THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Context-Infused Automated Software Test Generation

AFONSO FONTES

Division of Software Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2025

Context-Infused Automated Software Test Generation

AFONSO FONTES

Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2025.

Abstract

Automated software testing is essential for modern software development, ensuring reliability and efficiency. While search-based techniques have been widely used to enhance test case generation, they often lack adaptability, struggle with oracle automation, and face challenges in balancing multiple test objectives. This thesis expands the scope of search-based test generation by incorporating additional system-under-test context through two complementary approaches: (i) integrating machine learning techniques to improve test case generation, selection, and oracle automation, and (ii) optimizing multi-objective test generation by combining structural coverage with non-coverage-related system factors, such as performance and exception discovery.

The research is structured around four key studies, each contributing to different aspects of automated testing. These studies investigate (i) machine learning-based test oracle generation, (ii) the role of search-based techniques in unit test automation, (iii) a systematic mapping of machine learning applications in test generation, and (iv) the optimization of multi-objective test generation strategies. Empirical evaluations are conducted using real-world software repositories and benchmark datasets to assess the effectiveness of the proposed methodologies.

Results demonstrate that incorporating machine learning models into search-based strategies improves test case relevance, enhances oracle automation, and optimizes test selection. Additionally, multi-objective optimization enables balancing various testing criteria, leading to more effective and efficient test suites.

This thesis contributes to the advancement of automated software testing by expanding search-based test generation to integrate system-specific context through machine learning and multi-objective optimization. The findings provide insights into improving test case generation, refining oracle automation, and addressing key limitations in traditional approaches, with implications for both academia and industry in developing more intelligent and adaptive testing frameworks.

Acknowledgment

To my wife, for her unwavering support.
To my daughter, for being my greatest inspiration.
To my dog, for always reminding me to take breaks.

List of Publications

Appended publications

This thesis is based on the following publications:

[A] Afonso Fontes, Gregory Gay. Using Machine Learning to Generate Test Oracles: A Systematic Literature Review.
Proceedings of the 1st International Workshop on Test Oracles (TORACLE'21). Athens, Greece, August 2021.

[B] Afonso Fontes, Gregory Gay, Francisco Gomes de Oliveria Neto, Robert Feldt. Automated Support for Unit Test Generation.
Book chapter, Optimising the Software Development Process with Artificial Intelligence. Springer, 2022.

[C] Afonso Fontes, Gregory Gay. The Integration of Machine Learning into Automated Test Generation: A Systematic Mapping Study.
Software Testing, Verification and Reliability (STVR), 2023.

[D] Afonso Fontes, Gregory Gay, Robert Feldt. Exploring the Interaction of Code Coverage and Non-Coverage Objectives in Search-Based Test Generation.
Under revision in Software Testing, Verification, and Reliability (STVR).

Other publications

The following publications were published during my PhD studies. However, they are not appended to this thesis, due to contents not directly related to the thesis.

[a] Hamid Ebadi, Mahshid Helali Moghadam, Markus Borg, Gregory Gay, Afonso Fontes, Kasper Socha. Efficient and Effective Generation of Test Cases for Pedestrian Detection–Search-based Software Testing of Baidu Apollo in SVL. *Proceedings of 3rd IEEE International Conference on Artificial Intelligence Testing, Challenge Track (AiTest'21). Bari, Italy, August 2021.*

Research Contribution

As the main author of Papers A, B, C, and D, I was responsible for the core research, implementation, experimentation, and analysis.

In Paper A, I conducted a systematic literature review on machine learning-based test oracle generation, defining search strategies, selecting studies, and analyzing trends. I also implemented supporting scripts to assist in data extraction and synthesis.

In Paper B, I designed and implemented search-based unit test generation techniques, executing experiments, and analyzing results.

Paper C involved conducting a systematic mapping study on machine learning applications in test case generation, where I categorized existing approaches, identified trends, and structured research gaps.

Finally, in Paper D, I developed and evaluated a multi-objective test generation framework, analyzing the interaction between coverage and fault-based objectives, refining optimization strategies, and running large-scale empirical evaluations.

Across all studies, I conceptualized methodologies, implemented testing frameworks, executed experiments, and interpreted results.

Contents

Introduction

1.1 Problem Statement

Automated test generation is a crucial aspect of modern software engineering, aiming to enhance testing efficiency while reducing manual effort [1]. Among the various automated testing techniques, *search-based test generation* has emerged as one of the most effective methods [2]. Search-based test generation formulates test case creation as an optimization problem, using metaheuristic algorithms—such as genetic algorithms and simulated annealing—to iteratively refine test cases based on fitness functions [3]. These fitness functions commonly optimize for structural coverage (e.g., branch and statement coverage) and fault detection capabilities.

However, despite its effectiveness, existing approaches suffer from several limitations that hinder their practical application in real-world scenarios:

- **Over-reliance on Code Coverage:** Many test generation techniques prioritize structural coverage as a primary objective [1], but high coverage does not necessarily translate to meaningful fault detection [4]. Tests generated purely for coverage may lack real-world relevance, leading to false confidence in software reliability.

