More Effective Test Case Generation with Multiple Tribes of AI

Olsthoorn, Mitchell

DOI
10.4233/uuid:14dde577-54bb-46a8-a61a-85a940e416e2

Publication date
2024

Document Version
Final published version

Citation (APA)
Olsthoorn, M. (2024). *More Effective Test Case Generation with Multiple Tribes of AI*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:14dde577-54bb-46a8-a61a-85a940e416e2

MORE EFFECTIVE TEST CASE GENERATION WITH MULTIPLE TRIBES OF AI

MITCHELL OLSTHOORN

More Effective Test Case Generation with Multiple Tribes of AI

More Effective Test Case Generation with Multiple Tribes of AI

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus, prof. dr. ir. T.H.J.J. van der Hagen,
chair of the Board for Doctorates
to be defended publicly on
Wednesday, 19 June 2024 at 12:30 o'clock

by

Mitchell Jean Gijsbert OLSTHOORN

Master of Science in Computer Science,
Delft University of Technology, the Netherlands,
born in Houten, the Netherlands.

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was funded by the University Blockchain Research Initiative (UBRI).

Keywords: Search-Based Software Testing, Test Case Generation, Fuzzing, Artificial Intelligence

Printed by: ProefschriftMaken

Cover by: Mitchell Olsthoorn

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

ISBN 978-94-6366-871-2

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

A team is not a group of people that work together. A team is a group of people that trust each other.

Simon Sinek

Contents

Summary

Software testing is important to make sure that code works as intended. Traditionally, this verification process has relied on manual testing, which is not only time-consuming but also susceptible to human errors. Through the use of automated test case generation techniques, we can automate this process and reduce the time and effort needed to test software. One of the promising techniques for automated test case generation is Search-Based Software Testing (SBST), which uses search-based metaheuristics to automatically generate test cases. SBST has been shown to be effective in generating test cases for a variety of programming languages and levels of testing (*e.g.,* unit, integration, and system testing). However, SBST is not without its challenges. One of the main challenges is the size of the search space that needs to be explored.

In this thesis, we explore the potential to improve the effectiveness and efficiency of automated test case generation by combining multiple tribes of Artificial Intelligence (AI) to narrow down the search space. First, we introduce two novel approaches that incorporate domain-specific knowledge into the search process to reduce the search space for automated test case generation. Then, we present two novel crossover operators. One uses hierarchical clustering to identify and preserve promising patterns within test cases. The other combines multiple crossover operators at different levels (*i.e.,* structure and data) to increase the diversity within the population. Next, we propose a model inference approach that infers dynamic types to allow automated test case generation of dynamically-typed languages. Finally, we introduce a new testing framework for two languages (*Solidity* and *JavaScript*) where no existing tooling existed.

The results of this thesis show that both approaches for incorporating domain-specific knowledge into the search process are effective in reducing the search space for automated test case generation. Thereby improving the effectiveness and efficiency of automated test case generation and increasing the structural coverage and fault detection capabilities of the generated test cases. Furthermore, the first crossover operator managed to detect and preserve promising patterns within test cases, thereby maintaining the structure of the test cases throughout the search process. The results of the second crossover operator show an increase in structural code coverage resulting from an improvement in the diversity of the population. Moreover, our results show that the model inference approach improves structural code coverage, bringing automated test case generation for dynamically-typed languages one step further. Finally, our new testing framework has demonstrated to be effective at generating test cases for *Solidity* and *JavaScript*.

In summary, this thesis introduced various novel approaches to improve the effectiveness and efficiency of automated test case generation by combining multiple tribes of AI to narrow down the search space. The results show that these approaches improve upon the state-of-the-art and hopefully are a step towards increasing the adoption of automated test case generation techniques in industry.

Samenvatting

Het testen van software is belangrijk om ervoor te zorgen dat code werkt zoals bedoeld. Normaliter wordt dit verificatieproces handmatig gedaan, wat niet alleen tijdrovend is maar ook gevoelig is voor menselijke fouten. Door het gebruik van technieken zoals geautomatiseerde test generatie kunnen we dit proces automatiseren en de tijd en moeite die nodig is om software te testen verminderen. Een van de veelbelovende geautomatiseerde test generatie-technieken is Search-Based Software Testing (SBST), dat zoekgebaseerde metaheuristieken gebruikt om automatisch testen te genereren. SBST heeft bewezen effectief te zijn in het genereren van testen voor verschillende programmeertalen en niveaus van testen (bijvoorbeeld *unit*-testen, integratietesten en systeemtesten). Toch heeft SBST ook zijn uitdagingen. Een van de belangrijkste uitdagingen is de omvang van de zoekruimte die moet worden verkend.

In deze scriptie onderzoeken we de mogelijkheid om de effectiviteit en efficiëntie van geautomatiseerde test generatie te verbeteren door meerdere richtingen van Artificiële Intelligentie (AI) te combineren om zo de zoekruimte te verkleinen. Eerst introduceren we twee nieuwe benaderingen die domeinspecifieke kennis opnemen in het zoekproces om hiermee de zoekruimte voor geautomatiseerde test generatie te verkleinen. Daarna presenteren we twee nieuwe kruisingsoperators. De eerste maakt gebruik van hiërarchische clustering om veelbelovende patronen binnen testen te identificeren en te behouden. De andere combineert meerdere kruisingsoperatoren op verschillende niveaus (*i.e.,* structuur en data) om de diversiteit binnen de populatie te vergroten. Vervolgens stellen we een modelinferentiebenadering voor die dynamische typen afleidt om geautomatiseerde test generatie van dynamisch getypeerde talen mogelijk te maken. Ten slotte introduceren we een nieuw test framework voor twee talen (*Solidity* en *JavaScript*) waarvan nog geen bestaande tools bestonden om de adoptie van geautomatiseerde test generatie in de industrie te bevorderen.

De resultaten van deze scriptie tonen aan dat beide benaderingen om domeinspecifieke kennis op te nemen in het zoekproces effectief zijn in het verkleinen van de zoekruimte voor geautomatiseerde test generatie. Hierdoor wordt de effectiviteit en efficiëntie van geautomatiseerde test generatie verbeterd en worden de structurele dekking en foutdetectiemogelijkheden van de gegenereerde testgevallen vergroot. Bovendien slaagde de eerste kruisingsoperator erin veelbelovende patronen binnen testgevallen te detecteren en te behouden, waardoor de structuur van de testen gedurende het zoekproces behouden bleef. De resultaten van de tweede kruisingsoperator tonen een toename in de structurele code-dekking als gevolg van een verbetering in de diversiteit van de populatie. Daarnaast tonen onze resultaten aan dat de modelinferentiebenadering effectief is in het afleiden van dynamische typen, waardoor geautomatiseerde test generatie voor dynamisch getypeerde talen een stap verder wordt gebracht. Ten slotte heeft ons nieuwe test framework aangetoond effectief testen te kunnen genereren voor *Solidity* en *JavaScript*.

Samenvattend introduceert deze scriptie verschillende nieuwe benaderingen om de effectiviteit en efficiëntie van geautomatiseerde test generatie te verbeteren door meerdere stammen van AI te combineren om de zoekruimte te verkleinen. De resultaten tonen aan dat deze benaderingen een verbetering zijn ten opzichte van de stand van de techniek en hopelijk een stap zijn in de richting van een grotere adoptie van geautomatiseerde test generatie-technieken in de industrie.

Acknowledgments

My PhD journey has flown by, which is not something I would have expected when I first started out. When looking back at the past four years, I realize that I have been very fortunate to have had the opportunity to work with so many great people. It was undoubtedly challenging, and filled with ups and downs, but I emerged with a wealth of newfound knowledge and experiences. I not only learned more about my research topic but also about myself in the process. Therefore, I would like to take this opportunity to thank everyone who has contributed to my journey.

First and foremost, I would like to thank my supervisors, Annibale and Arie, for their guidance and support throughout my PhD. Under your supervision and mentorship, I have become the researcher I am today. In addition, I want to thank the members of my thesis committee for their valuable feedback and for the time they have taken to read my thesis. Lastly, I would like to thank Ripple and in particular Lauren for making it possible to do this PhD and for our continued collaboration. In the next few paragraphs, I would like to thank some of the people who have been instrumental in my success:

Annibale: I could not have undertaken this journey without you. You have been more than a supervisor, you have been a mentor, a friend, and a role model. When you first approached me with the idea of doing a PhD, I was hesitant, but you convinced me that it was the right path for me. I believe that the supervisor is more important than the topic, and I am grateful that I had the opportunity to work with you. You have always been there for me, providing guidance, support, and encouragement. Through this, I have learned so much from you, not only about research but also about life. I am thankful for the many opportunities you have given me to grow as both a researcher and as a person. You have always believed in me, even when I did not believe in myself. *Grazie di tutto.*

Arie: I am very happy to have gotten you as a promotor. You're one of the most genuine and kind-hearted individuals I've had the pleasure of getting to know. I am grateful for the many insightful discussions we have had and for the guidance and wisdom you have provided me with. Even though you were often quite busy, you always made time for me, and I am thankful for that. In addition, as the head of the Software Engineering Research Group, you have created a welcoming and supportive environment for me to work in. This environment has made my PhD journey all the more enjoyable.

Xavier: You are a wonderful person and a great friend. I really appreciate you being there for me whenever I needed someone to talk to. Our conversations and your advice have been invaluable to me and have helped me open up and grow as a person. I will always cherish the moments with the (Post) Corona group in the city center with a drink in our hands. I am grateful for the many fun times we have had together, and for the many more to come.

Pouria: I have always looked up to you. You are a great researcher and a great friend. I am grateful for the many talks, discussions, and laughs we have had. I hope that we can continue to collaborate together in the future.

Caro: Thank you for being such a lovely human being! I always enjoy our conversations and always appreciate getting your feedback. We should definitely have more of our coffee breaks and board game evenings in the future.

Dimitri: It was a pleasure to work and collaborate with you! I want to thank you for your dedication to the *SynTest* project and I hope that you are happy at your new job.

Zeki: Thank you for the many coffee chats and the fun times we have had. I really appreciate your advice about both work and life.

June: Thank you for being such a great friend. Hanging out with you is always fun.

Antony, Imara, Leonhard: Thank you for being my office mates and making our office a fun place to work. *Antony*, you only joined our office recently, but you are a really cool dude. *Imara*, you are such a sweet person and I really appreciate the many laughs we have had in our office. *Leonhard*, even though we did not always agree on everything, I have always appreciated your honesty and the discussions we had.

SERG: I have been very fortunate to be part of the Software Engineering Research Group. This group is filled with kind, hard-working, and dedicated individuals. I want to thank all my current and past colleagues for the many fun times and all the nice coffee breaks and lunches we have had together. *Amir (Mir), Elvan, Joseph, Medhi:* Thank you for sharing this journey with me. *Amir (Deljouyi) and Ali (Khatami):* Thank you for the positive and friendly attitude that you both have. *Aru and Lorena:* Thank you for being so kind and for your dedication to the group. *Minaksie and Kim:* Thank you both for the positivity and energy that you bring to the group and for all the help during my PhD.

I want to also thank *Calin, Lisette, and Longfei:* and all the BSc thesis students with whom I have had the pleasure of working on their thesis projects and paper collaborations. I wish you all the success in your future endeavors.

Martijn: Wij kennen elkaar nu al meer dan 10 jaar, en in die tijd ben jij een van mijn beste vrienden geworden. Ik kijk altijd uit naar onze kookavonden en onze gesprekken die vaak tot diep in de nacht doorgingen. Elke keer dat we zeiden dat we nu naar huis gaan en dan toch nog 2 uur bleven praten over van alles en nog wat.

Michael: It is quite a scary thought that without this PhD journey, we probably would have never met. I am so grateful that we did, and to have you in my life. You have made these last 1.5 years so much more enjoyable and memorable, and have fundamentally changed my perspective on what is important in life. Love you so much!

Pap, Mam, Thom: Bedankt dat jullie er altijd voor mij zijn geweest en mij altijd hebben gesteund in alles wat ik heb gedaan. Zonder jullie zou ik niet zijn wie ik nu ben. Ik ben jullie ontzettend dankbaar voor alles wat jullie voor mij hebben gedaan.

While writing these acknowledgments, I realize that despite the challenges that a PhD poses, I will think back to this time with great memories and a smile on my face. Some of the best moments of my life have been during my PhD, and I am grateful for the many people who have made this possible. I am proud of what I have accomplished and am excited for what the future holds. Thank you all!

Mitchell
Delft, April 2024

1 Introduction

Software testing is a critical activity in the software development life cycle for quality assurance. This manual task, however, is tedious, expensive, and error-prone. As a consequence, researchers have developed various techniques for automating the process of generating test cases, thereby reducing the time needed for testing and debugging software. Among these techniques, Search-Based Software Testing (SBST) has shown promising results in finding bugs and achieving high coverage, and demonstrated its effectiveness in real-world applications. SBST is the application of search-based optimization techniques to the domain of software testing, where the goal is to find test cases that satisfy specific criteria, such as structural code coverage, fault detection, or some other pre-defined quality metric.

In this thesis, we introduce novel approaches for automated test case generation that combine multiple tribes of Artificial Intelligence (AI) to narrow down the search space and improve the effectiveness and efficiency of these techniques. These novel approaches (i) improve the current state-of-the-art in search-based automated test case generation, (ii) preserve promising structures within test cases, (iii) improve the diversity of the individuals in the population, (iv) enable automated test case generation for dynamically-typed languages, and (iv) introduce a modular and extensible ecosystem for automated test case generation that can generate tests for multiple programming languages.

This chapter is partly based on 🗎 *Mitchell Olsthoorn. More effective test case generation with multiple tribes of AI. 44th ACM/IEEE International Conference on Software Engineering 2022 (ICSE'22): Companion Proceeding* [1].

Software testing is an important part of quality assurance. Manually writing test cases, however, is a tedious and error-prone task that can take up to 50 % of developers' time [2, 3]. Over the last decades, researchers have developed techniques for automating the process of generating test cases [4]. These techniques significantly reduces the time needed for testing and debugging software [5]. Additionally, recent studies have shown that search-based approaches can achieve higher code coverage compared to manually written test cases [6, 7] and can identify unknown bugs [8–10]. Furthermore, automated test case generation tools have been successfully used in industry (*e.g.*, [11–13]).

Search-Based Software Testing (SBST) stands out as one of the most promising approaches for automated test case generation [4, 14]. SBST reformulates software testing as a meta-heuristic optimization problem, where a fitness function is used to measure the quality of different potential solutions. The purpose of a fitness function is to guide the search process to more promising solutions from a potentially infinite search space, where the search space represents the domain of all potential solutions.

The current state-of-the-art automated test case generation approaches use Evolutionary Algorithms (EAs) to evolve an initial set of randomly generated test cases over time. One of the reasons why EAs are so effective is 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 [15].

One of the key challenges in automated test case generation is the size of the search space [16]. Although EAs could, in theory, generate any possible input data given enough time, this would, however, be inefficient for complex data [17, 18]. The size and complexity of the search space depend on the System Under Test (SUT) and the chosen fitness functions. With the ever-increasing complexity of modern applications, generating test cases, which consist of input data, test structures, and assertions [19], that satisfy difficult constraints is challenging.

Although automated test case generation is becoming more common in large software companies, the widespread adoption of these techniques is lagging behind [20]. An additional factor for the lack of adoption of automated test case generation techniques in industry is the shortage of easy-to-use production-level tooling [21].

The overall goal of this thesis is to improve automated test case generation to eventually increase the adoption of these techniques by developers.

Hypothesis: *By combining multiple tribes of artificial intelligence to narrow down the search space, we can improve the effectiveness and efficiency of automated test case generation.*

If we look at the computer science field as a whole, there are techniques developed in different fields that could be used to help guide EAs.

First, we focus on **narrowing down the search space** by incorporating domain-specific knowledge into the search process. Often programs require input data that is highly structured, requires specific formats, or has underlying assumptions. We introduce two novel approaches where (i) uses grammars to augment EAs by limiting potential solutions thereby generating more structured input data and (ii) uses interprocedural control dependency analysis to uncover implicit assumptions (pre- and post-conditions) in the program under test.

Secondly, we focus on **preserving promising structures within test cases**. By evolving test cases over time, we risk breaking promising patterns (*i.e.,* sequences of method calls) that were present in previous test cases. We introduce a novel approach that uses hierarchical clustering to detect these sequences and preserve them during the search process.

Thirdly, we focus on **improving the diversity of the individuals in the population** by combining two different levels of crossover operators. Diversity within the population is important as it allows an EA to explore different areas of the search space and prevent premature convergence. We introduce a novel hybrid multi-level crossover operator that combines a crossover operator at the test structure and test data level.

Fourthly, we focus on **enabling automated test case generation for dynamically-typed languages**. Dynamically-typed languages are becoming more popular, but automated test case generation techniques for these type of languages are still in their infancy. These techniques focussed primarily on statically-typed languages as type information is needed to sample appropriate values for the parameters of a method call. We introduce a novel approach that uses model inference to infer the types of variables in dynamically-typed languages.

Lastly, we present an **extensible testing framework** capable of automatically generating test cases for two programming languages for which no such existing tooling existed.

1.1 Background

This section provides an introduction to search-based techniques, how they have been applied to software engineering and testing problems, and the different aspects of search-based automated test case generation.

1.1.1 Search-Based Software Engineering

Search-Based Software Engineering (SBSE) is the application of search-based optimization techniques to software engineering problems [22, 23]. Many of the problems and challenges we face as software engineers can be stated as optimization problems [24]:

- What is the smallest set of test cases that cover all branches in this program?
- What is the best way to structure the architecture of a particular system?
- What is the set of requirements that balances software development cost and customer satisfaction?
- What is the best allocation of resources to this software development project?
- How do we modify a program to make it more efficient in terms of speed and resources?
- What is the best sequence of refactoring steps to apply to this system?

Over the years, SBSE has been applied to a wide range of software engineering problems, including requirements engineering [25], design [26], testing [4, 27], and maintenance [28, 29]. In addition, several surveys have been conducted to summarize the field [23, 30, 31].

SBSE techniques are commonly used for software engineering problems as they impose few assumptions on the problem structure and can find near-optimal solutions. Only two key ingredients are needed to apply search-based techniques to a software engineering problem [22, 30]:

Representation The representation determines how a software engineering problem is encoded as a search problem. This encoding determines the size of the search space and the potential solutions that can be generated. Potential solutions must be encoded in a way that they can be manipulated by the search algorithm.

Fitness Function The fitness function is used to evaluate the quality of a potential solution and is used to guide the search to promising areas of the search space. It is tailored to a specific problem and must be redefined for each new problem. Luckily many problems in software engineering already have domain-specific metrics that can serve as a good initial candidate [32].

1.1.2 Search-based Software Testing

Search-Based Software Testing (SBST) is a subfield of SBSE that focuses on the application of search-based optimization techniques to software testing problems [4, 14, 27, 33]. It has been applied to a wide range of software testing problems, including unit testing [34], integration testing [35, 36], system-level testing [37], regression testing [38–40], and mutation testing [41].

Search-based techniques can be used throughout the whole software testing process. We can use fuzzing or fuzz testing to generate input data to test the SUT. The purpose of this technique is to find inputs that cause the SUT to crash or behave unexpectedly. Fuzzing requires a pre-defined main entry point, such as a function or method, to start the search process. Test case generation, on the other hand, generates test inputs, test structures (*i.e.,* entry points), and assertions, with the goal of testing the functional behavior of software applications. It generates test cases that maximize specific adequacy criteria, such as structural code coverage, fault detection, or some other pre-established quality metric. When a crash happens in a software application, we can use crash reproduction and fault localization to create a test case replicating the crash and finding the location of the fault. Finally, we can use patch synthesis to remedy the fault. In this thesis, we will mostly be focussing on automated unit-level test case generation.

SBST techniques can be categorized into three main categories: black-box, white-box, and grey-box [42]. Black-box techniques do not use any information about the internal structure of the program under test. White-box techniques, on the other hand, use the internal structure of the program under test to guide the search process. Finally, grey-box techniques use a combination of both black-box and white-box techniques.

Search-Based Optimization Algorithms

Random Search (RS) is an unguided heuristic optimization technique. It is quite common in the literature, as it is the simplest optimization algorithm, both conceptually and from an implementation perspective, and can be used as a baseline against which to compare more sophisticated approaches. The algorithm randomly selects points or candidates from within the search space until the goal is fulfilled or a stopping condition is met. Because of this simplistic approach, RS is not good at finding solutions when they occupy a small part of the overall search space or the search space is flat (flag problem).

Potential solutions can be found faster, by introducing guidance to the search process. These so-called fitness-guided search algorithms use a fitness function to evaluate the quality of a potential solution and guide the search to promising areas of the search space.

Algorithm 1: High-level overview of a hill climbing algorithm (maximizing)

```
   input  : Solution space $S = \{s_1, \ldots, s_n\}$
   output : Final solution s
1  begin
2      s ∈ S
3      repeat
4          ∃s' ∈ Neighborhood(s) : Fitness(s') > Fitness(s)
5          s ⟵ s'
6      until Fitness(s) ≥ Fitness(s') or ∀s' ∈ Neighborhood(s)
```

The simplest fitness-guided optimization algorithm is Hill Climbing. Alg. 1 shows a high-level overview of how a hill climbing algorithm works. Similar to RS, Hill Climbing starts with an initial random candidate solution (line 2). Potential solutions in the direct neighborhood are evaluated using the fitness function in an attempt to find a better solution. If a better candidate solution is found, the current solution is replaced with the better solution (lines 4 - 5). The algorithm then re-evaluates the direct neighborhood until no better solution can be found (line 6). This solution is a local optimum, as it is not guaranteed to be the best solution in the entire search space. Local search algorithms, like Hill Climbing, only consider one solution at a time and can only search in close proximity to the current candidate solution, and are therefore not able to escape local optima.

Algorithm 2: High-level overview of a genetic algorithm

```
   output : Final population P
1  begin
2      P ⟵ RandomPopulation()
3      repeat
4          ∀s ∈ P : EvaluateFitness(s)
5          parents ⟵ Selection(P)
6          offspring ⟵ Crossover(parents)
7          P' ⟵ Mutation(offspring)
8          P ⟵ Reinsertion(P')
9      until Stopping Condition Reached
```

A Genetic Algorithm (GA), on the other hand, is a global search algorithm inspired by the process of natural evolution. As a global search algorithm, GAs sample many potential solutions in the search space at once and, therefore, does not suffer from the local optima problem. EAs are one of the most commonly used classes of meta-heuristics in SBST. EAs have been used to generate both test data [4] and test cases [19]. Alg. 2 shows a high-level overview of how a GA works. The algorithm starts by creating an initial population of random candidate solutions (line 2). The fitness of each candidate solution is then evaluated using a pre-defined fitness function (line 4). A selection operator is applied to select the best candidate solutions from the population to be used as parents for the next generation (line 5). These parents are then recombined using a crossover (also called recombination) operator, swapping elements between the parents, to create new candidate solutions, called offspring (line 6). Then, the algorithm mutates the offspring to introduce new genetic material (line 7). Finally, the next generation of the population is chosen through the reinsertion operator (line 8). This process is repeated until all objectives have been achieved or a stopping condition is met (line 9).

Search Heuristics
Fitness-guided algorithms rely on test adequacy criteria, like structural code coverage and mutation score, to define search heuristics to optimize. The *approach level* and *branch distance* are well-known heuristics for *line* and *branch* coverage [3, 4]. The approach level is the number of control-dependent nodes, within the Control Flow Graph (CFG), between the nodes covered by the solution and the target node [43]. The CFG represents the flow-dependent relationships between the statements in the program. The nodes in the graph represent basic blocks, which are sequences of statements that are executed as a whole. In more technical terms, it is a sequence of program instructions with a single entry and exit point. The edges in the graph represent the flow of control between the basic blocks. A more intuitive understanding of the approach level is the number of if-statements that still need to be satisfied to reach the target node. The branch distance is much simpler and specifies how far away a solution (*i.e.,* test case) is removed from satisfying the condition, measured at the node where the flow of control had split. When these two heuristics are plotted within the search space, they form a fitness landscape. This landscape can be used to visualize the search process and to identify promising areas.

Flag Problem
The flag problem is a common issue in SBST that occurs when the condition within a control-dependent node is not explicit [44, 45]. Examples of such conditions are inline methods calls (*e.g.,* `if (isNull(y))`) or when conditions read boolean variables (*e.g.,* `if(x && y)`). To address this problem, researchers have proposed *testability transformations* [45]. These transformations modify the program under test in a way that preserves its semantics but replaces conditions with predicates that involve non-boolean variables. Prior studies have shown that testability transformations dramatically improve code coverage without the need for adapting the underlying search algorithms [44, 46, 47].

Test Oracles
When automatically generating test cases, we need to be able to determine whether the test case is correct or not. Most existing literature makes the conscious decision to assume that the output of the automatically generated test is correct and leaves the verification

to the end-user. However, determining whether a test case is correct or not is a difficult problem, as it requires a human to understand the program under test and the test case. This is often seen as one of the drawbacks of automated test case generation techniques.

1.1.3 Large Language Models (LLMs)

Recent developments and advances in Large Language Models (LLMs) have been used in various approaches within the context of automated test case generation [48–51]. Although LLMs have promise in the field of automated test case generation as they generally excel at test case readability, they are not without their limitations.

Schafer *et al.* [48] evaluated the effectiveness of OpenAI's GPT3.5-Turbo without additional training. They found that the model can generate state-of-the-art statement coverage on all 25 NPM packages included in their benchmark. They have observed, however, that the size and training set of the LLM have a significant impact on the effectiveness of the generated test cases. Siddiq *et al.* [49] evaluated three popular LLMs (Codex, GPT3.5-Turbo, and StarCoder) on two different datasets (HumanEval and the SF110 benchmark from *EvoSuite*) to investigate the compilation rates, test correctness, coverage, and test smells. They found that for HumanEval between 37.5 % and 70 % of the generated tests were compilable depending on the model they used. For SF110 (arguably the more interesting benchmark as it is commonly used in literature), however, only 2.7 % of the generated tests were compilable. Of these compilable tests, only 52 % were correct (*i.e.,* all test methods passed). Even though these models generated tests with between 67 % and 92.8 % of structural overage for HumanEval, all models produced less than 2 % coverage for SF110. These numbers are 11-19 times lower than the coverage achieved by *EvoSuite* for SF110. Finally, the generated tests also suffered from test smells, such as duplicated assertions and empty test cases. Dakhel *et al.* [50] performed a similar study but with a focus on fault detection capability. Their findings show that although LLMs can serve as a useful tool to generate test cases, they require specific post-processing steps to enhance the effectiveness of the generated test cases, which may suffer from syntactic or functional errors and may be ineffective at detecting faults. Finally, Ouyang *et al.* [51] found that LLMs can be highly unstable because of their non-deterministic behavior, where very different responses are returned for the same prompt. In comparison SBST techniques have actually shown to generate fewer code smells than their manually written counterparts [52].

1.2 Challenges

This section discusses key challenges that we seek to address in this thesis.

1.2.1 Size of the Search Space

One of the key challenges in search-based automated test case generation is the size of the search space [16]. Even when we limit the search space to a single method, the number of possible inputs can be very large. Moreover, with modern applications becoming more complex, the search space is growing exponentially. Programs that require complex input data are usually highly structured, require specific formats, and/or domain-dependent. Previous studies have shown that automatically generated input is often unstructured and can be difficult to read and interpret [17, 18]. Furthermore, domain-dependent input data can be difficult to generate without more information about the underlying implicit requirements of the input data.

These implicit requirements cause a lack of guidance for the search process and can be difficult to infer. Even though solutions are limited to a specific region of the search space, the search-based techniques waste valuable time trying potential solutions that will not work. For example, a program that requires a date as input can accept any date, but the program might only accept dates in a specific format (*e.g.,* ISO 8601). Additionally, if a generic data type (*e.g.,* string) is used to represent the date, the chance of generating a valid date is very low. Programs that require highly structured input data are similarly difficult to generate as structural constraints need to be satisfied. Highly structured data is quite common in applications today with common data formats like JavaScript Object Notation (JSON) and Extensible Markup Language (XML) that are used to send information between applications.

1.2.2 Preserving Promising Structures

Another challenge in search-based automated test case generation is the preservation of promising structures within test cases. Test cases consist of input data, test structures (*i.e.,* sequences of method calls that are executed in a specific order), and assertions [19]. The building block hypothesis states that recombining multiple fit building blocks (*i.e.,* partial solutions) into even fitter larger building blocks is the primary source behind the success of EAs, where these smaller building blocks are reused across multiple test cases [53]. However, while crossover/recombination operators are effective at combining promising building blocks, they they do not directly recognize and preserve them [54]. As a result, the search process can break promising structures that were present in previous test cases.

In particular, these crossover operators can cause the quality, in terms of achieved coverage and comprehensibility, of the test cases to deteriorate as they do not understand what makes a test case effective. For example, in a collection (*i.e.,* list of items) class where the `remove` method is called before the `add` method, the `remove` method will always fail. Another example is a SUT where a `login` method needs to be called before any other method. Without identifying these sequences of method calls, the search process will waste valuable time trying potential solutions that will not work.

1.2.3 Population Diversity

A common issue with genetic algorithms is premature convergence, which is strongly tied to the loss of diversity within the population [55, 56]. Premature convergence occurs when the population narrows down to a solution, before the search space has been explored properly. Genetic algorithms strive to maintain a balance between exploration (*i.e.,* exploring new areas in the search space) and exploitation (*i.e.,* refining existing solutions) [55, 57]. This balance is adjusted by making changes to the selection, crossover, and mutation operators. The selection operator determines the selection pressure, and the higher the selection pressure, the more likely it is that the best individuals are selected for crossover (*i.e.,* exploitation). Conversely, the crossover and mutation operator determine how an encoding is changed, with a higher probability of crossover and mutation making it more likely that the offspring will be different from the parents (*i.e.,* exploration).

Having high diversity in the population gives three main benefits [56]. Firstly, diversity allows genetic algorithms to explore different regions of the search space simultaneously, increasing the chance of finding the global optimum. Secondly, multi-objective optimization involves optimizing multiple conflicting objectives simultaneously. Diversity ensures that the genetic algorithm can captures a wide range of trade-offs between different objectives. Lastly, in certain optimization problems, the landscape changes over time due to factors such as evolving objectives. When this happens, individuals with diverse characteristics may become more suitable for the new situation. If the population lacks diversity, it might struggle to adapt to such changes effectively. An example of this can be seen with automated test case generation, where objectives change over time to cover different parts of the SUT [58].

1.2.4 Dynamically-Typed Language Support

Most search-based automated test case generation research is focused on statically-typed programming languages like Java (*e.g., EvoSuite* [34]) and C (*e.g.,* AUSTIN [59]). They make use of the static type information to (i) generate suitable input data, and (ii) search guidance calculations. Without this type information, these approaches have to randomly guess which types are compatible with the parameter specification of the constructor or function call, greatly increasing the search space. As reported by Lukasczyk *et al.* [60], state-of-the-art approaches used for statically-typed languages do not perform well on *Python* programs when type information is not available.

Dynamically-typed programming languages introduce new challenges for unit-level automated test case generation. The lack of type information makes it difficult to generate suitable input data. According to the survey from Stack Overflow[1], however, *Python* and *JavaScript* are the two most commonly-used programming languages. As both of these languages are dynamically-typed, there is a need for better automated test case generation techniques for these types of languages.

1.2.5 Lack of Adoption in Industry

The last challenge that we will highlight is the lack of (general) adoption in industry of search-based testing approaches. Although automated test case generation is becoming more common in large software companies, the widespread adoption of these techniques is lagging behind [20]. One of the factors for this lack of adoption in industry is the shortage of easy-to-use production-level tooling [21]. Most of the research in this field focuses on the effectiveness of the automated test case generation and not on the usability of the tooling. As state-of-the-art genetic algorithms are often complex and hard to understand, it is difficult for developers to use these tools in practice. Additionally, these types of algorithms have many optimization parameters that need to be tuned to the specific problem at hand. Given this complexity, it is not trivial to build tooling for each language and/or testing framework. Another factor is that the tooling is often not integrated into the existing development environment, making it more cumbersome to use.

[1]https://survey.stackoverflow.co/2022/#most-popular-technologies-language

1.3 Research Goals & Questions

The goal of this thesis is to tackle the existing challenges of search-based automated test case generation to eventually increase the adoption of these techniques by developers. Although Evolutionary Algorithms (EAs) could, in theory, generate any possible input data given enough time, this would, however, be inefficient for programs requiring complex data as the search space grows exponentially [17, 18]. We 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. To make the goal more concrete and limit the scope of this work, we focus on the following research questions:

The problem with programs that require complex input data is that the data is usually highly structured, requires specific formats, and/or is domain-dependent. Without more information about the input data, EAs struggle to generate such data. Therefore, the first research question focuses on how we can leverage domain-specific knowledge to steer the search process toward better solutions.

RQ$_1$: *How can we reduce the search space for automated test case generation by using domain-specific knowledge?*

Generating input data is only the first step in the automated test case generation process. After we manage to create a suitable input for the program under test, we need to make sure that the algorithm does not break promising patterns of method sequences within the test cases. Therefore, the second research question focuses on detecting and preserving these patterns within the test cases.

RQ$_2$: *How can we preserve promising structures within test cases throughout the search process?*

Without enough diversity within the population, the search process can prematurely converge to a solution before the search space has been properly explored. Therefore, the third research question focuses on how we can increase the diversity of the individuals in the population.

RQ$_3$: *How can we increase the diversity of individuals in the population?*

Current state-of-the-art approaches for automated test case generation focus on statically-typed languages. However, dynamically-typed languages are gaining popularity. Therefore, the fourth research question focusses on integrating dynamic type information into the search process.

> **RQ$_4$:** *How do we increase the effectiveness and efficiency of automated test case generation for dynamically-typed languages?*

Finally, with the shortage of easy-to-use production-level tooling [21], the last research question focusses on how we can construct a platform that allows multiple different programming languages to be tested within one ecosystem.

> **RQ$_5$:** *How can we make a platform for automated test case generation that supports multiple programming languages?*

1.4 Research Methodology

This thesis answers the aforementioned research questions by following the Design Science Research Methodology (DSRM) [61–63]. DSRM is a widely employed approach in empirical computer science and other engineering disciplines. This methodology uses a structured approach to address real-world problems by combining scientific research with practical engineering to produce both scientifically sound and practically valuable solutions.

We chose this methodology because it is well-suited for exploratively solving practical problems in software engineering. Our goal consists of using applied research to develop new techniques and tools to improve automated test case generation and then evaluate these techniques and tools in an empirical and/or industrial setting.

DSRM consists of four steps:

1. **Problem Identification:** The first step in DSRM is to identify the problem that needs to be addressed. This step involves surveying what has been done in related literature and identifying the gap that needs to be addressed.
2. **Design and Creation of Artifacts:** When the problem has been identified, the next step is to design and create artifacts to address the unsolved problem. This is the core concept of the design science methodology. Artifacts can take many forms, including software prototypes, algorithms, or conceptual models.
3. **Evaluation:** After the artifact has been created, it needs to be evaluated. The evaluation assesses the artifact's effectiveness, quality, and efficiency for the identified problem. The evaluation can be done in an artificial environment or through the use of a case study. The method used to evaluate the artifact depends on the artifact's nature and the problem domain.
4. **Reflection:** Based on the results gathered from the evaluation, researchers reflect on the artifact's performance and identify trends and insight that could provide value for future research.

This process is inherently iterative, involving the generation of multiple artifact versions or prototypes. Each version is refined and improved based on information gathered from the evaluation and may cycle through the different stages several times to refine the artifact and improve its quality.

In this thesis, we focus on the challenges of search-based automated test case generation as laid out in Section 1.2. We make use of a quantitative research approach to answer the research questions. When designing our study, we make use of the guidelines for empirical studies in software engineering [64]. For each study, we identify if an existing code base and/or benchmark suite is available or if we need to create our own. We then identify the research questions that we want to answer and the metrics that we will use to answer these questions. To evaluate the effectiveness of our approaches, we use the well-established evaluation metrics of structural code coverage and mutation score. In addition, we use coverage over time to measure the efficiency of an algorithm. To establish a baseline, we compare our approaches to the state-of-the-art in the field of automated test case generation. To mitigate the risk of bias, we make use of the guidelines for assessing randomized algorithms in software engineering [65, 66]. We perform multiple repetitions of the experiment and make use of statistical analysis to assess the results. In particular, we make use of the unpaired Wilcoxon rank-sum test [67] for the statistical significance and the Vargha-Delaney statistic [68] for the effect size.

1.5 Research Outline

This section gives an overview of the chapters included in this thesis. Table 1.1 outlines the connections between the different research questions and chapters in the thesis.

Chapter 2: Certain types of applications require highly structured input data, parsers are an example of this. Automated test case generation has limitations in creating such data. Previous studies have shown that automatically generated input is mostly unstructured [17, 18]. Grammar-based fuzzing, on the other hand, is very effective in generating highly structured input data based on a user-specified grammar [69, 70]. For this reason, fuzzing has been widely used for security and system testing [71, 72]. However, since grammar-based fuzzing only generates input data, developers need to manually create the structure of the test case, and come up with their own assertions. To address these limita-

Table 1.1: Connection of chapters with research questions

Research Question	Chapters
RQ_1: How can we reduce the search space for automated test case generation by using domain-specific knowledge?	2, 3
RQ_2: How can we preserve promising structures within test cases throughout the search process?	4
RQ_3: How can we increase the diversity of individuals in the population?	5
RQ_4: How do we increase the effectiveness and efficiency of automated test case generation for dynamically-typed languages?	6
RQ_5: How can we make a platform for automated test case generation that supports multiple programming languages?	7

tions, we propose a novel approach that combines the strength of grammar-based fuzzing and EAs to narrow than the search space within the context of the JSON data format. 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 to limit the number of possible actions that can be performed on a SUT and improve the quality of the test data. We evaluated our approach on three popular Java JSON parsers and showed that it can improve the effectiveness of automated test case generation without negatively impacting the performance of non-JSON-related classes. On average, the proposed approach achieves +15 % of branch coverage compared to the baseline. The largest improvement that was observed in the study was +50 % of branch coverage for one of the classes in the benchmark.

Chapter 3: 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). Transaction-reverting statements are key constructs within *Solidity* that are extensively used for such checks. These statements protect smart contracts against invalid requests by reverting the transaction when the conditions are not met [73]. These statements, however, are not part of the control flow of the method they are applied to, making it difficult for the EA to satisfy the condition within them. This creates partially flat fitness landscapes, forcing the search algorithm to resort back to random testing. To address this problem, we propose a novel approach that restores 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 search algorithm. We evaluated our approach on 100 real-world smart contracts and showed that it can improve the effectiveness of automated test case generation. On average, we improve transaction-reverting statement coverage by 14 % (up to 35 %), line coverage by 8 % (up to 32 %), and vulnerability-detection capability by 17 % (up to 50 %).

Chapter 4: EvoMaster is a state-of-the-art automated test case generation tool for Java REpresentational State Transfer (REST) Application Programming Interface (API) testing [37]. While the state-of-the-art algorithms can successfully create promising sequences of Hypertext Transfer Protocol (HTTP) requests, they do not directly recognize and preserve them when creating new test cases [54]. As REST APIs are stateful, 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. In Chapter 4, we argue that detecting and preserving these patterns, referred to as *linkage structures*, improves the effectiveness of the automated test case generation process and maintains the quality of the test cases. We propose a novel approach, called *LT-MOSA*, that uses Agglomerative Hierarchical Clustering (AHC) to infer these linkage structures from automatically generated test cases. These linkage structures are then 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, we implemented this approach within *EvoMaster* and performed an empirical study with 7 real-world benchmark web/enterprise applications from the EvoMaster Benchmark (EMB) dataset. The results show that 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*). In particular, *LT-MOSA* achieves significantly higher structural coverage in 4 and 5 out of the 7 applications compared to *MIO* and *MOSA*, respectively, and can detect, on average, more unique real-faults that were not detected by the baselines.

