

A Retrospective on Whole Test Suite Generation: On the Role of SBST in the Age of LLMs

Gordon Fraser*, Andrea Arcuri†

*University of Passau, Germany

†Kristiania University College and Oslo Metropolitan University, Norway

Abstract—This paper presents a retrospective of the article “*Whole Test Suite Generation*”, published in the *IEEE Transactions on Software Engineering*, in 2012. We summarize its main contributions, and discuss how this work impacted the research field of Search-Based Software Testing (SBST) in the last 12 years. The novel techniques presented in the paper were implemented in the tool EvoSuite, which has been so far the state-of-the-art in unit test generation for Java programs using SBST. SBST has shown practical and impactful applications, creating the foundations to open the doors to tackle several other software testing problems besides unit testing, like for example system testing of Web APIs with EvoMaster. We conclude our retrospective with our reflections on what lies ahead, especially considering the important role that SBST still plays even in the age of Large Language Models (LLMs).

Index Terms—EvoSuite, SBST, LLM, Pynguin, EvoMaster

I. INTRODUCTION

Search-Based Software Engineering (SBSE) refers to the research field in which the application of search algorithms (e.g., Genetic Algorithms [1]) to software engineering problems is studied [2], [3]. Software engineering problems are cast into *optimization* problems, and then different search algorithms can be designed and investigated to tackle them. Search-Based Software Testing (SBST) is a (large) subset of SBSE, focused on the testing of software [4], [5]. Given the search space of all possible tests for any piece of software, metrics to optimize via search are for example fault detection or code coverage. Although the work on SBST took off in the 90s, applications of SBST can be traced back to at least 1976 [6].

The paper titled “*Whole Test Suite Generation*” [7] introduced a major paradigm shift in SBST. At that time, before 2012, most work on SBST focused on generating test cases for *single* targets, such as specific branches or statements in the system under test (SUT). Different heuristics to *smooth* the search landscape were designed, like the so-called *branch distance* [8] and *testability transformations* [9]. However, to generate not just individual tests but an entire test suite, several different independent searches were needed, one for each testing target in the SUT (e.g., for all branches). This was problematic for several reasons: targets could be infeasible, and any effort spent on covering them would be wasted; targets have different complexity that is unknown beforehand, such that allocating a balanced search budget (i.e., for how long to run the search before giving up) equally among targets would be inefficient; also, targets often have dependencies among them (e.g., nested `if` statements), and independent search sessions would not be able to exploit this information.

Whole test suite generation introduces a conceptually simple, but very effective approach: Instead of modeling each individual testing target as different optimization problem wrapped into an overall test generation loop, we consider *all* targets at the same time, in a single optimization search, addressed with an evolutionary algorithm. Instead of evolving individual test cases, we evolve *whole* test suites. The above named problems go away, the test generation algorithm is simplified by an entire layer, and the driving fitness function is a simplified version of heuristics for all targets in the SUT.

In this retrospective, we discuss how whole test suite generation impacted SBST and software testing research, and why. In particular, we discuss this in the light of successful tools such as EvoSuite (Section II), why this was successful (Section III), and successor tools like EvoMaster or Pynguin (Section IV). Despite the influence of whole test suite generation [7] on the last decade of software testing, it is important to consider what impact it may still have in the future. Disruptive AI technologies such as Large Language Models (LLMs) have been applied to many software testing problems [10]. Is whole test suite generation still relevant in the age of LLMs? We provide our reflections on this topic in Section V.

II. EVOSUITE

The novel techniques presented in the 2012 paper [7] were first implemented in a tool called EvoSuite [11]. This is an open-source project¹, aimed at unit test generation for Object-Oriented software written in Java. Throughout the years, several novel techniques (including whole test suite generation, as well as others like specialized memetic algorithms [12] and seeding strategies [13]) were implemented in this tool.

On the one hand, EvoSuite was a research playground for experimenting and designing novel techniques in SBST. On the other hand, significant engineering effort was invested (including plugins for popular tools such as Maven, IntelliJ and Jenkins [14]), making it usable in practice, and enabling large empirical studies [15]. This not only led it to win most unit test competitions throughout the years (e.g., 2013 [16] to 2023 [17]), but also it made it possible for other researchers to use EvoSuite for their work. In 2023, this culminated in an ACM SIGSOFT Impact Paper Award.² The use of whole test suite generation was a main key for EvoSuite’s impact on the software engineering research community.