- **Limited Context Awareness:** Human testers leverage domain knowledge and past experience to craft meaningful test cases [5]. Current search-based methods fail to integrate such contextual understanding, often producing test cases that are formally valid but ineffective in revealing defects.

  Paper C reviews a range of machine learning techniques that have been explored in test generation research, including methods that aim to improve adaptivity in test selection. Additionally, contextual fitness functions have been introduced as an alternative way to incorporate information from the system under test (SUT) into test generation. By dynamically adjusting optimization goals based on system behavior, these approaches move beyond static test generation. For example, fitness functions based on performance or exception discovery, as examined in Paper D, provide mechanisms to prioritize test cases that expose performance bottlenecks or trigger exceptional conditions. While this thesis does not directly implement new machine learning models, prior studies analyzed in Paper C demonstrate how ML-driven strategies and our experimental results in Paper D on fitness functions based on system-related factors—such as performance or

exception discovery—can help bridge the gap between automated test creation and the contextual understanding typically leveraged by human testers.

- **Challenges in Test Oracle Definition:** Defining reliable test oracles remains a bottleneck in test automation. Manually specified oracles require significant human effort [5], while machine learning-based oracles face challenges in generalization and reliability [6].

- **Static and Non-Adaptive Test Generation:** Many existing methods apply fixed fitness functions and optimization goals, ignoring the evolving characteristics of the software under test (SUT) [7]. Adaptive test generation strategies that tailor objectives based on program behavior remain underdeveloped [8].

Addressing these challenges requires a shift towards intelligent, adaptive test case generation techniques that incorporate **multi-objective optimization, contextual awareness, and machine learning-driven inference**. This research explores how combining search-based techniques with AI-driven methods can improve test effectiveness, automate oracle inference, and enhance fault detection while maintaining efficiency.

1.1.1 Importance of Automated Test Case Generation

Automated test case generation is a critical aspect of software testing that focuses on systematically creating test cases to verify software correctness. As software complexity increases, manually designing test cases becomes impractical due to its high cost, time constraints, and susceptibility to human error [1]. Automated test case generation addresses these challenges by enabling systematic, repeatable, and scalable creation of test inputs and oracles.

One of the primary benefits of automated test case generation is its ability to improve test coverage while reducing human effort [7]. By automating the process of generating test cases, this approach ensures that a broader range of software behaviors is systematically explored, leading to enhanced fault detection. Additionally, search-based test generation techniques, such as genetic algorithms, iteratively refine test cases to maximize coverage and detect defects more effectively [3].

Beyond increasing coverage, automated test case generation contributes to software reliability by reducing human bias in test design. Manually written test cases often reflect a developer's expectations, potentially overlooking unforeseen edge cases [5]. In contrast, automated approaches systematically explore the input space, uncovering unexpected behaviors that might go undetected in manually designed test suites.

Modern software development methodologies, such as Agile and DevOps, emphasize rapid feedback loops and frequent software releases, requiring automated testing strategies to keep pace [9]. Automated test case generation supports these workflows by enabling continuous and adaptive generation of new test inputs, ensuring that evolving software components remain adequately tested. Despite its advantages, automated test case generation presents challenges, including the need for reliable test oracles, the integration of contextual information into test selection, and scalability concerns. Addressing these challenges requires advancements in search-based optimization and AI-driven inference techniques to improve adaptability, accuracy, and efficiency in test case generation.

1.1.2 Motivation for the Study

The increasing complexity of software systems, coupled with the growing demand for rapid deployment cycles, has highlighted the limitations of traditional testing methodologies. Manual testing remains a bottleneck in modern software development, requiring significant human effort while struggling to keep pace with continuous integration and deployment practices. Automated test generation offers a promising solution, yet current approaches face fundamental challenges that limit their practical effectiveness.

One of the primary motivations for this study is the realization that conventional automated test generation techniques, which focus primarily on structural coverage, often fail to detect critical faults. Structural coverage refers to a set of testing criteria that measure how much of a program's source code has been exercised by a test suite. Common coverage metrics include statement coverage (ensuring every line of code is executed at least once) and branch coverage (verifying that all decision points in the program flow are tested) [1, 9]. While achieving high coverage is beneficial, it does not inherently guarantee that software behaves correctly in real-world scenarios. This limitation necessitates the development of more advanced test generation strategies that go beyond mere structural metrics and incorporate domain-specific insights to improve fault detection.

Another key motivation is the ongoing struggle with defining reliable test oracles. In practice, writing effective assertions requires expert knowledge, and manually specifying expected outputs for all possible test cases is infeasible. Machine learning-based approaches have emerged as potential solutions [1, 5], yet their application in test oracle automation remains limited due to challenges in generalization and robustness [6]. This research seeks to explore how ML-driven test oracles can be enhanced to reduce reliance on manual specifications while maintaining accuracy [5].