Chapter 5: The encoding used to represent a test case within an EA consists of both test data and method sequences [19]. Current state-of-the-art EAs for automated test case generation use a crossover operator (*i.e.,* single-point) that swaps a group of method sequences between two test cases [58, 74]. These operators only change the test structure and simply copy over the corresponding test data Consequently, the input data of the offspring is often very similar to the input data of the parents. In Chapter 5, we argue that a hybrid crossover operator that alters both the structure of the test cases as well as the test data can improve the test case diversity. To validate this hypothesis, we propose a new operator, called Hybrid Multi-level Crossover (HMX), that combines different crossover operators on multiple levels (*i.e.,* data and method level). We implemented this hybrid operator within *EvoSuite* [34] and performed an empirical study with 116 classes from the Apache Commons and Lucene Stemmer projects, which include classes for numerical operations and string manipulation. The results show that *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). On average, *HMX* achieves 6.4 % and 7.2 % more branches and lines covered than the baseline, respectively (with a max improvement of 19.1 % and 19.4 %) and 3.9 % (max. 14 %) and 2.1 % (max. 12.1 %) for weak and strong mutation, respectively.

Chapter 6: Search-based automated test case generation approaches make use of type information to determine what input data to generate for a given method. The lack of type information in dynamically-typed languages makes it difficult to generate suitable input data. As a result, the search space grows exponentially, making the overall search process less effective. We propose a novel unsupervised probabilistic type inference approach that infers the data types of the parameters of a method. More specifically, we statically analyze the source code of the method under test to extract the relationships between the parameters and the variables used within the method and deduct the most likely data type for each parameter. The inferred types are then used to guide the search process towards better solutions. We evaluate our inference model on a benchmark of 98 units under test (*i.e.,* exported classes and functions) and compare the results to a random type sampling baseline *w.r.t.* branch coverage. Our results show that our type inference approach achieves a statistically significant increase in 56 % of the test files with up to 71 % of branch coverage compared to random type sampling.

Chapter 7: One of the reasons for the lack of adoption of automated test case generation techniques in industry is the absence of easy-to-use production-level tooling [21]. There are several state-of-the-art tools for automated test case generation, but they are often primarily focused on research and not on usability. Examples of these are *AUSTIN* [59], *EvoSuite* [34], and *Pynguin* [60]. Within the context of white-box testing, tools are tightly coupled to the underlying programming language, making it challenging to build tooling for each language and/or testing framework. In Chapter 7, we present *SynTest-Framework*, a modular and extensible framework for automated test case generation. The main goal of the framework is to provide an ecosystem of testing tools that are easy to use and come with a collection of pre-defined presets for different search algorithms. *SynTest-Framework* is designed to be language agnostic and can be extended to support new languages and testing frameworks. Currently, the framework supports *JavaScript* and *Solidity*. To improve the usability of the framework, we provide an online web service for users to generate test cases without the need to install any software.

Chapter 8: In the last chapter, we revisit the research questions posed in this chapter, summarize our findings, and make conclusions based on the results of the studies. Additionally, we discuss the limitations of this work and provide recommendations for future directions.

1.6 Origins of the Chapters

This thesis is based on a collection of papers written during the Ph.D. that act as the foundation of this work. The included papers have previously been published in peer-reviewed conferences. Each paper in this portfolio-style thesis can therefore be read independently and contains a dedicated background, related work, and conclusion section.

To better fit the style of this thesis, the layout and formatting of the papers have been adapted. Additionally, figures and tables were enlarged to make them easier to read. Even though the papers have been adapted and in some cases merged the main content of the papers has not been changed.

For all chapters, except Chapters 4 and 6, the author of this thesis was the first author. For papers in collaboration with B.Sc. and M.Sc. students (*i.e.,* Chapters 4 and 6) the authors are ordered by seniority. The author of this thesis was the main responsible for the design of the algorithms and experiments, the analysis of the results, and the writing of the paper.

- **Chapter 2** was published in the paper "Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing" by Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella at the 35th IEEE/ACM International Conference on Automated Software Engineering 2020 (ASE'20) [75].

- **Chapter 3** was published in the paper "Guiding Automated Test Case Generation for Transaction-Reverting Statements in Smart Contracts" by Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella at the 38th IEEE International Conference on Software Maintenance and Evolution 2022 (ICSME'22) [76].

- **Chapter 4** was published in the paper "Improving Test Case Generation for REST APIs Through Hierarchical Clustering" by Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella at the 36th IEEE/ACM International Conference on Automated Software Engineering 2021 (ASE'21) [77].

- **Chapter 5** was published in the paper "Hybrid Multi-level Crossover for Unit Test Case Generation" by Mitchell Olsthoorn, Pouria Derakhshanfar, and Annibale Panichella

Table 1.2: Connection of chapters with replication packages.

Chapter	Host	DOI	Replication Package
2	Zenodo	10.5281/ZENODO.4001744	[84]
3	Zenodo	10.5281/ZENODO.6787666	[85]
4	Zenodo	10.5281/ZENODO.5106027	[86]
5	Zenodo	10.5281/ZENODO.5102597	[87]
6	Zenodo	10.5281/ZENODO.7088684	[88]
7			

at the 13th International Symposium on Search-Based Software Engineering 2021 (SSBSE'21) [78].

- **Chapter 6** was published in the paper "Guess What: Test Case Generation for Javascript with Unsupervised Probabilistic Type Inference" by Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella at the 13th International Symposium on Search-Based Software Engineering 2022 (SSBSE'22) [79].
- **Chapter 7** is based on

  "SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts" by Mitchell Olsthoorn, Dimitri Stallenberg, Arie van Deursen, and Annibale Panichella at the 44th ACM/IEEE International Conference on Software Engineering 2022 (ICSE'22): Companion Proceeding [80] and

  "SynTest-JavaScript: Automated Unit-Level Test Case Generation for JavaScript" by Mitchell Olsthoorn, Dimitri Stallenberg, and Annibale Panichella at the 17th IEEE/ACM International Workshop on Search-Based and Fuzz Testing 2024 (SBFT'24) [81].

1.7 Open Science

Open Science is the "movement that aims at more open and collaborative research practices in which publications, data, software and other types of academic output are shared at the earliest possible stage and made available for reuse" [82]. It is a broad term that encompasses many different aspects of the scientific process. Three principles of Open Science are (i) Open Source, (ii) Open Data, and (iii) Open Access [83]. Open Source relates to making use of open-source technology whenever possible and making your own code available as open source. Open Data refers to making datasets used for research publicly available so others can replicate the results based on the data. Open Access ensures that scientific publications and research findings are freely accessible to the public. By making use of Open Science, we can ensure that our research is transparent, reproducible, and accessible to everyone.

1.7.1 Open Datasets

As part of Open Data, we make the replication packages used for the experiments publicly available. By doing this, we hope to increase the reproducibility of our results and

Table 1.3: Overview of created experimental benchmarks.

Benchmark	Chapter	Host
Solidity contracts testing benchmark	3, 7	GitHub
JavaScript libraries testing benchmark	6, 7	GitHub

Table 1.4: Overview of software where the various approaches have been integrated with.

Chapter	Tool
2	EvoSuite
3	SynTest-Solidity
4	EvoMaster
5	EvoSuite
6	SynTest-JavaScript
7	

allow others to build upon our work. Table 1.2 shows the connection between the chapters and their corresponding replication packages. All replication packages have been made available on Zenodo[2]. Chapter 7 is a tool demonstration and therefore does not have a replication package. The software package for this chapter can be found in the next subsection (See Section 1.7.2). To increase the replicability of the experiments, we make use of open-source software and Docker[3] containers for the experiment runner infrastructure.

Additionally, in this thesis, we created two new benchmarks for running experiments. These benchmarks have been used in multiple chapters of this thesis as can be seen in Table 1.3. The first benchmark contains a diverse set of Solidity smart contracts that can be used for testing smart contract systems on Ethereum. Even though existing benchmarks for Solidity smart contracts exist, previous related literature has criticized these benchmarks as they do not contain a diverse selection of contracts with different complexities, versions, and language features [89]. The second benchmark contains a diverse set of JavaScript libraries that can be used for evaluating unit-level automated test case generation for JavaScript. To the best of our knowledge, no such benchmark existed for JavaScript previously. Both benchmarks are hosted on GitHub under the SynTest Framework organization[4].

By complying with Open Science, authors have the responsibility to make sure their research data is consumable by others. One set of guiding principles for scientific data management and stewardship are the FAIR principles [90, 91]. FAIR stands for Findable, Accessible, Interoperable, and Reusable. We have used these principles throughout this thesis to make sure that our data complies with the Open Science guidelines.

Table 1.5: Overview of main code contributions.

Project	Chapters	Host	License	#Commits	SLOC	Language (Main)
EvoSuite Experiment Runner	2, 5	GitHub	GNU GPLv3	16	324	Bash
SynTest-Framework	3, 6, 7	GitHub	Apache 2.0	797	19400	TypeScript
SynTest-Solidity	3, 7	GitHub	Apache 2.0	488	6062	TypeScript & JavaScript
SynTest-JavaScript	6, 7	GitHub	Apache 2.0	219	18088	TypeScript
Σ				1520	43874	

1.7.2 Open Software

In addition to the data, we have also made the software implementations of the various approaches in this thesis publicly available. Table 1.4 shows the connection between the chapters and the tool that the approach has been implemented on and integrated with. By implementing the approaches in existing tools, we hope to increase the adoption of our research by the community. The tools chosen for the different approaches are all state-of-the-art tools in their respective domains. EvoSuite [34] is a state-of-the-art unit-level automated test case generation tool for Java. It has been used extensively in previous research and is one of the most popular tools in the field. EvoMaster [37] is a state-of-the-art system-level automated test case generation tool for REST APIs. It contains both black-box and white-box approaches for testing REST APIs.

For some of the challenges that this thesis addresses, no existing tools were available. In these cases, we have created new tools and frameworks to implement the approaches. By doing this, we hope it will be easier for others to build upon our work. Table 1.5 shows an overview of the main code contributions made by the author of this thesis.

EvoSuite Experiment Runner *EvoSuite* is a command-line tool that can be used to generate test cases for Java programs. However, it does not contain any infrastructure for running experiments. Therefore, we have created a new experiment runner for EvoSuite and published it under the GNU GPLv3 license. This runner allows us to run experiments in a reproducible way and collect the results in a structured format. Additionally, it allows us to run experiments in parallel across many cores. It accepts a configuration file that specifies the parameters of the experiment and what benchmark classes to run.

SynTest-Framework SynTest-Framework is an open-source ecosystem for automated test case generation and fuzzing based on TypeScript, published under the Apache License 2.0. It contains a collection of language-independent search algorithms that are optimized for automated test case generation. The framework uses a modular extensible architecture to allow testing tools for different programming languages to be built on top of it. Our main goal with the framework is to make it easier for researchers to implement new approaches for automated test case generation. Additionally, we hope that the framework will make it easier for practitioners to adopt automated test case generation in their projects.

[2]https://zenodo.org/
[3]https://www.docker.com/
[4]https://github.com/syntest-framework

SynTest-Solidity SynTest-Solidity is an open-source unit-level testing tool built on top of the SynTest-Framework for automated test case generation and fuzzing for Solidity smart contracts, published under the Apache License 2.0. This project contains the different interfacing components (*e.g.,* static analysis, instrumentation, guidance, and coverage collection) to test a language-specific application.

SynTest-JavaScript Similarly to SynTest-Solidity, SynTest-JavaScript is an open-source unit-level testing tool built on top of the SynTest-Framework but focusses on automated test case generation and fuzzing for server-side JavaScript libraries, published under the Apache License 2.0. It contains a state-of-the-art type inference model to infer the types of variables in the application.

1.8 Other Contributions

In addition to the publications included as part of this thesis, I co-authored a number of papers that I shortly describe in the following:

- In the FSE'24 industry paper "Evolutionary Generative Fuzzing for Differential Testing of the Kotlin Compiler", we propose a novel approach for differential testing of compilers. Our approach uses evolutionary generative fuzzing to generate Kotlin programs that are syntactically and semantically valid, and exploit a diverse set of language features. We perform an empirical evaluation with two different versions of the Kotlin compiler front-end and discover multiple critical bugs.

- In the ICSE'24 tool demo paper "TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion", we propose an IntelliJ IDEA plugin that incorporates multiple automated test case generation techniques into the Integrated Development Environment (IDE). Our tool allows developers to make use of state-of-the-art testing techniques in a user-friendly way without leaving their IDE. TestSpark is available on the JetBrains Marketplace[5].

- In the SBFT'23 paper "Grammar-Based Evolutionary Fuzzing for JSON-RPC APIs" [92], we propose the first (black-box) approach for automated fuzzing of JSON-Remote Procedure Call (RPC) APIs. In particular, we empirically evaluate the effectiveness of grammar-based evolutionary fuzzing against random grammar-based fuzzing on the XRP ledger, a large-scale industrial blockchain system that uses JSON-RPC APIs. Our approach uses hierarchical clustering [77, 93] to group similar API responses to determine what type of input is most likely to uncover new path traces in the underlying application.

- In the SSBSE'20 challenge paper "An Application of Model Seeding to Search-Based Unit Test Generation for Gson" [94], we investigate if model seeding can improve the effectiveness of search-based test generation for the popular Java JSON parsing library, called GSON. Seeding consists of injecting additional information (*e.g.,* manually-written test suites) for use in the search process [95]. A previous study

[5]`https://plugins.jetbrains.com/plugin/21024-testspark`

proposed a method to infer a behavioral model from the usage patterns of applications in the context of crash reproduction [96]. This paper adapts this approach to the context of unit-level test generation and performs and empirical study.

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

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.*

This chapter has been published as *Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing. 35th IEEE/ACM International Conference on Automated Software Engineering 2020 (ASE'20)* [75].

2.1 Introduction

Software testing is a critical activity for quality assurance and can take up to 50 % of developers' time [97]. 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 [4]. Recent advances show that search-based approaches can achieve higher code coverage compared to manually written test cases [6, 7]. They can also detect unknown bugs [8–10] and have been successfully used in industry (*e.g.,* [11–13]). Moreover, automatic test case generation significantly reduces the time needed for testing and debugging [5].

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 [17, 18]. 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 [37]. 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 [98].

Grammar-based fuzzing is very effective in generating highly-structured input data based on a user-specified grammar [69, 70]. For this reason, fuzzing has been widely used for security and system testing [71, 72]. 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* [34], 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 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 [84].

2.2 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.

2.2.1 Search-based Test Case Generation

Various search-based test case generation approaches have been proposed in the literature (*e.g.,* [4, 34, 99]). These approaches rely on test adequacy criteria (*e.g.,* branch coverage [4, 100]) and evolutionary algorithms (*e.g.,* genetic algorithms [4, 74, 101]). 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 [4]. 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 [74]. *EvoSuite* can optimize multiple adequacy criteria simultaneously [100, 102]. 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) [58]. 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 [74, 101]. 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* [58].

2.2.2 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 [103]. To fuzz, developers need to specify the entry points of the application under test. Whitebox fuzzing aims to generate test inputs that, when applied to specified entry points, allow satisfying/covers different program conditions [104]. Fuzzing can use different engines for the test data generation [104], such as symbolic execution [71], meta-heuristics [105], grammars [69], and hybrid approaches [106].

Grammar-based fuzzing generates well-formed inputs by relying on a user-specified grammar [69, 107]. 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 [104]. Typically, the grammars encode application-specific knowledge of the program under test [69]. As shown by previous work, grammar-based fuzzers are very effective in creating highly-structured inputs for applications like compilers and interpreters [69, 70].

2.2.3 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.

2.3 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 2.2). In the following paragraphs, we explain the changes we introduced in DynaMOSA to incorporate the grammar-fuzzer.

2.3.1 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

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

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.

2.3.2 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 [58]. 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.

2.3.3 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.

2.4 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 2.1) 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 [6, 9, 108].

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 [109]. 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.

2.4.1 Results

Table 2.1 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**).

Table 2.1: 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)

2

2.5 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.

3 Guiding Automated Test Case Generation for Transaction-Reverting Statements in Smart Contracts

Transaction-reverting statements are key constructs within Solidity that are extensively used for authority and validity checks. Current state-of-the-art search-based testing and fuzzing approaches do not explicitly handle these statements and therefore can not effectively detect security vulnerabilities. In this paper, we argue that it is critical to directly handle and test these statements to assess that they correctly protect the contracts against invalid requests. To this aim, we propose a new approach that improves the search guidance for these transaction-reverting statements based on interprocedural control dependency analysis, in addition to the traditional coverage criteria. We assess the benefits of our approach by performing an empirical study on 100 *smart contracts w.r.t. transaction-reverting statement coverage and vulnerability detection capability. Our results show that the proposed approach can improve the performance of DynaMOSA, the state-of-the-art algorithm for test case generation. On average, we improve transaction-reverting statement coverage by* 14 % *(up to* 35 %*), line coverage by* 8 % *(up to* 32 %*), and vulnerability-detection capability by* 17 % *(up to* 50 %*).*

This chapter has been published as *Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. Guiding Automated Test Case Generation for Transaction-Reverting Statements in Smart Contracts. 38th IEEE International Conference on Software Maintenance and Evolution 2022 (ICSME'22)* [76].

3.1 Introduction

Ever since its launch in 2015, *Ethereum* has been the largest and most prominent smart contract platform [110]. One key property of these smart contracts is that once a contract has been deployed, it cannot be updated [111]. This property makes sure that contracts that are in use on the platform cannot be altered by the creators of the contract for their benefit. However, this creates certain challenges, *e.g.,* what happens if a bug is discovered. This greatly increases the importance of quality assurance in the smart contract development lifecycle. In the last few years, various search-based methods have been developed to assist developers with this problem, like *fuzzing* [112–117] and *test case generation* [118].

Smart contracts on the *Ethereum* platform are written in *Solidity*, a high-level smart contract language. *Solidity* is a transactional language, meaning transactions either succeed or fail. Hence, it is impossible for the contract to be in a broken state. To accomplish this, the language makes use of *transaction-reverting statements*, which allow developers to check the validity of requests. When the conditions of such statements are not met, an exception is raised and all modifications made by a given request to the current state are reverted [119]. For the purpose of checking the (external) inputs or the validity of the state to receive such inputs, *Solidity* provides the `require` routine – this transaction-reverting statement makes the contract robust against improper usage. Typical examples of `require` statements are: (i) checking that certain requests can only be done by the owner of the contract —*i.e.,* `require(msg.sender==owner)`— or (ii) that a transaction amount is positive —*i.e.,* `require(amount>0)`.

A recent study by Liu *et al.* [73] shows that transaction-reverting statements are extensively used within *Solidity* smart contracts for authority and validity checks. They found that removing or modifying these statements may compromise the security of the smart contract. Additionally, the study showed that existing *Solidity* testing tools cannot effectively detect security vulnerabilities caused by these statements. Internally, the `require` statements are just normal function calls that are handled in a special way by the interpreter. Existing search-based approaches [112, 118], however, treat these statements as any other function call without taking this critical construct of *Solidity* into account. In particular, there is no gradient in the fitness landscape that the search algorithm could use as guidance to satisfy the condition of these statements, the search algorithm has to resort back to random testing.

In this paper, we argue that these transaction-reverting statements should be treated as first class citizens during testing, since any error in them likely corresponds to a security vulnerability. To this aim, we propose a new approach to improve the search guidance for transaction-reverting statements, without changing the semantics of the contract under test. First, we statically analyze the contract under test and identify the transaction-reverting statements and modifiers. *Modifiers* are interprocedural constructs that group transaction-reverting statements that are executed by the Ethereum Virtual Machine (EVM) as a dependency for certain methods. Then, we perform interprocedural dependency analysis to link the *Control Flow Graph* (CFG) of the method under test with the associated modifiers. Lastly, we calculate an interprocedural-level fitness value (*i.e.,* to guide the search process) based on the runtime data collected by a context-sensitive instrumentation of the transaction-reverting statements. The new fitness function allows to measure how far a test case is from satisfying the condition within these statements.

To evaluate the effectiveness of our approach, we implemented it within *Syntest-Solidity* [80] — a test case generation framework for *Solidity* that implements *DynaMOSA*. *DynaMOSA* is the state-of-the-art algorithm for unit test case generation originally designed for Java programs [58, 120] and recently applied to *Solidity* [118]. It is guided by state-of-the-art unit-level fitness functions that consider structural coverage [121]. We performed an empirical study on 100 real-world smart contracts gathered from *etherscan.io.* We compare the results achieved by *DynaMOSA* with and without the improved guidance with regard to vulnerability detection capability and structural coverage.

Our results show that *DynaMOSA* covers significantly more transaction-reverting statements with the improved guidance in 37 % of the smart contracts, with an average increase in transaction-reverting statement coverage of 12 %. This also leads to an average increase in line coverage by 8 %. Finally, our approach helps *DynaMOSA* detect more vulnerabilities, with 17 % (on average) more captured vulnerabilities for those contracts on which we observe an increase in transaction-reverting statement coverage.

In summary, this work makes the following contributions:

1. A lightweight approach based on interprocedural analysis to improve the search guidance for transaction-reverting statements (Section 3.3).
2. An implementation of our approach[1] within a state-of-the-art *Solidity* smart contract testing tool, named *Syntest-Solidity* [80].
3. A *Solidity* smart contract benchmark consisting of a diverse set of 100 real-world smart contracts (Section 3.4.2).
4. An empirical study demonstrating the benefit of the proposed approach (Section 3.5).
5. A full replication package including the code, results, and the scripts to analyze the results [85].

While in this paper we focus on *Solidity* smart contracts, this approach can be beneficial for any programming language with explicit contracts or declarative input validation rules.

3.2 Background and Related Work

This section provides an overview of basic concepts and related work on smart contracts, fuzzing, and test case generation.

3.2.1 Smart Contracts and Ethereum

In the last decades, there has been an increased focus on creating decentralized services to cut out intermediaries from the interaction between people. One example of this trend is smart contracts —digital agreements between multiple parties on how certain tasks need to be executed— and in particular *Ethereum*, the most popular smart contract platform [110]. The main benefits that smart contracts can provide are trustless interactions, automated task handling, and hosting of decentralized applications (dApps). Smart contracts are built

[1]https://github.com/syntest-framework/syntest-solidity

on top of a blockchain, a tamper-proof ordered ledger. When a smart contract gets deployed, it creates a transaction containing code (a collection of functions) and data (state) that resides at a specific address on this ledger. Users can make requests to this address, using the functions to modify the state of the contract. Each state modification creates a new transaction on the blockchain. This chain of blocks will grow over time with the addition of new contracts and requests. Since the logic that can be applied to the state is fixed and the state is publicly available, users in the network can verify if a transaction was properly executed.

Ethereum runs on a decentralized network of nodes. These nodes process the requests made to the contracts and create the blocks needed to modify state and deploy new contracts. To secure the platform against attacks, there should be consensus between the nodes. To get consensus within the network, *Ethereum* makes use of the mechanism called *Proof of Work* (PoW). PoW relies on a computationally-expensive mathematical problem that is difficult to calculate, but easy to verify. The random node that solves the problem first gets to decide which transactions are accepted.

3.2.2 Transaction-Reverting Statements

Since smart contracts cannot be modified once deployed, it is crucial that they are thoroughly tested to detect and remove potential vulnerabilities. In addition, transaction-reverting statements are used by developers to further assess the validity of requests and verify that the contract remains in a valid state. Hence, it is critical that these statements are correctly added to assess the important properties of the contract under analysis.

To better show how these reverting statements work, let us consider the simplified example of a *Solidity* smart contract shown in Example 3.1 that represents a bank account. On lines 6-8, the owner of the account is set to the creator of the contract. Lines 10-13 define a `modifier`, consisting of a transaction-reverting statement that is executed by the Ethereum Virtual Machine (EVM) as a dependency for the `withdraw` method. Lastly, the method on lines 15-22 allows users to withdraw money from the account. The `withdraw` method makes use of the `isOwner` modifier to guarantee that only the owner of the account can withdraw money. In addition, the method uses a local reverting statement (line 16) to check if the amount to withdraw is positive. When the `require` check on line 16 fails, state-of-the-art coverage heuristics (like used by existing search-based approaches [112]) would assume that line 17 and 21 are also covered. In reality, however, only line 16 is covered and the execution is halted.

3.2.3 Testing Solidity Smart Contracts

Various techniques have been used in literature to test *Solidity* smart contracts. An overview of the different techniques is available in the recent survey by Ren *et al.* [89].

Static Analysis [122–124]: Static analysis tools analyze a contract for vulnerabilities without running it. This can be done at both a source code and a byte-code level. The benefit of analyzing a contract statically is that the entire contract can be scanned at once. However, static analysis tools often have a high false-positive rate requiring manual verification [89].

Symbolic Execution [125, 126]: Symbolic execution tools also statically analyze a contract. What differentiates symbolic execution tools is that they keep track of all constraints they encounter on every path through the code. This allows these tools to perform constraint solving to determine which range of input values will lead to certain branches. Symbolic Execution, however, unavoidably suffers from problems like path explosion [5].

Formal Verification [127]: Formal verification methods transpose the source code of the contract to a mathematical proof language. Within this proof language, this method mathematically checks the source code against a manually constructed model of the code's behaviour. This method provides the most security, however, it requires developers to construct a complex model in a different language than the contract.

Fuzzing [112–117]: Fuzzing automatically generates test data. This data is fed to the contract under test to see how the contract responds to it. This technique is very effective at finding inputs that make the contract crash. However, it cannot be used for verifying the behavior of the contract. Besides, it only focuses on test data without generating complete test cases (*e.g.,* without assertions).

Test Case Generation [118]: Test case generation generates test data, method sequences, and assertions. One study used this technique. However, this study [118] uses existing algorithms without adapting them to Solidity.

3.2.4 Search-based Testing and Fuzzing

Search-based software testing (SBST) is a well studied research area that focuses on automating the generation of test data and test cases. Automatic test case generation significantly reduces the time needed for testing applications [5] and has been successfully used in industry [11, 128]. Various studies have been performed that use meta-heuristics to test programs at different levels *e.g.,* unit [34], integration [36], and system-level [37]. These studies have shown that these techniques are effective at achieving high coverage [74] and detecting faults [8, 129, 130].

One of the most commonly used classes of meta-heuristics is Evolutionary Algorithms (EAs) [19, 74, 101]. EAs are inspired by the process of natural selection. They evolve an *initial population* of randomly generated individuals (test data or test cases). These individuals are then *evaluated* based on a predefined fitness function. After the evaluation, the individuals with the best fitness values are selected for *reproduction.* Reproduction creates new offspring by applying mutation (small delta changes to an individual) and crossover (exchanging information between two individuals). Lastly, the new population is created by *selecting* the best individuals across the parents (current population) and the offspring (newly created test data or test cases). These three steps *evaluation*, *reproduction*, and *selection* happen in a loop until a stopping condition has been met. After the search process ends, an archive is created with the best individuals from the population [58, 120].

EAs are often used in fuzzing for generating input data. For example, Nguyen *et al.* [112] used an efficient genetic algorithm for fuzzing *Solidity* smart contracts. The main difference between fuzzing and test case generation is that the former focuses on generating test inputs while the latter aims to generate full test cases, including input data, method sequence, and assertions.

3.2.5 Unit-level Fitness function

The purpose of a fitness function is to measure and indicate how far off the individual (test) is from satisfying a test objective, *e.g.,* branches. In SBST, the *de facto* fitness function is made up of two *heuristics*: *approach-level* and *branch distance* [4, 34, 58, 112]. The *approach-level* relies heavily on the Control Flow Graph (CFG). A CFG represents the flow of the logic within a function of a program — all paths that might be traversed during the execution of the program. CFGs are created from the Abstract Syntax Tree (AST) provided by the parser of the language, in our case the *Solidity* compiler. A node in the CFG is called a *basic block* and corresponds to a sequence of statements that are always executed altogether [131], *i.e.,* with no branches inside the block. The *approach-level* uses the CFG and the data that is gathered from the instrumentation during runtime to measure how far, in terms of graph distance, the execution flow is removed from the targeted branch point. More precisely, the instrumentation data is used to determine which branches of the CFG have been covered by the test case. Afterwards, the fitness function calculates the shortest difference along the CFG between the targeted branch node and the closets covered node. Once the execution path reaches the targeted branch node, the fitness function uses the *branch distance* to calculate how far the input variable is from satisfying the condition of the target *true* or *false* branch.

3.2.6 Testability Transformations

The *flag* problem is a common issue in SBST [44, 45] that manifests when the conditions in the if-statements are not explicit (*e.g.,* an inline method call like `if (isNull(y))`) or it reads boolean variables (*e.g.,* `if(y==true)`). To address this problem, researchers have proposed *testability transformations* [45], which transform the program under test into an equivalent one (*i.e.,* by preserving the semantics) where the conditions are replaced with predicates reading non-boolean variables. Prior studies have shown that testability transformations dramatically improve code coverage without the need for adapting the underline search algorithms [44, 46, 47].

Compared to these prior studies, we do not apply testability transformation for two reasons. First, creating testability transformations that fully preserve the semantics of program is challenging, limiting its practical applicability [45]. Second, state-reverting conditions are internal subroutines executed by the EVM at run-time and not part of the branch conditions of the source program under test and, therefore, they cannot be transformed.

```solidity
pragma solidity ^0.5.0;
contract Account {
  address public owner;
  mapping (address => uint) private balances;

  constructor() public {
        owner = msg.sender;
  }

  modifier isOwner() {
    require(msg.sender == owner, "You are not the owner");
    _;
  }

  function withdraw(int amount) public isOwner {
    require(amount > 0, "Amount too low");
    if (amount <= balances[msg.sender]) {
      balances[msg.sender] -= amount;
      msg.sender.transfer(amount);
    }
    return balances[msg.sender];
  }
  ...
}
```

Example 3.1: Example *Solidity* smart contract

3.3 Approach

This section outlines our approach to improve the search guidance (*i.e.,* restoring the gradient) for transaction-reverting statements using the contract shown in Example 3.1 as a running example.

3.3.1 Problem Definition

A primary challenge in SBST is defining an effective fitness function that guides the search algorithm toward covering an uncovered branch. As an example of test objectives, let us consider the false branch of the `if` condition in line 17 for the method `withdraw` in Example 3.1 and its CFG depicted in Figure 3.2a. If we apply the state-of-the-art unit-level fitness function, we obtain the *fitness landscape* depicted in Figure 3.1. This fitness landscape shows the fitness values for the false branch with varying inputs for the `amount` parameter. The inputs where the fitness function is zero lead to covering the target branch. Ideally, the fitness function should have a gradient to effectively guide the search algorithms. However, in Figure 3.1, we can observe that the landscape is *flat* for all negative values of `amount`. This is due to the program execution ending when the condition within the `require` in line 16 of Example 3.1 is not met, without providing any information on how close the execution is to satisfying that condition.

The problem of the flat landscape does not apply only to our example but it generalizes to all contracts that have transaction-reverting statements. As shown by Liu *et al.* [73], these statements are extensively used in smart contracts for authority and validity checks. Therefore, explicitly considering these constructs when computing the fitness function is critical to restore the gradient and make the search more effective. Otherwise, the search algorithm has to resort to random testing when encountering such transaction-reverting statements. This approach is not ideal as random testing (*i.e.,* without guidance) is slow and might not lead to a solution within the allocated search budget. In practice, this means the search algorithm either randomly guesses the input values needed to satisfy the condition or gets stuck.

Additionally, in Example 3.1, we can see that the `withdraw` method defines a dependency on the `isOwner` modifier. In this example contract, the `require` statement within the `isOwner` modifier (line 11) has to be satisfied before the main branch of the `withdraw` function can be executed. As a consequence, the search algorithm has to overcome two independent obstacles without guidance through random testing before it can reach the branch in line 17.

3.3.2 Overview

The goal of our approach is to restore the gradient for *Solidity* smart contracts containing transaction-reverting statements, by providing a quantitative measurement on how far a test case is from satisfying these statements. To this aim, we first statically analyze the Abstract Syntax Tree (AST) of the contract under test and identify the transaction-reverting statements and modifiers (**Step 1**). Then, we perform interprocedural control dependency analysis to determine the control flow across the different methods and subroutines (**Step 2**). Lastly, we define a new interprocedural fitness function based on the runtime data collected by the context-sensitive instrumentation of transaction-reverting statements (**Step 3**). The last two steps will be further explained in the next subsections.

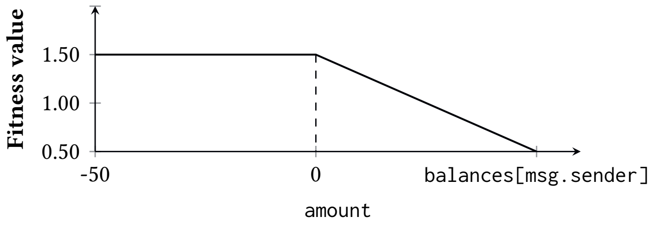
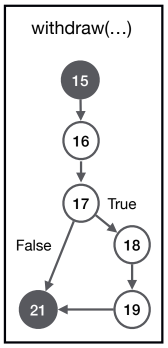
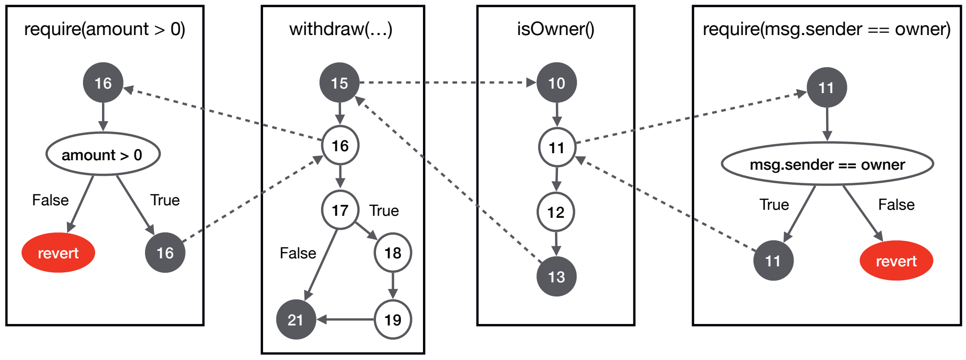


Figure 3.1: Fitness landscape for the false branch of the `if` in line 17 of the method `withdraw` in Example 3.1



(a) Traditional CFG



(b) CFG with interprocedural dependency analysis

Figure 3.2: Control Flow Graphs (CFGs) of the *withdraw* function in Example 3.1

3.3.3 Interprocedural Dependency Analysis

The idea behind the interprocedural dependency analysis is to determine how the transaction-reverting statements and modifiers impact the execution of the method under test at run-time. To explain how this analysis works, we will use the example in Figure 3.2. Figure 3.2a depicts the traditional CFG for the `withdraw` method in Example 3.1 while Figure 3.2b shows the results of enriching it with our interprocedural dependency analysis. In these two figures, the gray nodes represent the flow entry and exit blocks of the CFG. The numbers within the nodes indicate the line number of the statement that the block represents. Lastly, the solid edges indicate how the execution flows through the nodes.

Linking Transaction-Reverting Statements

Transaction-reverting statements are special sub-routines within the EVM and, therefore, they do not have a corresponding CFG nor branching nodes. We apply context-sensitive instrumentation around the transaction-reverting statements to capture their impact on the dependent methods. The instrumentation allows to capture these interprocedural dependencies and build an artificial control flow representation of the sub-routines. In the example of Figure 3.2b, we build the control flow of the statement in line 16 (*i.e.*, the box with the header `require(amount > 0)`, which is linked (dashed edges) to the CFG of the `withdraw` method. The red nodes in the sub-routine represent the *Solidity* revert mechanism.

The context-sensitive instrumentation injects two additional instrumentation statements, namely *pre-trs* and *post-trs* (where *trs* stands for transaction-reverting statements). The *pre-trs* and *post-trs* are injected before and after each of these statements, respectively. The *pre-trs* indicates if the execution of the contract reached the statement, meaning the search process is at the revert point. If the *post-trs* is reached, it indicates that the condition of the transaction-reverting statement has been met. If the *pre-trs* has been reached and the *post-trs* has not, the condition has not been met and the execution is halted and reverted.

However, the *pre-* and *post-trs* do not provide information on how to satisfy the particular condition but only if the condition has been met or not. To collect information on how far a test case is from satisfying the conditions, we add additional instrumentation statements (the context) to record the type of operator and the values of the operands from the memory stack at runtime. For example, for the statement `require(amount > 0)`, our instrumentation records the operator > and the runtime value of the `amount` operand and the constant value `0`. This data can be integrated into the fitness function as discussed in Section 3.3.4 to restore its gradient.

Linking Modifiers

In step 1 of the approach, we analyze the Abstract Syntax Tree (AST) of the contract to compile a list of all modifiers that each method is dependent on. As an example, the method `withdraw` in Figure 3.2b depends on a single modifier, called `isOwner`. Note that a modifier cannot be directly invoked but can be tested only through the methods that define it as a dependency. In general, a modifier acts like a template (or around advice in terms of aspect-oriented programming), wrapping its logic around the method that depends on it. Modifiers use a special identifier (`_;`), as can be seen on line 12 of Example 3.1, to indicate where the function's logic should be executed. In the example, all statements within the method `withdraw` are post-dominated by the conditions of the `isOwner` modifier. Hence, the statements in `withdraw` are not covered by simply invoking the function if the conditions of `isOwner` are not met.

To capture the interprocedural dependencies we build the control flow graph of the modifier and link it to the entry or exit point within the method depending on where the template identifier is located. If a method depends on multiple modifiers, the CFG of each modifier is linked to the dependent method Z in the order they appear in signature of Z in a layered approach.

As an example, consider Figure 3.3, which defines two modifiers, named `X` and `Y`, together with their extracted parts (`A`, `B`) and (`C`, `D`), respectively. Method `Z` uses both mod-

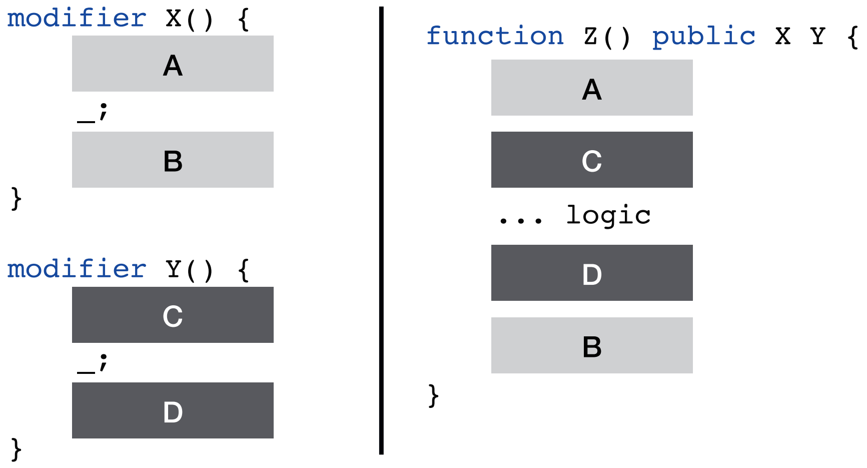


Figure 3.3: Modifier structure and execution order

ifiers in the order they are listed: `X`, `Y`. The overall dependency graph links part `A`, part `C` to the entry point of the body of the method `Z` while its exit point is linked to part `D`, and lastly part `B`.

If the modifier contains transaction-reverting statements, we apply the same procedure described in the previous subsection to the CFG of the modifier. An example of such a case is depicted in Figure 3.2b where the `isOwner` modifier contains a `require` statement with the condition `msg.sender == owner`. This condition checks that the request is only made by the owner of the contract. Finally, if a modifier is declared by multiple methods of a contract, the linking procedure (from the modifier CFG to the method CFG) is applied separately for each of these as the context differs among the different methods.