¹<https://github.com/EvoSuite/evosuite>

²<https://www2.sigsoft.org/awards/impactpaper/>

III. WHY WAS IT SUCCESSFUL?

By all standard metrics such as citations, whole test suite generation and the accompanying EvoSuite tool were a success. But why? We can but speculate about the reasons:

- **Simplicity:** Single target test generation uses a search algorithm, contained in a test generation algorithm, requiring even more algorithms to calculate fitness values. Whole test suite generation is a single algorithm and a much simpler fitness function, making it not only easier to apply to other domains, but also to adapt and extend.
- **Tooling:** An evolutionary search algorithm may be easy to understand, but instantiating it for test generation is anything but easy. The actual search algorithm probably is less than 10% of EvoSuite’s source code, whereas the challenging bits lie in details of representing and running tests for Java. EvoSuite solved these bits once and for all (alas, until Java received a breaking update), making it easy to focus more on algorithmic fun.
- **Maintenance:** An incredible effort went into maintaining EvoSuite. A standard empirical study with EvoSuite consisted of running the tool on thousands of classes, and like any program analysis tool EvoSuite would crash on substantial shares of target programs. It is fair to assume that 90% of the maintenance effort for EvoSuite went into looking into cases where the tool crashed, over and over again, and making it work better.
- **Usability:** Anyone can use EvoSuite out of the box, without having to first resolve dependencies, edit configuration files, or setting up some sort of virtual environment or Docker container. There’s just one “fat” jar with all dependencies—download it, run it, and there be tests.

All these points made EvoSuite easily adoptable, and with it the whole test suite generation approach. The fact that we wrote many papers using EvoSuite certainly helped in publicizing the tool, but so did many other researchers. Maybe more than anything, we might speculate, the success was influenced by EvoSuite serving as a catalyst for other research for which it was available at just the right time: Countless recent software engineering research topics rely on the availability of tests, for example fault localization, automated program repair, or deep learning applications. EvoSuite provides the tests required to study these fields, and it does so for free.

IV. BEYOND EVOSUITE

A. Beyond Whole Test Suite Generation

While whole test suite generation represented a milestone in the field of SBST that overcame some previous limitations, it provides its own limitations that called for further improvements. First and foremost there is the issue that even after a test for an individual target has been found, that target remains part of the overall optimization goal, rather than letting the search focus on what has not been covered yet. The solution to this problem is to make the optimization dynamic: Once a testing target has been covered the corresponding test is stored in an archive, and the search continues only for the remaining targets. Even better, established techniques from multi-objective optimization can improve diversity throughout

the search, yielding far superior results. This is implemented in the Many Objective Sorting Algorithm (MOSA) [18], which is now the default algorithm used in EvoSuite.

At this point, the algorithm itself maybe is no longer the central issue inhibiting further improved code coverage, but rather the problematic fitness landscape that test generation represents. The venerable branch distance [8] provides gradients in the search space by estimating how close conditional statements are to evaluating to true or false, but this estimation cannot provide any guidance when comparing Boolean values or object references. While gradients can be recovered through testability transformations [9] for Boolean variables [19], we found that Booleans are not the main culprit [20], and instead the failure to properly construct complex objects prevents the search from getting to a point where missing gradients even matter [21]. This is not a solved problem to date.

B. Beyond Java

EvoSuite implemented whole test suite generation for Java, and much of EvoSuite’s complexity lies in dealing with specifics of that language and environment. While Java is a popular target in the software engineering research world, whole test suite generation and similar principles have made appearances for example for Python [22], Rust [23], or JavaScript [24]. Our own dabbling in the world of Python with the Pynguin test generator [22], a re-implementation of EvoSuite for Python, have if anything confirmed that often the *engineering* challenges maybe outshadow the research challenges (e.g., dynamic typing in Python) in adapting test generation approaches for new programming languages.