Additionally, existing test generation techniques often adopt static optimization objectives, failing to adapt dynamically to different types of software under test [3]. The lack of adaptability leads to inefficient test generation, where certain aspects of the system may be over-tested while others remain insufficiently explored [8]. By investigating multi-objective optimization strategies [4] and machine learning techniques [6], this study aims to develop adaptive test generation strategies that intelligently balance multiple testing goals.

This research is motivated by the need to advance automated test generation methodologies by integrating AI-driven techniques that enhance efficiency, reliability, and scalability [10]. By addressing the existing gaps in test case generation, oracle inference, and adaptive testing strategies, this study aims to contribute towards the development of more robust and practical software testing solutions [9].

1.1.3 Research Objectives

This study aims to enhance the efficiency and effectiveness of automated test generation by integrating advanced search-based and machine learning techniques. The research focuses on developing adaptive test generation strategies that move beyond static fitness functions, enabling dynamic adjustment of testing goals based on program behavior. Another key objective is to investigate the impact of multi-objective optimization in balancing structural coverage, fault detection, and execution efficiency.

Another objective of this research is to systematically analyze and categorize AI-driven test generation techniques, providing an overview of their applications,

limitations, and future potential. This is achieved through the systematic mapping study presented in Paper C, which informs the direction of subsequent research by identifying key challenges and trends in ML-based test generation.

Furthermore, this study explores the potential of machine learning techniques to automate test oracle inference, reducing manual effort in specifying expected outputs. Finally, it aims to evaluate the scalability and applicability of AI-driven test generation across diverse software projects to ensure generalizability and practical adoption.

1.1.3.1 Research Questions

To provide a clear overview of how this research addresses the challenges in automated test generation, we formulated five key research questions. Table 1.1 presents these questions and maps them to their corresponding papers, showing how each component of the thesis contributes to our understanding of intelligent test automation.

Research Question
RQ1: How can the integration of search-based and machine learning-driven techniques improve automated test case generation?
RQ2: What are the challenges and benefits of multi-objective test generation as a comprehensive optimization strategy?
RQ3: Specifically, how does combining structural coverage with context-based fitness functions affect test generation effectiveness?
RQ4: What role does machine learning play in test oracle automation, and how can it improve test reliability?
RQ5: What principles and strategies can guide the design of adaptive test generation frameworks?

Table 1.1: Research Questions

In this study, context-based fitness functions refer to testing objectives beyond structural coverage, ensuring that test generation is optimized for diverse criteria such as fault detection and execution efficiency. This broader perspective on test objectives allows for a more comprehensive evaluation of test suite quality and effectiveness.

This thesis does not present a fully implemented adaptive test generation framework. Instead, Papers B and C provide insights into adaptive strategies by exploring ML-based heuristics and dynamic test generation techniques.

Research Question	**Addressed by Paper(s)**
RQ1	B, C, D
RQ2	D
RQ3	D
RQ4	A
RQ5	B, C, D

Table 1.2: Mapping of Research Questions to the Corresponding Papers

1.1.3.2 Research Context

The study is positioned within the broader field of software test case generation, and focusing on the intersection of search-based test generation, and artificial intelligence. Traditional approaches to automated test generation rely on structural coverage metrics to guide test generation, yet these methods often fail to detect functional faults or adapt to diverse software contexts. By leveraging context-driven techniques, this research seeks to bridge the gap between theoretical test generation strategies and practical testing applications, providing scalable, efficient, and adaptive solutions for modern software development practices.

1.2 Background

1.2.1 Software Testing

Software testing is a fundamental process in software engineering that ensures software systems function as intended and meet their requirements [11]. The primary goal of testing is to identify defects, improve software reliability, and validate expected behavior before deployment.

Testing methodologies can be broadly classified into manual and automated approaches, applied at various stages of software development. Automated testing increases efficiency by reducing human effort, enabling systematic and repeatable verification of software behavior [5]. It plays a crucial role in modern development workflows, particularly in continuous integration and deployment pipelines, where rapid feedback is essential.

A structured testing process typically involves executing test cases that evaluate different aspects of software functionality and performance. The effectiveness of test cases is often assessed using coverage criteria, such as statement and branch coverage, which quantify how thoroughly the software has been tested [1, 9]. While high coverage provides confidence in test adequacy, it does not guarantee defect-free software.

Testing strategies vary based on granularity, with unit testing focusing on individual components, integration testing assessing interactions between modules, and system-level testing evaluating overall functionality [12]. The adoption of automated tools and techniques, including search-based and machine learning-driven test generation, enhances test efficiency by improving coverage, reducing manual effort, and enabling more adaptive testing strategies.

1.2.1.1 Test Oracles

A critical aspect of software testing is determining whether a test passes or fails, which is accomplished using a *test oracle* [5]. A test oracle is a mechanism that defines the expected behavior of the system-under-test (SUT) and automatically verifies whether the observed outputs match the expected ones.