3.3.4 Interprocedural Fitness Function

For each branch in the code, we do not simply apply the unit-level fitness function discussed in Section 3.2.5 but enrich it with context data collected by the interprocedural dependency analysis.

We define the *interprocedural approach level* as an extension to its unit-level variant. Let t be a test case and b_i be a branch to cover. The interprocedural approach level $IAL(b_i, t)$ is the number of interprocedural control dependencies between the closest executed branch and b_i. The interprocedural control dependencies includes the classic unit-level control nodes (in the CFG) and the interprocedural dependencies related to modifiers and transaction reverting statements. For example in Figure 3.2b, the branch 17⟶21 of the `withdraw` method is control dependent on nodes 15-16 (unit-level dependencies) but also on nodes 10-13 of the *isOwner* modifier and the conditions of the two require statements (nodes 11 and 16).

When the execution of a test t is halted because of a transaction-reverting statement TRS_i, we introduce the *trs-distance*. This distance measures how far t is from satisfying the condition in TRS_i by using Korel's rules [121] for conditions. For example, the *trs-distance* for the statement `require(x == 0)` is computed as $|x - 0|$ [121], which is equal to zero only when the condition $x == 0$ is satisfied.

Therefore, the interprocedural fitness function (f) for a test t *w.r.t.* an uncovered branch b_i is computed as follows:

$$f = \begin{cases} IAL(b_i, t) + \dfrac{d(TRS_i, t)}{d(TRS_i, t) + 1} & \text{if halted at } TRS_i \\ IAL(b_i, t) + \dfrac{d(b_i, t)}{d(b_i, t) + 1} + 1) & otherwise \end{cases} \tag{3.1}$$

where IAL denotes the interprocedural approach level, $d(TRS_i, t)$ is the *trs-distance* for the transaction-reverting statement TRS_i and $d(b_i, t)$ is the traditional branch distance.

3.4 Empirical Study

We carried out an empirical study to assess the effectiveness of the proposed interprocedural fitness function compared to its state-of-the-art unit-level variant. To this aim, we use these functions to guide the state-of-the-art testing algorithm, *DynaMOSA*. We evaluate the impact of the proposed fitness function *w.r.t.* to the following testing criteria: (i) structural (branch, transaction-reverting statement, and line) coverage and (ii) *vulnerability detection capability.*

3.4.1 Research Questions

Our empirical evaluation aims to answer the following two research questions:

RQ1 *To what extent does the proposed approach improve the structural coverage achieved by DynaMOSA?*

RQ2 *To what extent does the proposed approach improve the vulnerability detection of DynaMOSA?*

These two research questions aim to evaluate if the proposed approach improves the effectiveness of the state-of-the-art test case generation algorithm *DynaMOSA*. RQ2 reflects the main goal, which is to determine if the proposed approach allows the two algorithms to detect more vulnerabilities in the *Solidity* smart contract under test. We additionally report the structural coverage as test data and test cases cannot detect or capture vulnerabilities in code regions that are uncovered.

3.4.2 Benchmark

To evaluate the proposed approach, we created a benchmark consisting of 100 *Solidity* smart contracts. We collected all contracts submitted between January and April of 2021 with *Solidity* versions 5 and 6 from *etherscan.io.* We then selected smart contracts with a cyclomatic complexity of $cc >= 2$, *i.e.,* contracts with at least one conditional statement, *i.e., branch, loop.*

A recent study by Ren *et al.* [89] empirically and theoretically criticizes the benchmarks used in prior studies, even those that include the entirety of *etherscan.io.* Moreover,

Table 3.1: Statistics (min, max, median and quartiles) of the 100 smart contracts in our benchmark

Code Elements	Min.	Q_1	Median	Q_3	Max
# Functions	2	10	21.5	39	111
# Branches	0	6	12	22	62
# Reverting statements	0	12	27	40.5	102
# Lines	6	53	109.5	154	431

previous studies did not explicitly report the source of the contracts [118] or did not check the cyclomatic complexity [112] as suggested in the literature [132, 133]. This study proposes a benchmark that is more transparent by removing trivial smart contracts ($cc < 2$) and specifying the date and time on which the contracts were submitted to *etherscan.io*.

We ensured that the benchmark contains (i) contracts from different application domains (*e.g.,* wallets, auctions, tokens, financial staking, DAO, voting, insurances); (ii) contracts with and without transaction-reverting statements (70 % use modifiers, 18 % use a single require statement, 62 % use multiple require statements, 5 % use no reverting statements) to validate that the proposed approach does not negatively impact contracts without these constructs; (iii) contracts with a diverse size and complexity. Table 3.1 reports the statistics of the 100 *Solidity* smart contracts in our benchmark. In particular, the table reports the minimum, maximum, median, and quartiles (Q_i) of the functions, branches, lines, and transaction-reverting statements in the contracts. The benchmark is available within the replication package.

3.4.3 Benchmark Tool & Baseline

To answer the research questions, we implemented our approach within *Syntest-Solidity* [80]. We have used this tool because it generates complete test cases with assertions, which are necessary for capturing vulnerabilities automatically. Instead, other *Solidity* testing tools were either solely built to work as a fuzzer [112] or were not sufficiently extensible to integrate the proposed approach [118]. We briefly describe the state-of-the-art unit-level test case generation algorithm used in *Syntest-Solidity*.

DynaMOSA

Dynamic Many-Objective Sorting Algorithm (DynaMOSA) is the state-of-the-art evolutionary search algorithm for test case generation [58]. It models test case generation as a many-objective problem by targeting each test target (*e.g.,* branch, line) simultaneously using a many-objective genetic algorithm. As any evolutionary algorithm, *DynaMOSA* evolves a set of randomly generated test cases (see Section 3.2.4). The fitness of each test case (or individual) is determined based on the approach level, and the branch distance for the remaining uncovered targets. *DynaMOSA* makes use of a dynamic selection of the targets, where test targets are dynamically added based on the control dependency hierarchy when the current target is covered. This dynamic selection improves the efficiency of the search process for smaller search budgets [58]. After evaluating and creating new test cases (offspring), environmental selection is used to select the fittest individuals in the population to survive using the *preference criterion*, *non-dominated sorting*, and *crowd-*

ing distance. The *preference criterion* first selects the best test case (the one with the best fitness) for each just-missed branch (front zero). Then, the *non-dominated sorting* selects the remaining test cases based on the concept of Pareto optimality, which is the standard criterion in SBST. Finally, *crowding distance* is in place to promote the diversity among the test cases that are equally good according to the Pareto optimality.

3.4.4 Parameter Setting

Previous studies empirically showed [109] that although parameter tuning has an impact on the effectiveness of a search algorithm, the default values, which are commonly used in literature, provide reasonable and acceptable results. For this study, we have chosen to use the following default parameter settings recommended in the literature [34, 58, 109, 134–136].

Population size

We use a population size of 10 individuals (test cases); We performed a preliminary experiment to determine the size of the population. A population that is too small will not allow for enough exploration and will quickly converge. A population size that is too big will consume more of the search budget per iteration of the search process. Since *Solidity* smart contract tests are performed through an API (in comparison to testing frameworks at unit-level), running tests is drastically slower. In addition, before each test case can be run, the contract has to be deployed to the smart contract network. Therefore, we established that a population of 10 individuals provides sweet spot in the trade off between efficiency and coverage. Our choice of using a relatively small population size is also in line with the recommended population for expensive fitness functions [134, 137].

Mutation Operator

We use the *uniform mutation*, which changes each test case by adding, deleting, or replacing method calls. We use a mutation probability p_m=1/n, where n is the number of statements in the test case as recommended in the literature [34, 58, 109]. For primitive statements (*e.g., int*), the values are mutated using the *polynomial mutation* [135] that is applied with a probability of 80%. For the remaining 20%, the operator applies random sampling.

Crossover Operator

We use the single-point tree crossover with a crossover probability of p_c=0.8, which is within the recommended range $0.50 \le p_c \le 0.90$ [136, 138].

Selection

We use the binary tournament selection to sample individuals from the population for reproduction [139].

Search Budget

As a stopping criterion for the search process, we use a search budget based on time instead of the number of executed tests. This was done as a time-based stopping criterion provides the fairest comparison of the different approaches, given that the proposed heuristics add a small computational overhead to the search process. Additionally, practitioners will often only allocate a specific amount of time for the algorithm to run as the time it takes to run a certain number of iteration differs across contracts and across tests for the same contract.

The search budget for the algorithm was set to 30 minutes as this provides a balance between giving the algorithm enough time to explore the search space (considering the slower execution time of a single test case) and making the study infeasible to execute. The algorithm will end prematurely if all its test objectives have been covered. Note that time-based search budgets are considered a less biased stopping criterion than a budget based on the number of executed tests (or fitness evaluation) as not all tests have the same running time [7, 34, 130, 140].

3.4.5 Vulnerability Detection

To evaluate how the proposed approach influences the effectiveness of *DynaMOSA* at detecting/capturing vulnerabilities, we considered multiple vulnerable versions of the contracts in our benchmark. We synthesize vulnerable versions that differ from the secure ones by either (i) missing transaction-reverting statements or (ii) transaction-reverting statements with incorrect conditions. As an example, for the contract in Example 3.1, one vulnerable version could be obtained by removing the `require` function in line 11. In that case, anyone can withdraw the money from the bank account, not only the owner. Another example of a vulnerable contract version would be if we inverted the condition of the `require` function in line 16. This would allow an attacker to increase the balance of an account by withdrawing a negative amount. Studies have shown that the transaction-reverting statements play a crucial role in the behavior of the contract when testing for faults that cause vulnerabilities [73, 141, 142]. Therefore, we analyze the ability to detect the vulnerability associated with these missing or incorrect statements.

For each contract (with transaction-reverting statements) in the benchmark, we generated 10 vulnerable versions. To assess the vulnerability detection capability, we run the test cases that were generated for the non-vulnerable version of the contract on these vulnerable versions to determine if the test cases fail, and thereby, capturing the vulnerability. Finally, We assess the performance of the testing algorithm with and without our approach measuring the number of vulnerabilities detected by the generated test cases.

3.4.6 Experimental Protocol

For each contract in the benchmark, we run *DynaMOSA* with and without the improved guidance. The resulting coverage information for the different evaluation metrics (*i.e.,* branch, reverting statements, line) is collected and stored along with the generated test cases.

Since *DynaMOSA* is a randomized algorithm, we can expect a fair amount of variation in the results of the empirical study. To prevent potential biases in the results, we repeated every experiment 20 times, with a different random seed, and computed the average (median) results. In total, we performed 4000 executions: two configurations of *DynaMOSA* on 100 *Solidity* smart contracts with 20 repetitions each. With each execution taking 30 minutes, the total execution time is 83.5 days of consecutive running time. We ran the experiment on a system with two AMD EPYC™ 7452 using 120 cores running at 2.35 GHz.

To answer *RQ1*, we compare the structural coverage results of the two configuration with each other. To evaluate the vulnerability detection capability of the different approaches (*RQ2*), we compare the same configurations as for *RQ1* but now using the procedure described in Section 3.4.5.

We use the unpaired Wilcoxon rank-sum test [67] with a threshold of 0.05 to determine if the results of the proposed approach are statistically significant. The Wilcoxon rank-sum is a non-parametric statistical test that determines if two data distributions are significantly different. This is the standard test for evaluating randomized algorithms such as *DynaMOSA* [65]. In addition, we use the Vargha-Delaney statistic [68] to measure the effect size of the result, which indicates how large the difference between the two configurations is.

3.5 Results

This section discusses the results of our empirical study with the aim of answering the research questions formulated in Section 3.4.1.

3.5.1 Result for RQ1: Structural coverage

Table 3.2 shows the statistical results for the structural coverage achieved by *DynaMOSA* with the proposed approach, compared to *DynaMOSA* without it, on the *Solidity* smart contracts in the benchmark. *#Win* indicates the number of contracts for which the search algorithms with the improved guidance have a statistically significant improvement (p-value $\leq$ 0.05) over the algorithms without this guidance. *#Lose* indicates the number of contracts for which the proposed approach did not provide a statistically improvement (p-value $>$ 0.05), and lastly, *#No diff.* indicates the number of contracts for which there is no statistical difference in the results between the search algorithms with and without the improved guidance. In addition, the *#Win* and *#Lose* columns also include the magnitude of the difference through the $\hat{A}_{12}$ effect size, classified in *Negligible* (N), *Small* (S), *Medium* (M), and *Large* (L).

From Table 3.2, we can see that the proposed approach only provides a statistically significant improvement for branch coverage in very few cases (4). This result is as expected as without the additional information that the guidance provides, the search process falsely assumes that the branches containing the transaction-reverting statements are fully covered. Consequently, with the improved guidance, we can observe a statistically significant improvement in 37 and 35 contracts for transaction-reverting statement and line coverage, respectively. This indicates that without this guidance *DynaMOSA* cannot reach the code regions after these statements. For the transaction-reverting statement coverage, *DynaMOSA* improves with a large magnitude for 35 contracts and medium for 2 contracts. For line coverage, *DynaMOSA* improves with a large magnitude for 29 contracts, medium for 4 contracts, and small for 2 contracts.

Figures 3.4 and 3.5 show the absolute difference in the average (mean) line and transaction-reverting statement coverage achieved by *DynaMOSA* with the improved guidance, compared to *DynaMOSA* without this guidance, for the significant cases. The proposed approach on average improves the line coverage by +8.66 %, with a maximum improvement of +27.97 % for `GreenMarkTrust` (id = C31), and the transaction-reverting statement coverage by +12.29 %, with a maximum improvement of +31.07 % for `MARVELCOIN` (id = C25).

Table 3.2: Statistical results for *DynaMOSA* with and without the improved guidance. We report the number of times the proposed approach statistically improve (#Win) or decrease (#Lose) the effectiveness of *DynaMOSA*. Negligible (N), Small (S), Medium (M), and Large (L) denote the $\hat{A}_{12}$ effect size.

Metric	#Win				#Lose				#No diff.
	N	S	M	L	N	S	M	L	
Branch	-	-	2	2	-	-	-	-	96
Rev. statement	-	-	2	35	-	-	-	-	63
Line	-	2	4	29	-	-	-	-	65

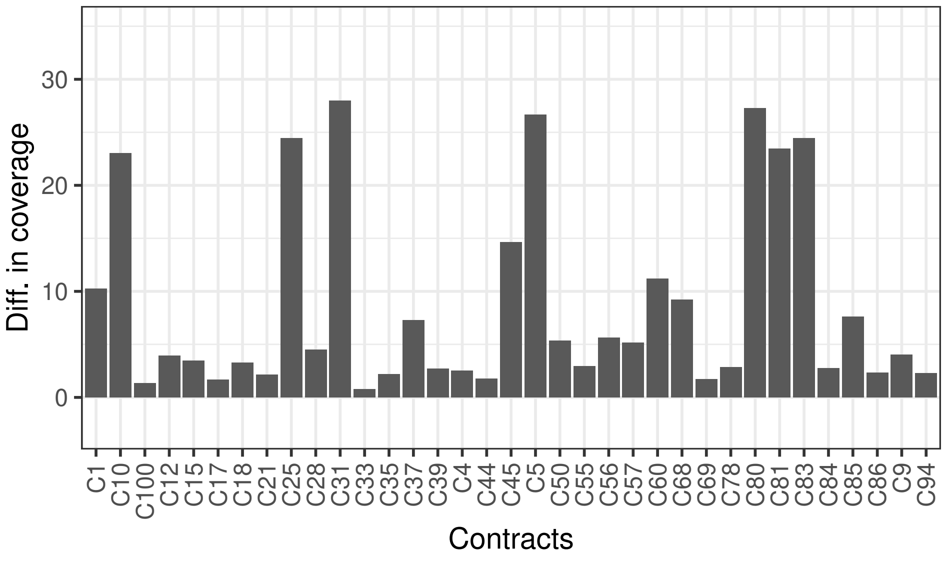


Figure 3.4: Absolute difference in line coverage for *DynaMOSA* with and without the improved guidance

3.5.2 Result for RQ2: Vulnerability Detection

Figure 3.6 shows the percentage of vulnerabilities that were detected by *DynaMOSA* when comparing the unit-level fitness function to the proposed interprocedural one. As we can observe, there is no or small differences in the minimum and first quartile in the box-plots. That means that for 25 % of the contract there is no difference in the vulnerability detection capability. This is also in line with the results we observe in RQ1, considering that covering the line and transaction-reverting statement is a prerequisite to reach the vulnerability. However, we observe larger differences in the second and third quartiles, as well as in the maximum value.

In particular, we observe that the percentage of captured vulnerabilities achieved by *DynaMOSA* increases by 2 % in the 2nd quartile and 8 % in the 3rd quartile, as depicted in Figure 3.6. For the contracts with a difference in the number of captured vulnerabilities, our approach improves on average by 17 %. The largest improvement is obtained for

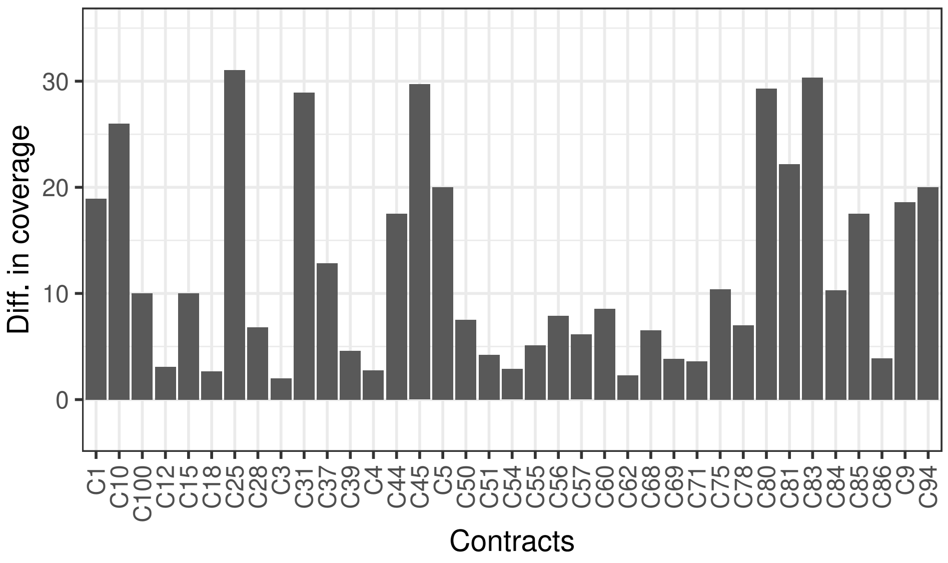


Figure 3.5: Absolute difference in reverting statement coverage for *DynaMOSA* with and without the improved guidance

`HTDD_contract` with an increase in the number of vulnerabilities captured of 38 %. We also report a *moderate* positive Pearson's r correlation between the increases in the vulnerability detection capability and the increases in line coverage (r=0.48, p-value=<0.01) and transaction-reverting statement coverage (r=0.40, p-value=<0.01) achieved when using the improved guidance with *DynaMOSA*. We applied the Pearson's r correlation coefficient since the difference in these metrics are normally distributed.

To provide a practical example, let us consider the vulnerability reported in line 7 of Example 3.2 for the contract `INS`. This vulnerability is caused by changing the condition (from <= to >) in the second `require` statement. The vulnerability is captured by *DynaMOSA* when using the improved guidance but remains undetected when our approach is not applied. The test case that captures the vulnerability is reported in Example 3.3. This test case covers both require statements in the function (line 3 and 7) and asserts the reverting operation of the EVM in line 7, *i.e.,* if the transaction-reverting statement is not satisfied, all performed transactions are reverted. The test correctly captures the transaction-reverting statement and fails (via the expected `to.be.rejectedWith(Error)` code) when such a condition is modified. Instead, *DynaMOSA* without the improved guidance could not even reach the `require` in line 7 as it did not manage to satistfy the condition of the `require` statement in line 3.

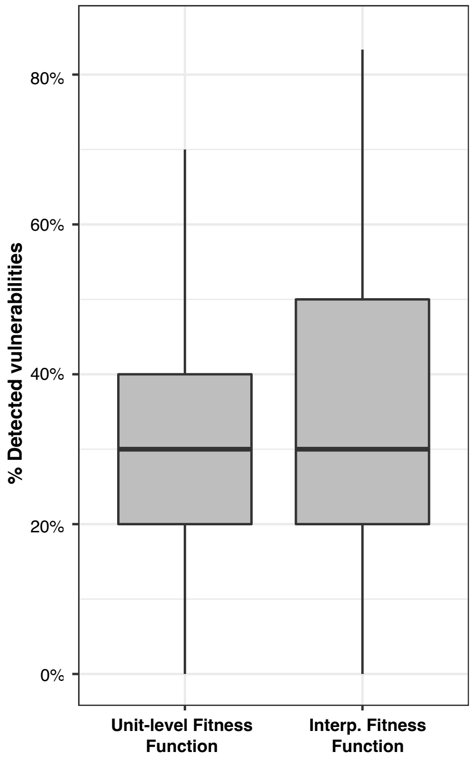


Figure 3.6: Vulnerability detection results for *DynaMOSA*

```
function burnFrom(address _from, uint256 _value) public ... {
  // Check if the targeted balance is enough
  require(balanceOf[_from] >= _value);

  // Check allowance
  //require(_value<=allowance[_from][msg.sender]); <- SECURE
  require(_value>allowance[_from][msg.sender]);  // <- VULNER.

  // Subtract from the targeted balance
  balanceOf[_from] -= _value;

  // Subtract from the sender's allowance
  allowance[_from][msg.sender] -= _value;
  totalSupply -= _value;

  // Update totalSupply
  emit Burn(_from, _value);
  return true;
}
```

Example 3.2: Vulnerable variant for the contract `INS.sol`

```
it('test for INS', async () => {
  const INS0 = await INS.new(BigInt("139"), "fKQs..",
          "lihM...", {from: accounts[2]});

  const bool0 = await INS0.burn.call(BigInt("1361"),
          {from: accounts[2]});

  assert.equal(bool0, true)

  await expect(
          INS0.burnFrom.call(accounts[1], BigInt("1212"),
          {from: accounts[2]})
        ).to.be.rejectedWith(Error);
});
```

Example 3.3: Generated test case that detects the vulnerability (Example 3.2) for the contract `INS.sol`

3.6 Discussion

Our experiment empirically shows that applying state-of-the-art test case generation approaches cannot effectively detect vulnerabilities (or produce structural coverage) without treating all constructs of the language to be tested as first class citizins. The success of search-based software testing is, in practice, dependent on many components, including the ability of the search algorithm to get insight on all aspects of the program execution through the fitness function. Our empirical study shows the importance of modelling these language-level constructs in the fitness function.

The benefits of this approach are not only applicable for test case generation, but also to fuzzing approaches and have the potential to improve the testing landscape for *Solidity* smart contracts. Based on a preliminary study, our approach can improve line coverage for *sFuzz* [112], a state-of-the-art fuzzer, by on average +8.42 %, with a maximum improvement of +31.76 %, and the transaction-reverting statement coverage by +13.08 %, with a maximum improvement of +33.09

Additionally, this approach does not only apply to *Solidity* but can be generalized to any programming language with explicit contracts or declarative input validation rules. For example, Java makes use of annotations (*e.g.,* @NotNull) that help control contracts throughout method hierarchies. In general, interprocedural analysis can benefit testing programs that use design by contract constructs.

This paper focusses on *Solidity* as contracts can not be updated once they are deployed, increasing the importance of detecting vulnerabilities related to the transaction-reverting statements as early as possible [73].

3.7 Threats to Validity

3.7.1 Construct Validity

The study makes use of well-established metrics in software testing to compare the different approaches: structural coverage (*i.e.,* branch, line) and vulnerability detection capability (how well do the generated tests detect vulnerabilities). A time budget is used as the stopping condition for the search algorithm instead of the number of evaluations. Given that the approaches compared in the study use different genetic operators, with a different execution overhead, search time is a fairer metric for budget allocation [74].

3.7.2 External Validity

To make sure that the study's results can be generalized, the benchmark used to evaluate it has to contain a diverse set of smart contracts of a wide range of complexities. We created a benchmark with 100 real-world smart contracts gathered from *etherscan.io*. This benchmark contains contracts with different sizes and cyclomatic complexities.

3.7.3 Conclusion Validity

Evolutionary algorithms make use of randomness to search the problem space. To minimize the risk that the results were caused by favourable randomness, we have performed the experiment 20 times with different random seeds. We have followed the best practices for running experiments with randomized algorithms as laid out in well-established guidelines [66] and analyzed the possible impact of different random seeds on our results. We used two non-parametric tests: the unpaired Wilcoxon rank-sum test and the Vargha-Delaney $\hat{A}_{12}$ effect size to assess the significance and magnitude of our results.

3.8 Conclusions and Future Work

Previous studies focused on coverage-oriented heuristics to test and fuzz *Solidity* smart contract. However, they do not directly handle transaction-reverting statements, a vital mechanism within *Solidity* to protect the contract against invalid requests. To overcome this limitation, we proposed a novel fitness function based on interprocedural dependency analysis and context-sensitive instrumentation to exercise and test directly these statements.

We implemented the novel fitness function in the *Syntest-Solidity* [80] testing framework. The framework implements the state-of-the-art testing algorithm, called *DynaMOSA* [58], guided by well-established unit-level fitness functions. Our results show that our interprocedural fitness function improves the number of the vulnerabilities detected as well as structural coverage compared to the state-of-the-art unit-level alternative. Our results suggest that our approach has a wide range of applications being able to improve both test case generation and fuzzing algorithms.

Given our promising results, there are multiple potential directions for future work, including (i) a topology study on common transaction-reverting statement vulnerabilities and their prevalence, and (ii) constructing a build pipeline for smart contracts to prevent vulnerable contracts to go live.

4 Improving Test Case Generation for REST APIs Through Hierarchical Clustering

With the ever-increasing use of web APIs in modern-day applications, it is becoming more important to test the system as a whole. In the last decade, tools and approaches have been proposed to automate the creation of system-level test cases for these APIs using evolutionary algorithms (EAs). One of the limiting factors of EAs is that the genetic operators (crossover and mutation) are fully randomized, potentially breaking promising patterns in the sequences of API requests discovered during the search. Breaking these patterns has a negative impact on the effectiveness of the test case generation process. To address this limitation, this paper proposes a new approach that uses Agglomerative Hierarchical Clustering (AHC) to infer a linkage tree model, which captures, replicates, and preserves these patterns in new test cases. We evaluate our approach, called LT-MOSA, by performing an empirical study on 7 real-world benchmark applications w.r.t. branch coverage and real-fault detection capability. We also compare LT-MOSA with the two existing state-of-the-art white-box techniques (MIO, MOSA) for REST API testing. Our results show that LT-MOSA achieves a statistically significant increase in test target coverage (i.e., lines and branches) compared to MIO and MOSA in 4 and 5 out of 7 applications, respectively. Furthermore, LT-MOSA discovers 27 and 18 unique real-faults that are left undetected by MIO and MOSA, respectively.

This chapter has been published as *Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Improving Test Case Generation for REST APIs Through Hierarchical Clustering. 36th IEEE/ACM International Conference on Automated Software Engineering 2021 (ASE'21)* [77].

4.1 Introduction

Over the last decade, the software landscape has been characterized by the shift from large monolithic applications to component-based systems, such as microservices. These systems, together with their many diverse client applications, make heavy use of web APIs for communication. Web APIs are almost ubiquitous today and rely on well-established communication standards such as SOAP [143] and REST [144]. The shift towards component-based systems makes it ever-increasingly more important to test the system as a whole since many different components have to work together. Manually writing system-level test cases is, however, time-consuming and error-prone [145, 146].

For these reasons, researchers have come up with different techniques to automate the generation of test cases. One class of such techniques is search-based software testing. Recent advances have shown that search-based approaches can achieve a high code coverage [74], also compared to manually-written test cases [7], and are able to detect unknown bugs [8, 129, 130]. Search-based test case generation uses Evolutionary Algorithms (EAs) to evolve a pool of test cases through randomized *genetic operators*, namely *mutations* and *crossover/recombination*. More precisely, test cases are encoded as chromosomes, while statements (*i.e.,* method calls) and test data are encoded as the genes [19]. In the context of REST API testing, a test case is a sequence of API requests (*i.e.,* HTTP requests and SQL commands) on specific resources [37, 130].

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). Example 4.1 shows an example of HTTP requests made to a REST API that manages `products`. In the example, the first request authenticates the client to the API with the given `username` and `password`. In return, the client receives a `token` that can be used to make subsequent requests. The second request creates a new product by specifying the `id`, `price`, and the `token`. The `price` is then updated in the third request and the changes are retrieved in the last request.

The example above contains patterns of HTTP requests that strongly depend on the previous ones. The `GET` request can not retrieve a product that does not exist, and therefore, can not be successfully executed without request 2. Similarly, the `UPDATE` request can not be executed before the product is created. Lastly, request 2, 3, and 4, all depend on request 1 for the authentication token. Hence, HTTP requests should not be executed in any random order [147].

Test generation tools rely on EAs to build up sequences of HTTP requests iteratively through genetic operators, *i.e.,* crossover and mutation [37, 130, 140]. While these operators can successfully create promising sequences of HTTP requests, they do not directly recognize and preserve them when creating new test cases [54]. For example, the genetic operators may remove request 2 from the test case in Example 4.1, breaking requests 3 and 4 unintentionally.

In this paper, we argue that detecting and preserving patterns of HTTP requests, hereafter referred to as *linkage structures*, improves the effectiveness of the test case generation process. We propose a new approach that uses Agglomerative Hierarchical Clustering (AHC) to infer these linkage structures from automatically generated test cases in the context of REST APIs testing. In particular, 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

```
POST   authenticate?user=admin&password=pwd
POST   product?id=1&price=10.99&token={key}
UPDATE product/1?price=8.99&token={key}
GET    product/1?token={key}
```

Example 4.1: Motivating example of patterns in API requests.

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 our approach, we implemented a prototype based on *EvoMaster*, the state-of-the-art test case generation tool for Java-based REST APIs. We performed an empirical study with 7 benchmark web/enterprise applications from the *EvoMaster* Benchmark (EMB) dataset. We compared our 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).

Our results show that *LT-MOSA* covers significantly more test targets in 4 and 5 out of the 7 applications compared to *MIO* and *MOSA*, respectively. On average, *LT-MOSA*, 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 %). Furthermore, *LT-MOSA* could detect, on average, 27 and 18 unique real-faults that were not detected by *MIO* and *MOSA*, respectively.

In summary, we make the following contributions:

1. A novel approach that uses Agglomerative Hierarchical Clustering to learn and preserve *linkage structures* embedded in REST API.
2. An empirical evaluation of the proposed approach with the two state-of-the-art algorithms (*MIO* and *MOSA*) on a benchmark of 7 web/enterprise applications.
3. A full replication package including code and results [86].

The remainder of this paper is organized as follows. Section 4.2 summarizes the related work and background concepts. Section 4.3 introduces our approach called, *LT-MOSA*, and gives a detailed breakdown of how it works. Section 4.4 describes the setup of our empirical study. Section 4.5 discusses the obtained results and presents our findings. Section 4.6 discusses the threats to validity and Section 4.7 draws conclusions and identifies possible directions for future work.

4.2 Background and Related Work

This section provides an overview of basic concepts and related work in search-based software testing, REST API testing, test case generation, and linkage learning.

4.2.1 Search-based software testing

Search-based software testing has become a widely used and effective method of automating the generation of test cases and test data [4, 14]. Automatic test case generation significantly reduces the time needed for testing and debugging applications (*e.g.*, [5]) and it has been successfully used in industry (*e.g.,* [11, 128]). Popular tools for automatically generating test cases include EvoSuite [34], for unit testing, and Sapienz [11, 148], for Android testing.

Evolutionary Algorithms (EAs) are one of the most commonly used class of meta-heuristics in search-based testing. EAs have been used to generate both test data [4] and test cases [19]. The latter includes test data, method sequences and assertions. EAs are inspired by the biological process of evolution. It initializes and evolves a population of randomly generated individuals (test cases). These individuals are then evaluated based on a predefined fitness function. The individuals with the best fitness value are *selected* for reproduction through *crossovers* (swapping elements between two individuals) and random *mutations* (small changes to individuals). The offspring test cases resulting from the reproduction are then *evaluated.* Finally, the population for the next generation is created by selecting the best individuals across the previous population and the newly generated tests (*elitism*). This loop (reproduction, evaluation, and selection) continues until a stopping condition has been reached. The final test suite is created based on the population's best individuals.

4.2.2 REST API Testing

A REpresentational State Transfer (REST) API is oriented around resources. This differs from a command-oriented API like for example the Remote Procedure Call (RPC) standard. A REST API request performs an action on a specific resource. These actions are encoded by the different methods defined in the HTTP protocol. Common HTTP methods include `GET`, `HEAD`, `POST`, `PUT`, `PATCH`, and `DELETE`. These actions are performed on an endpoint, which is the location of the resource. An example of this would be performing a `GET` action on the `/user/3` endpoint to retrieve the information of a user with a user id of 3. Another example would be performing a `POST` action on the `/user/` endpoint to create a new user.

With the recent rise in popularity of REST APIs in the last decade, it is becoming more important to test this critical communication layer. There are two different ways system-level API testing can be approached: black-box and white-box testing. Black-box testing frameworks (*e.g.,* RESTTESTGEN [149], EvoMaster black-box [37]) examine the functionality of the system without looking at the internals of the system.

In contrast, white-box testing approaches rely on the internal structure of the system and measure the adequacy of the tests based on coverage criteria (*e.g.,* branch coverage). This allows the algorithm to easily identify which paths have been covered and which have not. Prior studies show white-box techniques achieve better results than their black-box counterparts [37]. Additionally, white-box techniques allow to integrate SQL databases in the test case generation process [130].

4.2.3 Test Case Generation for REST APIs

EvoMaster is a tool that aims to generate system-level test cases for REST APIs. It internally uses evolutionary algorithms to evolve the test cases iteratively. At the time of writing, *EvoMaster* provides two EAs. The first algorithm is the Many Independent Objective (MIO) algorithm proposed by Arcuri *et al.* [140]. The second algorithm is a variant of the Many-Objective Sorting Algorithm (MOSA) proposed by Panichella *et al.* [120]. Both of these algorithms are specifically designed for test case generation and consider the peculiarities of these systems. The pseudo-code of these algorithms can be found in the respective papers.

MIO

The Many Independent Objective (MIO) algorithm is an evolutionary algorithm that aims to improve the scalability of many-objective search algorithms for programs with a very large number of testing targets (in the order of millions) [140]. It works under the assumption that: (i) each target can be independently optimized; (ii) targets can be strongly related, for example when nested, or completely independent; (iii) not all targets can be covered. Based on these assumptions, *MIO* maintains a separate population for each target of the System Under Test (SUT). The fitness function for each population consists solely of the objective of that population. So, each population compares and ranks the individuals based on a single testing target. At the start of the algorithm, all populations are empty. Each iteration, the algorithm either samples a random new test case with a certain probability or it samples a test case from one of the populations with an uncovered target. This test case is then added to all populations with an uncovered target and evaluated independently. The population that is chosen when a test case is sampled from one of the populations, is based on the number of times a test case has been sampled from that population before. Each time a population is sampled, a counter is incremented. If a test case with a better fitness value is added to the population, the counter is reset to zero. The sampling mechanism chooses the population with the lowest counter. This makes sure that the algorithm won't get stuck on an unreachable target. When a population reaches a certain predefined size, it removes the test case with the worst fitness value. At the end of the algorithm, a test suite is built with the best test case from each population.

MOSA

The Many Objective Sorting Algorithm (MOSA) is an evolutionary algorithm that focuses on optimizing multiple objectives (*e.g.,* branches) at the same time [120]. It adapts the NSGA-II algorithm, which is one of the most popular multi-objective search algorithms [150]. In *MOSA*, a test case is represented as a chromosome. Each testing target (*e.g.,* branch, line) in the code corresponds to a separate objective measuring the distance (of a given test) toward reaching that target. The fitness of the test cases is measured according to a vector of scalar values that represent these different objectives. Since *MOSA* has many different objectives, it uses two preference criteria to determine which test cases should be selected and evolved first: (i) minimal distance to the uncovered target; (ii) test case length. More precisely, it first looks for the subset of the Pareto front that contains test cases with a minimum distance for each uncovered objective. When multiple test cases are equally close to cover the same target, the smallest test case will be selected. In each generation, an archive collects the test cases that cover previously uncovered targets. The archive is updated every time a newly generated test case covers new targets or covers already covered targets but with fewer statements.

Comparison

Both *MIO* and *MOSA* produce good results in both unit and system-level tests. In the context of system-level testing, Arcuri [140] showed that *MIO* achieves the best average results, but there are web/enterprise applications in which *MOSA* achieves higher coverage.

In unit testing, Campos *et al.* [74] showed that *MOSA* (and its variants) achieves overall better coverage than *MIO*. Therefore, in this paper, we consider both *MIO* and *MOSA* as they excel in different scenarios and are the state-of-the-art in test case generation for REST APIs.

Notice that an extension of *MOSA*, called DynaMOSA [58], has been proposed in the related literature for unit testing. Compared to *MOSA*, DynaMOSA organizes the coverage targets (*e.g.*, branches) of a given code unit into a global hierarchy based on their structural dependencies. Then, the list of search objectives is updated dynamically based on their structural dependencies and the previously covered targets. While previous studies in unit-testing showed that DynaMOSA outperforms its predecessor *MOSA* [58, 74], it cannot be applied for REST APIs as no global hierarchy exists across the coverage targets of different microservices or functions/classes within the same microservice[1].

Chromosome Representation

Test cases in both search algorithms included in *EvoMaster* are represented by two genes: action gene and input gene. The action gene represents the structure and order of the HTTP requests in the test case. *EvoMaster* extracts these actions from the Swagger/OpenAPI documentation that has to be provided for each system under test. An action gene consists of the HTTP method and the REST endpoint. An example of an action gene would be `POST /authentication`.

The input gene represents the input data for the HTTP request. An example of this input data would be the username and password that are required by the `/authentication` endpoint. This input data is sampled from the source code of the SUT.

4.2.4 Linkage Learning in EAs

Linkage-learning refers to a large body of work in the evolutionary computation community that aims to infer *linkage structures* present in promising individuals [151]. Linkage structures are groups of "good" genes that contribute to the fitness of a given population. Accurate inference of *linkage structures* has been used to design "competent" genetic operators [152] for numerical problems. These operators are designed to replicate rather than break groups of genes (patterns) into the offspring.

To learn linkage structures from numerical chromosomes, researchers have used different unsupervised machine learning algorithms. BOA [153] constructs a Bayesian Network and creates new numerical chromosomes using the joint distribution encoded by the network. DSMGA [154] uses Dependency Structure Matrix (DSM) clustering and applies the crossover by exchanging gene clusters between parent chromosomes. 3LO [155] employs local optimization as an alternative method for linkage learning.

Two state-of-the-art EAs for numerical problems are LT-GA [156] and GOMEA [93]. Both algorithms use clustering to infer linkage-trees, representing the linkage structures between genes (problem variables) using tree-like structures. GOMEA uses agglomerative hierarchical clustering as a faster and more efficient way to learn linkage-trees [93]. GOMEA uses the *gene-pool optimal mixing* to create new solutions by applying a local search within the recombination procedure. More precisely, it creates offspring solutions from one single parent by iteratively replicating (copying) gene clusters from different

[1]Micro-services are loosely coupled and deployable independently.

donors. In each iteration, the new solution is evaluated; if its fitness improves, the replicated genes are kept; otherwise, the change is reverted.