C. Beyond Unit Testing

Where whole test suite generation was originally investigated in the context of *unit testing*, it can be applied in other contexts as well. An example is *system testing* for Web APIs (e.g., REST [25], GraphQL [26] and RPC APIs [27]) with search-based tools such as EvoMaster [28], [29].

One challenge though is that, compared to *unit testing*, in *system testing* there are a few orders of magnitude higher number of testing targets to optimize for. The algorithm presented in the whole test suite generation paper [7] would not directly scale in this testing context. But, as for any research result, it can be used as a stepping stone to build more advanced algorithms. In this particular case, it led to the design of the MIO algorithm [30]. This was a major success, with thousands of downloads of EvoMaster from practitioners [28], including its use in large Fortune 500 enterprises such as Meituan [27], [31] and Volkswagen [32], [33]. The concept of whole test suite generation [7] played a central role in the design and implementation of EvoMaster.

Not all endeavors in system test generation were as fruitful as the work on EvoMaster, though. For a while, Android testing was all the rage in software engineering research, with new papers and tools appearing frequently. However, we found that neither whole test suite generation nor newer many-objective search algorithms could really clearly outperform simple random testing [34], and neither replacing algorithm,

representation, or fitness function, could change that. Once again it seems that engineering challenges inhibit progress [35]

D. Beyond Deterministic Testing

All previously named application areas of whole test suite generation and derived algorithms make the assumption that a test is a sequence of interactions with a program under test; e.g., sequences of API calls for unit testing, sequences of REST-API calls or UI-interactions for system testing. A fundamental assumption here is that executing the same sequence on the same system twice will lead to the same result. Already when studying whole test suite generation on Java unit tests, we found that this assumption often does not hold. We had stumbled on the problem of *flaky* tests even before it was a hyped topic in software engineering research, and used heavy machinery in the form of program instrumentation to make EvoSuite tests deterministic [36]. However, non-deterministic behavior may be unavoidable in systems interacting with the real world, or even desirable in computer games that aim to challenge players. Such systems nevertheless need to be tested, and SBST represents a feasible way to do so. For example, the same evolutionary search algorithms and the same fitness functions that have been repeatedly mentioned in this paper can not only optimize sequences, but also train neural networks that learn to adapt to behavioral differences while still ensuring eventual coverage of testing targets [37].

E. Beyond Code Coverage

As a final example research that whole test suite generation enabled, let us point out one more commonality of all approaches named previously in this section: They all aim to optimize code coverage. While code coverage is certainly an important quality metric for tests and a worthwhile optimization goal for automated test generation approaches, it only solves half the problem. Whole test suite generation gives you test *inputs* that achieve code coverage, but in order to find bugs in programs, this is not sufficient. The tests also need test *oracles* that can distinguish correct from incorrect behavior, and if they do reveal a bug. Until automated program repair solves this problem for us once and for all, the generated tests need to be readable and understandable. Consequently, even though these problems are orthogonal to whole test suite generation, they need to be addressed in any implementation that aims to be usable, for example by finding sensible names for variables [38] or by predicting test oracles [39].

V. BEYOND SBST: SBST IN THE AGE OF LLMs

Evolutionary computation techniques have achieved tremendous successes in many fields of science and engineering. As such, their introduction in software engineering [2], [3] opened the door to tackle with scalable solutions important problems, especially in software testing [4], [5]. In a similar way, at the time of writing, Large Language Models (LLMs) have brought disruptive innovation worldwide. It is not surprising that a lot of research has been carried out to evaluate them on software engineering problems, such as software testing [10].

In a world where you can prompt an LLM to generate test cases, will SBST techniques like whole test suite generation still be relevant? We cannot peek into the future, but we can make an educated guess after two decades of research. Our current answer is a strong *yes*, as we will corroborate next.

Several studies [40], [41] (in which we were not involved) have investigated LLMs to generate unit tests, and compared the results with the state-of-the-art whole test suite generation EvoSuite. On average, current LLMs achieved worse results. For example, it has been reported [40] that “*EvoSuite generally achieves higher coverage*”, based on an experiment with 690 Java classes. Furthermore, LLMs have been observed [41] to have “*a close performance with the state-of-the-art test generation tool for the HumanEval dataset, but their performance is poor for open-source projects from Evosuite based on coverage*”. A major issue, besides hallucinations, is that when using an LLM “*several generated tests were not compilable*” [41]. In our own work on the application of EvoMaster in industry at Volkswagen [32] on testing REST APIs, generally SBST techniques using whole test suite generation provided better results than LLM ones.

