tools for Java are EvoSuite [19] (unit-level) and EvoMaster [6] (system-level). Recently, a TCG tool was published for Python [25]. One of the limiting factors for the lack of adoption of TCG techniques in industry is the absence of production-level tooling [14]. I plan to create a new testing framework for Javascript that will integrate the different research focus areas. I will evaluate my proposed framework using the following research questions:

**RQ7.1** *How effective is the proposed testing framework w.r.t. code coverage?*

**RQ7.2** *What is the fault-detection capability of the proposed testing framework?*

# 3 TIMELINE

I am currently halfway through my 4 year Ph.D. program. In my third year, I plan to work on my research focus areas 4 and 5. Tooling is an ongoing project that I plan to complete in my fourth year.

# 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] Maurício Aniche, Christoph Treude, and Andy Zaidman. 2021. How Developers Engineer Test Cases: An Observational Study. arXiv:2103.01783 [cs.SE]

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

[7] Andrea Arcuri and Juan P Galeotti. 2020. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–31.

[8] 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.

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

[10] Matteo Brunetto, Giovanni Denaro, Leonardo Mariani, and Mauro Pezzè. 2021. On Introducing Automatic Test Case Generation in Practice: A Success Story and Lessons Learned. arXiv:2103.00465 [cs.SE]

[11] 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.

[12] 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, August (2018), 207–235. https://doi.org/10.1016/j.infsof.2018.08.010

[13] George Candea and Patrice Godefroid. 2019. Automated software test generation: some challenges, solutions, and recent advances. In *Computing and Software Science*. Springer, 505–531.

[14] Adnan Causevic, Daniel Sundmark, and Sasikumar Punnekkat. 2011. Factors limiting industrial adoption of test driven development: A systematic review. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 337–346.

[15] Tsong Yueh Chen and Man Fai Lau. 1996. Dividing strategies for the optimization of a test suite. *Inform. Process. Lett.* 60, 3 (Nov. 1996), 135–141.

[16] William Jay Conover. 1998. *Practical nonparametric statistics*. Vol. 350. John Wiley & Sons.

[17] 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.

[18] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. 2020. Search-based crash reproduction using behavioural model seeding. *Software Testing, Verification and Reliability* 30, 3 (2020), e1733.

[19] 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*. 416–419.

[20] Gordon Fraser and Andrea Arcuri. 2013. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE sixth international conference on software testing, verification and validation*. IEEE, 362–369.

[21] 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.

[22] 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.

[23] 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.

[24] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing Transaction-Reverting Statements in Ethereum Smart Contracts. arXiv:2108.10799 [cs.CR]

[25] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. In *International Symposium on Search Based Software Engineering*. Springer, 9–24.

[26] Federico Mancini, Dag Hovland, and Khalid A Mughal. 2010. Investigating the limitations of java annotations for input validation. In *2010 International Conference on Availability, Reliability and Security*. IEEE, 513–518.

[27] 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.

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

[29] Phil McMinn. 2004. Search-based software test data generation: A survey. *Software Testing Verification and Reliability* 14, 2 (2004), 105–156.

[30] 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.

[31] Mitchell Olsthoorn, Pouria Derakhshanfar, and Annibale Panichella. 2021. Hybrid Multi-level Crossover for Unit Test Case Generation. In *International Symposium on Search Based Software Engineering*. Springer, 72–86.

[32] Mitchell Olsthoorn and Annibale Panichella. 2021. Multi-objective test case selection through linkage learning-based crossover. In *International Symposium on Search Based Software Engineering*. Springer, 87–102.

[33] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1224–1228.

[34] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2016. Automatic test case generation: What if test code quality matters?. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 130–141.

[35] 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.

[36] 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.

[37] A. Panichella, R. Oliveto, M. Di Penta, and A. De Lucia. [n.d.]. Improving Multi-Objective Test Case Selection by Injecting Diversity in Genetic Algorithms. *IEEE Transactions on Software Engineering* ([n. d.]). https://doi.org/10.1109/TSE.2014.2364175 To appear.

[38] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. 1999. BOA: The Bayesian optimization algorithm. In *Proceedings of the genetic and evolutionary computation conference GECCO-99*, Vol. 1. Citeseer, 525–532.

[39] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366–401.

[40] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. 1998. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In *Proceedings of the International Conference on Software Maintenance*. IEEE CS Press, 34–44.

[41] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. 1999. Test case prioritization: an empirical study. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*. 179–188. https://doi.org/10.1109/ICSM.1999.792604

[42] Takfarinas Saber, Florian Delavernhe, Mike Papadakis, Michael O'Neill, and Anthony Ventresque. 2018. A hybrid algorithm for multi-objective test case selection. In *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1–8.

[43] 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.

[44] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2021. Improving Test Case Generation for REST APIs Through Hierarchical Clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[45] Dirk Thierens. 2010. The linkage tree genetic algorithm. In *International Conference on Parallel Problem Solving from Nature*. Springer, 264–273.

[46] Paolo Tonella. 2004. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 119–128.

[47] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[48] Joakim Verona. 2016. *Practical DevOps.* Packt Publishing Ltd.

[49] Richard A Watson, Gregory S Hornby, and Jordan B Pollack. 1998. Modeling building-block interdependency. In *International Conference on Parallel Problem Solving from Nature*. Springer, 97–106.

[50] Richard A Watson and Thomas Jansen. 2007. A building-block royal road where crossover is provably essential. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 1452–1459.

[51] 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.

[52] Shin Yoo. 2010. A Novel Mask-Coding Representation for Set Cover Problems with Applications in Test Suite Minimisation. In *Proceedings of the 2nd International Symposium on Search-Based Software Engineering (SSBSE 2010)*. IEEE.

[53] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case selection. In *Proc. International Symposium on Software Testing and Analysis*. ACM, 140–150.

[54] Shin Yoo and Mark Harman. 2010. Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software* 83, 4 (2010), 689–701.

[55] S. Yoo and M. Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (March 2012), 67–120.