Linkage models have been successfully applied to evolutionary algorithms for numerical [157], permutation [158], and binary optimization problems [156, 159] with fixed length chromosomes. However, test cases for REST APIs are characterized by a more complex structure [37]: each test is a sequence of HTTP requests towards a RESTful service, each with input data, such as HTTP headers, URL parameters, and payloads for `POST`/`PUT`/`PATCH` methods. Besides, a test case might include SQL data and commands for microservices that use databases [130]. Finally, test cases have a variable size, and their lengths can also vary throughout the generations. Therefore, we need to tailor existing *linkage learning* methods according to the test case characteristics discussed above.

4.3 Approach

This section presents our approach, called *LT-MOSA*, for system-level test cases generation and that incorporates and tailors linkage learning into *MOSA* [120]. We selected *MOSA* as the base algorithm to apply linkage learning because it evolves a single population of test cases, which is a requirement for the learning process. Additionally, *MOSA* has been proved to be very competitive in the context of RESTful API testing [140], unit testing [74, 101], and DNN testing [160].

Alg. 3 outlines the pseudo-code of *LT-MOSA*. The parts where *LT-MOSA* deviates from *MOSA* are highlighted with a blue color. *LT-MOSA* starts with initializing the population P and computing the corresponding objective scores (line 2). Each test case is composed of HTTP calls (*actions*) and SQL commands (*database actions*) [130]. The RANDOM-POPULATION function also executes the generated tests and computes their objective scores using the *branch distance* [121]. The *branch distance* is a well-known heuristic in search-based testing to measure how far each test case is from reaching a given coverage target (*e.g.,* branch). Then, the test cases are sorted in sub-dominance fronts using the *preference sorting algorithm* [120], in line 4. The test cases within the first front (Front[0]) are the closest ones in P to reach the coverage targets and, therefore, the fittest individuals to consider for model learning.

Afterwards, the population P is evolved through subsequent generations within the loop in lines 5-20. Each generation starts by training a *linkage tree model* on the first non-dominated front (line 6) with the goal of learning patterns of HTTP and SQL actions that strongly contribute to the "optimality" of the population. We discuss the learning procedure in detail in Section 4.3.1. Once the *linkage tree model* is obtained, *LT-MOSA* selects the *fittest* test cases using the *tournament selection* (line 9 and 11) and creates an offspring population P' by using a *linkage-based recombination* [93] (line 12) and *mutation* [130] (line 13 and 15). The *linkage-based recombination* is a specialized *crossover* that relies on the *linkage tree model* to decide which patterns of genes (HTTP requests) can be copied into the offspring test cases. We describe the *linkage-based recombination* operator in Section 4.3.2.

LT-MOSA adds the newly generated tests into the offspring population (line 16), executes them, and updates the archive (line 17) in case new coverage targets have been reached. The generation ends by selecting the best M test cases across the existing population P and the offspring population P'. This selection is made by combining the two

Algorithm 3: *LT-MOSA*

```
Input:
Coverage targets $\Omega = \{\omega_1, \dots, \omega_n\}$
Population size $M$
Frequency $K$ for updating the linkage tree model
Result: A test suite $T$
1  begin
2      $P \longleftarrow$ RANDOM-POPULATION($M$)
3      archive $\longleftarrow$ UPDATE-ARCHIVE($\emptyset$, $P$)
4      Fronts $\longleftarrow$ PREFERENCE-SORTING($R$)
5      while not (stop_condition) do
6          L $\longleftarrow$ LEARN-LINKAGE-MODEL(Fronts[0], $K$)
7          $P' \longleftarrow \emptyset$
8          for index = 1..$M$ do
9              parent $\longleftarrow$ TOURNAMENT-SELECTION($P$)
10             if apply_recombination then
11                 donor $\longleftarrow$ TOURNAMENT-SELECTION($P$)
12                 offspring $\longleftarrow$ LINKAGE-RECOMB(parent, donor, L)
13                 offspring $\longleftarrow$ MUTATION(offspring)
14             else
15                 offspring $\longleftarrow$ MUTATION(parent)
16             $P' \longleftarrow P' \bigcup$ {offspring}
17             archive $\longleftarrow$ UPDATE-ARCHIVE(archive, offspring)
18         $R \longleftarrow P \bigcup P'$
19         Fronts $\longleftarrow$ PREFERENCE-SORTING($R$)
20         $P \longleftarrow$ ENVIRONMENTAL-SELECTION(Fronts, $M$)
21     $T \longleftarrow$ archive
```

population into one single pool R of size $2 \times M$ (line 18), applying the *preference sorting* (line 19), and selecting M solutions from the non-dominated fronts starting from Front[0] until reaching the population size M (line 20).

The search stops when the termination criteria are met (condition in line 5), the final test suite will then be composed of all test cases that have been stored in the *archive* throughout the search. Note that *LT-MOSA* updates the *archive* in each generation by storing the shortest test case covering each target ω_i. Finally, the list of objectives is updated such that the search focuses only on the targets (branches) that are left uncovered.

In the following sub-sections, we detail the key novel ingredients in *LT-MOSA*, namely the *linkage model learning* (LT) (Section 4.3.1), the *linkage-based recombination operator* (Section 4.3.2), and the *mutation* operator (Section 4.3.3).

4.3.1 Linkage tree learning

In this section, we describe the main changes we applied to the traditional *linkage learning* and adapt it to our context, *i.e.,* test case generation for RESTful APIs.

Linkage Encoding

The first problem we had to solve is encoding the test cases into discrete vectors of equal length, which can be interpreted and analyzed via hierarchical clustering. To this aim, we opted for *encoding* test cases as binary vectors whose entries denote the presence (or not) of the possible HTTP requests. Given an SUT (software under tests), there are N possible HTTP requests to the available APIs. This information can be extracted from the Swagger/OpenAPI definition [37], which is a widely-used tool for REST API documentation. A Swagger definition contains the HTTP operations available for each API endpoint. Each operation contains both fixed and variable parts. The fixed part includes the type of operation (`POST`, `GET`, `PUT`, `PATCH`, and `DELETE`), the IP address or the URL of the target API, and the HTTP headers. For the variable part, the Swagger definition includes information about the input data (*e.g., string*, *double*, *date*, etc.) that can vary. Therefore, for each API endpoint, we identify the available HTTP operations, hereafter referred to as *actions*, by parsing the Swagger definition.

Let $\mathcal{S} = \{S_1, \dots, S_N\}$ be the set of N *HTTP actions* available for the target SUT. We encode each test case T as a binary string of size N as follows:

$$E(T) = \langle e_1, \dots, e_N \rangle \;\; with \;\; e_i = \begin{cases} 0 & if\ S_i \notin T \\ 1 & if\ S_i \in T \end{cases} \tag{4.1}$$

In other words, each element e_i in the encoded vector $E(T)$ is set to 1 if the test case T contains the *action* S_i; 0 otherwise. The linkage model is trained on the binary-coded vectors rather than on the original test cases. This encoding is used to determine, via statistical analysis, which group of HTTP *actions* often appear together within the fittest test cases, and which ones never occur together. This information is used to create more efficient *recombination* operators.

Linkage Model Training

In this paper, we use *agglomerative hierarchical clustering* (AHC) over other techniques (e Bayesian Networks) for linkage tree learning. This is because prior studies show AHC is more efficient [93]. In particular, we apply the UPGMA (unweighted pair group method with arithmetic mean) algorithm [161]. In each iteration, UPGMA merges two clusters that are most similar based on the average distance across the data points (genes in our case) in the two clusters. The similarity between two HTTP actions genes S_i and S_j is computed using the mutual information as suggested by Thierens and Bosman [93]:

$$MI(S_i, S_j) = H(S_i) + H(S_j) - H(S_i, S_j) \tag{4.2}$$

where $H(.)$ denotes the information entropy [162].

Note that *LT-MOSA* infers the *linkage tree* for the most promising part of the population, *i.e.,* the first non-dominated front (line 6 in Alg. 3). Furthermore, the training process is applied to the encoded test cases according to the schema described in Section 4.3.1

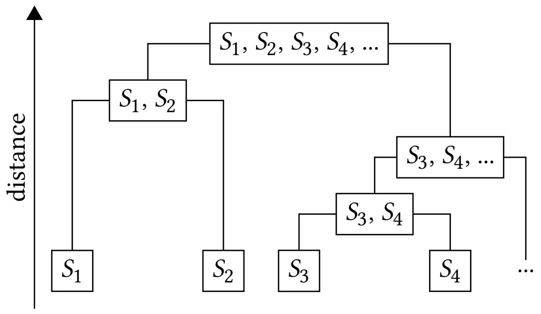


Figure 4.1: Example of *linkage tree* model and the Family Of Subset (FOS)

rather than on the actual test cases. Hence, the *linkage tree* obtained with UPGMA captures the hierarchical relationship between HTTP actions in our case.

For example, let us consider the *linkage tree* depicted in Figure 4.1. In the example, the set of actions $\mathcal{S} = \{S_1, S_2, S_3, S_4, ...\}$ (the root of the tree) are partitioned into two clusters: S_1, S_2 and $S_3, S_4, ...$; each sub-cluster can be further divided in sub-cluster until reaching the leaf node. In general, the linkage tree has N leaves and $N-1$ internal nodes. The root node contains all HTTP actions of the SUT. Each internal node divides the set of HTTP actions into two mutually exclusive clusters (the child nodes). Finally, the leaves contain the individual HTTP actions, which are the starting point of the UPGMA algorithm.

The *linkage tree* nodes are often referred to as *Family of Subsets* ($\mathcal{F}$) in the related literature [93, 156]. Each node (or subset) $\mathcal{F}' \in \mathcal{F}$ with $|\mathcal{F} > 2|$ has two mutually exclusive subsets (or child nodes) $\mathcal{F}_x$ and $\mathcal{F}_y$ such that $\mathcal{F}_x \bigcap \mathcal{F}_y = \emptyset$ and $\mathcal{F}_x \bigcup \mathcal{F}_y = \mathcal{F}'$. Each subset $\mathcal{F}' \in \mathcal{F}$ represents a cluster of HTTP actions that often appear together and characterized the best test cases in the population. Therefore, the recombination operator should be applied by preserving these subsets (patterns) when creating new offspring tests. The next subsection describes the subsets-preserving recombination operator we implemented in *LT-MOSA*.

The computation complexity of UPGMA is $O(N \times M^2)$, where N is the number of genes and M is the population size. To reduce its overhead, the linkage tree learning procedure is not applied in each generation. Instead, the linkage tree model is re-trained every K generations (line 6 of Alg. 3).

4.3.2 Linkage-based Recombination

MOSA creates offspring tests using the *single-point crossover* [120]. This crossover operator is the classic *recombination operator* used in genetic algorithms [163], and test case generation [19, 34]. This operator generates two offsprings by randomly swapping statements between two parent tests T_1 and T_2. As argued in Section 4.1, exchanging statements between test cases in a randomized manner can lead to breaking gene patterns (HTTP actions) that characterized the fittest individuals. Randomized recombination is also disruptive toward building good partial solutions (building blocks), negatively affecting the overall convergence [93].

Therefore, *LT-MOSA* uses a *linkage-based recombination operator* rather than the classical *single-point crossover* to preserve the patterns of HTTP actions identified by the linkage tree model. The *recombination operator* generates only one offspring starting from two existing test cases, called *parent* and *donor*. Both test cases are selected from the current population P as indicated in lines 9 and 11. The offspring is created by copying all genes (HTTP actions with input data) from the *parent* and further injecting only some genes from the *donor*. These genes are selected by exploiting the *linkage tree model* trained according to Section 4.3.1.

More precisely, we first identify the gene patterns (*i.e.,* the subsets $\mathcal{F}' \in \mathcal{F}$) that the *donor* contains. This is done by iterating across all subsets in the linkage tree model $\mathcal{F}$ and identifying the subsets $\mathcal{F}' \subset \mathcal{F}$ that appear in the encoded vector (see Section 4.3.1) of the *donor*. *LT-MOSA* randomly selects one of the identified subsets in $\mathcal{F}'$ and inserts it into the offspring. The *injection point* is randomly chosen, and the selected genes (HTTP actions with test data) are inserted into the offspring in the exact order as they appear in the *donor*.

If the *donor* does not contain any subset according to the linkage tree (*i.e.,* $\mathcal{F}' = \emptyset$), then the offspring is generated by applying the traditional *single-point crossover*. This operator can be applied to the latter case since the *linkage tree model* could not identify any useful gene pattern within the *donor*.

4.3.3 Mutation

In *MOSA*, each test case is mutated with a probability $p_m = 1/L$, where L is the test case length [120]. This also reflects the existing guidelines in evolutionary computation [164, 165], which suggest using a mutation probability p_m proportional to the size of the chromosome.

In recent years, Arcuri [140] improved the *mutation operator* in the context of system-level test case generation by using a variable mutation rate. Indeed, the mutation operator in *MIO* [140] increases the number of mutations applied to each test case from 1 (start of the search) up to 10 (end of the search) with linear incremental steps. The importance of having a large mutation rate for RESTful API testing has also been confirmed by a recent study results [130].

Based on these observations, *LT-MOSA* uses the same mutation rate of *MIO* (*i.e.,* increasing mutation rate from 1 up to 10 mutations) rather than the fixed mutation rate of *MOSA*.

4.4 Empirical Study

This section details the empirical study that we carried out to evaluate the effectiveness of the proposed solution, called *LT-MOSA*, and compare it with the state-of-the-art algorithms (*MIO*, *MOSA*) *w.r.t.* to the following testing criteria: (i) code (line and branch) coverage and (ii) *fault detection* capability.

4.4.1 Benchmark

This study uses the *EvoMaster* Benchmark (EMB)[2] version 1.0.1. This benchmark was specifically created as a set of web/enterprise applications for evaluating the test case

[2]https://github.com/EMResearch/EMB/releases/tag/v1.0.1

Table 4.1: Web applications from the *EvoMaster* Benchmark (EMB) used in the empirical study. Reports the number of Java classes, test coverage targets (*i.e.,* lines and branches), and the number of endpoints.

Application	Classes	Coverage Targets	Endpoints
CatWatch	69	2182	23
Features-Service	23	513	18
NCS	10	652	7
OCVN	548	8010	258
ProxyPrint	68	3758	74
Scout-API	75	3449	49
SCS	13	865	11
Total	1655	19 429	440

generation algorithms implemented in *EvoMaster*. We selected this benchmark since it has been widely used in the literature to assess test case generation approaches for REST APIs [37, 140].

In this study, we used five real-world open-source Java web applications and two artificial Java web applications. *CatWatch* is a metrics dashboard for GitHub organizations. *Features-Service* is a REST Microservice for managing feature models of products. *OCVN* (Open Contracting Vietnam) is a visual data analytics platform for the Vietnam public procurement data. *ProxyPrint* is a platform for comparing and making requests to print-shops. *Scout-API* is a RESTful web service for the hosted monitoring service "Scout". *NCS* (Numerical Case Study) is an artificial application containing numerical examples. *SCS* (String Case Study) is an artificial application containing string manipulation code examples. We use *NCS* and *SCS* since they have been designed for assessing test generation tools. These artificial web applications allow to cover many different scenarios (*e.g.,* deceptive branches [140]). Compared to previous studies [37, 140], we added the *OCVN* application as it is the largest real-world system in the benchmark. We additionally removed the *rest-news* application as it contains artifical examples that are used for classroom teaching.

Table 4.1 summarizes the main characteristics of the applications in the benchmark, such as the number of classes, the number of test coverage targets, and the number of endpoints included in the service. This benchmark contains a total of 1655 classes with around 20 000 test coverage targets and 440 endpoints, not including tests or third-party libraries.

EvoMaster requires a test driver for the application under test. This test driver contains a controller that is responsible for starting, resetting, and stopping the SUT. We used the test drivers available in the EMB benchmark for the web applications used in this study.

4.4.2 Research Questions

Our empirical evaluation aims to answer the following research questions:

RQ1 *How does LT-MOSA perform compared to the state-of-the-art approaches with regard to code coverage?*

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

RQ3 *How effective is LT-MOSA at covering test targets over time compared to the state-of-the-art approaches?*

The first two research questions aim to evaluate if preserving patterns in HTTP requests through linkage learning can improve the *effectiveness* of test case generation for REST APIs by reaching a higher coverage and detecting more faults.

The last research question aims to answer if our approach, *LT-MOSA*, is more *efficient* in covering these test targets by measuring how many test targets are covered at different times within the search budget.

4.4.3 Baseline

To answer our research questions, we compare *LT-MOSA* with the two state-of-the-art search-based test case generation algorithm for REST APIs as a baseline:

- *Many Independent Objective (MIO)* is the state-of-the-art for REST API testing, and it is the default search algorithm in *EvoMaster*. *MIO* aims to improve the scalability of many-objective search algorithms for programs with a very large number of testing targets (see Section 4.2.3).

- *Many-Objective Sorting Algorithm (MOSA)* is the base algorithm we use to build and design *LT-MOSA*. Therefore, we want to assess that our approach outperforms its predecessor. Furthermore, *MOSA* has been proven to be very competitive in the context of REST APIs testing (see Section 4.2.3).

4.4.4 Prototype Tool

We have implemented *LT-MOSA* in a prototype tool that extends *EvoMaster*, an automated system-level test case generation framework. In particular, we implemented the approach as described in Section 4.3 within *EvoMaster*.

The variant of *MOSA* implemented in *EvoMaster* differs from the original algorithm proposed by Panichella *et al.* [120]. The *EvoMaster* variant does not use the crossover operator but merely relies on the mutation operator to create new test cases. Therefore, we implemented the *single-point* crossover as described in [120] and adapted it to the encoding schema used for representing REST API requests in *EvoMaster*. See Section 4.2.3 for more details.

We chose *EvoMaster* because it already implements the state-of-the-art test case generation algorithms, and it is publicly available on GitHub. Besides, *EvoMaster* implements *testability transformations* to improve the guidance for search-based algorithms [166] and can handle SQL databases [130].

4.4.5 Parameter Setting

For this study, we have chosen to adopt the default search algorithm parameter values set by *EvoMaster*. It has been empirically shown [109] that although parameter tuning has an impact on the effectiveness of a search algorithm, the default values, which are commonly used in literature, provide reasonable and acceptable results. Thus, this section only lists a few of the most important search parameters and their values:

Search Budget
We chose a search budget (stopping condition) based on time instead of the number of executed tests. This choice was made as search time provides the fairest comparison given that we consider different kinds of algorithms with diverse internal routines (also in terms of computational complexity). Additionally, practitioners will often only allocate a certain amount of time for the algorithm to run. The search budget for all algorithms was set to 30 minutes as this strikes a balance between giving the algorithms enough time to explore the search space and making the study infeasible to execute. If the algorithm has covered all its test objectives, it will stop prematurely. Note that running time is considered a less biased stopping condition than counting the number of executed tests since not all tests have the same running time [7, 34, 130, 140]. We further discuss this aspect in the threats to validity.

***MIO* parameters**
For *MIO*, we used the default settings as provided in the original paper by Arcuri *et al.* [140, 167].

- *Population size:* We use the default population size of 10 individuals per testing target. Notice that *MIO* uses separate populations for the different targets.
- *Mutation:* We use the default number of applied mutations on sampled individuals, which linearly increases from 1 to 10 by the end of the search.
- *F:* We use the default percentage of time after which a focused search should start of 0.5.
- P_r: We use the default probability of sampling at random, instead of sampling from one of the populations, of 0.5. This value will linearly increase/decrease based on the consumed search budget and the value of F.

***MOSA* parameters**
For *MOSA*, we used the default settings described in the original paper *et al.* [58].

- *Population size:* 50 individuals (test cases).
- *Mutation:* We use the *uniform mutation*, which either changes the test case structures (adding, deleting, or replacing API requests) or the input data. Test structure and test data mutation are equally probable, i.e, each has 50% probability of being applied. The mutation probability for each statement/data gene is equal to $1/n$, where n is the number of statements in the test case.
- *Recombination Operator:* We use the single-point crossover with a crossover probability of 0.75.
- *Selection:* We use the tournament selection with the default tournament size of 10.

***LT-MOSA* parameters**
For *LT-MOSA*, we used the same parameters as for the *MOSA* algorithm except for the mutation operator, for which we use the mutation described in Section 4.3.3. Additionally, we use the following parameter values for the *linkage learning* model:

- *Frequency:* We use a frequency of 10 generations for generating a new Linkage-Tree model. From a preliminary experiment that we have performed, this provides a balance between having too much overhead (< 10) and having an outdated model (> 10).
- *Recombination Operator:* We use the linkage-based recombination with a probability of 0.75.

4.4.6 Real-fault Detection

To find out the number of unique faults that the search algorithms can detect, *EvoMaster* checks the returned status codes from the HTTP requests for 5xx server errors, as an indicator for a fault. Since web applications handle many different clients, when an error occurs it is not desirable for the application to crash or exit as this would also impact the other clients. Thus, web applications return a status code in the 5xx range, indicating an error has occurred on the server's side. *EvoMaster* keeps track of the last executed statement in the SUT (excluding third-party libraries) when a 5xx status code is returned, to distinguish between different errors that happen on the same endpoint.

4.4.7 Experimental Protocol

For each web application, all three search algorithms (*MOSA*, *MIO*, *LT-MOSA*) are separately executed, and the resulting number of test targets that are covered is recorded.

Since all three search algorithm used in the study are randomized, we can expect a fair amount of variation in the results. To mitigate this, we repeated every experiment 20 times, with a different random seed, and computed the average (median) results. In total, we performed 420 executions, three search algorithms for seven web applications with 20 repetitions each. With each execution taking 30 minutes, the total execution time is 8.75 days of consecutive running time.

To determine if the results (*i.e.,* code coverage and fault detection capability) of the three different algorithms are statistically significant, we use the unpaired Wilcoxon rank-sum test [67] with a threshold of 0.05. This is a non-parametric statistical test that determines if two data distributions are significantly different. Since we have three different data distributions, one for each search algorithm, we perform the Wilcoxon test pairwise between each configuration pair: (i) *LT-MOSA* and *MOSA*; (ii) *LT-MOSA* and *MIO*. We combine this with the Vargha-Delaney statistic [68] to measure the effect size of the result, which determines how large the difference between the two configuration pairs is.

To determine how the two configuration pairs compare in terms of efficiency, we analyze the code coverage at different points in time. While the effectiveness measures the code coverage only at the end of the allocated time, we also want to analyze how algorithms perform during the search. One way to quantify the efficiency of an algorithm is by plotting the number of test targets at predefined intervals during the search process. This is called a convergence graph. We collected the number of targets that have been covered for every generation of each independent run. To express the efficiency of the experimented algorithms using a single scalar value, we computed the overall convergence rate as the Area Under the Curve (AUC) delimited by the convergence graph. This metric is normalized by dividing the AUC in each run by the maximum possible AUC per application[3].

4.5 Results

This section details the results of the empirical study with the aim of answering our research questions.

4.5.1 RQ1: Code Coverage

Table 4.2 reports the median and inter-quartile range (IQR) of the number of test targets covered by *MIO*, *MOSA*, and *LT-MOSA* for each of the seven applications.

From Table 4.2, we observe that *LT-MOSA* achieved the highest median value (avg. +334.75 targets) for four out of the seven applications, and *MOSA* and *MIO* both achieved the highest median value (+10.00 and +0.5 targets, respectively) for 1 out of the 7 applications. The largest increase in code coverage is observable for *OCVN*, for which *LT-MOSA* covered +1100.00 more targets. For *SCS*, both *LT-MOSA* and *MIO* covered the same number of targets (853.00). For both artificial applications, namely *NCS* and *SCS*, the difference between the search algorithms is minimal (≤ 1).

In terms of variability (IQR), there is no clear trend with regard to the applications under test and/or the search approaches. For example, in some cases, the winning configuration (*LT-MOSA* on *CatWatch*) has the highest IQR with a significant margin (161.75 vs. 132.00 or 12.75). On *Scout-API*, *LT-MOSA* yields the lowest IQR by a significant margin (33.25 vs. 8.00 or 5.50). Within and across each search algorithm, the IQR varies.

Table 4.3 reports the statistical significance (p-value), calculated by the Wilcoxon test, of the difference between the number of targets covered by *LT-MOSA* and the two baselines, *MIO* and *MOSA*. It also reports the magnitude of the differences according to the Vargha-Delaney $\hat{A}_{12}$ statistic.

From Table 4.3, we can observe that for the non-artificial web applications, *LT-MOSA* achieves a significantly higher code coverage than *MIO* in four out of five applications with a *large* effect size ($\hat{A}_{12}$ statistics). *LT-MOSA* significantly outperforms *MOSA* in all five applications. The effect size is *large* in four applications and *small* for *CatWatch*. For the two artificial applications, *NCS* and *SCS*, there is no statistical difference between the results of *LT-MOSA* and the two baselines (*MIO* and *MOSA*). This confirms our preliminary results reported in Table 4.2. Moreover, the difference between *LT-MOSA* and *MIO* is not significant for *Features-Service*. Finally, in none of the applications in our benchmark, neither of the baselines achieved a significantly larger coverage than *LT-MOSA*.

In summary, *LT-MOSA* achieves significantly higher (most of the cases) or equal code coverage when applied to REST APIs as compared to both *MIO* and *MOSA*.

[3]Which corresponds to the area of the box with a height of the maximum code coverage and a width equal to the search budget.

Table 4.2: Median number of covered test targets.

Application	*MIO*		*MOSA*		*LT-MOSA*	
	Median	IQR	Median	IQR	Median	IQR
CatWatch	1173.00	12.75	1177.00	132.00	1215.50	161.75
Features-Service	488.00	72.25	455.50	33.25	478.00	5.00
NCS	622.50	1.25	623.00	4.00	622.00	3.25
OCVN	2421.50	374.75	2931.50	271.00	4031.50	338.75
ProxyPrint	1485.50	16.25	1501.00	78.25	1602.50	59.00
Scout-API	1727.50	54.75	1707.00	69.00	1826.50	33.25
SCS	853.00	5.50	852.00	8.00	853.00	3.00

Table 4.3: Statistical results (p-value and $\hat{A}_{12}$) for the covered test targets (*RQ1*). Significant p-values (*i.e.,* p-value < 0.05) are marked gray.

Application	*LT-MOSA* vs *MIO*		*LT-MOSA* vs *MOSA*	
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
CatWatch	<0.01	0.87 (Large)	0.04	0.66 (Small)
Features-Service	0.34	0.54	<0.01	0.83 (Large)
NCS	0.86	0.49	0.90	0.38 (Small)
OCVN	<0.01	1.00 (Large)	<0.01	1.00 (Large)
ProxyPrint	<0.01	1.00 (Large)	<0.01	0.86 (Large)
Scout-API	<0.01	0.96 (Large)	<0.01	0.96 (Large)
SCS	0.86	0.40 (Small)	0.50	0.50

4.5.2 RQ2: Fault Detection Capability

Table 4.4 reports the median number of real-faults (and the corresponding IQR) detected by *MIO*, *MOSA*, and *LT-MOSA* for each of the seven applications.

We observe that for both the artificial applications, *NCS* and *SCS*, the number of faults that have been detected by any search algorithm is zero. This is because these artificial applications are not designed to fail softly by returning 5xx faults. For the open-source applications, *LT-MOSA* detects the largest number of faults (avg. +3.40 faults) in all five cases. The largest increase in fault-detection rate is observable for the *OCVN* application, with +10.5 more faults detected by *LT-MOSA* than the baselines. It is noteworthy that the largest difference between *LT-MOSA* and the baselines is on the *OCVN* application, which is the application with by far the most classes (*i.e.,* 548) and endpoints (*i.e.,* 258) in our benchmark. This could be explained by the fact that *LT-MOSA* also achieved a much higher code coverage for this application. However, the difference in detected faults for *OCVN* is larger than for the other applications in the benchmark, which could indicate that *LT-MOSA* is especially effective for testing large REST APIs. The faults detected by *LT-MOSA* are a superset of the faults detected by *MIO* and *MOSA*. These newly discovered faults originate from the additional coverage that *LT-MOSA* achieves.

Table 4.5 reports the results of the statistical test, namely the Wilcoxon test, applied

Table 4.4: Median number of detected real-faults.

Application	*MIO*		*MOSA*		*LT-MOSA*	
	Median	IQR	Median	IQR	Median	IQR
CatWatch	13.00	0.25	12.00	2.00	13.50	2.25
Features-Service	17.00	0.00	17.00	0.00	18.00	0.50
NCS	0	0.00	0	0.00	0	0.00
OCVN	34.00	5.25	37.50	5.25	48.00	3.50
ProxyPrint	32.50	1.00	33.00	1.00	34.00	0.25
Scout-API	54.50	3.75	60.00	1.50	64.00	3.00
SCS	0	0.00	0	0.00	0	0.00

Table 4.5: Statistical results (p-value and $\hat{A}_{12}$) for the detected real-faults (*RQ2*). Significant p-values (*i.e.,* p-value < 0.05) are marked gray.

Application	*LT-MOSA* vs *MIO*		*LT-MOSA* vs *MOSA*	
	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$
CatWatch	<0.01	0.7 (Large)	<0.01	0.84 (Large)
Features-Service	<0.01	0.92 (Large)	<0.01	0.98 (Large)
NCS	-	-	-	-
OCVN	<0.01	0.99 (Large)	<0.01	0.92 (Large)
ProxyPrint	<0.01	0.89 (Large)	0.03	0.66 (Small)
Scout-API	<0.01	1.00 (Large)	<0.01	0.91 (Large)
SCS	-	-	-	-

to the number of faults detected by *LT-MOSA* and the two baselines, *MIO* and *MOSA*. It also reports the magnitude of the differences (if any) obtained with the Vargha-Delaney $\hat{A}_{12}$ statistic. Significant p-values (*i.e.,* p-value < 0.05) are highlighted with gray color. From Table 4.5, we can observe that *LT-MOSA* detects a significantly higher number of faults than *MIO* and *MOSA* in all non-artificial applications. The effect size ($\hat{A}_{12}$) is *large* in all comparisons, except for *ProxyPrint*, where the effect size is *small* when comparing *LT-MOSA* and *MOSA*. Since none of the algorithms detected any faults in the artificial applications, Table 4.5 does not report any p-value or $\hat{A}_{12}$ statistics for these applications.

In summary, we can conclude that *LT-MOSA* detects more faults than the state-of-the-art approaches, namely *MIO* and *MOSA*, for all applications in our benchmark.

4.5.3 RQ3: Code Coverage over Time

Table 4.6 reports the median Area Under the Curve (AUC) related to the number of targets covered over time by *MIO*, *MOSA*, and *LT-MOSA* for each of the seven applications. The AUC indicates how efficient the search algorithms are at reaching a certain code coverage.

Table 4.6: Median normalized AUC for the number of covered test targets. The highest values are marked in gray.

Application	*MIO*	*MOSA*	*LT-MOSA*
CatWatch	0.77	0.78	0.78
Features-Service	0.78	0.75	0.82
NCS	0.99	0.99	0.99
OCVN	0.50	0.50	0.66
ProxyPrint	0.87	0.82	0.88
Scout-API	0.84	0.81	0.86
SCS	0.95	0.96	0.96

For more information on how the AUC is calculated and normalized see Section 4.4.7. Table 4.6 highlights the search algorithm (in gray color) that achieved the highest AUC value.

We observe that for the open-source applications, *LT-MOSA* has the highest AUC (avg. +0.06) in four out of five applications, with the largest difference (+0.16) in the *OCVN* application. For *CatWatch*, both *MOSA* and *LT-MOSA* have the same AUC (*i.e.,* 0.78). From Tables 4.2 and 4.3 however, we can see that *LT-MOSA* covers significantly more targets (+38.5) after 30 minutes of search budget. This means that *MOSA* reaches a higher coverage in the beginning but loses to *LT-MOSA* over time.

Figure 4.2 shows the (median) number of targets covered over time by the different search algorithms for *OCVN*, which is the largest application in our benchmark. In the beginning of the experiment (0-500 seconds), *MIO* and *LT-MOSA* perform roughly equal. After the first 500 seconds, *LT-MOSA* outperforms *MIO*. This results in a much larger AUC value (+ 0.16) for *LT-MOSA* compared to *MIO* as indicated in Table 4.6. We conclude that *LT-MOSA* significantly outperforms both *MOSA* and *MIO* in term of effectiveness and efficiency on this application. To reaffirm this, we can observe that in Figure 4.2, *MIO* never reaches 2700 covered targets, *MOSA* takes 1311 seconds to reach that many targets, and *LT-MOSA* performs this in just 713 seconds, almost half the time of *MOSA*.

For the two artificial applications, *NCS* and *SCS*, the difference in AUC between the three search algorithms is very minimal (≤ 0.01). From Table 4.2, we can also see that *LT-MOSA* covers one target more than *MOSA* on *SCS* and one target less on *NCS*. However, they both yield the same AUC, *i.e.,* 0.96 (*SCS*) and 0.99 (*NCS*). These results are in line with the results from *RQ1*.

In summary, we can conclude that *LT-MOSA* achieves higher AUC values than the baselines, *i.e.,* it covers more targets and in less time.

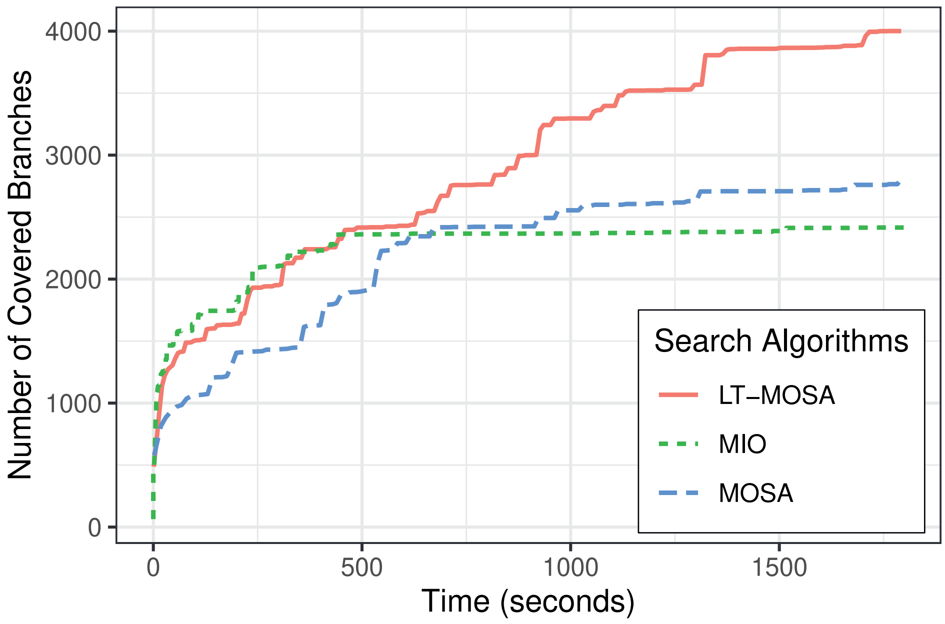


Figure 4.2: Average number of targets covered by our approach (*LT-MOSA*) and the baselines (*MOSA*, *MIO*) for *OCVN*.

4.6 Threats to Validity

This section discusses the potential threats to the validity of the study performed in this paper.

4.6.1 Construct Validity

We rely on well-established metrics in software testing to compare the different test case generation approaches, namely code coverage, fault detection capability, and running time. As a stopping condition for the search, we measured the search budget in terms of running time (*i.e.,* 30 minutes) rather than considering the number of executed tests, or HTTP requests. Given that the different algorithms in the comparison use different genetic operators, with different overhead, execution time provides a fairer measure of time allocation.

4.6.2 External Validity

An important threat regards the number of web services in our benchmark. We selected seven web/enterprise applications from the EMB benchmark. The benchmark has been widely used in the related literature on testing for REST APIs. The applications are diverse in terms of size, application domain, and purpose. Further experiments on a larger set of web/enterprise applications would increase the confidence in the generalizability of our study. A larger empirical evaluation is part of our future agenda.

4.6.3 Conclusion Validity

Threats to *conclusion validity* are related to the randomized nature of EAs. To minimize this risk, we have performed each experiment 20 times with different random seeds. We have followed the best practices for running experiments with randomized algorithms as laid out in well-established guidelines [66] and analyzed the possible impact of different random seeds on our results. We used the unpaired Wilcoxon rank-sum test and the Vargha-Delaney $\hat{A}_{12}$ effect size to assess the significance and magnitude of our results.

4.7 Conclusions and Future Work

In this paper, we have used agglomerative hierarchical clustering to learn a *linkage tree model* that captures promising patterns of HTTP requests in automatically generated system-level test cases. We proposed a novel algorithm, called *LT-MOSA*, that extends state-of-the-art approaches by tailoring and incorporating *linkage learning* within its genetic operators. Linkage learning helps to preserve and replicate patterns of API requests that depend on each other.

We implemented *LT-MOSA*, in *EvoMaster* and evaluated it on seven web applications from the EMB benchmark. Our results show that *LT-MOSA* significantly improves code coverage and can detect more faults than two state-of-the-art approaches in REST API testing, namely *MIO* [140] and *MOSA* [120]. This suggests that using unsupervised machine learning (and agglomerative hierarchical clustering in our case) is a very promising research direction.

Based on our promising results, there are multiple potential directions for future works. In this paper, we used the UPGMA algorithm for hierarchical clustering. Therefore, we intend to investigate more learning algorithms within the hierarchical clustering category. We also plan to investigate other categories of machine learning methods alternative to hierarchical clustering, such as Bayesian Network [153]. Finally, *LT-MOSA* uses a fixed parameter *K* for the linkage learning frequency. We plan to investigate alternative, more adaptive mechanisms to decide whether the linkage tree model needs to be retrained or not. Finally, we intend to implement and apply linkage learning to unit-test case generation as well.

5
Hybrid Multi-level Crossover for Unit Test Case Generation

State-of-the-art search-based approaches for test case generation work at test case level, where tests are represented as sequences of statements. These approaches make use of genetic operators (i.e., mutation and crossover) that create test variants by adding, altering, and removing statements from existing tests. While this encoding schema has been shown to be very effective for many-objective test case generation, the standard crossover operator (single-point) only alters the structure of the test cases but not the input data. In this paper, we argue that changing both the test case structure and the input data is necessary to increase the genetic variation and improve the search process. Hence, we propose a hybrid multi-level crossover (HMX) operator that combines the traditional test-level crossover with data-level recombination. The former evolves and alters the test case structures, while the latter evolves the input data using numeric and string-based recombinational operators. We evaluate our new crossover operator by performing an empirical study on more than 100 classes selected from open-source Java libraries for numerical operations and string manipulation. We compare HMX with the single-point crossover that is used in EvoSuite w.r.t. structural coverage and fault detection capability. Our results show that HMX achieves a statistically significant increase in 30 % *of the classes up to* 19 % *in structural coverage compared to the single-point crossover. Moreover, the fault detection capability improved up to* 12 % *measured using strong mutation score.*

This chapter has been published as *Mitchell Olsthoorn, Pouria Derakhshanfar, and Annibale Panichella. Hybrid Multi-level Crossover for Unit Test Case Generation. 13th International Symposium on Search-Based Software Engineering 2021 (SSBSE'21)* [78].

5.1 Introduction

Genetic operators are a fundamental component of evolutionary search-based test case generation algorithms. These operators create variation in the test cases to help the search process explore new possible paths. The main genetic operators are mutation, which makes changes to a single test case, and crossover, which exchanges information between two test cases.

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 [4]. The test case-level crossover exchanges statements between two parent test cases [19]. Lastly, the test suite-level crossover swaps test cases within two test suites [34]. Recent studies have shown that the test case-level schema combined with many-objective (MO) search is the most effective at generating test cases with high coverage [58, 74].

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 [19]. 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.