Test oracles take various forms, including manually specified assertions, outputs from previous software versions, formally specified properties, or even models trained on historical execution data [5]. Automated oracles significantly reduce the need for manual validation and support regression testing by ensuring consistent verification across software versions.

```
@Test
public void testPrintMessage() {
  String str = "Test␣Message";
  TransformCase tCase = new TransformCase(str);
  String upperCaseStr = str.toUpperCase();
  assertEquals(upperCaseStr, tCase.getText());
}
```

Figure 1.1: Example of a unit test case written using JUnit. The `assertEquals` statement acts as a test oracle, comparing the expected and actual outputs.

However, defining reliable test oracles remains a challenge, particularly for complex, nondeterministic systems or those lacking formal specifications. Machine learning-based approaches have been explored as potential solutions, aiming to infer test oracles from past execution data or learned program behaviors [12]. While these techniques can enhance automation, they also introduce challenges related to generalization and false positives.

1.2.1.2 Unit Testing and Coverage Criteria

Unit testing is a widely used technique that focuses on testing individual software components in isolation [1]. It ensures that functions, methods, or classes perform as expected before integrating them into a larger system. Unit tests are typically written as executable code and maintained in test suites, enabling repeated execution throughout development.

The effectiveness of unit tests is often measured using *coverage criteria*, which assess how thoroughly the source code is tested [9]. Common structural coverage metrics include:

- **Statement coverage**: Ensures that every executable statement in the program is executed at least once.
- **Branch coverage**: Requires that all possible control flow paths, including conditional branches (`if`, `case`, loops), are tested at least once [1, 4].

Higher coverage generally increases confidence in the quality of test suites. However, achieving high coverage does not necessarily mean that all software faults are detected [13]. Effective testing requires a combination of structural and functional criteria, along with well-defined test oracles.

Automated test generation techniques, particularly search-based and ML-driven methods, have been developed to improve test coverage while minimizing human effort [5]. These approaches generate test cases that maximize coverage, improve fault detection, and reduce redundancy in manually written tests.

1.2.2 Automated Test Case Generation

Automated test case generation is a process in software testing where test cases are automatically created without direct human intervention. This approach aims to improve testing efficiency and effectiveness by leveraging advanced algorithms to explore the software under test (SUT) systematically and generate test cases that meet specific testing criteria [12]. The automation of test case generation is particularly

beneficial in large and complex systems, where manual test design can be time-consuming and prone to human error.

Traditional test case generation methods often rely on predefined scenarios or developer-written scripts. While effective, these approaches can introduce bias and may not adequately cover edge cases or unexpected software behaviors [1]. Automated generation techniques mitigate these limitations by employing automated processes, often driven by dynamic analysis, model-based strategies, or heuristic search algorithms [9].

One common technique for automated test case generation is *search-based testing*, which formulates test generation as an optimization problem. Search algorithms, such as genetic algorithms, are utilized to explore the input space of the system under test (SUT), aiming to find test inputs that maximize predefined fitness functions, such as code coverage or fault detection [3]. A *fitness function* is a quantitative measure that evaluates the quality of generated test cases based on specific testing objectives. It assigns a numerical score to each test case, guiding the search algorithm toward more effective test inputs by favoring those that improve coverage, detect faults, or satisfy other testing criteria [8]. The choice of fitness function significantly influences the effectiveness of search-based test generation, as different formulations may lead to varied testing outcomes.

Recently, *large language models* (LLMs) have also been explored for automated test case generation. LLMs leverage pre-trained deep learning models to generate test inputs based on natural language descriptions of software behavior, code documentation, or past test cases. Unlike search-based approaches, which optimize test inputs iteratively based on a fitness function, LLM-based test generation relies on learned patterns from large-scale code datasets to produce semantically meaningful test cases. While promising, the effectiveness of LLMs in generating structurally and functionally valid test cases remains an open research question, particularly regarding their ability to generalize across different software domains [14, 15].

Other approaches include *model-based testing*, where test cases are derived from abstract models of the software's expected behavior, and *symbolic execution*, which generates test inputs by analyzing the program paths and solving logical constraints [16].

Automated test generation not only improves testing coverage but also supports the automation of regression testing and continuous integration processes by enabling the rapid generation of new test cases as software evolves. The integration of machine learning (ML) techniques further enhances this process by enabling adaptive and intelligent test generation strategies [6]. For example, ML models can predict high-risk code areas and guide the generation of targeted test cases, improving the efficiency of the testing process.

While automated test case generation offers significant benefits, it also presents challenges. These include the computational cost of executing complex algorithms, the potential need for high-quality training data when using ML-based methods, and the difficulty in generating valid test oracles for complex systems [5]. Despite these challenges, automated approaches are increasingly adopted in both research and industry, demonstrating their potential to enhance software quality and reduce testing costs.