Obviously, this can and likely will change in the future, as novel, more sophisticated LLM techniques could be designed that lead to better results. Still, regardless of whether LLM will replace SBST or not in the future, at the moment they can be combined, covering each other weaknesses (in a similar way in which SBST techniques can be combined with Dynamic Symbolic Execution [42]). For example, when an SBST search process is stuck in local optima, an LLM can be used to derive inputs to help escape them, as done for example by CodaMosa [43]. Furthermore, important aspects for a usable test suite might be hard to encode in fitness functions for SBST. The *readability* of the generated test cases is one such important aspect that is hard to codify programmatically. In this case, LLMs can be used to improve the readability of the generated test suite outputs of an SBST process [44].

Another critical aspect of LLMs that needs to be considered is that, as for any machine learning process, the quality of the results are strongly correlated with the quality of the data used for training. In the case of programming tasks and test case generation, this usually would be open-source projects, hosted for example on GitHub. Large enterprise systems, using for example large microservice architectures, might not be well represented among open-source projects, compared to, for example, utility libraries and student projects. As such, the success of LLMs at generating test cases is likely strongly correlated with how many existing test cases for a specific testing domain can be found among open-source projects. Where we can see improvements in unit testing with LLMs in the near future, for system testing of industrial systems our expectations are more limited. Still, also for current SBST techniques large industrial systems are challenging. As such, we can see an exciting time ahead in the software testing research field, where novel techniques like LLMs can be evaluated and combined with fundamental stepping stones such as whole test suite generation [7].