In this paper, we argue that better genetic variation can be obtained by designing a crossover operator that alters the structure of the test cases and also the input data by creating new data that is in the neighborhood of the parents' data. To validate this hypothesis, we propose a new operator, called Hybrid Multi-level Crossover (*HMX*), that combines different crossover operators on multiple levels. We implement *HMX* within *EvoSuite* [34], the state-of-the-art unit-test generation tool for Java.

To evaluate the effectiveness of our operator, we performed an empirical study where we compare *HMX* with the single-point crossover used in *EvoSuite*, a state-of-the-art test case generation tool for Java, *w.r.t.* structural coverage and fault detection capability. To this aim, we build a benchmark with 116 classes from the Apache Commons and Lucene Stemmer projects, which include classes for numerical operations and string manipulation.

Our results show 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 our baseline, respectively (with a max improvement of 19.1 % and 19.4 %). Additionally, the proposed operator 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.

In summary, we make the following contributions:

1. A novel crossover that works at both test case and input data-level to increase genetic variation in the search process. The data-level recombination combines multiple different techniques depending on the data type.

2. An open-source implementation of our operator in *EvoSuite*.

3. A full replication package containing the results and the analysis scripts [87].

The outline for the remainder of this paper is as follows. Section 5.2 explains the fundamental concepts used in the paper. Section 5.3 introduces our new crossover operator, called *HMX*, and breaks down how it works. Section 5.4 sets out our research questions and describes the setup of our empirical study. Section 5.5 details our results and highlights our findings. Section 5.6 discusses the threats to validity and Section 5.7 draws conclusions and identifies possible directions for future work.

5.2 Background and Related Work

5.2.1 Search-Based Unit Test Generation

Prior studies introduced search-based software test generation (SBST) approaches utilizing meta-heuristics (*e.g.,* genetic algorithm) to automate test generation for different testing levels [4], such as unit [34], integration [36], and system-level testing [37]. Search-based unit-test generation is one of the widely studied topics in this field, where a search process generates tests fulfilling various criteria (*e.g.,* structural coverage, mutation score) for a given class under test (CUT). Studies have shown that these techniques are effective at achieving high code coverage [74, 101] and fault detection [9].

5.2.2 Single-Objective Unit Test Generation

Single-objective techniques specify one or more fitness functions to guide the search process towards covering the search targets according to the desired criteria. Rojas *et al.* [100] proposed an approach that aggregates all of the fitness functions for each criterion using a weighted sum scalarization and performs a single-objective optimization to generate tests. Additionally, Gay [168] empirically showed that combining different criteria in a single-objective leads to detect more faults compared to using each criterion separately.

5.2.3 Dynamic Many-Objective Sorting Algorithm (*DynaMOSA*)

In contrast with single-objective unit test generation, Panichella *et al.* have proposed a many-objective evolutionary-based approach, called *DynaMOSA* [58]. This approach considers each coverage targets from multiple criteria as an independent search objective. *DynaMOSA* utilizes the hierarchy of dependencies between different coverage targets (*e.g.,* line, branch, mutants) to select the search objectives during the search dynamically. Moreover, recent work [102] introduced a multi-criteria variant of *DynaMOSA* that extends the idea of dynamic selection of the targets, based on an enhanced hierarchical dependency analysis. This recent study showed that this multi-criteria variant outperforms single-objective search-based unit test generation *w.r.t.* structural and mutation coverage and, therefore, can achieve a higher fault detection rate. These results have also been confirmed independently by Campos *et al.* [74]. Consequently, *DynaMOSA* is currently used as the default algorithm in *EvoSuite*.

5.2.4 Crossover Operator

Like any other evolutionary-based algorithms, all variations of *DynaMOSA* need crossover and mutation operators for evolving the individuals in the current population to generate the next population. Since *DynaMOSA* encodes tests at a test case-level, the mutation operator alters statements in a selected test case according to a given *mutation probability.*

This search algorithm uses the single-point crossover to recombine two selected individuals (parents) into new tests (offspring) for the next generation. This crossover operator randomly selects two positions in the selected parents and split them into two parts. Then, it remerges each part with the opposing part from the other parent. A more detailed explanation of this operator is available in Section 5.3.

While the single-point crossover brings diversity to the structure of the generated test cases, it does not work at the data-level (*i.e.,*crossover between the test inputs). Hence, this study introduces a hybrid multi-level crossover, called *HMX*, for the state-of-the-art in search-based unit test generation.

5.3 Approach

This section details our new crossover operator, called Hybrid Multi-level Crossover (*HMX*). This operator combines the traditional *single-point* test case-level crossover with multiple data-level crossovers.

Alg. 4 outlines the pseudo-code of our crossover operator. *HMX* first performs the traditional *single-point* crossover at line 2. The *single-point* crossover is chosen for the test case-level operator as previous studies have shown that it is effective in producing a variation in the population over time [19]. It is also the default crossover operator used in the state-of-the-art test case generation tool *EvoSuite* [19]. This operator takes two parent test cases as input and selects a random point among the statements within the parents test cases. The parents are then split at this point, and their resulting parts are then recombined with its opposing part of the other parent to produce two new offspring test cases. Since these offspring test cases use a random crossover point, they might contain incomplete sequences of statements (*e.g.,* missing variable definition) and, therefore, will not compile. To make the crossover more effective, these broken references are fixed by introducing new random variable definitions that match the type of the broken reference [34]. Lines 3-22 contain the selection logic of the data-level crossover. Unlike the test case-level crossover, the data-level crossover can not be applied to every combination of input data. Performing the crossover on input data with different types (*e.g.,* strings and numbers) would not produce any meaningful output as there is no logical way to combine these dissimilar types. Furthermore, we should not perform a crossover on two identical data types from different methods. If the data-level crossover would be applied to parameters of the same type that belong to different methods, it could produce offspring that are farther from the desired objective than the original. Hence, the algorithm has to select which combinations of input data are compatible. *HMX* achieves this by selecting compatible functions (*i.e.,* constructors and methods calls) and applying the crossover pairwise to the function's parameters.

In lines 3-6, two pairs of maps are created that store the compatible functions for each parent for both constructors and methods. Each map stores a list of functions that share the same signature; The signature is the key of the map, and the functions are the values. The signature of the function is a string derived from the class name, function name, parameters types, and return type using the following format:

```
CLASS_NAME|FUNCTION_NAME(PARAM1_TYPE, PARAM2_TYPE, …)RETURN_TYPE
```

Algorithm 4: *HMX*: hybrid multi-level crossover

```
Input: Two parent test cases $P_1$ and $P_2$
Output: Two offspring test cases $O_1$ and $O_2$
begin
    $O_1, O_2 \leftarrow$ SINGLE-POINT-CROSSOVER($P_1$, $P_2$)
    // Constructor data store
    $C_1 \leftarrow$ Map<signature, constructor[ ]> // For $P_1$
    $C_2 \leftarrow$ Map<signature, constructor[ ]> // For $P_2$
    // Method data store
    $M_1 \leftarrow$ Map<signature, method[ ]> // For $P_1$
    $M_2 \leftarrow$ Map<signature, method[ ]> // For $P_2$
    forall ($S_1$, $S_2$), in $S_1 \in O_1$ and $S_2 \in O_2$ do
        if SIGNATURE($S_1$) == SIGNATURE($S_2$) then
            if $S_1$ is constructor then
                $C_1$[SIGNATURE($S_1$)].add($S_1$)
                $C_2$[SIGNATURE($S_2$)].add($S_2$)
            else if $S_1$ is method then
                $M_1$[SIGNATURE($S_1$)].add($S_1$)
                $M_2$[SIGNATURE($S_2$)].add($S_2$)
    foreach $SIG \in C_1.keys \cup C_2.keys$ do
        // choose random constructor with same signature
        $S_1 \leftarrow random.choice(C1[SIG])$
        $S_2 \leftarrow random.choice(C2[SIG])$
        $O_1, O_2 \leftarrow$ DATA-CROSSOVER($O_1$, $O_2$, PARAMS($S_1$), PARAMS($S_2$))
    foreach $SIG \in M_1.keys \cup M_2.keys$ do
        // choose random method with same signature
        $S_1 \leftarrow random.choice(M1[SIG])$
        $S_2 \leftarrow random.choice(M2[SIG])$
        $O_1, O_2 \leftarrow$ DATA-CROSSOVER($O_1$, $O_2$, PARAMS($S_1$), PARAMS($S_2$))
    return $O_1$, $O_2$
```

In lines 7-14, *HMX* loops over all combinations of statements S_1 and S_2 in the offspring produced by the single-point crossover. For each combination, it checks if the signatures of the two functions match (line 8). If both statements are either constructors or methods, they are stored in their corresponding map with the signature as a key in lines 10-11 and 13-14, respectively. Note that if the test case contains constructor or method calls for other classes than the CUT, these are also considered by the selection of compatible functions. For example, additional objects (*e.g.*, strings, lists) might be needed as an input argument to one of the CUT's functions.

When all possible matching functions have been found, the operator loops through the signatures of the two function types separately in lines 15-18 and 19-22. For each signature, *HMX* selects a random function instance matching the signature from each parent. The operator then performs the data-level crossover on the parameters of these two randomly selected functions in lines 18 and 22. For each signature in the map, *HMX* only selects one function instance per parent to proceed with the genetic recombination.

The data-level recombination pairwise traverses the parameters of the two compatible functions selected in lines 16-17 (for constructors) and 20-21 (for methods). For each pair of parameters, Alg. 4 checks their types and determines if they are numbers or strings, the two supported types of *HMX*. If the two parameters are numbers (*i.e.,* byte, short, int, long, float, double, boolean, and char), the operator applies the *Simulated Binary Crossover* (SBX), which is described in Section 5.3.1. If the parameters are strings, it applies the string crossover described in Section 5.3.2. Lastly, in line 23, *HMX* returns the produced offspring.

```java
@Test
public void test1() {
  Fraction f0 = new Fraction(2, 3);
  Fraction f1 = new Fraction(2, -1);
  f0.divideBy(f1);
  f0.add(Fraction.ZERO);
}
```

Example 5.1: Parent 1

```java
@Test
public void test2() {
  Fraction f0 = new Fraction(3, 1);
  Fraction f1 = new Fraction(1, 3);
  f0.add(f1);
  f0.pow(2.0);
}
```

Example 5.2: Parent 2

To provide a practical example, let us consider the two parent test cases in Examples 5.1 and 5.2. Both parent 1 and parent 2 contain two invocations of the `Fraction` constructor. Since these constructors share the same signature: `Fraction|<init>(int, int)Fraction`; they are compatible. Similarly, the method add of the `Fraction` class is present in both parents, with the same signature: `Fraction|add(Fraction)V`; and are compatible, as well. In contrast, for example, method `divideBy`, in parent 1, and method add, in parent 2, are not compatible since their signatures are different.

5.3.1 Simulated Binary Crossover

The *Simulated Binary Crossover* (SBX) is a recombination operator commonly used in numerical problems with numerical decision variables and fixed-length chromosomes. It has been shown that Evolutionary Algorithms (EAs) that use this crossover operator produce better results compared to traditional numerical crossover operators [169]. The equation below outlines the algorithm of *SBX*:

$$u = rand_u \tag{5.1}$$

$$\beta = \begin{cases} 2 \cdot u^{1/(\eta_c+1)} & \text{if } u < 0.5 \\ 1 & \text{if } u = 0.5 \\ \frac{0.5}{1.0-u}^{1/(\eta_c+1)} & \text{if } u > 0.5 \end{cases} \tag{5.2}$$

$$b = rand_b \tag{5.3}$$

$$v = \begin{cases} ((v_1 - v_2) \cdot 0.5) - (\beta \cdot 0.5 \cdot |v_1 - v_2|) & \text{if } b = true \\ ((v_1 - v_2) \cdot 0.5) + (\beta \cdot 0.5 \cdot |v_1 - v_2|) & \text{if } b = false \end{cases} \tag{5.4}$$

where v (Equation (5.4)) is the new value of parameter v_1, v_1 is the original value of the parameter, and v_2 is the value of the opposing parameter (the corresponding parameter from the matched function). η_c is the *distribution index* and it measures how close the new values should be to original values (proximity). For *HMX*, this variable is set to 2.5 as this is within the recommended range [2;5] [169]. *SBX* first creates a random *uniform* variable u (Equation (5.1)), which is used to select one of three strategies for β. This scaling variable β (Equation (5.2)), is used to scale an offset. This offset is either subtracted or added depending on the random *boolean* variable b. In general, *SBX* generates new values centered around the original parents, either in between the parents' values (contracting) or outside this range (expending) depending on the value of u. The algorithm is performed on both matching parameters, and the resulting new values are used as a replacement of the original values.

As an example, consider the two compatible constructors `Fraction(2,3)` (line 3 in Example 5.1) and `Fraction(1,3)` (line 4 in Example 5.2). The *SBX* recombination operator is applied for the following pairwise combinations: (2, 1) and (3, 3). To calculate the new value of the first element of the first pair, $v_1 = 2$ and $v_2 = 1$. Similarly, the second element can be calculated by switching the values of v_1 and v_2. The same procedure can be applied to calculate the new values of the second pair.

5.3.2 String Crossover

The single-point string crossover is used to exchange information between two string parameters of matching functions [4]. By recombining parts of each string, it makes it possible for promising substrings to collect together. The operator achieves this by picking two random numbers, $0 \leq x_i < \text{length}(x)$ and $0 \leq y_i < \text{length}(y)$ for both strings, respectively. It then recombines the two strings by concatenating the substrings in the following way: $x = x[: x_i] \,||\, y[y_i :]$ and $y = y[: y_i] \,||\, x[x_i :]$.

For example, given the following string $x = "lorem"$ and $y = "ipsum"$ and the random variables $x_i = 1$ and $y_i = 3$, the new values will be: $x = "lom"$ and $y = "ipsurem"$.

5.4 Empirical Study

To assess the impact of *HMX* on search-based unit test generation, we perform an empirical evaluation to answer the following research questions:

Table 5.1: Projects in our empirical study. # indicates the number of CUTs. `cc` indicates the cyclomatic complexity of CUTs. σ indicates the standard deviation. `min` and `max` indicate the minimum and maximum value of the metric, respectively. Also, $\overline{\text{str-par}}$ and $\overline{\text{nr-par}}$ are the average number of string and number input parameters for the selected CUTs.

Project	**#**	**CCN**				**String parameter**				**Number parameter**			
		$\overline{\text{cc}}$	σ	min	max	$\overline{\text{str-par}}$	σ	min	max	$\overline{\text{nr-par}}$	σ	min	max
CLI	4	1.7	0.9	3.0	1.1	14.5	14.2	34.0	4.0	8.5	13.7	29.0	1.0
Geometry	13	1.8	0.4	2.5	1.2	3.4	5.5	21.0	1.0	10.2	6.7	21.0	1.0
Lang	34	3.0	1.6	7.4	1.1	17.4	36.7	209.0	1.0	26.6	48.3	249.0	1.0
Logging	1	3.0	-	3.0	3.0	6.0	-	6.0	6.0	3.0	-	3.0	3.0
Math	27	2.9	1.6	7.7	1.1	2.5	1.8	9.0	1.0	10.0	10.5	45.0	1.0
Numbers	5	2.8	1.1	4.5	1.6	1.4	0.9	3.0	1.0	31.6	33.5	89.0	4.0
RNG	4	3.3	1.4	5.0	1.7	2.2	2.5	6.0	1.0	2.0	1.4	4.0	1.0
Stemmer	16	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

RQ1 *To what extent does HMX improve structural coverage compared to the single-point crossover?*

RQ2 *How does HMX impact the fault-detection capability of the generated tests?*

5.4.1 Benchmark

For this study, we selected the CUTs from the Apache Commons and Snowball Stemmer libraries. The former is a commonly-used project containing reusable Java components for several applications [1]. The latter is a well-known library for stemming strings, which is part of the Apache Lucene [2]. As described in Section 5.3, *HMX* brings more advantages for search-based test generation in projects that utilize strings and numbers. Hence, to show the effect of this new crossover operator, we selected 100 classes from 9 components in Apache Commons that have numeric and string input data: (i) Math a library of lightweight, self-contained mathematics and statistics components; (ii) Numbers includes utilities for working with complex numbers; (iii) Geometry provides utilities for geometric processing; (iv) RNG a library of Java implementations of pseudo-random generators; (v) Statistics a project containing tools for statistics; (vi) CLI an API processing and validating a command line interface; (vii) Text a library focused on algorithms working on strings; (viii) Lang contains extra functionality for classes in `java.lang`; and (ix) Logging an adapter allowing configurable bridging to other logging systems.

In addition, we added the main 16 classes in Snowball Stemmer to the benchmark, as these focus on string manipulation and were previously used in former search-based unit test generation studies [120].

Due to the large number of classes in the selected Apache Commons components, we used CK [170], a tool that calculates the method-level and class-level code metrics in Java projects using static analysis. We collect the Cyclomatic Complexity (CC) and type of input parameters for each method in the selected 9 components. Using the collected information, we filter out the classes that do not have methods accepting strings or numbers

[1] `https://commons.apache.org`

[2] `https://github.com/weavejester/snowball-stemmer`

(integer, double, long, or float) as input parameters. Then, we sort the remaining classes according to their average CC and pick the top 100 cases for our benchmark. Table 5.1 reports CC, number of string, and number arguments for each project used in this study. By doing a preliminary run of *EvoSuite* on the 116 selected classes, we noticed that this tool fails to start the search process in 9 of the CUTs. These failures stem from an issue in the underlying test generation tool *EvoSuite*. The tool fails to gather a critical statistic (*i.e.,* TOTAL_GOALS) for these runs in both the baseline and *HMX*. We also encountered 4 classes that did not produce any coverage for both the baseline and our approach. Consequently, we filtered out these classes from the experiment and performed the final evaluation on 103 remaining classes.

5.4.2 Implementation

We implemented *HMX* in *EvoSuite* [34], which is the state-of-the-art tool for search-based unit test generation in Java. By default, this tool uses the single-point crossover for test generation. We have defined a new parameter *multi_level_crossover* to enable *HMX*. Our Implementation is openly available as an artifact [87].

5.4.3 Preliminary Study

We performed a preliminary study to see how the probability of applying our data-level crossover influences the result. The single-point test case-level crossover is applied with a predefined probability. We experimented with how often the data-level crossover should be applied whenever the test case-level crossover was applied. From the probabilities we tried (*i.e.,* 0.25, 0.50, 0.75, 1.00), we found out that always applying the data-level crossover when the test case-level crossover produced the best results according to statistical analysis.

5.4.4 Parameter Settings

We run each search process with *EvoSuite*'s default parameter values. As confirmed by prior studies [109], despite the impact of parameter tuning on the search performance, the default parameters provide acceptable results. Hence, we run each search process with a two-minute search budget and set the population size to 50 individuals. Moreover, we use mutation with a probability of $1/n$ (n = length of the generated test). For both crossover operators that we used in this study (single-point crossover for the baseline and our novel *HMX*), the crossover probability is 0.75. For the Simulated Binary Crossover (SBX), we used the *distribution index* $\eta_c = 2.5$ [169]. The search algorithm is the multi-criteria DynaMOSA [102], which is the default one in *EvoSuite* `v1.1.0`.

5.4.5 Experimental Protocol

We apply both default *EvoSuite* with single-point crossover and *EvoSuite* + *HMX* to each of the selected CUTs in the benchmark. To address the random nature of search-based test generation tools, we repeat each execution 100 times, with a different random seed, for a total number of 23 200 independent executions. We run our evaluation on a system with an AMD EPYC™ 7H12 using 240 cores running at 2.6 GHz. With each execution taking 5 minutes on average (*i.e.,* search, minimalization, and assertion generation), the total running time is 80.6 days of sequential execution.

For our analysis, we report the average (median) results across the 100 repeated runs. To determine if the results (*i.e.,* structural code coverage and fault detection capability) of the two crossover operator are statistically significant, we use the unpaired Wilcoxon rank-sum test [67] with a threshold of 0.05. The Wilcoxon test is a non-parametric statistical test that determines if two data distributions are significantly different. Additionally, we use the Vargha-Delaney statistic [68] to measure the magnitude of the result, which determines how large the difference between the two operators is.

5.5 Results

This section discusses the results of our study with the aim of answering the research questions formulated in Section 5.4. All differences in results in this section are presented in absolute differences (percentage points).

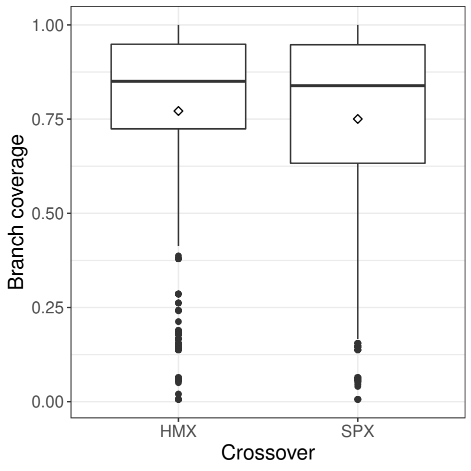
5.5.1 Result for RQ1: Structural Coverage

Figure 5.1 shows the structural coverage achieved by our approach, *HMX*, compared to the baseline, SPX, on the benchmark. In particular, Figure 5.1a shows branch coverage and Figure 5.1b shows line coverage. The boxplots show the median, quartiles, variability in the results, and the outliers for all classes together. The diamond point indicates the mean of the results.

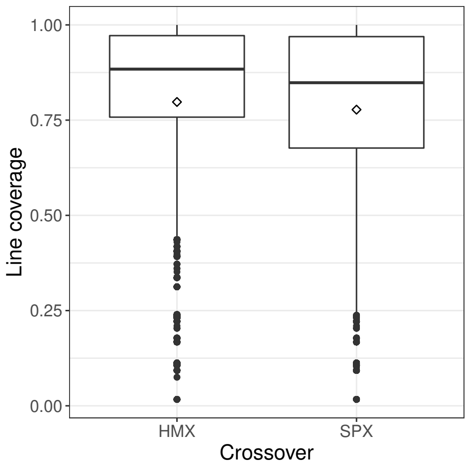
Figure 5.1a and Figure 5.1b show that, on average, *HMX* has higher 1^{st} quartile, median, mean, and 3^{rd} quartile values than the baseline, SPX, for both test metrics. On average, *HMX* improves the branch coverage by +2.0 % and the line coverage by +1.9 %. The largest differences are visible for the lower whisker and for the first quartile (25th percentile). In particular, the differences for the lower whisker are around +20% branch and line coverage when using *HMX*; the improvements in the first quartile are around +10% and +8% for branch and line coverage, respectively. These results indicate that *HMX* improves both line and branch coverage for some of the CUTs in our benchmark. Finally, as we can see in both of the plots in Figure 5.1, the variation in the results for *HMX*, measured by the Interquartile Range (IQR), is smaller than for SPX. This observation shows that *HMX* helps *EvoSuite* to generate tests with a more stable structural coverage.

Table 5.2 shows the results of the statistical comparison between *HMX* and the baseline, SPX, based on a p-value ≤ 0.05. *#Win* indicates the number of times that *HMX* has a statistically significant improvement over SPX. *#Equal* indicates the number of times that there is no statistical difference in the results between the two operators; *#Lose* indicates the number of times that *HMX* has statistically worse results than SPX. The *#Win* and *#Lose* columns also include the magnitude of the difference through the $\hat{A}_{12}$ effect size, classified in *Small*, *Medium*, *Large*, and *Negligible*.

From Table 5.2, we can see that *HMX* has a statistically significant non-negligible improvement in 30 and 23 classes for branch and line coverage, respectively. For the branch coverage metric, *HMX* improves with a large magnitude for 22 classes, medium for 3 classes, and small for 5 classes. For line coverage, *HMX* improves with a large magnitude for 19 classes, medium for 3 classes, and small for 1 class. *HMX* only loses in one case in comparison to the baseline for both branch and line coverage: `StrSubstitutor` from the `Lang` project. However, in this case, the effect size is small (magnitude).



(a) Branch Coverage



(b) Line Coverage

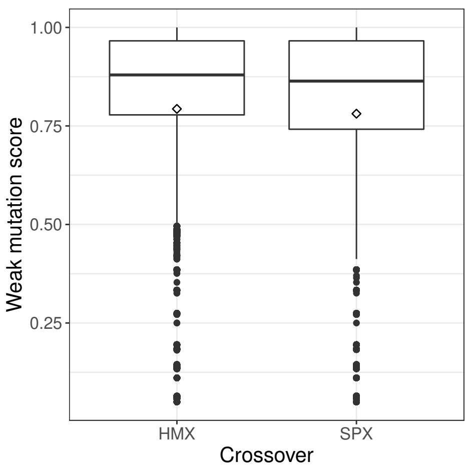
Figure 5.1: Boxplot of structural coverage comparing *HMX* to the baseline SPX. The diamond point indicates the mean coverage of the benchmark.

Table 5.2: Statistical results of *HMX* vs. SPX on structural coverage. #Win indicates the number of times that *HMX* is statistically better than SPX. #Lose indicates the opposite. #No diff. indicates that there is no statistical difference. Negl., Small, Medium, and Large denote the $\hat{A}_{12}$ effect size.

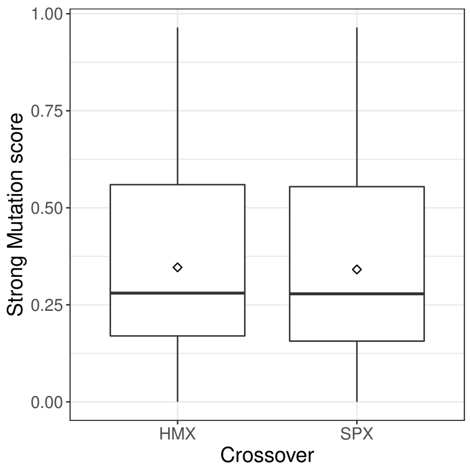
Metric	#Win				#Lose				#No diff.
	Negl.	Small	Medium	Large	Negl.	Small	Medium	Large	
Branch	2	5	3	22	0	1	0	0	70
Line	3	1	3	19	0	1	0	0	76

For branch coverage, we observe a maximum increase in coverage of +19.1 % for the *finnishStemmer* class from the `Stemmer` project. For line coverage, the class with the maximum increase in coverage is `hungarianStemmer` (also from `Stemmer`) with an average improvement of +19.4 %. Compared to the baseline, all classes in the Snowball Stemmer string manipulation library improve based on branch and line coverage with an average improvement of +11.4 % and +11.0 %, respectively. For the Apache Commons library, *HMX* significantly improves the branch and line coverage in 16 (9 string-related and 7 number-related) and 10 (6 string-related and 4 number-related) classes, respectively.

In summary, the proposed *HMX* crossover operator achieves significantly higher (~30 % of the cases) or equal structural code coverage for unit test case generation compared to the baseline SPX.



(a) Weak Mutation Score



(b) Strong Mutation Score

Figure 5.2: Boxplot of structural coverage comparing *HMX* to the baseline SPX.

Table 5.3: Statistical results of *HMX* vs. SPX for fault-detection capability.

Metric	#Win				#Lose				#No diff.
	Negl.	Small	Medium	Large	Negl.	Small	Medium	Large	
Weak mutation	3	3	3	21	0	1	0	0	72
Strong mutation	0	8	0	15	0	3	0	0	77

5.5.2 Result for RQ2: Fault Detection Capability

Figure 5.2 shows the fault detection capability of *HMX* compared to SPX measured through the mutation score. Figure 5.2a shows the weak mutation score and Figure 5.2b shows the strong mutation score. The boxplots show the median, quartiles, variability in the results, and the outliers for all classes in the benchmark together. The diamond point indicates the mean of the results. From Figure 5.2a, we can see that, on average, *HMX* improves the weak mutation score by +1.2 % compared to SPX. However, from Figure 5.2b we can see that overall, the strong mutation scores only show marginal improvements (+0.5 %).

Table 5.3 shows the statistical comparison between *HMX* and SPX, based on a p-value ≤ 0.05. Similarly to Table 5.2, *#Win* indicates the number of times that *HMX* has a statistically significant improvement over SPX, *#Equal* indicates the number of times that there is no statistical difference in the results of the two operators, and *#Lose* indicates the number of times that *HMX* has statistically worse results than SPX. The *#Win* and *#Lose* columns additionally also indicate the magnitude of the difference through the $\hat{A}_{12}$ effect size. From Table 5.3, we can see that *HMX* has a statistically significant non-negligible improvement in 27 and 23 cases for weak and strong mutation, respectively. For weak mutation, *HMX* improves with a large magnitude for 21 classes, medium for 3 classes, and small for 3 classes. For strong mutation, *HMX* improves with a large magnitude for 15 classes and a small magnitude for 8 classes. *HMX* performed worse in one case (`Fraction` from the `Lang` project) for weak mutation and three cases (`AdaptiveStepsizeFieldIntegrator` and `MultistepIntegrator` from the `Math` project, and `SphericalCoordinates` from the `Geometry` project) for strong mutation, all with a small effect size.

We observe a maximum increase in weak mutation score of +14.0 % for the `hungarianStemmer` class (`Stemmer`) and +12.2 % for the `ExtendedMessageFormat` class (`Text`) on strong mutation score. Among the classes that improve on weak and strong mutation score, 27 and 20, respectively, also improve *w.r.t.* branch coverage. Interestingly, four classes among both mutation scores improve *w.r.t.* mutation score without improving the structural coverage.

In summary, *HMX* achieves significantly higher (~25 % of the cases) or equal fault detection capability compared to SPX and is outperformed in one and three classes for weak and strong mutation, respectively.

5.6 Threats to Validity

This section discusses the potential threats to the validity of our study.

5.6.1 Construct Validity

Threats to *construct validity* stem from how well the chosen evaluation metrics measure the intended purpose of the study. Our study relies on well-established evaluation metrics in software testing to compare the proposed hybrid multi-level crossover with the current state-of-the-art, namely structural coverage (*i.e.,* branch and line) and fault detection capability (*i.e.,* weak and strong mutation). As the stopping condition of the search process, we used a time-based budget rather than a budget based on the number of test evaluations or generations. A time-based budget provides a fairer measure since the two crossover operators have a different overhead and execution time and might otherwise provide an unfair advantage to our operator.

5.6.2 Internal Validity

Threats to *internal validity* stem from the influence of other factors onto our results. The only difference between the two approaches in our study is the crossover operator. Therefore, any improvement or diminishment in the results must be attributed to the difference in the two crossover operators.

5.6.3 External Validity

Threats to *external validity* stem from the generalizability of our study. We selected 116 classes from popular open-source projects based on their cyclomatic complexity and type of input parameters to create a representative benchmark. These classes have previously been used in the related literature on test case generation [58, 120].

5.6.4 Conclusion Validity

Threats to *conclusion validity* stem from the deduction of the conclusion from the results. To minimize the risk of the randomized nature of EAs, we performed 100 iterations of the experiment in our study with different random seeds. We have followed the recommended guidelines for running empirical experiments with randomized algorithms using sound statistical analysis as recommend in the literature [66]. We used the unpaired Wilcoxon rank-sum test and the Vargha-Delaney $\hat{A}_{12}$ effect size to determine the significance and magnitude of our results.

5.7 Conclusions and Future Work

In this paper, we have proposed a novel crossover operator, called *HMX*, that combines different crossover operators on both a test case-level and a data-level for generating unit-level test cases. By implementing such a hybrid multi-level crossover operator, we can create genetic variation in not only the test statements but also the test data. We implemented *HMX* in *EvoSuite*, a state-of-the-art Java unit test case generation tool. Our approach was evaluated on a benchmark of 116 classes from two popular open-source projects. The results show that *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). Based on these promising results, there are multiple potential directions for future work to explore. In this paper, we detailed the crossover operator for two types of primitive test data inputs (*i.e.,* numbers and strings). In future work, we are planning to extend this with additional operators for arrays, lists, and maps. Additionally, we want to experiment with alternative crossover operators for numbers (*e.g.,* parent-centric crossover, arithmetic crossover) and strings (*e.g.,* multi-point crossover).

Acknowledgements

We gratefully acknowledges the Horizon 2020 (EU Commission) support for the project *COSMOS*, Project No. 957254-COSMOS.

6

Guess What: Test Case Generation for Javascript with Unsupervised Probabilistic Type Inference

Search-based test case generation approaches make use of static type information to determine which data types should be used for the creation of new test cases. Dynamically typed languages like JavaScript, however, do not have this type information. In this paper, we propose an unsupervised probabilistic type inference approach to infer data types within the test case generation process. We evaluated the proposed approach on a benchmark of 98 *units under test (i.e., exported classes and functions) compared to random type sampling w.r.t. branch coverage. Our results show that our type inference approach achieves a statistically significant increase in* 56 % *of the test files with up to* 71 % *of branch coverage compared to the baseline.*

This chapter has been published as *Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Guess What: Test Case Generation for Javascript with Unsupervised Probabilistic Type Inference. 13th International Symposium on Search-Based Software Engineering 2022 (SSBSE'22)* [79].

6.1 Introduction

Over the last few decades, researchers have developed various techniques for automating test case generation [4]. In particular, search-based approaches have been shown to (1) achieve higher code coverage [108] and (2) have fewer smells [52] compared to manually-written test cases, and (3) detect unknown bugs [8–10]. Furthermore, generated tests significantly reduce the time needed for testing and debugging [5], and have been successfully used in industry (*e.g.,* [11–13]).

These approaches make use of static type information to (1) generate primitives and objects to pass to constructors and function calls, and (2) determine which branch distance function to use. Without this type information, the test case generation process has to randomly guess which types are compatible with the parameter specification of the constructor or function call and would not have guidance to solve the binary flag problem. This greatly increases the search space and, therefore, makes the overall process less effective and efficient. Consequently, most of the work in this research area has focused on statically-typed programming languages like Java (*e.g., EvoSuite* [34]) and C (*e.g.,* AUSTIN [59]).

Dynamically-typed programming languages introduce new challenges for unit-level test case generation. As reported by Lukasczyk *et al.* [60], state-of-the-art approaches used for statically-typed languages do not perform well on *Python* programs when type information is not available. According to the survey from Stack Overflow[1], *Python* and *JavaScript* are the two most commonly-used programming languages. Both languages are dynamically-typed, strengthening the importance of addressing these open challenges with the goal of increasing the adoption of test case generation tools in general.

In this paper, we focus on test case generation for *JavaScript* as, to the best of our knowledge, this is a research gap in the literature. In building our research, we build on top of the reported experience by Lukasczyk *et al.* [171] for *Python* programs. They addressed the input type challenge by incorporating Type4Py [172] —a deep neural network (DNN)— into the search process.

We propose a novel approach that incorporates unsupervised probabilistic type inference into the search-based test case generation process to infer the type information needed. An unsupervised type inference approach has two benefits compared to a DNN: (1) it does not require a labeled dataset with extensive training time, and (2) the model is explainable (*i.e.,* the decision can be traced back to a rule set). We build a prototype tool which implements the state-of-the-art many-objective search algorithm, *DynaMOSA*, and the probabilistic type inference model for *JavaScript*. We investigate two different strategies for incorporating the probabilistic model into the main loop of *DynaMOSA*, namely *proportional sampling* and *ranking*.

To evaluate the performance of the proposed approach, we performed an empirical study that investigates the baseline performance of our prototype (*i.e.,* using random type sampling) and the impact of the unsupervised probabilistic type inference *w.r.t.* branch coverage. To this aim, we constructed a benchmark consisting of 98 Units under Test (*i.e.,* exported classes and functions) of five popular open-source *JavaScript* projects, namely `Commander.js`, `Express`, `Moment.js`, `Javascript Algorithms`, and `Lodash`.

[1]https://survey.stackoverflow.co/2022/#most-popular-technologies-language

Our results show that integrating unsupervised probabilistic type inference improves branch coverage compared to random type sampling. Both the *ranking* and *proportional sampling* strategies significantly increase the number of branches covered by our approach (+9.3 % and +12.6 %, respectively). Out of the two strategies, *proportional sampling* outperforms *ranking* in 20 cases and loses in 4. In summary, we make the following contributions:

1. An unsupervised probabilistic type inference approach for search-based unit-level test case generation of *JavaScript* programs.
2. A prototype tool for automatically generating *JavaScript* unit-level test cases that incorporates this approach.[2]
3. A Benchmark consisting of 98 units under test from five popular open-source *JavaScript* projects.
4. A full replication package containing the results and the analysis scripts [88].

6.2 Background and Related Work

This section explains the background concepts and discusses the related work.

6.2.1 Test Case Generation

Writing test cases is an expensive, tedious, yet necessary activity for software quality assurance. Hence, researchers have proposed various techniques to semi-automate this process since the 1970s [173]. These techniques include symbolic execution [174], random testing [175], and meta-heuristics [4] (*e.g.,* genetic algorithms). The latter category is often referred to as search-based software testing (SBST). SBST techniques have been successfully used in the literature to automate the creation of test cases for different testing levels [4], such as unit [34], integration [36], and system-level testing [37]. At unit-level, SBST techniques aim to generate test cases that optimize various test adequacy criteria, such as *e.g.,* structural coverage and mutation score. Many different meta-heuristic search algorithms have been proposed over the years (*e.g.,* whole suite [176], MIO [140], MOSA [120], or DynaMOSA [58]). Recent studies have shown that *DynaMOSA* is more effective and efficient than other genetic algorithms for unit test generation of *Java* [74] and *Python* [60] programs.

6.2.2 Type Inference

A recent study by Gao *et al.* [177] showed that the lack of static types within *JavaScript* leads to bugs that could have been easily identified with a static type system. To combat this problem, various approaches have been proposed to infer/predict types for generating type annotations or assertions. Anderson *et al.* [178] proposed a formal approach for inferring types using constraint solvers based on a custom *JavaScript*-like language. Chandra *et al.* [179] proposed a formal type inference approach for static compilation of *JavaScript* programs. These approaches, however, only support a subset of the *JavaScript* syntax and, therefore, will not work on all programs. JSNice [180] and DeepTyper [181]

[2]https://github.com/syntest-framework/syntest-javascript

are two other approaches that train a model based on training data and use it to predict future type information. These approaches have the shortcoming that they can only predict basic JavaScript types. Meaning that they are unable to predict/assert user-defined types. Additionally, these approaches cannot consider the context of the literals and objects within a program or function. Type4Py [172] is a similar approach that uses a Deep Neural Network (DNN) to infer types for *Python* projects and suffers from similar limitations.

6.2.3 Testing for JavaScript

JavaScript started out as a client-side programming language for the browser. Most work related to testing for *JavaScript* is, therefore, also focused on web applications within the browser (*e.g.,* [182–185]). Existing client-side testing approaches either focus on specific subsystems such as the browser's event handling system [183, 184] or the interaction of *JavaScript* with the *Document Object Model* of the browser [182]. Nowadays, *JavaScript* is also a very commonly-used language for back-end development on *Node.js*. Tanida *et al.* [185] proposed a symbolic execution approach that uses a constraint solver for input data generation. Other approaches focused on mutation testing [186] or contract-based testing [187]. However, to the best of our knowledge, there exists no approach for automatic unit-level test case generation for *JavaScript*.

6.3 Approach

This section details our test case generation approach for *JavaScript* programs that relies on Unsupervised Type Inference. Our approach consists of three phases, which are detailed in the next subsections.

6.3.1 Phase 1: Static Analysis

The first phase inspects the Subject Under Test (SUT) and its dependencies. First, this phase builds the Abstract Syntax Trees (ASTs) and extracts all identifiers and literals from the code; these will be referred to as *elements*. Afterward, the static analyzer extracts the relations between those elements and all user-defined objects, *i.e.,* classes, interfaces, or prototyped objects.

Elements

As mentioned before, the elements consist of *identifiers* and *literals*. The former are the named references to variables, functions, and properties. The latter are constant values assigned to variables; examples are strings, numbers, and booleans. The types of the literal are straightforward and do not require inference. However, the identifiers do not have explicit types in dynamically typed languages like *JavaScript*. Hence, their types need to be inferred based on the *contextual* information (or *relations*) of the extracted elements.