1.2.2.1 Search-Based Test Generation Techniques

Search-based test generation formulates the creation of test cases as an optimization problem, leveraging metaheuristic algorithms to explore the input space of the system-under-test (SUT) [2]. This approach treats each test case as a potential solution to a testing objective, with fitness functions guiding the search process toward high-quality test inputs. Commonly used metaheuristics include genetic algorithms, simulated annealing, and particle swarm optimization, each offering distinct strategies for balancing exploration and exploitation of the input space [7].

Genetic algorithms, for example, simulate natural selection by evolving a population of test cases over successive generations. Each generation involves selection, crossover, mutation, and evaluation steps, gradually refining the population toward optimal test cases based on defined fitness criteria such as code coverage or fault detection [3]. This method is particularly effective in identifying edge cases and generating inputs that challenge the robustness of the SUT.

Search-based techniques are adaptable to various testing goals, including functional testing, performance testing, and security testing. By defining appropriate fitness functions, these methods can focus the search on specific aspects of the software's behavior, enhancing both the breadth and depth of test coverage [10]. However, the success of search-based approaches depends heavily on the quality of the fitness functions and the computational resources available, as complex software may require extensive search iterations to achieve meaningful results [8].

1.2.2.2 Common Test Generation Techniques

In addition to search-based methods, several other automated test generation techniques contribute to thorough and effective software testing. These include random testing, model-based testing, symbolic execution, combinatorial testing, and large language model (LLM)-based test generation, each offering unique advantages and applications depending on the testing context [7].

Random testing generates test inputs randomly, providing a simple yet powerful method for identifying unexpected behaviors in the SUT [17]. While random testing is computationally inexpensive and easy to implement, its effectiveness depends on the ability to cover a broad input space, which may require a large volume of tests.

Model-based testing involves creating abstract models of the software's expected behavior and deriving test cases systematically from these models [18]. This approach is particularly effective when the software's behavior can be accurately represented through state machines, flowcharts, or other formal models. Model-based testing supports automated test case generation and validation, reducing the manual effort required.

Symbolic execution generates test inputs by analyzing the program's code paths and solving logical constraints associated with each path [16]. By symbolically evaluating possible execution paths, this technique can generate high-coverage test cases that explore edge conditions and identify potential faults.

Combinatorial testing aims to systematically cover all possible combinations of input parameters, ensuring that interactions between inputs are thoroughly tested [19]. This method is particularly useful for systems with configurable parameters or decision-making logic, where specific input combinations may trigger hidden issues.

LLM-based test generation leverages large language models trained on extensive software repositories to generate test cases based on natural language descriptions,

existing code, or past test cases. These models can infer meaningful test scenarios and assertions without requiring explicit rule-based formulations. While LLM-based test generation offers promising automation potential, its effectiveness depends on the quality of training data, the model's ability to generalize across different software domains, and its capacity to generate syntactically and semantically valid test cases [14, 15].

Each of these techniques can be integrated into automated testing frameworks to enhance test coverage, improve fault detection, and streamline the testing process. The choice of technique depends on the software's complexity, the testing objectives, and the available computational resources.

1.2.3 Integrating Machine Learning and Contextual Fitness Functions into Search-Based Test Case Generation

Search-based test generation relies on metaheuristic optimization algorithms to generate test cases by systematically exploring the input space of the system under test (SUT). Traditionally, these approaches have focused on maximizing structural coverage or fault detection through predefined fitness functions. However, this research explores the integration of *machine learning* (ML) and *context-aware fitness functions* into search-based test generation to enhance adaptability and effectiveness.

A key component of search-based test generation is the use of *evolutionary algorithms*, which apply principles of natural selection to evolve test cases over generations. Among these, genetic algorithms (GAs) have been widely studied and applied due to their ability to iteratively refine test inputs. GAs employ selection, crossover, and mutation to optimize test cases according to predefined fitness functions, which guide the search process by rewarding test cases that improve coverage, detect faults, or satisfy system-related objectives [3]. The choice of fitness function significantly influences test generation effectiveness, and recent studies have explored alternative formulations that incorporate *non-coverage objectives*, such as exception discovery and performance analysis, to provide a broader evaluation of test quality.

Machine learning techniques complement search-based test generation by introducing adaptive and predictive capabilities. *Supervised learning* models can be trained on historical test data to predict fault-prone regions of the code, allowing the search process to prioritize high-risk areas [6]. Similarly, *unsupervised learning* techniques, such as clustering, can identify patterns in execution traces, guiding the generation of test cases that expose unexpected behaviors. The incorporation of ML-based heuristics allows for more intelligent exploration of the input space, reducing reliance on manually engineered fitness functions.

Search-based test generation can also be enhanced by incorporating *contextual fitness functions* that dynamically adjust based on system characteristics. These fitness functions extend beyond traditional structural coverage metrics by incorporating execution properties, system-specific constraints, and functional correctness objectives. For example, multi-objective optimization strategies enable balancing different testing criteria, such as maximizing coverage while minimizing test suite size or execution time [8]. By integrating adaptive fitness functions, search-based approaches can better align with real-world software requirements.