REFERENCES

- [1] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.
- [2] M. Harman and B. F. Jones, "Search-based software engineering," *Journal of Information & Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.
- [3] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.
- [4] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.
- [5] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 6, pp. 742–762, 2010.
- [6] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, 1976.
- [7] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.
- [8] B. Korel, "Automated software test data generation," *IEEE Transactions on software engineering*, vol. 16, no. 8, pp. 870–879, 1990.
- [9] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.
- [10] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, 2024.
- [11] G. Fraser and A. Arcuri, "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*, 2011, pp. 416–419.
- [12] G. Fraser, A. Arcuri, and P. McMinn, "A memetic algorithm for whole test suite generation," *Journal of Systems and Software*, vol. 103, pp. 311–327, 2015.
- [13] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 366–401, 2016.
- [14] A. Arcuri, J. Campos, and G. Fraser, "Unit test generation during software development: Evosuite plugins for maven, intellij and jenkins," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016.
- [15] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–42, 2014.
- [16] —, "Evosuite at the SBST 2013 tool competition," in *International Workshop on Search-Based Software Testing (SBST)*, 2013, pp. 406–409.
- [17] S. Schweikl, G. Fraser, and A. Arcuri, "Evosuite at the sbst 2023 tool competition," in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, 2023, pp. 65–67.
- [18] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 2015, pp. 1–10.
- [19] Y. Lin, J. Sun, G. Fraser, Z. Xiu, T. Liu, and J. S. Dong, "Recovering fitness gradients for interprocedural boolean flags in search-based testing," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 440–451.
- [20] S. Vogl, S. Schweikl, and G. Fraser, "Encoding the certainty of boolean variables to improve the guidance for search-based test generation," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2021, pp. 1088–1096.
- [21] Y. Lin, Y. S. Ong, J. Sun, G. Fraser, and J. S. Dong, "Graph-based seed object synthesis for search-based unit testing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1068–1080.
- [22] S. Lukaczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.
- [23] V. Tymofeyev and G. Fraser, "Search-based test suite generation for rust," in *International Symposium on Search Based Software Engineering*. Springer, 2022, pp. 3–18.
- [24] M. Olsthooorn, D. Stallenberg, and A. Panichella, "Syntest-javascript: Automated unit-level test case generation for javascript," in *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, 2024, pp. 21–24.
- [25] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, 2019.
- [26] A. Belhadi, M. Zhang, and A. Arcuri, "Random testing and evolutionary testing for fuzzing graphql apis," *ACM Transactions on the Web*, vol. 18, no. 1, pp. 1–41, 2024.
- [27] M. Zhang, A. Arcuri, Y. Li, Y. Liu, and K. Xue, "White-box fuzzing RPC-based APIs with EvoMaster: An industrial case study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–38, 2023.
- [28] A. Arcuri, M. Zhang, S. Seran, J. P. Galeotti, A. Golmohammadi, O. Duman, A. Aldasoro, and H. Ghianni, "Tool report: Evomaster—black and white box search-based fuzzing for rest, graphql and rpc apis," *Automated Software Engineering*, vol. 32, no. 1, p. 4, 2025.
- [29] A. Arcuri, "Evomaster: Evolutionary multi-context automated system test generation," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 394–397.
- [30] —, "Test suite generation with the many independent objective (mio) algorithm," *Information and Software Technology*, vol. 104, pp. 195–206, 2018.
- [31] M. Zhang, A. Arcuri, P. Teng, K. Xue, and W. Wang, "Seeding and mocking in white-box fuzzing enterprise rpc apis: An industrial case study," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2024–2034.
- [32] A. Poth, O. Rjollli, and A. Arcuri, "Technology adoption performance evaluation applied to testing industrial rest apis," *Automated Software Engineering*, vol. 32, no. 1, p. 5, 2025.
- [33] A. Arcuri, A. Poth, and O. Rjollli, "Introducing black-box fuzz testing for rest apis in industry: Challenges and solutions," 2025.
- [34] L. Sell, M. Auer, C. Frädrieh, M. Gruber, P. Werli, and G. Fraser, "An empirical evaluation of search algorithms for app testing," in *Testing Software and Systems: 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15–17, 2019, Proceedings 31*. Springer, 2019, pp. 123–139.
- [35] M. Auer, I. Arcuschin Moreno, and G. Fraser, "Wallmuer: Robust code coverage instrumentation for android apps," in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, 2024, pp. 34–44.
- [36] G. Fraser and A. Arcuri, "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, 2013, pp. 362–369.
- [37] P. Feldmeier and G. Fraser, "Neuroevolution-based generation of tests and oracles for games," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.
- [38] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 57–67.
- [39] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2130–2141.
- [40] W. C. Ouédraogo, K. Kaboré, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyandé, "Llms and prompting for unit test generation: A large-scale evaluation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2464–2465.
- [41] M. L. Siddiq, J. C. Da Silva Santos, R. H. Tanvir, N. Ulfat, F. Al Rifat, and V. Carvalho Lopes, "Using large language models to generate junit tests: An empirical study," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 313–322.
- [42] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," in *2013 IEEE 24th international symposium on software reliability engineering (issre)*. IEEE, 2013, pp. 360–369.
- [43] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.
- [44] M. Biagiola, G. Ghislotti, and P. Tonella, "Improving the readability of automatically generated tests using large language models," *arXiv preprint arXiv:2412.18843*, 2024.



Gordon Fraser Prof. Gordon Fraser is a Professor in Computer Science at the University of Passau, Germany. He received a PhD in computer science from Graz University of Technology, Austria, in 2007, worked as a post-doc at Saarland University, and was a Senior Lecturer at the University of Sheffield, UK. The central theme of his research is improving software quality, and his recent research concerns the prevention, detection, and removal of defects in software, but he is also actively researching software testing and programming education.



Andrea Arcuri Prof. Andrea Arcuri is a Professor of Software Engineering at Kristiania University of Applied Sciences and Oslo Metropolitan University, Oslo, Norway. His main research interests are in software testing, especially test case generation using evolutionary algorithms. Having worked 5 years in industry as a senior engineer, a main focus of his research is to design novel research solutions that can actually be used in practice. Prof. Arcuri is the main-author of EvoMaster and a co-author of EvoSuite, which are open-source tools that can automatically generate test cases using evolutionary algorithms. He received his PhD in search-based software testing from the University of Birmingham, UK, in 2009.