Relations

Relations correspond to operations performed on code elements and describe how these elements are used and relate to other elements, providing hints on their types. For example, let us consider the *assignment* relation $L = R$, where R (right-hand element) is a boolean

```
1 function example (a) {
2   if (a < 6) {
3     return 0
4     }
5   return a
6 }
7
```

(a) Example Code

1. $[L_R, example, a]$
2. $[L < R, a, 6]$
3. $[L \longrightarrow R, example, 0]$
4. $[L \longrightarrow R, example, a]$
5. $[L(R), example, 5]$

(b) Extracted Relations

Figure 6.1: Extracting relations from code

```
const x = (a == b ? 6 : 10)
```

(a) Nested Relation

1. $[L = R, x, y*]$
2. $y* = [C?L : R, z*, 6, 10]$
3. $z* = [L == R, a, b]$

(b) Extracted Relations

Figure 6.2: Extracting relations from nested code

literal; we can logically derive (or infer) that L (left-hand element) must also be a boolean variable.

These relations are extracted from the AST and are converted to a consistent format that allows for easy identification of the relation type. Let us assume that there is a *lower than* relation between variable *a* and literal *6*, as shown in Figure 6.1a on line 2. This relation is converted and recorded as $[L < R, a, 6]$, as shown in Figure 6.1b. In general, a relation is stored as a tuple containing (1) the type of operation ($L < R$ in our example) and (2) the list of operands (*i.e.,* *a* and *6* in our example). The full list of extracted relations for the code snippet in Figure 6.1a is reported in Figure 6.1b.

In total, we designed 75 possible relations based on the MDN web documentation by Mozilla [3]. These operations/relations are classified into 15 categories, namely (1) *primary*, (2) *left-hand side*, (3) *increment/decrement*, (4) *unary*, (5) *arithmetic*, (6) *relational*, (7) *equality*, (8) *bitwise shift*, (9) *binary bitwise*, (10) *binary logical*, (11) *ternary*, (12) *optional chaining*, (13) *assignment*, (14) *comma*, and (15) *function* expressions. The complete list of relations is available in our replication package.

Nested relations are special types of relations whose composing elements are relations themselves. As an example, let us consider the code snippet in Figure 6.2a. The corresponding relation for the assignment is $[L = R, x, y*]$, where y^* is an *artificial* element that points to the whole right-hand side of the assignment. This element corresponds to a ternary relation $[C?L : R, z*, 6, 10]$, which also includes an artificial element, called z^*, that points to the equality relation in the conditional part of the ternary statement. So z^* points to the final relation $[L == R, a, b]$. Although the code in Figure 6.2a seems rather simple, it

[3]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators

```
1    const x = 5
2
3    function example(a) {
4        const x = "Hello "
5        return x + a
6    }
7
```

Figure 6.3: Scopes

corresponds to three relations, two of which are nested, as shown in Figure 6.2b.

Scopes

A critical aspect of the elements we have not yet discussed is scoping. The scope of an identifier determines its accessibility. To better understand the importance of the scope, let us consider the example in Figure 6.3. First, the constant x is assigned the value 5. The constant x is defined in the so-called *global scope*. Next, a function is defined, creating a new scope. This scope has access to references of the global scope. Still, it can also have its own references, which are only available within its sub-scopes. In our example, another constant x is defined within the function scope. Note that from line 4, every reference to x in the scope of the function refers to the newly defined constant, not the x constant of the global scope. This type of operation is called *variable shadowing*. In a nutshell, variable shadowing is when the code contains an identifier for which there are multiple declarations in separate scopes. In these situations, the narrower scope *shadows* the other identifier declarations.

This shadowing principle is fundamental during the first phase of our approach because variables in the global scope are not the same variables as those in the function scope (*e.g.,* x in Figure 6.3). In fact, variables with the same identifier names but within different scopes can also have different types. In the example of Figure 6.3, x from the global scope is numerical, while the x from the function scope is a string. In conclusion, the relations include the involved elements together with their scope.

Complex Objects

In *JavaScript*, objects are the building blocks of the language and are stored as key-value pairs. Apart from primitive types like booleans and numbers, almost everything is represented as an object. An array, for example, is a special object where the keys are numbers. In recent *JavaScript* versions, developers can define classes and interfaces through a prototype-based object model, inducing a more object-oriented approach to *JavaScript*. Since these objects play such a prominent role in *JavaScript*, it is important that object types can be inferred as well. Hence, our approach extracts all object descriptions available in the program under test, including class, interface definitions, and standard objects (*e.g.,* functions).

6.3.2 Phase 2: Unsupervised Static Type Inference

The second phase builds a probabilistic type model for the elements extracted from the first phase. For literal elements, the type inference is straightforward as the type can be directly inferred from the literal type. However, for non-literal elements, our probabilistic model considers all *type hints* that can be inferred from the relations extracted in the previous phase.

For example, the assignment `x = 5` corresponds to the relation $[L = R, x, 5]$. From such a relation, we can derive that, at this particular point in the code, `x` must be numerical since it is assigned the literal value *5*. However, for statements like `x = y + z`, there are various possibilities for the type of `x` depending the on types of `y` and `z`. To illustrate, the + operator can be applied to both numbers (arithmetic sum) and strings (string concatenation). Besides, in *JavaScript*, it is also possible to concatenate numbers with strings. For example, `1 + “1”` returns the number 11. Therefore, multiple types can be assigned to elements that have relations/operations compatible with multiple data types.

To account for this, our model assigns scores to each type depending on the number of hints that can be derived for that type by its relations in the code. In general, given the element e and the set of relations $R = \{r_1, \dots r_n\}$ associated to e as extracted from a program P, our model assigns each type t a score equal to the number of relations that can be applied to t (*i.e.,* the number of hints):

$$score(e,t) = |hints(e,t)| \text{ where } hints(e,t) = \{r_i \in R : r_i \text{ applies to } t\} \tag{6.1}$$

Finally, the element e has a probability of being assigned the type t proportional to the number of hints received for t:

$$p(e,t) = \frac{score(e,t)}{\sum_{t_i} score(e,t_i)} \tag{6.2}$$

The higher the score of a particular type, the larger the probability that the element is of that type. The probabilities are later used to sample argument types in the search phase.

For example, let us consider the statement `x = y + z`, which can be applied to both strings and numbers. In this case, our probabilistic model would assign +1 hint for numbers and +1 hint for strings. Hence, both types will have an equal probability of 50 %.

Nested Types

The probability model takes into account both simple and nested relations. For example, let us consider the *JavaScript* statement: `c = a > b`. Such a statement corresponds to two relations (one of which is nested): $[L = R, c, d*]$ and $d* = [L > R, a, b]$. The outcome for $d* = [L > R, a, b]$ is boolean no matter the types of a and b. Therefore, we can infer the variable (or element) c should be as well. Hence, the hints and scores are obtained by considering all relations, including the nested ones.

Resolving Complex Objects

Complex objects are characterized by *property accessor* relations, *i.e.,* operations that aim to access properties of certain objects (*e.g.,* using the dot notation `object.property`). If an element is involved in one or more *property accessor* relations, the accessed properties are compared to the available object descriptions. If there is an overlap between the element’s properties and the properties of an object description, the object description receives +1 hint. In addition to matching object descriptions, an anonymous object type is created and assigned as a possible type. This anonymous object type exactly matches the properties of the element. This object is used when no other object matches are found.

6.3.3 Phase 3: Test Case Generation

The third phase generates test cases using meta-heuristics with the goal of maximizing branch coverage. As explained in Section 6.2, we use the Dynamic Many-Objective Sorting Algorithm (DynaMOSA) [58] as suggested in the literature [74, 101, 171]. Previous studies have shown that *DynaMOSA* outperforms other meta-heuristics in unit test case generation for *Java* [74, 101], `python` [171], and `solidity` [80] programs. Assessing other meta-heuristics in the context of unit test generations for *JavaScript* programs is part of our future agenda.

Our implementation applies the probabilistic model described in Section 6.3.2 to determine what is the potential type of each input parameter. We have implemented two different strategies to incorporate the type inference model into the main *DynaMOSA* loop, namely *proportional type sampling* and *ranking*.

Proportional Sampling

This strategy can assign various types to each input parameter. As explained in Section 6.3.2, our model assigns scores to multiple types (see Equation (6.1)) based on the number of positive hints received by analyzing the associated relations. When creating a new test case (either in the initial population or during mutation), each input parameter is assigned one of the types. Each candidate type has a probability of being selected equal to the value obtained by applying Equation (6.2). Notice that each data type is sampled for each newly generated test case. Therefore, the same input parameter (for the same function) may be assigned different types every time a new test case is created.

Ranking

This strategy assigns only one type to the input parameter. In particular, this strategy sorts all types with positive hints in descending order of their score values. Then, this method selects the type with the largest probability (or the largest number of hints).

Test Execution

Once generated, each generated test case will contain a sequence of function calls with their input data. These tests are then executed against the program under test, and the coverage information is stored. The "fitness" of a test is measured according to its distance to cover all unreached branches in the code, as typically done in *DynaMOSA*. The distance to each uncovered branch is computed using two well-known coverage heuristics [4]: (1) the *approach level* and (2) the normalized *branch distance*.

6.4 Empirical Study

To assess the impact of the unsupervised probabilistic type inference on the performance of search-based unit test generation for *JavaScript*, we perform an empirical evaluation to answer the following research questions:

RQ1 *How does unsupervised static type inference impact structural coverage of DynaMOSA for JavaScript?*

Benchmark	**#Units**	**CC**	**SLOC**	**Avg. n. branches**
Commander.js	4	23	208	29
Express	15	20	222	25
Moment.js	54	7	33	8
Javascript Algorithms	30	5	68	8
Lodash	10	11	63	16

Table 6.1: Benchmark statistics

RQ2 *What is the best strategy to incorporate type inference in DynaMOSA?*

6.4.1 Benchmark

To the best of our knowledge, there is no existing *JavaScript* benchmark for unit-level test case generation. Hence, for our empirical study, we build a benchmark comprising of five *JavaScript* projects: *Express*[4], *Commander.js*[5], *Moment.js*[6], *JavaScript Algorithms*[7], *Lodash*[8]. These projects were selected based on their popularity in the JavaScript community (measured through the number of stars on GitHub) and represent a diverse collection of *JavaScript* syntax and code styles. From these projects, we selected a subset of `units` (*i.e.,* classes or functions) based on two criteria: (1) the unit has to be testable (*i.e.,* the unit has to be exported), and (2) the unit needs to be non-trivial (*i.e.,* have a Cyclomatic Complexity of $CC \geq 2$ as calculated by *Plato*[9]). The latter criterion is in line with existing guidelines for assessing test case generation tools [101]. Table 6.1 provides the main characteristics of our benchmark at the project-level, including the average Cyclomatic Complexity per project (**CC** column), the average Source Lines Of Code (**SLOC** column), and the average number of branches. It is worth noting that some of the files in the selected projects had to be excluded or modified. For example, in the `Commander.js` project there are two files that contain statements that terminate the running process. This has the effect of also terminating the test case generation process. Therefore, we have excluded this file from the benchmark and modified it, so that any other files depending on it will not be affected.

6.4.2 Prototype

To evaluate the proposed approach, we have developed a prototype for unit-level test case generation that implements our unsupervised dynamic type inference, written in `Typescript`. The prototype also implements the state-of-the-art search algorithm for test case generation, namely *DynaMOSA* [58], as well as the guiding heuristics [4], *i.e.,* the approach level and branch distance.

[4]https://expressjs.com/
[5]https://tj.github.io/commander.js/
[6]https://momentjs.com/
[7]https://github.com/trekhleb/javascript-algorithms
[8]https://lodash.com/
[9]https://github.com/es-analysis/plato

6.4.3 Parameter Settings

For this study, we have chosen to mainly adopt the default search algorithm parameter values as described in literature [58]. Previous studies have shown that although parameter tuning impacts the search algorithm's performance, the default parameter values provide reasonable and acceptable results [109]. Hence, the search algorithm uses a single point crossover with a crossover probability of 0.75, mutation with a probability of $1/n$ (n = number of statements in the test case), and tournament selection. For the population size, however, we decided to deviate from the default (50). We went for a size of 30 as our preliminary experiment showed this worked best for a benchmark this size. The search budget per unit under test is 60 s. This is a common value used in related work [75].

6.4.4 Experimental Protocol

To answer RQ1, we compare the two variants of our approach with *DynaMOSA* without type inference. In particular, for this baseline, the type for the input data is randomly sampled among all types that can be extracted using the relations described in Section 6.3.1. To answer RQ2, we compare the two variants of our approach: (1) proportional type sampling, and (2) ranking.

To account for the stochastic nature of the approach, each unit under test was run 20 times. We performed 20 repetitions of 3 configurations (*i.e.,* random type sampling, ranking, and proportional sampling) on 98 units under test, for a total of 5880 runs. This required $(5880\ \text{runs} \times 60\,\text{s})/(60\,\text{s} \times 60\,\text{min} \times 24\,\text{h}) \approx 4.1\,\text{d}$ computation time. At the end of each run, we stored the maximum branch coverage achieved by the approach for the active configuration (**RQ1** and **RQ2**). The experiment was performed on a system with an AMD Ryzen 9 3900X (12 cores 3.8 GHz) with 32 GB of RAM. Each experiment was given a maximum of 8 GB of RAM. To determine if one approach performs better than the others, we applied the unpaired Wilcoxon signed-rank test [67] with a threshold of 0.05. This non-parametric statistical test determines if two data distributions are significantly different. In addition, we apply the Vargha-Delaney $\hat{A}_{12}$ statistic [68] to determine the effect size of the result, which determines the magnitude of the difference between the two data distributions.

6.5 Results

This section discusses the results of our empirical study with the aim of answering the research questions formulated in Section 6.4. All differences in results are presented in absolute differences (percentage points).

6.5.1 Result for RQ1: Structural Coverage

Table 6.2 summarizes the results achieved by our approach on the benchmark with the winning configuration highlighted in gray color. It shows the median branch coverage and the Inter-Quartile-Range (IQR) for the two possible strategies to incorporate the type inference model (Ranking, Proportional) and a baseline that uses random type sampling (Random). The *Units* column indicates the number of units (*i.e.,* exported classes and functions) that are tested in the file of the benchmark project.

On average for all 57 files in the benchmark, *random* achieves 33.4 % branch coverage,

Benchmark	File Name	#Units	Random		Ranking		Proportional	
			Median	IQR	Median	IQR	Median	IQR
Commander.js	help.js	1	0.20	0.019	0.41	0.076	**0.53**	0.023
	option.js	2	0.33	0.056	0.33	0.056	**0.39**	0.000
	suggestSimilar.js	1	0.69	0.062	0.56	0.156	**0.75**	0.062
Express	application.js	1	0.63	0.019	0.63	0.019	**0.65**	0.019
	query.js	1	**0.67**	0.000	**0.67**	0.000	**0.67**	0.000
	request.js	1	0.25	0.000	**0.27**	0.023	0.25	0.023
	response.js	1	**0.14**	0.007	0.13	0.013	**0.14**	0.013
	utils.js	7	0.56	0.007	**0.62**	0.000	0.59	0.029
	view.js	1	**0.06**	0.000	**0.06**	0.000	**0.06**	0.000
JS Algorithms Graph	articulationPoints.js	1	0.00	0.000	0.00	0.000	**0.08**	0.000
	bellmanFord.js	1	0.00	0.000	0.17	0.000	**0.33**	0.000
	bfTravellingSalesman.js	1	0.00	0.000	**0.08**	0.000	**0.08**	0.000
	breadthFirstSearch.js	1	0.12	0.125	**0.38**	0.031	0.31	0.125
	depthFirstSearch.js	1	**0.00**	0.167	**0.00**	0.167	**0.00**	0.167
	detectDirectedCycle.js	1	0.00	0.000	0.12	0.000	**0.38**	0.000
	dijkstra.js	1	0.00	0.000	0.10	0.000	**0.20**	0.100
	eulerianPath.js	1	0.00	0.000	0.00	0.000	**0.21**	0.000
	floydWarshall.js	1	0.00	0.000	**0.67**	0.000	**0.67**	0.000
	hamiltonianCycle.js	1	**0.00**	0.000	**0.00**	0.000	**0.00**	0.050
	kruskal.js	1	0.10	0.100	**0.30**	0.000	**0.30**	0.000
	prim.js	1	0.08	0.000	0.08	0.083	**0.17**	0.000
	stronglyConnectedComponents.js	1	0.00	0.000	0.00	0.000	**0.25**	0.000
JS Algorithms Knapsack	Knapsack.js	1	**0.57**	0.000	0.50	0.000	**0.57**	0.000
	KnapsackItem.js	1	**0.50**	0.000	**0.50**	0.000	**0.50**	0.000
JS Algorithms Matrix	Matrix.js	12	0.79	0.053	0.74	0.026	**0.80**	0.158
JS Algorithms Sort	CountingSort.js	1	**0.92**	0.083	**0.92**	0.021	**0.92**	0.000
JS Algorithms Tree	RedBlackTree.js	1	0.21	0.000	0.26	0.000	**0.29**	0.037
Lodash	equalArrays.js	1	0.08	0.000	0.67	0.042	**0.75**	0.052
	hasPath.js	1	0.75	0.156	0.75	0.000	**0.88**	0.250
	random.js	1	**1.00**	0.000	**1.00**	0.000	**1.00**	0.000
	result.js	1	**0.90**	0.100	0.80	0.000	**0.90**	0.100
	slice.js	1	**1.00**	0.000	**1.00**	0.000	**1.00**	0.000
	split.js	1	**0.88**	0.000	**0.88**	0.000	**0.88**	0.000
	toNumber.js	1	0.60	0.000	**0.65**	0.000	**0.65**	0.050
	transform.js	1	**0.83**	0.000	**0.83**	0.000	**0.83**	0.083
	truncate.js	1	0.38	0.000	**0.59**	0.029	**0.59**	0.000
	unzip.js	1	**1.00**	0.000	**1.00**	0.000	**1.00**	0.000
Moment.js	add-subtract.js	1	0.00	0.000	**0.71**	0.018	**0.71**	0.000
	calendar.js	2	0.05	0.000	**0.45**	0.091	0.43	0.091
	check-overflow.js	1	0.05	0.000	**0.60**	0.000	**0.60**	0.000
	compare.js	6	**0.14**	0.000	**0.14**	0.000	**0.14**	0.000
	constructor.js	3	0.38	0.000	**0.53**	0.008	0.41	0.156
	date-from-array.js	2	**0.88**	0.000	**0.88**	0.000	**0.88**	0.000
	format.js	4	**0.08**	0.000	**0.08**	0.000	**0.08**	0.000
	from-anything.js	2	0.68	0.059	**0.71**	0.000	0.69	0.037
	from-array.js	1	0.02	0.000	**0.04**	0.000	**0.04**	0.000
	from-object.js	1	**0.50**	0.000	**0.50**	0.000	**0.50**	0.000
	from-string-and-array.js	1	0.00	0.000	**0.31**	0.000	**0.31**	0.000
	from-string-and-format.js	1	0.06	0.000	**0.56**	0.039	0.55	0.133
	from-string.js	3	0.06	0.000	**0.16**	0.000	**0.16**	0.000
	get-set.js	5	0.14	0.000	0.23	0.045	**0.36**	0.068
	locale.js	2	**0.17**	0.167	**0.17**	0.000	**0.17**	0.000
	min-max.js	2	**0.12**	0.000	**0.12**	0.000	**0.12**	0.000
	now.js	1	**0.50**	0.000	**0.50**	0.000	**0.50**	0.000
	parsing-flags.js	1	**0.50**	0.000	**0.50**	0.125	**0.50**	0.000
	start-end-of.js	2	**0.10**	0.000	**0.10**	0.000	**0.10**	0.000
	valid.js	2	**0.38**	0.000	**0.38**	0.000	**0.38**	0.000

Table 6.2: Median Branch Coverage and the Inter-Quartile-Range. The largest values are highlighted in gray color.

Table 6.3: Statistical results *w.r.t.* branch coverage

Comparison	#Win				#No diff.	#Lose			
	Negl.	Small	Medium	Large	Negl.	Negl.	Small	Medium	Large
Ranking vs. Random	-	3	1	23	26	-	1	-	3
Prop. sampling vs. Random	-	1	4	27	25	-	-	-	-
Prop. sampling vs. Ranking	-	4	-	16	33	-	3	1	-

ranking 42.7 %, and *proportional type sampling* 46.0 %. The baseline still performs quite well, as *random type sampling* can be effective in triggering assertion branches and can over time guess the correct types for primitives. For the *ranking* strategy, the average improvement in branch coverage is 9.3 %. The file with the least improvement is `suggestSimilar.js` from the `Commander.js` project with an average decrease of 13 %. The file with the most improvement is `add-subtract.js` from the `Moment.js` project with an average increase of 71 %, which corresponds to 10 additionally covered branches. For the *proportional* strategy, the average improvement in branch coverage is 12.6 %. There are 24 files for which the *proportional* strategy performs equally to the baseline. The file with the most improvement is again `add-subtract.js` from the `Moment.js` project with an average increase of 71 %.

Table 6.3 shows the results of the statistical comparison between the two strategies and the baseline, based on a p-value ≤ 0.05. *#Win* indicates the number of times that the left configuration has a statistically significant improvement over the right one. *#No diff.* indicates the number of times that there is no evidence that the two competing configurations are different; *#Lose* indicates the number of times that the left configuration has statistically worse results than the right one. The *#Win* and *#Lose* columns also include the $\hat{A}_{12}$ effect size, classified into *Small*, *Medium*, *Large*, and *Negligible*.

We can see that the *ranking* and the *proportional* strategy have a statistically significant non-negligible improvement over the baseline in 27 and 32 files for branch coverage, respectively. *Ranking* improves with a large magnitude for 23 classes, medium for 1 class, and small for 3 classes and *proportional* with 27 (large), 4 (medium), and 1 (small). The *Ranking* strategy loses in four cases when compared to the baseline: `response.js`, `response.js`, `Knapsack.js`, `Matrix.js`, and `results.js`.

6.5.2 Result for RQ2: Strategy

When we compare the two different strategies with each other, we can observe that the *proportional type inference* on average improves by 3.3 % over the *ranked* strategy based on branch coverage. The file with the least improvement is `constructor.js` from the `Moment.js` project with an average decrease of 12 %. While the file with the most improvement is `detectDirectedCycle.js` from the `JS Algorithms` project with an average increase of 36 %. From Table 6.3, we can see that the *proportional* strategy has a statistically significant non-negligible improvement over *ranking* in 20 cases (16 large and 4 small). While *ranking* improves over *proportional* in only 4 cases (1 medium and 3 small): `slice.js`, `constructor.js`, `from-string-and-format.js`, and `parsing-flags.js`.

6.6 Threats to Validity

This section discusses the potential threats to the validity of our study.

6.6.1 External Validity

An important threat regards the generalizability of our study. We selected five open-source projects based on their popularity in the *JavaScript* community. The projects are diverse in terms of size, application domain, purpose, syntax, and code style. Further experiments on a larger set of projects would increase the confidence in the generalizability of our study and, therefore, is part of our future work.

6.6.2 Conclusion Validity

Threats to *conclusion validity* are related to the randomized nature of *DynaMOSA*. To minimize this risk, we have executed each configuration 20 times with different random seeds. We have followed the best practices for running experiments with randomized algorithms as laid out in well-established guidelines [66]. Additionally, we used the unpaired Wilcoxon signed-rank test and the Vargha-Delaney $\hat{A}_{12}$ effect size to assess the significance and magnitude of our results. To ensure a controlled environment that provides a fair evaluation, all experiments have been conducted on the same system and interfering processes were kept to a minimum.

6.7 Conclusion and Future Work

In this paper, we presented an automated unit test generation approach for *JavaScript*, the most popular dynamically-typed language. It generates unit-level test cases by using the state-of-the-art meta-heuristic search algorithm *DynaMOSA* and a novel unsupervised probabilistic type inference model. Our results show that (1) the proposed approach can successfully generate test cases for well-established libraries in *JavaScript*, and (2) the type inference model plays a significant role in achieving larger code coverage (through *proportional sampling*). As part of our future work, we plan (1) to extend our benchmark, (2) to investigate more meta-heuristics, (3) assess different strategies to incorporate the type inference model within the search process, and (4) compare our type inference model to state-of-the-art deep learning approaches.

7 SynTest Ecosystem

Software testing is an important but time-consuming part of the software development process. To automate this process, various tools for automated unit-level test case generation (e.g., EvoSuite and Pynguin) have been created over the years. Despite this, there is still a shortage of user-friendly production-level tooling. To this aim, we introduce SynTest-Framework, an open-source framework for automated test case generation and quality assurance. The framework aims to provide a comprehensive ecosystem of testing tools targeted at multiple programming languages. Its modular and extensible architecture allows users to customize the framework for their use cases. In addition, we introduced two language-specific tools, namely Syntest-Solidity and Syntest-JavaScript, built as an extension of SynTest-Framework. These tools extend the automated test case generation capabilities of the SynTest ecosystem to Solidity smart contracts and JavaScript programs, respectively. In order to assess the performance of these tools, we performed an empirical study. The results show that both Syntest-Solidity and Syntest-JavaScript are effective at generating test cases for their respective programming languages.

This chapter is based on:

Mitchell Olsthoorn, Dimitri Stallenberg, Arie van Deursen, and Annibale Panichella. SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts. 44th ACM/IEEE International Conference on Software Engineering 2022 (ICSE'22): Companion Proceeding [80] and

Mitchell Olsthoorn, Dimitri Stallenberg, and Annibale Panichella. SynTest-JavaScript: Automated Unit-Level Test Case Generation for JavaScript. 17th IEEE/ACM International Workshop on Search-Based and Fuzz Testing 2024 (SBFT'24).

7.1 SynTest-Framework

Software testing is an important part of the software development process. This task is often performed manually, which can be both time-consuming and prone to errors [2, 3]. Automated unit-level test case generation can help alleviate these issues but is a complex process that requires domain-specific knowledge and expertise. Additionally, given the tight coupling between white-box automated test case generation tools and the language of the program under test, it is challenging to build tooling for multiple languages and/or testing frameworks. This led to a tooling landscape that is fragmented, with many tools focusing on specific languages (*e.g., Java*) and testing frameworks [74]. To make automated test case generation techniques more accessible to practitioners, we need to provide user-friendly tools that are language agnostic. Therefore, we set out to create a framework, called *SynTest-Framework*, that provides an ecosystem of automated test case generation and quality assurance tools for multiple programming languages.

7.1.1 Architecture

SynTest-Framework is built with a modular architecture that can accommodate diverse software analysis and testing requirements. At its core, it features language-agnostic libraries that provide common functionality (*e.g.,* configuration, process management, search algorithms, metric collection) applicable across various programming languages. These libraries form the foundation of the framework. On top of these libraries, we have built a Command-Line Interface (CLI) that provides a unified interface. One of the key features of the framework is its extensibility, which allows the dynamic loading of modules. Modules provide a mechanism for users to build upon and extend *SynTest*. They contain a set of related extensions that are loaded together. This modular and extensible structure allows users to tailor *SynTest-Framework* to their specific use case. Modules can be developed by external third parties completely separate from the framework. At the time of writing, we have created two language-specific modules, namely *Syntest-Solidity* and *Syntest-JavaScript*, which are discussed in Sections 7.2 and 7.3.

7

7.1.2 Extensions

This sections describes the different extension types supported within the *SynTest* ecosystem. Figure 7.1 gives a visualization of how the different extension types fit within the overall architecture of the *SynTest-Framework*.

Tools

The tool extension type is the most important extension type within the framework. It adds behavior to our CLI by providing a new set of commands and functionalities. A tool extension is considered a first-class citizen, as it is the only extension type that can be executed directly from the CLI. They are the main entry point for users to interact with the framework. Without them, the framework would be nothing more than a collection of libraries. To run a tool, users have to execute the following command: `syntest <tool> <command> [options]`, where `<tool>` is the name of the tool and `<command>` is the name of the command within the tool. The capabilities of a tool extension are only limited by the developer's creativity. By providing core commands and functionalities, tool extensions create a solid base upon which additional functionality can be added. This expansion

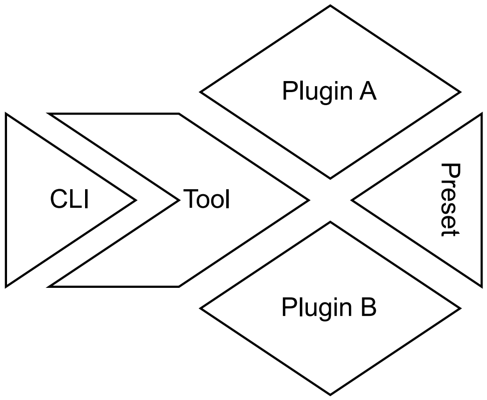


Figure 7.1: *SynTest-Framework* architecture.

opens up a world of possibilities, enabling users to perform a wide range of tasks within the framework's ecosystem.

Plugins

Plugins are a powerful extension type that allow users to customize the behavior and functionality of the framework to suit their specific needs. Tools can provide points of extension where plugins can hook into the execution flow of the overall framework. This allows plugins to respond to specific events or modify the behavior of a tool. This level of customization is crucial for making the framework flexible and allowing it to accommodate a wide range of use cases.

The CLI itself also provides two points of extension, namely event listener plugins and metric middleware plugins. Event listener plugins allow you to listen to all events within the framework and act upon them. These events can range from user actions to internal program states. As an example, the state storage plugin stores certain program states to file as they change throughout the lifecycle of the framework. This plugin is useful for logging, debugging, or triggering specific actions based on program events. The metric middleware plugins allow users to modify the behavior of our metric collection system. Two examples of this type of plugin are the statistics plugin which calculates additional statistical metrics based on other metrics, and the file writer plugin which writes all metrics to several () files.

Presets

Lastly, presets are an extension type that interact with the configuration system of *SynTest-Framework*. They provide a mechanism to override specific settings for both the tool's default configuration and the configuration provided by the user. Presets simplify the configuration process for users by offering predefined, ready-to-use options. Users are spared needing to look up the meaning of a configuration parameter or memorize an array of command-line values. Instead, they can simply select a preset that aligns with their use case. For test case generation this is especially useful as search algorithms often have many parameters that can be tweaked. Moreover, presets can play a crucial role in simplifying the sharing of empirical setups for scientific research. Rather than sharing an elaborate specification, users can conveniently share a single preset containing all the essential information required to replicate research results, promoting reproducibility and collaboration.

7.2 SynTest-Solidity

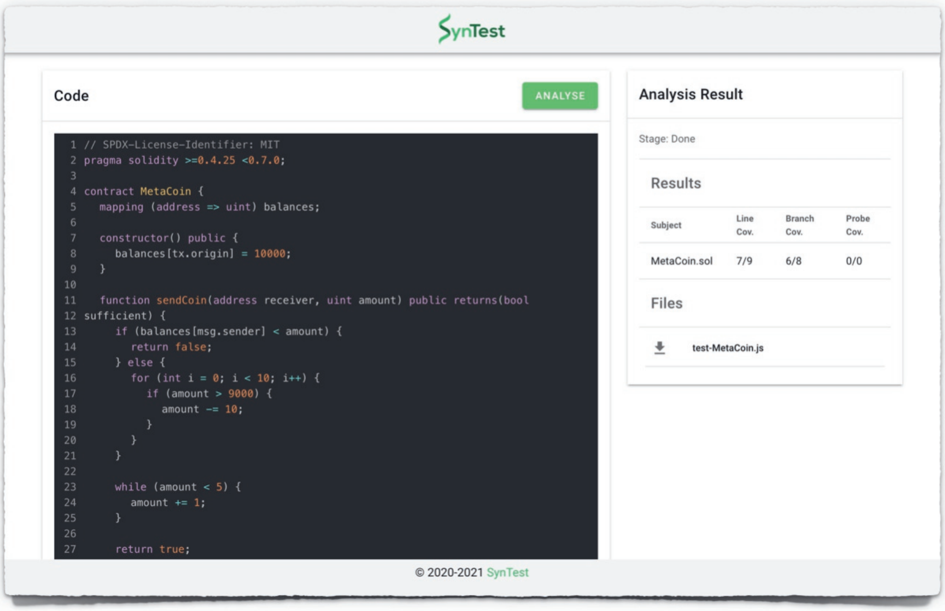
Smart contracts are agreements between multiple parties on how certain tasks (*e.g.,* releasing or transferring funds) need to be executed. More specifically, they are short programs deployed to a distributed ledger (blockchain) that run when predetermined conditions have been met. This allows automating the execution of an agreement with a deterministic outcome without a trusted intermediary. Smart contracts have been gaining popularity in recent years. The largest and most prominent smart contract platform is *Ethereum*, which uses the *Solidity* programming language [110].

One key property of smart contracts is that they can not be updated anymore after their deployment. This property prevents the creator of a smart contract modifying the contract for their own benefit (*e.g.,* only allowing themselves to retrieve funds). However, this introduces certain challenges, such as when a contract contains a bug. Therefore, it is critical to thoroughly test the behavior and constraints of the smart contract as early as possible in the development lifecycle. Since smart contracts have complex interactions, manual testing becomes very difficult and error-prone [73].

Over the last few years, various techniques have been used to assist developers with testing *Solidity* smart contracts, like fuzzing, formal verification, and test case generation. Tools like sFuzz [112] have successfully used fuzzing techniques to produce test input data that causes errors or unwanted effects within the contract. However, since fuzzers only generate input data but no actual (runnable) test cases, they cannot create compositional tests (*i.e.,* a test case with multiple requests) nor test for the desired behavior of the contract. On the other hand, formal verification approaches aim to mathematically verify the behavior of a contract by transposing the contract into a formal proof language. In general, this approach does not scale and requires developers to provide a complex model of the desired behavior [188]. To the best of our knowledge, we have not found any study that indicates this is different for smart contracts. Lastly, test case generation allows developers to test smart contracts for both bugs and behavior in a more efficient and scalable way. In addition, this allows generated tests to be added to the existing test suite for regression testing purposes. To the best of our knowledge, there exists only one related work that focusses on test case generation for *Solidity* [118]. However, the research prototype used in the study is not specifically adapted for the *Solidity* language. For example, it does take into account *Solidity*-specific types, such as different sizes for integers, nor distinguishes between signed and unsigned types. Besides, the tool does not generate assertions.

We have developed a tool extension for *SynTest-Framework*, called *Syntest-Solidity*[1], to allow developers to more effectively and efficiently test their smart contracts. It is publicly available on NPM[2] and GitHub[3]. Instructions on how to set up and run the tool can be found on both platforms. *Syntest-Solidity* makes use of a genetic algorithm to evolve a set of initial randomly generated test cases to satisfy predefined test criteria (*i.e.,* function, line, and branch coverage). It does this by extracting objectives from the contract, feeding these into the search algorithm, and evaluating the produced test cases using *Truffle* (de-facto testing library for *Solidity*) and *Ganache* (local development blockchain).

[1]`https://www.syntest.org/`
[2]`https://www.npmjs.com/package/@syntest/solidity`
[3]`https://github.com/syntest-framework/syntest-solidity`



```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.25 <0.7.0;

contract MetaCoin {
  mapping (address => uint) balances;

  constructor() public {
    balances[tx.origin] = 10000;
  }

  function sendCoin(address receiver, uint amount) public returns(bool
sufficient) {
    if (balances[msg.sender] < amount) {
      return false;
    } else {
      for (int i = 0; i < 10; i++) {
        if (amount > 9000) {
          amount -= 10;
        }
      }
    }

    while (amount < 5) {
      amount += 1;
    }

    return true;
```

Subject	Line Cov.	Branch Cov.	Probe Cov.
MetaCoin.sol	7/9	6/8	0/0

test-MetaCoin.js

Figure 7.2: Online web service for generating test cases with *Syntest-Solidity*.

7.2.1 Usage Scenarios

In addition to the CLI of *SynTest-Framework*, we provide an online web service[4] which allows users to interact with *Syntest-Solidity* without having to install the tool locally. Figure 7.2 depicts the main interface of the web service. A developer can submit their contract to the web service and request it to be analyzed by clicking on the *analyze* button. This will start the test case generation process, at the end of which the generated test cases can be downloaded directly from the webpage. The webpage will also display relevant statistics, such as the number of lines and branches that are covered by the test cases.

Figure 7.3 shows the architecture of the backend of the web service. This architecture consists of a webpage written in *Vue* that communicates with the *service* backend through websockets. The *service* backend is built with *Node.js*. Its role is to validate the user input and manage the sessions. Whenever a new contract is submitted, the *service* backend enqueues the job in the *RabbitMQ* message broker. The purpose of the message broker is to keep track of all the current submitted jobs and make sure that the *workers* process them. The workers are *Node.js* programs which retrieve jobs from the broker and perform the actual test case generation. The worker will create the files and folders required for *Syntest-Solidity* to run, after which it runs the tool and returns the resulting statistics and test files. The architecture is made such that the number of workers can be scaled based on the demand.

[4]`https://tool.syntest.org/`

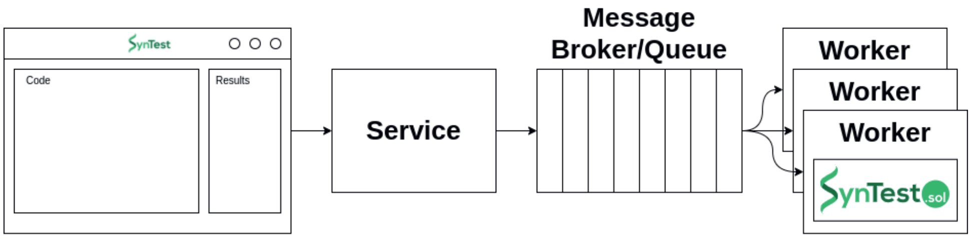


Figure 7.3: Architectural overview of the *Syntest-Solidity* web service.

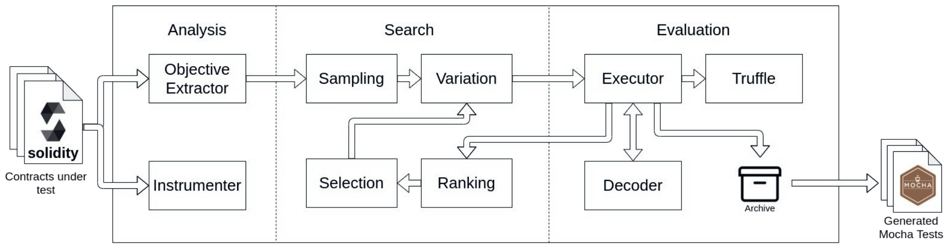


Figure 7.4: Internal architecture of *Syntest-Solidity*.

7.2.2 Tool Workflow

Analysis

To generate a test-suite, *Syntest-Solidity* uses white-box heuristics. Hence, the tool needs access to the source code for instrumentation which allows the tool to collect coverage information at runtime. The different steps of our tool are depicted in Figure 7.4. The tool takes as input a smart contract, parses the source code to build the Control Flow Graph (CFG), and extracts the search objectives.

Search

The search process starts after extracting the objectives and instrumenting the code, as shown in Figure 7.4. The tool boasts a number of different search algorithms, including random search, *NSGA-II*, *MOSA*, and *DynaMOSA*. For the purposes of this tool demo, we focus on the Dynamic Many-Objective Search Algorithm (*DynaMOSA*), which is the state-of-the-art meta-heuristic for white-box unit testing [58].

DynaMOSA evolves a set of randomly generated test cases. These test cases are generated using a *sampler* that provides the list of callable methods (*i.e.,* `public` and `external` methods) and constructors for the contract under test (CUT). A test case is encoded as a sequence of method calls. The *root* of the sequence is the contract deployment/instantiation made by invoking one of the public *constructors*. The remaining method calls in the sequence are obtained by randomly invoking some `public` and `external` methods in the CUT. Notice that the length of the test case is variable and can change through the generations.