While genetic algorithms remain a dominant search strategy in search-based test generation, other metaheuristic techniques, such as *simulated annealing* and *particle swarm optimization*, have also been explored for test input generation [2].

These algorithms, like GAs, iteratively refine candidate solutions but differ in their exploration-exploitation trade-offs. The key distinction lies not in their classification as metaheuristic algorithms but in their specific search strategies, making them potential alternatives depending on the nature of the SUT and optimization objectives.

Overall, this research investigates how search-based test generation can be improved by integrating machine learning techniques and diverse fitness functions to enhance adaptability, efficiency, and test case effectiveness. By leveraging predictive ML models, dynamic heuristics, and broader optimization goals, this approach moves beyond traditional structural coverage-driven methods, fostering a more intelligent and context-aware test generation process.

1.2.3.1 Genetic Algorithms

Genetic algorithms (GAs) are a class of evolutionary algorithms inspired by the principles of natural selection and genetics, widely used in search-based software testing (SBST) to automate test case generation [2]. GAs operate by evolving a population of candidate solutions (test cases) through iterative processes that mimic biological evolution, including selection, crossover, and mutation [3].

The process begins with the random initialization of a population of test cases, each encoded as a chromosome representing potential inputs to the system under test (SUT). The quality of each test case is assessed using a predefined fitness function, which evaluates how well the test meets specific testing objectives, such as achieving high code coverage or identifying software faults [10]. High-performing test cases are selected for reproduction, promoting the survival of the fittest.

Crossover, or recombination, combines segments of selected test cases to produce new offspring, introducing variability while retaining beneficial traits from parent solutions. Mutation further enhances diversity by randomly altering parts of a test case, helping the algorithm explore new areas of the input space and avoid local optima [8]. These genetic operators drive the population toward improved test cases over successive generations.

Traditionally, search-based test generation has focused on fitness functions based on structural coverage criteria (e.g., statement, branch, or path coverage). However, recent research has expanded the scope of fitness functions to incorporate context-aware and goal-based objectives that align more closely with real-world software behavior. These alternative fitness functions optimize test generation for objectives such as exception discovery, crash detection, and quality-related goals , including performance, energy consumption, and memory usage [3]. This shift allows test case generation to consider not only whether software executes different paths but also whether it meets key runtime and reliability criteria.

GAs are particularly effective in exploring large and complex input spaces, making them suitable for testing systems with intricate logic, numerous input parameters, or challenging edge cases. Their flexibility allows them to adapt to diverse testing goals by adjusting the fitness function, enabling targeted testing strategies such as boundary testing, robustness testing, and stress testing [3]. By incorporating domain-specific objectives into the fitness function, GAs can be tailored to focus on defect-prone areas and optimize test cases beyond structural coverage metrics .

Despite their advantages, GAs also present challenges. The choice of fitness function significantly influences the algorithm's effectiveness, and poorly designed functions may lead to suboptimal test cases [7]. Additionally, GAs can be computa-

tionally intensive, requiring careful tuning of algorithm parameters (e.g., population size, mutation rate) to balance exploration, exploitation, and performance.

Overall, genetic algorithms provide a powerful and flexible approach to search-based test case generation, contributing to the robustness and reliability of software systems by enhancing the efficiency, adaptability, and effectiveness of automated testing processes.

1.2.3.2 Types of Machine Learning Approaches

Machine learning (ML) approaches for search-based test case generation can be broadly categorized into *supervised learning*, *unsupervised learning*, and *reinforcement learning*. Each of these techniques contributes uniquely to the automation of software testing by leveraging data-driven models to optimize and predict testing outcomes [6].

Supervised learning requires labeled training data to learn mappings between inputs and expected outputs. In software testing, historical defect data can be used to train classifiers that predict fault-prone code segments, aiding in prioritizing test case generation [8]. A notable subset of supervised learning is the use of *large language models (LLMs)*, which are pre-trained on vast amounts of textual and code-based data. LLMs can be fine-tuned to generate unit tests, suggest assertions, or synthesize test cases based on software documentation and past test cases, improving automation in test generation [14, 15].

Unsupervised learning identifies patterns in unlabeled data, making it useful for anomaly detection in software testing. Clustering techniques, such as k-means, can group similar execution traces, helping identify deviations from expected behavior [10].

Reinforcement learning (RL) models learn optimal testing strategies by interacting with the software under test and receiving rewards based on test effectiveness. RL-based test case generation dynamically adapts as software evolves, improving test efficiency over time [6]. While RL remains an emerging area in automated testing, prior studies indicate its potential for improving test case prioritization and adaptive test selection.

Each of these ML approaches provides distinct advantages, and their integration into search-based test generation continues to evolve, supporting more intelligent and automated testing workflows.