The initial tests are iteratively evolved using *crossover* and *mutation*. The former operator creates new tests by swapping statements between pairs of tests (called *parents*). The latter operator applies small changes to the newly generated test cases, called *offspring*.

The population for the next generation is obtained by selecting the best test cases among parents and offspring using the *preference criterion* and the *non-dominated sorting* [58]. The goal of these two heuristics is to promote test cases that are closer to reaching the uncovered branches and lines in the code. The process is repeated until the predefined budget is depleted.

Note that *DynaMOSA* only optimizes the yet uncovered branches and lines. Every time an uncovered branch (or line) is reached, the corresponding test case is saved into the *archive*. The final test suite is composed of all test cases stored in the archive.

Text Execution

Our tool needs to execute each test case to determine how close that test is to covering the objectives. This is performed by the *objective functions*, which measure the distance to reaching an uncovered branch or line in the code using state-of-the-art heuristics, *i.e.*, the *approach level* and the normalized *branch distance* [58]. The flow of steps performed during test execution is also shown in Figure 7.4.

A test case is first converted into a JavaScript test. This test is then executed on a fresh *Ganache* blockchain instance. This local blockchain instance hosts the CUT deployed using the *Truffle framework*. The test execution results are then collected and used to compute the approach level and the branch distance for the yet uncovered branches and lines.

7.2.3 Solidity-specific Features

Syntest-Solidity provides support for all data types and other features that are unique and specific to *Solidity*. In the following, we briefly elaborate on these features and how they are handled.

Data Types

The *Solidity* programming language includes multiple data types, including boolean, number, bytes, strings, and arrays. Compared to other languages (*e.g.*, Java), *Solidity* includes multiple subtypes for both integers and doubles. There are two main subtypes of integers: signed integers (`int`) and unsigned integers (`uint`). These subtypes also have different sizes, ranging from `uint8` up to `uint256`, which correspond to 8 and 256 bits, respectively. Similarly, double numbers can be both signed (`fixed`) or unsigned (`ufixed`) and have different sizes in the number of bits (*e.g.*, `ufixed128x18`, etc.).

Syntest-Solidity handles all these subtypes as it encodes integers (and float numbers) with an extra bit for the sign and different upper and lower bounds depending on the number of required bits. Note that *Syntest-Solidity* generates tests in JavaScript, which uses 52 bits for numbers. To allow representing larger numbers (up to 256 bits) required for *Solidity*, *Syntest-Solidity* uses the `BigNumber` library that allows arbitrary-precision arithmetic.

Solidity Addresses

`Address` is a special data type in *Solidity* and represent the intended recipient of a *transaction*. An `address` has 160 bits or 40 hex characters. An address always has a `0x` prefix in its hexadecimal format (base 16 notation). *Syntest-Solidity* uses two strategies to handle and instantiate addresses. The first strategy extracts address literals from the source code of the CUT. These constants are used as seeds when generating test cases with 50 % probability. This means that constant addresses in the code are (with some probability) used in the generated test cases. The address `0x0` (or zero-address) is a special constant used to indicate that a new contract is being deployed.

The second strategy uses pre-allocated addresses that are allocated by the *Truffle* framework when the contract is deployed at the beginning of each test case. These pre-allocated addresses are accessible with the statement `account[index]`, where `index` points to the pre-allocated address to consider.

Transactions

Interactions with smart contracts are made via *transactions*. Transactions correspond (1) to either sending Ether to another account, (2) executing a contract method/function, or (3) adding a new contract to the network. Hence, a method call in the test case is required to have a recipient address in addition to the actual input parameters for the method being invoked. Therefore, a method call is encoded in *Syntest-Solidity* as an array of (1) input parameters, (2) return values, and (3) the address of the recipient.

Assertions

Syntest-Solidity generates assertions at the end of the search process as a post-process step. It does this by collecting the runtime values (*e.g.,* contract states and return values of invoked methods) from the final execution of the test cases. A specific type of assertion regards the runtime exceptions that can be thrown when the state-reverting security conditions (*i.e.,* `revert` and `require`) are not satisfied. If a test case triggers these runtime exceptions, the assertion generated by our tool asserts that the expected exception is triggered.

7.2.4 Evaluation

To evaluate *Syntest-Solidity*, we tested 20 *Solidity* smart contracts submitted to *etherscan.io*. In particular, the selected contracts are written in Solidity versions 5 and 6, which are currently supported by our tool. We randomly selected these contracts among those that have been submitted to *etherscan.io* multiple times for security checking between January and June 2021. For the selection, we excluded duplicates and analyzed their cyclomatic complexity (`CC`) to exclude trivial contracts with no branching statements. As suggested by existing guidelines in the literature [58, 133], test case generation tools should be assessed on code units (*e.g.,* classes in Java) with a cyclomatic complexity above two (`CC>2`). In our context, the 20 selected contracts have functions with cyclomatic complexity above three. The contracts and their characteristics are summarized in Table 7.1. The size of the contracts ranges from 23 LOC for `MetaCoin` to 307 for `Revive`.

While the benchmark might not be large enough for a complete empirical assessment, our goal is to show the ability of our tool in generating test cases with high code coverage and assertions for non-trivial, real-world smart contracts. Assessing the tools on a larger and more extensive benchmark is part of our future agenda.

Table 7.1: Average (median) coverage achieved by *Syntest-Solidity* over 20 independents runs

Contract	**LOC**	**Coverage**		
		Function	**Branch**	**Line**
AavePoolReward	108	0.92	0.50	0.60
Baz	33	1.00	0.95	0.95
BirdOracle	134	0.89	0.59	0.66
Core_Fi_V3	62	0.88	0.56	0.59
CryptoGhost	165	1.00	0.84	0.79
CryptoSecureBankToken	254	0.93	0.50	0.73
DJCoin	195	0.88	0.67	0.77
EdenCoin	67	1.00	1.00	1.00
FreakCoin	139	1.00	0.60	0.69
GAZ_ERC20	71	0.55	0.55	0.54
INS	109	1.00	0.50	0.50
MetaCoin	23	1.00	0.88	0.89
Revive	307	0.84	0.41	0.64
Rootkit_finance	61	0.88	0.59	0.61
SLTDETHlpReward	291	0.78	0.49	0.63
Straight_Fire_Finance	62	0.88	0.59	0.61
TetherToken	98	1.00	0.50	0.58
ThriftToken	96	0.91	0.50	0.63
TimeMiner	174	1.00	0.67	0.81
WOLF	80	1.00	0.40	0.68
Mean		0.91	0.64	0.70

Experimental Setup
In this evaluation, we use the parameter values suggested in the related literature [58]. More specifically, we run *DynaMOSA*, which is the state-of-the-art search algorithm for unit-level test case generation [58]. In addition, we use a population size of 10 test cases. New test cases are generated using the *single-point crossover* with probability $p_c = 0.80$. Test cases are further changed using the *uniform mutation* with the probability $p_m = 1/n$, where n is the length of the test case. This operator either adds, deletes, or changes statements within each test case. These three mutation operators are equally probable.

We set an overall search budget of 30 minutes per smart contract. This search budget is larger than the one usually used in unit-test generation in other languages (*e.g.*, Java [58]). This is because *Syntest-Solidity* has to deploy the contract under test to the smart contract network before each test case.

Empirical Results
We run *Syntest-Solidity* 20 times for each smart contract to account for the randomness of the search process. Table 7.1 reports the median results achieved by *SynTest-Framework* with regard to function, branch, and line coverage. In all cases, *Syntest-Solidity* was able to generate test suites with high function coverage, which is 91 % on average. For branch coverage, the results vary between 40 % achieved on `Wolf` and 100 % achieved for `EdenCoin`.

The produced branch coverage is greater than 50 % in all smart contracts except three. As a consequence, *Syntest-Solidity* yielded an average branch overage of 61 %. The results for line coverage are in-line with those achieved for branch coverage. Indeed, the mean line coverage is 68 %, with a minimum value of 54 % obtained for `GAZ_ERC20` and a maximum value of 100 % for `EdenCoin`.

7.3 SynTest-JavaScript

Various tools for unit-level test case generation (*e.g., AUSTIN* for *C*, and *EvoSuite* and *Randoop* for *Java*) have been created over the years. These tools mostly focus on statically-typed languages [74]. The most recent Stack Overflow developer survey[5], however, shows that *JavaScript* and *Python*, which are both dynamically-typed, are the most popular programming languages among professional developers. Recently, Lukasczyk *et al.* proposed *Pynguin*, an automated unit-level test case generation tool for *Python* [189]. However, despite *JavaScript*'s eleventh year in a row as the most popular programming language, automated tool support for test case generation for *JavaScript* is still lacking.

In the last decade, there has been a growing interest in developing tools for *JavaScript* [182–184, 190]. These tools, however, focus on *JavaScript* web applications that are characterized by their event-driven execution model and interaction with the Document Object Model (DOM) of the browser. *JavaScript* started out in 1995 as a client-side scripting engine for the browser, but through the years, additional *JavaScript* runtime engines like Node.js, Deno, and Bun have emerged, which allow developers to use *JavaScript* for server-side applications. These server-side *JavaScript* engines are used to create web servers and command-line tools and are heavily used by companies like Netflix[6], PayPal[7], and Uber[8].

A crucial problem with developing tools for dynamically-typed languages is that these types of languages do not provide any information on the types of variables and parameters. Types are instead inferred during the execution of the code. This characteristic, coupled with *JavaScript*'s weak typing —where variables can change types during execution— complicates the static determination of types. Without knowing the type of a function parameter, it will be challenging to generate the appropriate test inputs.

In this section, we introduce *Syntest-JavaScript*, an open-source automated unit-level test case generation tool for (server-side) *JavaScript*, which uses a probabilistic type inference approach we have introduced in our previous work [79]. It makes use of search-based algorithms to generate test cases that maximize function, branch, and path coverage. *Syntest-JavaScript* is implemented as a tool extension of the *SynTest-Framework*. This tool aims to provide a platform for researchers and practitioners to develop and evaluate new techniques for test case generation of *JavaScript* programs.

7.3.1 Workflow

The workflow of our tool, depicted in Figure 7.5, unfolds across five phases: (i) *initialization*, (ii) *pre-processing*, (iii) *processing*, (iv) *post-processing*, and (v) *cleanup*.

The *initialization phase* consists of setting up the environment, configuring all the required variables, and initializing the required classes. Next, the *pre-processing* phase uses static analysis methods to gather information about the targeted `units` (*i.e.,* exported functions or classes) that can be used to improve the search process. In this phase, we build the Control Flow Graph (CFG) starting from the Abstract Syntax Tree (AST) of the unit under test. The CFG allows us to extract the branch/function/path objectives from each `unit`. These objectives are used during the *processing* phase to guide the search algorithms

[5]`https://survey.stackoverflow.co/2023/#most-popular-technologies-language-prof`

[6]`https://netflixtechblog.com/debugging-node-js-in-production-75901bb10f2d`

[7]`https://paypal.github.io/PayPal-node-SDK/`

[8]`https://www.uber.com/en-NL/blog/uber-tech-stack-part-two/`

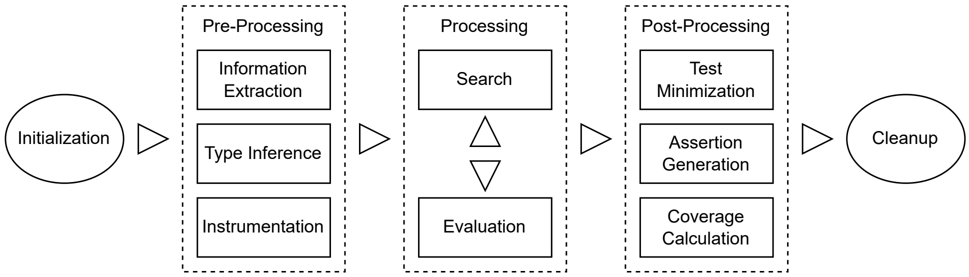


Figure 7.5: *Syntest-JavaScript* tool workflow.

towards maximum coverage. Next, we infer the variable types using the type inference techniques as proposed in our previous work [79]. Finally, we instrument the source code. This instrumentation allows us to record information about the performance of our generated test cases.

During the *processing* phase, each targeted file is considered separately. The information gathered in the pre-processing phase is used to sample encodings (test cases) during the search process. These encodings are then evaluated based on the distance from the objectives, which is calculated by executing the generated test cases (encodings) and using the coverage data generated by the instrumentation. For every objective that has been covered, we save an encoding in our archive [120]. Next to the original objectives (*e.g.,* branches), we also save error objectives that are discovered during the search process. The search and evaluation go back and forth until one of the stopping criteria is met (*e.g.,* running time).

In the *post-processing* phase, we optimize and prettify the encodings (test cases) in the archive. To achieve this, we first minimize the size of the test cases by iteratively removing spurious statements that do not contribute to the total coverage [58]. Next to the individual test case minimization, we also reduce the entire archive (test suite) by checking whether two test cases cover the same objectives and removing one of them. After minimization, the tool generates assertions for each function call result, or exception thrown. Finally, the resulting test suite is run to calculate the final coverage. An example of a generated test case with assertions is shown in Figure 7.7. In the last phase, the tool *cleans up* all the generated temporary files.

7.3.2 Components

Presets

Presets allow developers or researchers to create pre-specified configuration settings. Currently, we have four options, *random search*, *NSGA-II* [150], *MOSA* [120], and *DynaMOSA* [58]. Each preset is designed to align with the configurations detailed in their respective original articles.

Encoding

Our encoding for test cases is structured as a directed acyclic graph. An example of such an encoding is shown in Figure 7.6. At the top, we have the test case itself, which contains

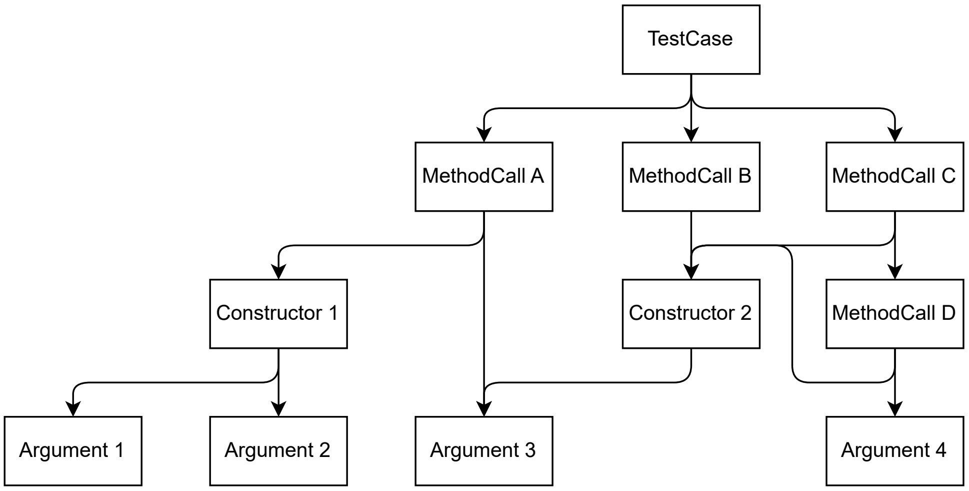


Figure 7.6: *Syntest-JavaScript* encoding structure

the root statements. In this example, the root statements consist of three method calls. Each method call requires an instance of an object to be called upon, for this reason, each method call has a constructor child. Next to the constructor, some method calls have arguments. These arguments can be primitives (*e.g.,* a boolean), objects, functions, or results of other method calls. In Figure 7.6, we see that method call C uses the result of method call D as an argument. Although not shown in the figure, the roots may also be object function calls or regular function calls, the regular function call does not require an object instance to be called.

Supported types

Our encoding supports two primitive types, namely complex and action statements. The primitive statements are a reflection of the primitive statements in *JavaScript* itself. These include: `boolean`, `integer`, `null`, `numeric`, `string`, and `undefined`. Note that in JavaScript there is no distinction between numeric and integer. However, to improve the capabilities of the tool we included a separate integer statement. Currently, the tool supports the following complex statements: `arrays`, `arrow functions`, and `objects`. Finally, the action statements include the `constructor`, `function`, and `method` calls. When the type matching engine finds a matching class type for a certain variable, we can import the matching class and instantiate it through a constructor call. If however no matching class can be found, we use the object statement to construct the required type. This enables the tool to support an infinite number of types.

Constant Pool

During the static analysis in the pre-processing phase, we gathered all constant values from the source code and put them into a constant pool. These constants can then be used during the sampling of primitive types such as strings and numbers.

Type Pool

Next to the constant pool, we also create a type pool, using the analysis files, which consists of all the user-defined object types (classes, interfaces, prototyped functions, *etc.*). These types can then be used when certain objects need to be sampled. We try to find the most likely match to the required object and then sample a constructor or import of that type. As mentioned before, if no matching type can be found, the sampler constructs the object itself through an object statement.

Statement Pool

For each test case, we maintain a statement pool that consists of each statement within the encoding tree. During the sampling of new statements for a test case, there is a chance of reusing already occurring statements from the encoding tree. This is done by sampling a matching statement from the statement pool. For this reason, our encoding is a directed acyclic graph instead of a tree. Figure 7.6 shows this by for example method calls B, C, and D all using constructor instance 2. Note that this only works when the types of the statements match.

Execution engine

To ensure that test case executions do not influence each other we created a test case execution engine that runs each test case in a new process. Running a test case in a separate process allows us to terminate the execution in case of a timeout or memory overflow (which can happen with generated test cases). The execution engine provides a separate process with the test to execute and the relevant environment. After execution, the results sent back by the process include instrumentation data, meta-data, and assertion data. To calculate the "fitness" of a test we measure its distance to cover all unreached branches in the code, as typically done in *DynaMOSA* [58]. The distance to each uncovered branch is computed using two well-known coverage heuristics [4]: (1) the *approach level* and (2) the normalized *branch distance*. We use the instrumentation data to calculate the approach level. To calculate the branch distance we use the meta-data which consists of the branch conditions together with the relevant variable values. Finally, the assertion data contains the results of function calls and is used to generate assertions.

Test splitting

As mentioned before, the post-processing phase minimizes the size of each test case by splitting them. Take the encoding shown in Figure 7.6 as an illustrative example. In this scenario, the original test case can be split into two separate ones: the first encompassing method call A, along with its associated child statements; the second comprising methods calls B and C, along with their child statements. The tool then runs these two test cases separately and checks whether their combined coverage is equal to (or higher than) the original test. In that case, the two new tests are stored and further considered for additional splits recursively.

Test de-duplication

After the test splitting, we end up with a large set of test cases, some of which might be redundant *w.r.t.* to the final coverage. For this reason, we have a de-duplication step in our workflow. During this step, each test case is compared to the other test cases to check

```
1  it("suggestSimilar returns correct suggestion for a misspelled word with special
     characters", async () => {
2    // Meta information
3    // Selected for objective: ./suggestSimilar.js:80:13:::82:7:::2311:2384
4    // ...
5    // Covers objective: ./suggestSimilar.js:81:8:::81:32:::2352:2376
6
7    // Test
8    const word = "cac@e-valiate";
9    const arrayElement = "cache-validate";
10   const candidates = [arrayElement]
11   const suggestSimilarReturnValue = await suggestSimilar(word, candidates)
12
13   // Assertions
14   expect(suggestSimilarReturnValue).to.equal("\n(Did you mean cache-validate?)")
15 })
16
```

Figure 7.7: Example of generated test case

for duplicate objective coverage. If two test cases cover exactly the same objective, the best one is picked based on secondary objectives such as length or readability.

Meta-commenting
To provide as much information as possible to the end user the tool provides meta-comments in each test case. These comments provide information about which objectives the test case covers and for which objective the test case was chosen. For error objectives, we also provide the stack trace in the comments. Figure 7.7 shows some meta-comments in line 2 to 5.

Naming strategy
To generate test cases that not only achieve high coverage but are also very readable, the names of the used variable names must be logical. To achieve this, the tool uses the names of the parameters of the called functions as the variable names for the corresponding arguments. If a variable name is already in use, we number them. For return values, we currently simply name the variable "`[function name]ReturnValue`" as can be seen on line 11 in Figure 7.7. In the future, we plan to improve this by using the name of the returned variable in the source code. We also plan to improve the test and variable names by using Large Language Models (LLMs) as a prettifier.

Assertion Generation
A test case is incomplete without proper assertions. To generate assertions we first execute the test cases without any assertions and record the result of each function call. In the case of an error, we catch and record the error. Then the recorded results are asserted in the final test suite. An example of this is shown on line 14 in Figure 7.7.

7.3.3 Evaluation

To evaluate the effectiveness of *Syntest-JavaScript*, we performed an experiment on the *SynTest-JavaScript-Benchmark*, previously introduced in [79]. To the best of our knowledge, this is the only benchmark targeted at unit-level test case generation for *JavaScript*.

Benchmark	Metrics			Achieved Branch Coverage			Statistical Significance		
	Files	#Units	Avg. CC	random	DynaMOSA	Difference	#Lose	#No Diff.	#Win
Artificial	4	4	5	47.92%	87.50%	39.58%	0	1	3
Commander.js	4	6	23	56.24%	75.43%	19.20%	0	0	4
Express	6	12	32	46.30%	46.41%	0.11%	2	3	1
JavaScript Algorithms	56	69	10	67.88%	73.67%	5.79%	3	28	25
Lodash	10	10	11	81.59%	89.13%	7.54%	0	7	3
Moment.js	19	41	18	45.39%	48.59%	3.20%	1	13	5
Average	17	24	17	57.55%	70.12%	12.57%	1	9	7

Table 7.2: Overview of the benchmark metrics, achieved coverage, and statistical significance

The current version of the benchmark contains 99 *JavaScript* source code files which consist of popular *JavaScript* libraries that represent a diverse set of *JavaScript* syntax and code styles. Table 7.2 provides the main characteristics of the benchmark projects, including the number of `files`, the number of `units` (*i.e.,* exported classes or top-level functions), and the average Cyclomatic Complexity per file (**CC** column).

We used the state-of-the-art search algorithm *DynaMOSA* [58] and compared it against *random search* as a baseline. We use the algorithm parameter values as suggested in the *DynaMOSA* paper[9]. We set a search budget of 180 seconds as often used in related work [58, 120]. To account for the stochastic nature of search-based approaches, each file under test was run 20 times. This resulted in 8.25 d of consecutive running time (3960 runs × 180 s). The experiment was performed on a system with two AMD EPYC 7H12 (64 cores, 2.6 GHz) CPUs and 512 GB of RAM.

To determine if one approach performs better than the others, we applied the unpaired Wilcoxon signed-rank test [67] with a threshold of 0.05. This non-parametric statistical test determines if two data distributions are significantly different. In addition, we apply the Vargha-Delaney $\hat{A}_{12}$ statistic [68] to determine the effect size of the result, which determines the magnitude of the difference between the two data distributions.

The results of our evaluation can also be found in Table 7.2. It shows the average branch coverage per benchmark project achieved by *random search* and *DynaMOSA* and how they perform compared to each other. As can be seen in the table, *DynaMOSA* achieves an average branch coverage above 70 % for four out of six projects, and close to 50 % for the remaining two. As shown in related work, *DynaMOSA* achieves higher code coverage than *random search* for most `units` under test. Additionally, Table 7.2 shows the statistical results of the comparison between the two search algorithms with regard to branch coverage across the various benchmarks. This section of the table is organized into three main categories: #Win, #No Diff, and #Lose. Analyzing the #Win category, we observe notable results in favor of *DynaMOSA* in all benchmarks. The table shows that in 41 cases *DynaMOSA* wins significantly, in 6 cases *random search* wins significantly, and in 52 cases there is no significant difference in performance.

[9]`https://github.com/syntest-framework/syntest-framework/blob/3f6b9612c030ffc79d5e79c5c1c126ca816a87a6/tools/base-language/lib/presets/DynaMOSAPreset.ts`

7.4 Conclusion and Future Work

In this paper, we introduced *SynTest-Framework*, an open-source framework for automated test case generation and quality assurance. The framework provides an ecosystem of testing tools for multiple programming languages. It is built with a modular and extensible architecture that allows users to tailor the framework to their specific use case. We have also introduced two language-specific modules, namely *Syntest-Solidity* and *Syntest-JavaScript*, which are built on top of the framework. These modules provide automated test case generation for *Solidity* smart contracts and *JavaScript* programs, respectively. To evaluate the performance of *Syntest-Solidity* and *Syntest-JavaScript*, we performed an empirical study and the results show that both tools were effective at generating test cases for their respective programming languages.

As part of our future work, we plan to extend the framework with additional search algorithms (*e.g.,* SPEA2, PESA, PSO) and LLM-based approaches. To make it easier for researchers to evaluate new approaches, we plan to provide infrastructure within the framework needed to easily run and compare experiments. Furthermore, we plan to incorporate a mutation-testing engine to better evaluate the quality of the test cases. Lastly, to make the proposed tool easier to use for practitioners, we plan to integrate them into the most popular IDEs (*e.g.,* VSCode and WebStorm) and CI/CD platforms (*e.g.,* GitHub, GitLab).

8 Conclusion

In this last chapter, we revisit the research questions posed in Section 1.3, summarize our findings, and make conclusions based on the results of the studies. Furthermore, we showcase the current and possible implications that this work may bring, and provide recommendations for future directions of search-based automated test case generation.

8.1 Research Questions Revisited

In this section, we revisit and answer our five research questions from Section 1.3.

RQ1 *How can we reduce the search space for automated test case generation by using domain-specific knowledge?*

In Chapter 1, we described one of the main challenges of automated test case generation as the size of the search space that needs to be explored. This challenge is especially prevalent for automated test case generation of complex programs, which usually require highly structured, specific, and/or domain-dependent input data. To answer *RQ1*, we investigated how domain-specific knowledge can be leveraged to steer the search process toward better solutions. In particular, we focus on *highly structured input data* (Chapter 2) and *input validity constraints* (Chapter 3).

In Chapter 2, we introduced a novel approach that combines the strengths of grammar-based fuzzing and automated test case generation. The approach is based on the observation that most complex input data for a program under test is highly structured and can be described by a grammar. By using the grammar of the input language, we can generate input data that is more likely to be valid and thus reduce the search space for automated test case generation. More specifically, we use automated test case generation techniques to evolve the test case structure while grammar-based fuzzing is used to evolve the complex input. We have evaluated the feasibility and effectiveness of our approach by implementing it in *EvoSuite* and applying it to the three most popular JavaScript Object Notation (JSON) parser libraries for *Java*. The results show that our approach is able to generate test cases that achieve higher structural coverage than the state-of-the-art without negatively impacting the coverage of classes without highly structured complex input data.

In Chapter 3, we highlight the importance of input validity constraints for automated test case generation of security vulnerabilities. Input validity constraints are a construct that exist within some languages (*e.g., Solidity* and *Java*) and are used to validate the input data before it is processed. Declarative input validation constraints are a form of domain-specific knowledge that can be used to reduce the search space for automated test case generation. To this aim, we proposed an approach that provides search guidance for these input validity constraints based on interprocedural control dependency analysis, in addition to the well-established test adequacy criteria. We have evaluated the effectiveness of our approach by comparing it to the state-of-the-art automated test case generation approach for *Solidity* smart contracts. The results show that our approach is able to generate test cases that achieve significantly higher structural coverage and improve the vulnerability-detection capability. This demonstrates that the effectiveness of automated test case generation for security vulnerabilities is highly dependent on the ability to generate valid input data.

RQ2 *How can we preserve promising structures within test cases throughout the search process?*

We described the second challenge of automated test case generation that we address in this thesis as the need to identify and preserve promising patterns within the structure of the test cases. While current crossover operators can randomly create effective test cases by combining the structure of two parent test cases, they do not preserve promising patterns (*i.e.,* certain permutations of method calls) within the structures that are present in the parent test cases. We hypothesized that if these promising patterns are broken, the search process will waste valuable time trying potential solutions that will not work.

In Chapter 4, we aim to answer *RQ2* by investigating how promising patterns within the structure of test cases can be identified and preserved for REpresentational State Transfer (REST) Application Programming Interface (API) testing. We introduced an approach with an improved crossover operator, named *LT-MOSA*, that uses Agglomerative Hierarchical Clustering (AHC) to infer patterns within the method sequences of the fittest test cases and preserve them. To evaluate *LT-MOSA*, we implemented it within *EvoMaster* and compared it to *MIO* and *MOSA*, the state-of-the-art automated test case generation approaches for REST APIs. The results show that by inferring and preserving the promising patterns, our approach is able to generate test cases that achieve significantly higher structural coverage and fault-detection capability within the same time compared to the state-of-the-art approaches. This shows that preserving promising patterns within the structure of test cases can significantly improve the effectiveness of automated test case generation.

RQ3 *How can we increase the diversity of individuals in the population?*

The third challenge addressed by this thesis, is the need to increase the diversity of individuals in the population to prevent premature convergence. Crossover operators are one of the main sources of diversity for genetic algorithms. However, current state-of-the-art crossover operators for automated test case generation (*i.e.,* single-point) only change the structure of the test cases and simply copy over the corresponding test data.

Consequently, the input data of the offspring is often very similar to the input data of the parents. To answer *RQ3*, we investigate how the diversity of individuals in the population can be increased by using a hybrid crossover operator that combines different crossover operators on multiple levels (*i.e.,* data and structure level).

In Chapter 5, we introduced a hybrid crossover operator, named *HMX*, that combines the single-point crossover operator with a data-level crossover operator. We have evaluated *HMX* by comparing it to the state-of-the-art single-point crossover operator on 116 classes from the Apache Commons and Lucene Stemmer projects. The results show that our operator significantly improves the structural coverage and fault detection capability of the generated test cases compared to the single-point crossover operator. This demonstrates that combining different crossover operators on multiple levels can increase the diversity of the population.

RQ4 *How do we increase the effectiveness and efficiency of automated test case generation for dynamically-typed languages?*

One of the other challenges of automated test case generation was the lack of type information for dynamically-typed languages. Current state-of-the-art approaches for automated test case generation generally focus on statically-typed languages as type information is needed to select suitable values for the parameters of a method call. Without this type information, the search space grows exponentially, making the overall search process less effective. Therefore, to answer *RQ4*, we focus on integrating dynamic type information into the search process.

In Chapter 6, we propose a novel unsupervised probabilistic type inference approach that infers the types of the parameters of a method and uses these to guide the search process. To evaluate our approach, we performed an empirical study on 10 open-source *JavaScript* projects. The results show that unsupervised probabilistic type inference approach achieves a statistically significant increase in 56 % of the test files with up to 71 % of branch coverage compared to random type sampling. This shows that the inferred types can be used to guide the search process and improve the effectiveness of automated test case generation for dynamically-typed languages.

RQ5 *How can we make a platform for automated test case generation that supports multiple programming languages?*

The last challenge that we address is the shortage of easy-to-use production-level tooling. Automated test case generation is a complex process that requires domain-specific knowledge and expertise. Therefore, to make automated test case generation techniques more accessible to practitioners, we need to provide easy-to-use tools that are language agnostic. However, given the tight coupling between the tool and the programming language to be tested (within the context of white-box testing), it is not trivial to build tooling for multiple languages and/or testing frameworks. Therefore, to answer our last research questions (*RQ5*), we investigate how we can make a platform for automated test case generation that supports multiple programming languages.

In Chapter 7, we present *SynTest-Framework*, a modular and extensible framework for automated test case generation. The main goal of the framework is to provide an ecosystem of testing tools that are easy to use and come with a collection of pre-defined presets for different search algorithms. *SynTest-Framework* is designed to be language agnostic and can be extended to support new languages and testing frameworks. Additionally, we present *Syntest-Solidity* and *Syntest-JavaScript*, two testing tools integrated into the *SynTest-Framework* for automated test case generation of *Solidity* smart contracts and *JavaScript* programs. We evaluated both testing tools and showed that they are able to effectively generate test cases for *Solidity* smart contracts and *JavaScript* programs, respectively. This demonstrates that the *SynTest-Framework* can be used to build tools for automated test case generation of different programming languages. Both the framework and the tools have been made publicly available to the research and industry community using the Apache 2.0 license.

8.2 Implications

In this section, we discuss the implications of the results of this thesis for the research and industry community.

8.2.1 Research Implications

The results of this thesis have several implications for the research community.

Datasets

We have made the datasets used in this thesis publicly available to the research community. Traditionally, datasets were not considered as a contribution of a research paper and were often not shared [191]. However, in recent years, the importance of datasets has increased. With a *Data Showcase* track at conferences like Mining Software Repositories (MSR)[1] it is now considered a contribution by itself. Public datasets are important for the research community as they allow researchers to replicate the results of a study, compare the results of an approach over time, and compare different approaches on the same dataset.

Replication Packages

Next to the datasets, we have made the replication packages associated with this thesis publicly available. The replication packages contain all the necessary information to replicate the results of this thesis and allow researchers to build upon our work.

8.2.2 Practical Implications

The results of this thesis also have several practical implications for the research and industry community.

SynTest Ecosystem

Considering the importance of software testing, it is unfortunate that readily accessible, production-level tools for automated test case generation are lacking [21]. In this thesis, we introduced *SynTest-Framework*, a modular, extensible, and language-agnostic framework for automated test case generation. The primary goal of this framework is to create

[1]https://www.msrconf.org/

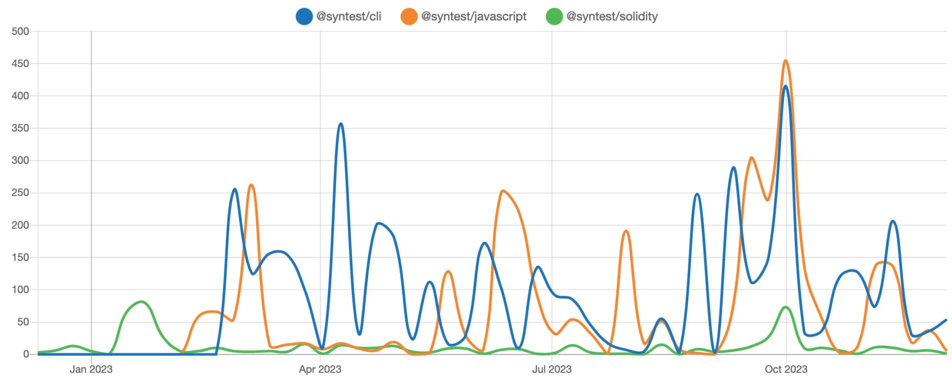


Figure 8.1: Number of downloads on NPM for the SynTest ecosystem

a user-friendly ecosystem of testing tools, open to expansion by researchers and practitioners who wish to incorporate additional search algorithms, language support, and/or testing frameworks. It currently offers testing capabilities for both *Solidity* smart contracts and *JavaScript* programs. Furthermore, we are currently working together with a different university to integrate their tool into the ecosystem.

By providing a cohesive ecosystem of testing tools for automated test case generation using a common interface, we hope to make search-based software testing techniques more accessible to practitioners. This way we can help bridge the gap between research and practice. In addition, we hope that the framework will encourage researchers to build upon our work and extend the ecosystem with novel techniques. The framework was designed with research in mind and can be easily extended without having to worry about the underlying infrastructure. Researchers can focus on the core of their work and leave the rest to the framework.

The *SynTest* ecosystem is publicly available under the Apache 2.0 license, accessible to both the research and industry communities. It has been downloaded over 1000 times from the NPM registry and has received 45 stars on GitHub. Figure 8.1 shows the number of downloads on NPM for the *SynTest* ecosystem over the last year.

In addition, we were invited to give a tutorial on the *SynTest* ecosystem at the Intl. Workshop on Search-Based and Fuzz Testing (SBFT) and the 4th International Workshop on Artificial Intelligence in Software Testing (AIST). Based on the feedback we received, we believe that the *SynTest* ecosystem has the potential to become a valuable tool for the research and industry community. Together with the organizers of the SBFT workshop, we are planning to start a tool competition around test generation for *JavaScript* and dynamic type inference based on the *SynTest* ecosystem.

EvoSuite Experiment Runner

In Chapters 2 and 5, we used *EvoSuite* as the platform to implement our approaches on for our empirical experiments. *EvoSuite*, however, does not provide an easy way to run experiments. Therefore, we have developed the *EvoSuite* experiment runner, a tool that allows researchers to easily run experiments with *EvoSuite* in a reproducible manner. It provides a simple interface to run experiments, collect results, and replicate studies. The tool is publicly available under the GNU GPLv3 license and has been contributed back to the original repository. We hope that by providing this tool, we can make it easier for other researchers to run their experiments on top of *EvoSuite*.

8.3 Future Work

This thesis has highlighted several key challenges of automated test case generation and proposed novel approaches to address them. However, there are still several ways to extend upon this work. In this section, we provide recommendations for future research directions of search-based automated test case generation.

8.3.1 Size of the Search Space

In Chapter 1, we mentioned that programs that require complex input data are usually highly structured, while generated input is often not. These implicit constraints make it challenging to generate valid input data. In this thesis, we have shown that by using domain-specific knowledge, we can reduce the search space for automated test case generation. However, this requires knowing the application domain beforehand. By investigating how we can automatically infer these constraints, we can further reduce the search space for automated test case generation.

8.3.2 Preserving Promising Structures

In Chapter 4, we have shown that using AHC (specifically UPGMA) to preserve promising patterns within the structure of test cases can significantly improve the effectiveness of automated test case generation. As part of a future research direction, it would be interesting to explore how other types of hierarchical clustering algorithms perform for this use case. Additionally, with the rise of Large Language Models (LLMs) for practical applications, it is worth investigating if these advanced language models can be used to identify promising structures within test cases based on the context of the program under test. As discussed in Chapter 1, LLMs are not yet mature enough for the purpose of automated test case generation [48–51], however, they have been successfully used to modify test cases [192]. Another interesting direction is to investigate if existing real-world usage patterns of the program under test (*e.g.,* manually-written test cases, API usage, or documentation) can be used to identify promising structures within test cases.

8.3.3 Population Diversity

In Chapter 1, we explained that the crossover and mutation operators are two of the main sources of diversity for genetic algorithms. Similar to the previous recommendation, it would be interesting to investigate if LLMs can be used to replace or augment the crossover and mutation operators. With their vast training data, LLMs might be able to understand the structure of the test cases and generate new ones with increased diversity.

8.3.4 Dynamically-Typed Language Support

In Chapter 6, we have shown that by using unsupervised probabilistic type inference, we can increase the effectiveness of automated test case generation for dynamically-typed languages. This technique relies heavily on the static analysis performed before the search process to identify the relations between different variables in the code. It would be integrating to investigate if the already dynamic nature of search-based approaches can be used to improve the inferred model based on the execution results of the generated test.

8.3.5 Tooling

In Chapter 7, we have presented an ecosystem of testing tools for automated test case generation that can generate test cases for both *Solidity* smart contracts and *JavaScript* programs. However, there are still many other programming languages and testing frameworks that could benefit from automated test case generation. Therefore, it would be interesting to investigate the limitations of implementing automated test case generations for multiple programming languages within a single language, given the tight coupling between the tool and the programming language to be tested.

Bibliography

References

[1] Mitchell Olsthoorn. More effective test case generation with multiple tribes of AI. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ACM, may 2022.

[2] Frederick P Brooks Jr. *The mythical man-month: essays on software engineering*. Pearson Education, 1995.

[3] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2015.

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

[5] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering*, 46(12):1294–1317, dec 2020.

[6] Urko Rueda Molina, Fitsum Kifetew, and Annibale Panichella. Java unit testing tool competition. In *Proceedings of the 11th International Workshop on Search-Based Software Testing*. ACM, may 2018.

[7] Annibale Panichella and Urko Rueda Molina. Java unit testing tool competition - fifth round. In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*. IEEE, may 2017.

[8] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. *Empirical Software Engineering*, 20(3):611–639, nov 2013.

[9] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 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, may 2017.