1.2.3.3 Applications of Machine Learning in Software Testing

ML techniques are applied across various software testing tasks, enhancing automation, fault detection, and efficiency. Key applications include:

- **Test Data Generation:** Machine learning techniques aid in the creation of test data by generating representative input values or datasets for software testing. By analyzing prior execution traces and system behavior, ML models can produce diverse test cases that increase code coverage and expose corner-case failures [3].

- **Test Input (Case) Generation:** ML-driven approaches, such as search-based methods and supervised learning models, automate test case generation by producing input sequences that optimize code coverage and reveal faults. These

methods significantly reduce manual effort while enhancing test effectiveness [8]. Large language models (LLMs) have emerged as a subset of supervised learning, capable of generating test cases based on code structure and learned patterns from large-scale repositories [6].

- **Test Oracle Automation:** ML models assist in constructing test oracles by inferring expected software behavior from past execution data. These techniques enhance automated verification by predicting expected outputs or detecting deviations, reducing reliance on manually defined assertions [5].
- **Defect Prediction and Prevention:** Supervised learning models utilize historical defect data to classify software components based on their likelihood of containing faults. This predictive capability enables targeted testing, ensuring that testing efforts are focused on high-risk areas to improve software reliability [10].

Example – ML-Assisted Parameter Tuning for Search-Based Test Generation: ML techniques, including RL, can also optimize search-based test generation by dynamically tuning algorithm parameters. Traditional search-based testing relies on manually configured mutation rates, crossover probabilities, and selection strategies, which may not be optimal for all software systems. RL-based approaches, such as adaptive fitness function selection (AFFS), have been successfully used to modify fitness function choices dynamically during test generation, improving fault detection and efficiency [20]. By learning from previous test generations, these adaptive strategies fine-tune search heuristics to balance exploration and exploitation, leading to more effective test suites.

1.2.4 Related Work

Several works have investigated the role of search-based software testing (SBST) in optimizing test generation. [2] provided a foundational analysis of search-based techniques for software testing, demonstrating how evolutionary algorithms, particularly genetic algorithms, can optimize test case selection. Expanding on this, [3] proposed fitness-guided test case evolution strategies that effectively increase fault detection rates by adapting test input mutations over multiple iterations.

Machine learning techniques have also been explored extensively in software testing. [6] performed a systematic mapping study examining how ML has been applied to various testing activities, including test input and oracle generation. Their findings indicate that supervised learning is the most frequently applied ML approach, with artificial neural networks being particularly common. Similarly, [21] reviewed ML applications in software testing, highlighting that both input and oracle generation tasks have benefited from ML-driven approaches.

The integration of ML into search-based test generation has also been studied in various contexts. [22] surveyed AI-driven techniques for white-box test generation, emphasizing the role of optimization techniques, including genetic algorithms, in achieving high structural coverage. They noted that ML techniques have been explored for guiding input generation, further expanding the capabilities of traditional search-based strategies.

Another critical aspect of test automation is oracle generation. [5] categorized different oracle mechanisms and analyzed how ML-derived oracles improve the automation of test result validation. ML-based oracle derivation falls into the ”derived”

oracle category, as it learns expected program behaviors from historical project artifacts. [12] further categorized oracles into four types—human-specified, automatically derived, implicit property-based, and human-in-the-loop—and discussed the role of ML in automating oracle inference.

The role of ML in hyper-heuristics for search-based testing has also been explored. [23] conducted a systematic mapping study on hyper-heuristics, which serve as secondary optimization mechanisms that adapt test generation strategies dynamically. Hyper-heuristics, including those leveraging ML-based techniques such as reinforcement learning, adjust search parameters based on system-under-test (SUT) behavior, improving the effectiveness of SBST approaches. For instance, ML-based hyper-heuristics have been used to fine-tune mutation rates, crossover probabilities, and fitness function selection, leading to more efficient and targeted test generation.

Furthermore, ML has been applied to dynamically adjusting test generation strategies. [24] explored the use of reinforcement learning in adaptive fitness function selection (AFFS), demonstrating that ML-guided adjustments to fitness function priorities can enhance test generation effectiveness. This study showed that dynamically altering test objectives based on execution feedback leads to better-balanced testing strategies compared to static fitness functions.

Overall, existing literature has laid a strong foundation for integrating ML into search-based test generation. While progress has been made in improving test efficiency and accuracy, challenges remain in refining fitness functions, ensuring test oracle reliability, and mitigating computational costs. Future work should continue to explore hybrid methodologies that integrate ML, evolutionary algorithms, and heuristic search techniques to further enhance software testing automation.

1.3 Research Methodology

This research integrates findings from four studies to explore advancements in automated test case generation, search-based optimization, and machine learning-driven approaches for software testing. The methodology consists of analyzing these contributions in a structured manner, aligning them with the research objectives to investigate and improve testing efficiency, test oracle generation, and fault detection capabilities.

1.3.1 Overview of the Contributions

To structure the research findings, the four primary studies contributing to this thesis are presented in Table 1.3.