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

[11] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying search based software engineering with sapienz at facebook. In *Search-Based Software Engineering*, pages 3–45. Springer International Publishing, 2018.

[12] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 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*. ACM, aug 2016.

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

[14] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, aug 2013.

[15] Mauricio Aniche, Christoph Treude, and Andy Zaidman. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[16] George Candea and Patrice Godefroid. *Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances*, pages 505–531. Springer International Publishing, 2019.

[17] Sheeva Afshan, Phil McMinn, and Mark Stevenson. 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, mar 2013.

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

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

[20] Matteo Brunetto, Giovanni Denaro, Leonardo Mariani, and Mauro Pezzè. On introducing automatic test case generation in practice: A success story and lessons learned. *Journal of Systems and Software*, 176:110933, June 2021.

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

[22] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, dec 2001.

[23] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering. *ACM Computing Surveys*, 45(1):1–61, nov 2012.

[24] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. 2009.

[25] Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. *Search Based Requirements Optimisation: Existing Work and Challenges*, pages 88–94. Springer Berlin Heidelberg.

[26] Outi Räihä. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, November 2010.

[27] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, November 2010.

[28] Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, April 2008.

[29] Michael Mohan and Des Greer. A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development*, 6(1), February 2018.

[30] Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE '07)*. IEEE, may 2007.

[31] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, pages 1–59. Springer Berlin Heidelberg, 2012.

[32] M. Harman and J. Clark. Metrics are fitness functions too. In *10th International Symposium on Software Metrics, 2004. Proceedings.* IEEE.

[33] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, mar 2011.

[34] Gordon Fraser and Andrea 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*. ACM, sep 2011.

[35] S Phani Shashank, Praneeth Chakka, and D Vijay Kumar. A systematic literature survey of integration testing in component-based software engineering. In *2010 International Conference on Computer and Communication Technology (ICCCT)*. IEEE, September 2010.

[36] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. Generating class-level integration tests using call site information. *IEEE Transactions on Software Engineering*, 49(4):2069–2087, apr 2023.

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

[38] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA07. ACM, July 2007.

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

[40] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21(3):445–478, July 2012.

[41] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, September 2008.

[42] Mohd Ehmer Khan. Different forms of software testing techniques for finding errors. *International Journal of Computer Science Issues (IJCSI)*, 7(3):24, 2010.

[43] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, December 2001.

[44] David W. Binkley, Mark Harman, and Kiran Lakhotia. FlagRemover: A testability transformation for transforming loop-assigned flags. *ACM Transactions on Software Engineering and Methodology*, 20(3):1–33, aug 2011.

[45] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song 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*. ACM, jul 2020.

[46] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. *ACM SIGSOFT Software Engineering Notes*, 29(4):108–118, jul 2004.

[47] Mark Harman, André Baresel, David Binkley, Robert Hierons, Lin Hu, Bogdan Korel, Phil McMinn, and Marc Roper. Testability transformation – program transformation to improve testability. In *Formal Methods and Testing*, pages 320–344. Springer Berlin Heidelberg.

[48] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *arXiv preprint arXiv:2302.06527*, 2023.

[49] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Exploring the effectiveness of large language models in generating unit tests. *arXiv preprint arXiv:2305.00418*, 2023.

[50] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. Effective test generation using pre-trained large language models and mutation testing. *arXiv preprint arXiv:2308.16557*, 2023.

[51] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. Llm is like a box of chocolates: the non-determinism of chatgpt in code generation. *arXiv preprint arXiv:2308.02828*, 2023.

[52] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering*, 27(7), sep 2022.

[53] Stephanie Forrest and Melanie Mitchell. *Relative Building-Block Fitness and the Building-Block Hypothesis*, pages 109–126. Elsevier, 1993.

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

[55] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys*, 45(3):1–33, June 2013.

[56] Dirk Sudholt. *The Benefits of Population Diversity in Evolutionary Algorithms: A Survey of Rigorous Runtime Analyses*, pages 359–404. Springer International Publishing, November 2019.

[57] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[58] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, feb 2018.

[59] Kiran Lakhotia, Mark Harman, and Hamilton Gross. AUSTIN: An open source tool for search based software testing of c programs. *Information and Software Technology*, 55(1):112–125, jan 2013.

[60] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. In *Search-Based Software Engineering*, pages 9–24. Springer International Publishing, 2020.

[61] Herbert A. Simon. The science of design: Creating the artificial. *Design Issues*, 4(1/2):67, 1988.

[62] Roel Wieringa. Design science methodology: principles and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10. ACM, May 2010.

[63] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[64] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[65] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, may 2011.

[66] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, nov 2012.

[67] William Jay Conover. *Practical nonparametric statistics*, volume 350. john wiley & sons, 1999.

[68] András Vargha and Harold D. Delaney. A critique and improvement of the <i>CL</i> common language effect size statistics of McGraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, jun 2000.

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

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

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

[72] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[73] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. Characterizing transaction-reverting statements in ethereum smart contracts. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, nov 2021.

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

[75] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, dec 2020.

[76] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. Guiding automated test case generation for transaction-reverting statements in smart contracts. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, oct 2022.

[77] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Improving test case generation for REST APIs through hierarchical clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, nov 2021.

[78] Mitchell Olsthoorn, Pouria Derakhshanfar, and Annibale Panichella. Hybrid multi-level crossover for unit test case generation. In *Search-Based Software Engineering*, pages 72–86. Springer International Publishing, 2021.

[79] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Guess what: Test case generation for javascript with unsupervised probabilistic type inference. In *Search-Based Software Engineering*, pages 67–82. Springer International Publishing, 2022.

[80] Mitchell Olsthoorn, Dimitri Stallenberg, Arie van Deursen, and Annibale Panichella. SynTest-solidity. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ACM, may 2022.

[81] Mitchell Olsthoorn, DM Stallenberg, and Annibale Panichella. Syntest-javascript: Automated unit-level test case generation for javascript. In *2024 ACM/IEEE International Workshop on Search-Based and Fuzz Testing*. ACM/IEEE, 2024.

[82] Open science, Oct 2023.

[83] Sam Parsons, Flávio Azevedo, Mahmoud M. Elsherif, Samuel Guay, Owen N. Shahim, Gisela H. Govaart, Emma Norris, Aoife O'Mahony, Adam J. Parker, Ana Todorovic, Charlotte R. Pennington, Elias Garcia-Pelegrin, Aleksandra Lazić, Olly Robertson, Sara L. Middleton, Beatrice Valentini, Joanne McCuaig, Bradley J. Baker, Elizabeth Collins, Adrien A. Fillon, Tina B. Lonsdorf, Michele C. Lim, Norbert Vanek, Marton Kovacs, Timo B. Roettger, Sonia Rishi, Jacob F. Miranda, Matt Jaquiery, Suzanne L. K. Stewart, Valeria Agostini, Andrew J. Stewart, Kamil Izydorczak, Sarah Ashcroft-Jones, Helena Hartmann, Madeleine Ingham, Yuki Yamada, Martin R. Vasilev, Filip Dechterenko, Nihan Albayrak-Aydemir, Yu-Fang Yang, Annalise A. LaPlume, Julia K. Wolska, Emma L. Henderson, Mirela Zaneva, Benjamin G. Farrar, Ross Mounce, Tamara Kalandadze, Wanyin Li, Qinyu Xiao, Robert M. Ross, Siu Kit Yeung, Meng Liu, Micah L. Vandegrift, Zoltan Kekecs, Marta K. Topor, Myriam A. Baum, Emily A. Williams, Asma A. Assaneea, Amélie Bret, Aidan G. Cashin, Nick Ballou, Tsvetomira Dumbalska, Bettina M. J. Kern, Claire R. Melia, Beatrix Arendt, Gerald H. Vineyard, Jade S. Pickering, Thomas R. Evans, Catherine Laverty, Eliza A. Woodward, David Moreau, Dominique G. Roche, Eike M. Rinke, Graham Reid, Eduardo Garcia-Garzon, Steven Verheyen, Halil E. Kocalar, Ashley R. Blake, Jamie P. Cockcroft, Leticia Micheli, Brice Beffara Bret, Zoe M. Flack, Barnabas Szaszi, Markus Weinmann, Oscar Lecuona, Birgit Schmidt, William X. Ngiam, Ana Barbosa Mendes,

Shannon Francis, Brett J. Gall, Mariella Paul, Connor T. Keating, Magdalena Grose-Hodge, James E. Bartlett, Bethan J. Iley, Lisa Spitzer, Madeleine Pownall, Christopher J. Graham, Tobias Wingen, Jenny Terry, Catia Margarida F. Oliveira, Ryan A. Millager, Kerry J. Fox, Alaa AlDoh, Alexander Hart, Olmo R. van den Akker, Gilad Feldman, Dominik A. Kiersz, Christina Pomareda, Kai Krautter, Ali H. Al-Hoorie, and Balazs Aczel. A community-sourced glossary of open scholarship terms. *Nature Human Behaviour*, 6(3):312–318, February 2022.

[84] Mitchell Olsthoorn and Annibale Panichella. Replication package of "generating highly-structured input data by combining search-based testing and grammar-based fuzzing", 2020.

[85] Mitchell Olsthoorn. mitchellolsthoorn/icsme-research-2022-syntest-security-conditions-replication: v0.1.1, 2023.

[86] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Replication package of "improving test case generation for rest apis through hierarchical clustering", 2021.

[87] Mitchell Olsthoorn, Pouria Derakhshanfar, and Annibale Panichella. Replication package of "hybrid multi-level crossover for unit test case generation", 2021.

[88] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Replication package of "guess what: Test case generation for javascript with unsupervised probabilistic type inference", 2022.

[89] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. Empirical evaluation of smart contract testing: what is the best choice? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2021.

[90] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J.G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A.C 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The fair guiding principles for scientific data management and stewardship. *Scientific Data*, 3(1), March 2016.

[91] Annika Jacobsen, Ricardo de Miranda Azevedo, Nick Juty, Dominique Batista, Simon Coles, Ronald Cornet, Mélanie Courtot, Mercè Crosas, Michel Dumontier,

Chris T. Evelo, Carole Goble, Giancarlo Guizzardi, Karsten Kryger Hansen, Ali Hasnain, Kristina Hettne, Jaap Heringa, Rob W.W. Hooft, Melanie Imming, Keith G. Jeffery, Rajaram Kaliyaperumal, Martijn G. Kersloot, Christine R. Kirkpatrick, Tobias Kuhn, Ignasi Labastida, Barbara Magagna, Peter McQuilton, Natalie Meyers, Annalisa Montesanti, Mirjam van Reisen, Philippe Rocca-Serra, Robert Pergl, Susanna-Assunta Sansone, Luiz Olavo Bonino da Silva Santos, Juliane Schneider, George Strawn, Mark Thompson, Andra Waagmeester, Tobias Weigel, Mark D. Wilkinson, Egon L. Willighagen, Peter Wittenburg, Marco Roos, Barend Mons, and Erik Schultes. Fair principles: Interpretations and implementation considerations. *Data Intelligence*, 2(1–2):10–29, January 2020.

[92] Lisette Veldkamp, Mitchell Olsthoorn, and Annibale Panichella. Grammar-based evolutionary fuzzing for JSON-RPC APIs. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, may 2023.

[93] Dirk Thierens and Peter A.N. Bosman. Optimal mixing evolutionary algorithms. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, jul 2011.

[94] Mitchell Olsthoorn, Pouria Derakhshanfar, and Xavier Devroey. An application of model seeding to search-based unit test generation for gson. In *Search-Based Software Engineering*, pages 239–245. Springer International Publishing, 2020.

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

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

[97] FP Brooks and Teh mythical Man-Month. Essays in software engineering. *Adisson-Wesley, Reading, Mass*, 1975.

[98] Sadeeq Jan, Cu D. Nguyen, and Lionel Briand. 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, aug 2015.

[99] I. S. Wishnu B. Prasetya. T3, a combinator-based random testing tool for java: Benchmarking. In *Future Internet Testing*, pages 101–110. Springer International Publishing, 2014.

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

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

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

[103] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of {TLS} implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, 2015.

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

[105] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. IFuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Computer Security – ESORICS 2016*, pages 581–601. Springer International Publishing, 2016.

[106] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering*, 22(2):928–961, jan 2016.

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

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

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

[110] Maximilian Wohrer and Uwe Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, mar 2018.

[111] Vimal Dwivedi, Vipin Deval, Abhishek Dixit, and Alex Norta. Formal-verification of smart-contract languages: A survey. In *Communications in Computer and Information Science*, pages 738–747. Springer Singapore, 2019.

[112] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, jun 2020.

[113] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. ConFuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&amp;P)*. IEEE, sep 2021.

[114] Bo Jiang, Ye Liu, and W. K. Chan. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, sep 2018.

[115] Valentin Wüstholz and Maria Christakis. Harvey: a greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, nov 2020.

[116] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, nov 2019.

[117] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. ReGuard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, may 2018.

[118] Stefan Driessen, Dario Di Nucci, Geert Monsieur, Damian A. Tamburri, and Willem-Jan van den Heuvel. Automated test-case generation for solidity smart contracts: the agsolt approach and its evaluation. February 2021.

[119] Expressions and control structures.

[120] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2015.

[121] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[122] Jiaming Ye, Mingliang Ma, Yun Lin, Yulei Sui, and Yinxing Xue. Clairvoyance: cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. ACM, jun 2020.

[123] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. ContractWard: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 8(2):1133–1144, apr 2021.

[124] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep*, pages 1–41, 2018.

[125] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. DefectChecker: Automated smart contract defect detection by analyzing EVM bytecode. *IEEE Transactions on Software Engineering*, 48(7):2189–2207, jul 2022.

[126] Christof Ferreira Torres, Mathis Steichen, et al. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1591–1607, 2019.

[127] Jian Zhu, Kai Hu, Mamoun Filali, Jean-Paul Bodeveix, and Jean-Pierre Talpin. Formal verification of solidity contracts in event-b. May 2020.

[128] Shaukat Ali, Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel C. Briand. Generating test data from OCL constraints with search techniques. *IEEE Transactions on Software Engineering*, 39(10):1376–1402, oct 2013.

[129] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, nov 2015.

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

[131] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[132] Xavier Devroey, Sebastiano Panichella, and Alessio Gambi. Java unit testing tool competition: Eighth round. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. ACM, jun 2020.

[133] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. SBST tool competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, may 2021.

[134] Man Rai. Robust optimal aerodynamic design using evolutionary methods and neural networks. In *42nd AIAA Aerospace Sciences Meeting and Exhibit*, page 778, 2004.

[135] Kalyanmoy Deb and ayan Deb. Analysing mutation schemes for real-parameter genetic algorithms. *International Journal of Artificial Intelligence and Soft Computing*, 4(1):1, 2014.

[136] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7(2):145–170, jun 2006.

[137] Hamidreza Eskandari and Christopher D. Geiger. A fast pareto genetic algorithm approach for solving expensive multiobjective optimization problems. *Journal of Heuristics*, 14(3):203–241, sep 2007.

[138] Helen G Cobb, John J Grefenstette, et al. Genetic algorithms for tracking changing environments. In *ICGA*, volume 1993, pages 523–530, 1993.

[139] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *Parallel Problem Solving from Nature PPSN VI*, pages 849–858. Springer Berlin Heidelberg, 2000.

[140] Andrea Arcuri. Test suite generation with the many independent objective (MIO) algorithm. *Information and Software Technology*, 104:195–206, dec 2018.

[141] Leonardo Alt and Christian Reitwiessner. SMT-based verification of solidity smart contracts. In *Lecture Notes in Computer Science*, pages 376–388. Springer International Publishing, 2018.

[142] Patrick Chapman, Dianxiang Xu, Lin Deng, and Yin Xiong. Deviant: A mutation testing tool for solidity smart contracts. In *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, jul 2019.

[143] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, mar 2002.

[144] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[145] Valentina Lenarduzzi and Annibale Panichella. Serverless testing: Tool vendors' and experts' points of view. *IEEE Software*, 38(1):54–60, jan 2021.

[146] Valentina Lenarduzzi, Jeremy Daly, Antonio Martini, Sebastiano Panichella, and Damian Andrew Tamburri. Toward a technical debt conceptualization for serverless computing. *IEEE Software*, 38(1):40–47, jan 2021.

[147] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. Resource-based test case generation for RESTful web services. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, jul 2019.

[148] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, jul 2016.

[149] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. RESTTESTGEN: Automated black-box testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, oct 2020.

[150] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, apr 2002.

[151] Richard A. Watson, Gregory S. Hornby, and Jordan B. Pollack. Modeling building-block interdependency. In *Lecture Notes in Computer Science*, pages 97–106. Springer Berlin Heidelberg, 1998.

[152] Martin Pelikan and David E Goldberg. Escaping hierarchical traps with competent genetic algorithms. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pages 511–518, 2001.

[153] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. Boa: The bayesian optimization algorithm. In *Proceedings of the genetic and evolutionary computation conference GECCO-99*, volume 1. Citeseer, 1999.

[154] Tian-Li Yu, David E. Goldberg, Kumara Sastry, Claudio F. Lima, and Martin Pelikan. Dependency structure matrix, genetic algorithms, and effective recombination. *Evolutionary Computation*, 17(4):595–626, dec 2009.

[155] Michal W. Przewozniczek and Marcin M. Komarnicki. Empirical linkage learning. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. ACM, jul 2020.

[156] Dirk Thierens. The linkage tree genetic algorithm. In *Parallel Problem Solving from Nature, PPSN XI*, pages 264–273. Springer Berlin Heidelberg, 2010.

[157] Anton Bouter, Tanja Alderliesten, Cees Witteveen, and Peter A. N. Bosman. Exploiting linkage information in real-valued optimization with the real-valued gene-pool optimal mixing evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, jul 2017.

[158] Peter A.N. Bosman, Ngoc Hoang Luong, and Dirk Thierens. Expanding from discrete cartesian to permutation gene-pool optimal mixing evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. ACM, jul 2016.

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

[160] Fitash Ul Haq, Donghwan Shin, Lionel C. Briand, Thomas Stifter, and Jun Wang. Automatic test suite generation for key-points detection DNNs using many-objective search (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2021.

[161] J. Martin Bland and DouglasG. Altman. STATISTICAL METHODS FOR ASSESSING AGREEMENT BETWEEN TWO METHODS OF CLINICAL MEASUREMENT. *The Lancet*, 327(8476):307–310, feb 1986.

[162] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

[163] SN Sivanandam and SN Deepa. *Genetic algorithms*. Springer, 2008.

[164] J David Schaffer, Rich Caruana, Larry J Eshelman, and Rajarshi Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proceedings of the 3rd international conference on genetic algorithms*, pages 51–60, 1989.

[165] J. E. Smith and T. C. Fogarty. Adaptively parameterised evolutionary systems: Self adaptive recombination and mutation in a genetic algorithm. In *Parallel Problem Solving from Nature — PPSN IV*, pages 441–450. Springer Berlin Heidelberg, 1996.

[166] Andrea Arcuri and Juan P. Galeotti. Testability transformations for existing APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, oct 2020.

[167] Andrea Arcuri. Many independent objective (MIO) algorithm for test suite generation. In *Search Based Software Engineering*, pages 3–17. Springer International Publishing, 2017.

[168] Gregory Gay. Generating effective test suites by combining coverage criteria. In *Search Based Software Engineering*, pages 65–82. Springer International Publishing, 2017.

[169] Kalyanmoy Deb, Karthik Sindhya, and Tatsuya Okabe. Self-adaptive simulated binary crossover for real-parameter optimization. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, jul 2007.

[170] Maurício Aniche. Ck calculator v0.0.6, 2015.

[171] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of automated unit test generation for python, 2021.

[172] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4py. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, may 2022.

[173] L.A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, sep 1976.

[174] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):1–39, may 2018.

[175] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, pages 320–329. Springer Berlin Heidelberg, 2004.

[176] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, feb 2013.

[177] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, may 2017.

[178] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *ECOOP 2005 - Object-Oriented Programming*, pages 428–452. Springer Berlin Heidelberg, 2005.

[179] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. Type inference for static compilation of JavaScript. *ACM SIGPLAN Notices*, 51(10):410–429, oct 2016.

[180] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from ”big code”. *ACM SIGPLAN Notices*, 50(1):111–124, jan 2015.

[181] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, oct 2018.

[182] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. JSEFT: Automated javascript unit test generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2015.

[183] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, may 2011.

[184] Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: automatic symbolic testing of JavaScript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, nov 2014.

[185] Hideo Tanida, Tadahiro Uehara, Guodong Li, and Indradeep Ghosh. Automated unit testing of javascript code through symbolic executor symjs. *International Journal on Advances in Software*, 8(1):146–155, 2015.

[186] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Efficient JavaScript mutation testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, mar 2013.

[187] Phillip Heidegger and Peter Thiemann. Contract-driven testing of JavaScript code. In *Objects, Models, Components, Patterns*, pages 154–172. Springer Berlin Heidelberg, 2010.

[188] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[189] Stephan Lukasczyk and Gordon Fraser. Pynguin: automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ACM, may 2022.

[190] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Pythia: Generating test cases with oracles for javascript applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, November 2013.

[191] Ronald Dekker. The importance of having data-sets. 2006.

[192] Gregory Gay. *Improving the Readability of Generated Tests Using GPT-4 and ChatGPT Code Interpreter*, pages 140–146. Springer Nature Switzerland, December 2023.

Glossary

AHC Agglomerative Hierarchical Clustering (AHC) is a hierarchical clustering algorithm that builds a hierarchy of clusters. It starts with each element as a separate cluster and then repeatedly merges the closest clusters until only one cluster remains.

AI Artificial Intelligence (AI) is intelligence demonstrated by machines, unlike the natural intelligence displayed by humans and animals, which involves consciousness and emotionality.

API An Application Programming Interface (API) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API.

CC Cyclomatic Complexity (CC) is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code.

CFG A Control Flow Graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a method during its execution.

CSV Comma Separated Values (CSV) is a delimited text file that uses a comma to separate values. A CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas.

DSRM Design Science Research Methodology (DSRM) is a research methodology for information systems research.

EA An Evolutionary Algorithm (EA) is a subset of evolutionary computation, a generic population-based metaheuristic optimization algorithm. An EA uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection.

EMB The EvoMaster Benchmark (EMB) is a benchmark for EvoMaster, a tool for automated test case generation for REST APIs.

GA A Genetic Algorithm (GA) is a subset of evolutionary algorithms, a generic population-based metaheuristic optimization algorithm. A GA uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection.

HMX Hybrid Multi-level Crossover (HMX) is a crossover operator for evolutionary algorithms described in Chapter 5.

HTTP Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting hypermedia documents, such as HTML. It was designed for communication between web browsers and web servers, but it can also be used for other purposes. HTTP follows a classical client-server model, with a client opening a connection to make a request, then waiting until it receives a response. HTTP is a stateless protocol, meaning that the server does not keep any data (state) between two requests. Though often based on a TCP/IP layer, it can be used on any reliable transport layer, that is, a protocol that doesn't lose messages silently, such as UDP.

IDE An Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of at least a source code editor, build automation tools, and a debugger.

JSON JavaScript Object Notation (JSON) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values). It is a very common data format, with a diverse range of applications, such as serving as a replacement for XML in AJAX systems.

LLM A Large Language Model (LLM) is a language model that is trained on a large corpus of text.

REST REpresentational State Transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, called RESTful Web services, provide interoperability between computer systems on the Internet. RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations.

RPC A Remote Procedure Call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC is called remote invocation or remote method invocation.

RS Random Search (RS) is a search algorithm that randomly samples the search space.

SBSE Search-Based Software Engineering (SBSE) is the application of metaheuristic search techniques to software engineering problems.

SBST Search-Based Software Testing (SBST) is the application of metaheuristic search techniques to software testing problems.

SUT The System Under Test (SUT) is the system that is being tested.

XML Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The design goals of XML emphasize simplicity, generality, and usability across the Internet.

Curriculum Vitæ

Mitchell Jean Gijsbert Olsthoorn

09/04/1994	Date of birth in Houten, The Netherlands

Education

9/2019-12/2023	Ph.D. Student, Software Engineering Research Group, Delft University of Technology, The Netherlands, *More Effective Test Case Generation with Multiple Tribes of AI* Promotors: Prof. Dr. A. van Deursen and Dr. A. Panichella
2/2017-9/2019	M.Sc. Computer Software, Delft University of Technology, Delft, The Netherlands
9/2013-11/2016	B.Sc. Computer Software & Engineering, Delft University of Technology, Delft, The Netherlands
9/2008-6/2013	VWO (Pre-university Education), Oosterlicht College, Nieuwegein, The Netherlands

Academic Service

Co-organizer	GI-Dagstuhl Seminar 23103 on "Testing and Debugging of Data Analysis Workflows"[2] 2023
Chair	Web Chair, IEEE International Conference on Software Testing, Verification and Validation (ICST) 2022
	🏆 Distinguished organizing committee member award 2022 edition of the International Conference on Software Testing, Verification and Validation (ICST)

[2]https://www.dagstuhl.de/seminars/seminar-calendar/seminar-details/23103

Program Committee Member	Intl. Workshop on Search-Based and Fuzz Testing (SBFT) 2023
	IEEE International Conference on Software Testing, Verification and Validation (ICST) 2022
	IEEE/ACM International Conference on Automated Software Engineering (ASE) 2022, Artifact Evaluation
	Symposium on Search Based Software Engineering (SSBSE) 2022
	Alice & Eve 2022
	Intl. Workshop on Search-Based Software Testing (SBST) 2021, 2022
Reviewer	Automated Software Engineering (ASE) 2022
	Empirical Software Engineering (EMSE) 2021, 2022, 2023
	ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) 2021
	Journal of Software: Evolution and Process (JSME) 2022, 2023
	IEEE Transactions on Reliability (IEEE-TR) 2022
	ACM Transactions on Software Engineering and Methodology (TOSEM) 2022, 2023
	IEEE Transactions on Software Engineering (TSE) 2021, 2023
(Co-)Supervisor	Yehor Kozyr's Research Project on Reinforcement Learning for Automated Test Case Generation, 2022-2023
	Longfei Lin's Master's Thesis "The Impact of Test Case Clustering on Comprehending Automatically Generated Test Suites", 2022-2023
	Calin Georgescu's Master's Thesis "Test Program-Based Generative Fuzzing for Differential Testing of the Kotlin Compiler", 2022-2023

Diego Viero’s Bachelor’s Thesis “Is PSO a valid option for search-based test case generation in the context of dynamically-typed languages?”, 2023

Sergey Datskiv’s Bachelor’s Thesis “Can RVEA with DynaMOSA features perform well at generating test cases?”, 2023

Erwin Li’s Bachelor’s Thesis “Investigating the performance of SPEA-II on automatic test case generation”, 2023

Apoorva Abhishek’s Bachelor’s Thesis “Performance of the Pareto Envelope-Based Search Algorithm - II in Automated Test Case Generation”, 2023

Alp Güneri’s Bachelor’s Thesis “Augmenting Pareto Corner Search Evolutionary Algorithm for Automatic Test Case Generation”, 2023

Martin Mladenov’s Honours Project “Identification of Differences Between Parsers and Specification Using Grammar-Based Fuzzing”, 2022-2023

Dimitri Stallenberg’s Master’s Thesis “Automated Test Case Generation using Unsupervised Type Inference for JavaScript”, 2021-2022

Lisette Veldkamp’s Master’s Thesis “Blockchains and Security: Grammar-Based Evolutionary Fuzzing for JSON-RPC APIs and the Division of Responsibilities”, 2020-2022

Ivan Stranski’s Bachelor’s Thesis “Training a Machine-Learning Model for Optimal Fitness Function Selection with the Aim of Finding Bugs”, 2022

Daniela Toader’s Bachelor’s Thesis “Machine-Learning for Optimal Fitness Function Selection in Automated Testing”, 2022

Stoyan Dimitrov’s Bachelor’s Thesis “Training a Machine-Learning Model for Optimal Fitness Function Selection with the Aim of Finding Bugs”, 2022

Toon Kling’s Bachelor’s Thesis “Improving Automatic Test Case Generation by predicting optimal Fitness Functions”, 2022

	Pandelis Symeonidis's Bachelor's Thesis "Log-Based Behavioral System Model Inference Using Reinforcement Learning", 2021
	Thomas Werthenbach's Bachelor's Thesis "Inferring Log-Based Behavioural System Models using Markov Chains", 2021
	Calin Georgescu's Bachelor's Thesis "Modeling System Behaviour from Log Analysis Using Meta-Heuristic Search", 2021
	Tommaso Brandirali's Bachelor's Thesis "Inferring DFAs from Log Traces Using Community Detection", 2021
	Dimitri Stallenberg's Bachelor's Thesis "Preserving Inter-gene Relations During Test Case Generation using Intelligent Evolutionary Operators", 2020
	Chiel de Vries's Bachelor's Thesis "Improving Test Case Generation for RESTful APIs through Seeded Sampling", 2020
	Jeroen Kappé's Bachelor's Thesis "Multi-Objective Hill Climbing for Automated RESTful API Test Case Generation", 2020
	Michael Kemna's Bachelor's Thesis "Guiding automated system test generation for RESTful APIs using log statements", 2020
	Francis Behnen's Bachelor's Thesis "Independent verification of validator performance in the XRP ledger", 2020
	Marijn Roelvink's Bachelor's Thesis "Log inference on the Ripple Protocol: testing the system with an empirical approach", 2020
	Wolf Bubberman's Bachelor's Thesis "TLS MITM attack on the Ripple XRP Ledger", 2020
	Sengim Karayalçin's Bachelor's Thesis "Improving rippled: Leveraging passive model inference techniques to test large decentralized systems", 2020
Teaching Assistant	Artificial Intelligence for Software Testing and Reverse Engineering (CS4110 - 1st-year Master's Course), 2022-2024

Software Engineering Methods (CSE2115 - 2nd-year Bachelor's Course), 2018-2024

Blockchain Engineering (CS4160 - 1st-year Master's Course), 2019-2020

Context Project (TI2806 - 2nd-year Bachelor's Course), 2018-2019

Signal Processing (TI2716-A - 2nd-year Bachelor's Course), 2018-2019

Object-Oriented Programming Project (CSE1105 - 1st-year Bachelor's Course), 2018-2019

Computer Networks (CSE1405 - 1st-year Bachelor's Course), 2017-2019

Invited Lectures, Talks & Tutorials

Guest Lectures	*Automated Software Engineering* at Dr. Xavier Devroey's Software Engineering course at the University of Namur, Namur, Belgium, 4.5.2022
	On the Challenges of Automated Test Case Generation and How To Overcome Them at Humboldt University of Berlin, Berlin, Germany, 16.6.2022
Talks	*SynTest-JavaScript: Automated Unit-Level Test Case Generation for JavaScript* at Intl. Workshop on Search-Based and Fuzz Testing (SBFT), Lisbon, Portugal, 2024
	Grammar-Based Evolutionary Fuzzing for JSON-RPC APIs at Intl. Workshop on Search-Based and Fuzz Testing (SBFT), Melbourne, Australia, 2023
	Testing and Debugging of Data Analysis Workflows at Dagstuhl, Wadern, Germany, 5.3.2023
	SynTest-JavaScript for Jetbrains, Delft, The Netherlands, 28.11.2022

Guess What: Test Case Generation for JavaScript With Unsupervised Probabilistic Type Inference at Symposium on Search Based Software Engineering (SSBSE), Singapore, 2022

Guiding Automated Test Case Generation for Transaction-Reverting Statements in Smart Contracts at IEEE International Conference on Software Maintenance and Evolution (ICSME), Limassol, Cyprus, 2022

SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts at International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 2022

More Effective Test Case Generation with Multiple Tribes of AI at International Conference on Software Engineering (ICSE), online, 2022

Improving Test Case Generation for REST APIs Through Hierarchical Clustering at IEEE/ACM International Conference on Automated Software Engineering (ASE), online, 2021

Blockchain and its Impact on Decentralized Computing at blockchain workshop, Utrecht, The Netherlands, 1.11.2021

SynTest-Solidity: Fuzzing and Test Case Generation for Solidity Smart Contracts at UBRIConnect, online, 12.10.2021

Multi-Objective Test Case Selection Through Linkage Learning-Driven Crossover at Symposium on Search Based Software Engineering (SSBSE), online, 2021

Testing Frameworks for XRP Ledger and Beyond at UBRIConnect, online, 12.10.2020

An Application of Model Seeding to Search-Based Unit Test Generation for Gson at Symposium on Search Based Software Engineering (SSBSE), online, 2020

Generating Highly-Structured Input Data by Combining Search-Based Testing and Grammar-Based Fuzzing at IEEE/ACM International Conference on Automated Software Engineering (ASE), online, 2020

Tutorials	*A Hands-on Tutorial for Automatic Test Case Generation and Fuzzing for JavaScript* at 4th International Workshop on Artificial Intelligence in Software Testing (AIST), Toronto, Canada, 2024
	Getting Started with SynTest-Framework: A Hands-on Tutorial for Automatic Test Case Generation and Fuzzing at Intl. Workshop on Search-Based and Fuzz Testing (SBFT), Lisbon, Portugal, 2024

Implemented Tools

2020-current	**EvoSuite Experiment Runner**[3]: A bash- and Docker-based infrastructure for running large scale experiments with EvoSuite.
2020-current	**SynTest-Framework**[4]: An open-source framework for search-based software testing. The framework is language-agnostic and can be used to implement search-based test case generation tools for different programming language.
2020-current	**SynTest-Solidity**[5]: An open-source search-based tool to generate synthetic tests for the Solidity platform.
2021-current	**SynTest-JavaScript**[6]: An open-source search-based tool to generate synthetic tests for the JavaScript and TypeScript languages.

Projects

2019-2023	*University Blockchain Research Initiative (UBRI)*[7]: A collaborative effort between 45+ universities across 20 countries to accelerate academic research, technical development and innovation in blockchain, cryptocurrency and digital payments.

[3]`https://github.com/EvoSuite/evosuite/blob/master/scripts/experiment_runner`

[4]`https://github.com/syntest-framework/syntest-framework`

[5]`https://github.com/syntest-framework/syntest-solidity`

[6]`https://github.com/syntest-framework/syntest-javascript`

[7]`https://ripple.com/impact/ubri/`

List of Publications

14. *Călin Georgescu, **Mitchell Olsthoorn**, Pouria Derakhshanfar, Marat Akhin, Annibale Panichella*: Evolutionary Generative Fuzzing for Differential Testing of the Kotlin Compiler. The ACM International Conference on the Foundations of Software Engineering (FSE), Porto de Galinhas, Brazil, 2024. doi: 10.1145/3663529.3663864.

📄 13. ***Mitchell Olsthoorn**, Dimitri Stallenberg, Annibale Panichella*: SynTest-JavaScript: Automated Unit-Level Test Case Generation for JavaScript. 17th IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT), Lisbon, Portugal, 2024. doi: 10.1145/3643659.3643928.

12. *Arkadii Sapozhnikov, **Mitchell Olsthoorn**, Annibale Panichella, Vladimir Kovalenko, Pouria Derakhshanfar*: TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. 46th ACM/IEEE International Conference on Software Engineering (ICSE): Companion Proceeding, Lisbon, Portugal, 2024. doi: 10.1145/3639478.3640024.

11. *Lisette Veldkamp, **Mitchell Olsthoorn**, Annibale Panichella*: Grammar-Based Evolutionary Fuzzing for JSON-RPC APIs. 16th IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT), Melbourne, Australia, 2023. doi: 10.1109/SBFT59156.2023.00026.

🏆📄 10. *Dimitri Stallenberg, **Mitchell Olsthoorn**, Annibale Panichella*: Guess What: Test Case Generation for Javascript with Unsupervised Probabilistic Type Inference. 13th International Symposium on Search-Based Software Engineering (SSBSE), Singapore, 2022. doi: 10.1007/978-3-031-21251-2_5.

📄 9. ***Mitchell Olsthoorn**, Arie van Deursen, Annibale Panichella*: Guiding Automated Test Case Generation for Transaction-Reverting Statements in Smart Contracts. 38th IEEE International Conference on Software Maintenance and Evolution (ICSME), Limassol, Cyprus, 2022. doi: 10.1109/ICSME55016.2022.00023.

📄 8. ***Mitchell Olsthoorn***: More effective test case generation with multiple tribes of AI. 44th ACM/IEEE International Conference on Software Engineering (ICSE): Companion Proceeding, Pittsburgh, PA, USA, 2022 (online). doi: 10.1145/3510454.3517066.

📄 7. ***Mitchell Olsthoorn**, Dimitri Stallenberg, Arie van Deursen, Annibale Panichella*: SynTest-solidity: automated test case generation and fuzzing for smart contracts. 44th ACM/IEEE International Conference on Software Engineering (ICSE): Companion Proceeding, Pittsburgh, PA, USA, 2022. doi: 10.1145/3510454.3516869.

📄 6. *Dimitri Stallenberg, **Mitchell Olsthoorn**, Annibale Panichella*: Improving Test Case Generation for REST APIs Through Hierarchical Clustering. 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 2021 (online). doi: 10.1109/ASE51524.2021.9678586.

📄 5. ***Mitchell Olsthoorn**, Pouria Derakhshanfar, Annibale Panichella*: Hybrid Multi-level Crossover for Unit Test Case Generation. 13th International Symposium on Search-Based Software Engineering (SSBSE), Bari, Italy, 2021 (online). doi: 10.1007/978-3-030-88106-1_6.

📄 4. ***Mitchell Olsthoorn***, *Annibale Panichella*: Multi-objective Test Case Selection Through Linkage Learning-Based Crossover. 13th International Symposium on Search-Based Software Engineering (SSBSE), Bari, Italy, 2021 (online). doi: 10.1007/978-3-030-88106-1_7.

📄 3. ***Mitchell Olsthoorn***, *Arie van Deursen, Annibale Panichella*: Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 2020 (online). doi: 10.1145/3324884.3418930.

2. ***Mitchell Olsthoorn***, *Pouria Derakhshanfar, Xavier Devroey*: An Application of Model Seeding to Search-Based Unit Test Generation for Gson. 12th International Symposium on Search-Based Software Engineering (SSBSE), Bari, Italy, 2020 (online). doi: 10.1007/978-3-030-59762-7_17.

1. *Martijn de Vos*, ***Mitchell Olsthoorn***, *Johan Pouwelse*: DevID: Blockchain-Based Portfolios for Software Developers. IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON), Newark, CA, USA, 2019. doi: 10.1109/DAPPCON.2019.00030.

📄 Included in this thesis.
🏆 Won a best paper, tool demonstration, or proposal award.

Titles in the IPA Dissertation Series since 2021

Software testing is important to make sure that code works as intended. Traditionally, this verification process has relied on manual testing, which is not only time-consuming but also susceptible to human errors. Through the use of automated test case generation techniques, we can automate this process and reduce the time and effort needed to test software. One of the promising techniques for automated test case generation is Search-Based Software Testing (SBST), which uses search-based metaheuristics to automatically generate test cases. SBST has been shown to be effective in generating test cases for a variety of programming languages and levels of testing (e.g., unit, integration, and system testing). However, SBST is not without its challenges. One of the main challenges is the size of the search space that needs to be explored.

In this thesis, we explore the potential to improve the effectiveness and efficiency of automated test case generation by combining multiple tribes of Artificial Intelligence (AI) to narrow down the search space. First, we introduce two novel approaches that incorporate domain-specific knowledge into the search process to reduce the search space for automated test case generation. Then, we present two novel crossover operators. One uses hierarchical clustering to identify and preserve promising patterns within test cases. The other combines multiple crossover operators at different levels (i.e., structure and data) to increase the diversity within the population. Next, we propose a model inference approach that infers dynamic types to allow automated test case generation of dynamically-typed languages. Finally, we introduce a new testing framework for two languages (Solidity and JavaScript) where no existing tooling existed.

9 789463 668712