1.3.2 Approach

The research employs a multi-faceted approach combining empirical experimentation, systematic literature analysis, and tool-based evaluation. By leveraging search-based techniques and machine learning models, it investigates new methodologies for improving test case generation, optimizing fitness functions, and automating test oracles.

The work is structured around analyzing existing approaches to automated test generation, developing adaptive optimization strategies for multi-objective test generation, integrating structure-based and context-based fitness functions, assessing ML models for test oracle inference and automated validation, and evaluating the effectiveness of these techniques through empirical studies.

Paper	Title and Contribution
Paper A	*Using Machine Learning to Generate Test Oracles: A Systematic Literature Review*. This paper provides an analysis of ML-based approaches for test oracle generation, identifying trends, challenges, and future directions.
Paper B	*Automated Support for Unit Test Generation*. This work focuses on the development of search-based test generation algorithms, providing a tutorial on search-based testing techniques. While it includes some theoretical exploration of how ML could be integrated, its primary contribution is in advancing search-based approaches.
Paper C	*The Integration of Machine Learning into Automated Test Generation: A Systematic Mapping Study*. This study analyzes ML-driven test case generation techniques, examining their applications, limitations, and potential improvements.
Paper D	*Exploring the Interaction of Code Coverage and Context-Based Fitness Functions in Multi-objective Test Generation*. This paper investigates the integration and interaction of coverage-based and context-based fitness functions in test case generation, balancing multiple testing objectives such as fault detection, execution efficiency, and structural coverage.

Table 1.3: Summary of the primary studies contributing to this research.

1.3.3 Algorithms

The research explores various search-based and ML-driven approaches, including genetic algorithms, reinforcement learning, supervised learning, and multi-objective optimization. Genetic algorithms iteratively refine test cases by optimizing fitness functions based on structural and fault-based coverage. Reinforcement learning models test case generation as a sequential decision-making problem, adjusting test selection based on feedback. Supervised learning applies ML models trained on labeled test execution data to predict high-risk areas and improve oracle generation. Multi-objective optimization strategies balance multiple competing objectives, such as coverage, mutation score, and test suite size, to enhance overall test quality.

1.3.4 Data Collection and Evaluation Metrics

The research employs a combination of benchmark datasets, real-world software projects, and controlled experiments to evaluate test generation effectiveness. Code coverage is used to measure structural coverage, including branch, statement, and path coverage. Mutation score assesses fault detection capability by determining how many artificial defects (mutants) are successfully detected. Test suite size is analyzed to evaluate efficiency by examining the number of generated test cases and their redundancy. Finally, correctness metrics are applied to oracle generation, measuring prediction accuracy, classification performance, and false positive/negative rates. These evaluation criteria ensure a comprehensive assessment of the proposed methodologies.

1.4 Research Results

This section presents the key findings from the four studies contributing to this thesis, highlighting their impact on automated test case generation, search-based testing, and machine learning-driven software testing approaches.

Paper A examined machine learning approaches for test oracle automation, identifying techniques such as neural networks and decision trees to predict test verdicts. The study highlighted that ML-derived oracles reduce manual effort but require substantial training data for reliability. The findings suggest that while ML-based oracles can complement traditional methods, their effectiveness varies depending on the software under test.

Paper B introduced a search-based approach for automated unit test generation in Python, demonstrating how metaheuristic search algorithms can systematically refine test inputs to maximize code coverage. The study framed unit test generation as an optimization problem and illustrated the application of search-based software testing techniques, specifically within Python's pytest framework. By leveraging evolutionary algorithms, the approach generated pytest-formatted unit tests that improved structural code coverage while reducing the manual effort involved in test creation. The paper provided a tutorial on key search-based testing concepts, such as solution representation, fitness function design, and mutation strategies, making it accessible to practitioners interested in adopting automated unit test generation.

Beyond the implementation details, the study discussed potential extensions for integrating machine learning techniques into search-based test generation. It outlined challenges such as selecting optimal hyperparameters, refining fitness functions, and balancing test effectiveness with computational efficiency. Although machine learning was not directly implemented in this work, the discussion highlighted avenues for future research, including using ML-based heuristics to guide search algorithms or employing reinforcement learning to dynamically adapt search strategies. The study's primary contribution was the development of a structured, reproducible methodology for search-based test generation in Python, offering both a practical implementation and theoretical insights for further advancements in automated testing.

Paper C systematically categorized ML-driven test generation techniques based on their application in test input and oracle generation. The study identified that ML techniques are applied across various testing practices, including system testing, GUI testing, unit testing, performance testing, and combinatorial interaction testing. Supervised learning emerged as the most commonly used approach, particularly in system testing, combinatorial interaction testing, and oracle generation, where models such as neural networks and decision trees are frequently applied. Reinforcement learning was noted for its effectiveness in scenarios requiring sequential decision-making, such as GUI and unit test generation, as well as in tuning existing test generation tools. Unsupervised learning methods, such as clustering, were explored for tasks like anomaly detection and structural pattern identification in software behavior.

The study highlighted that ML has been used to generate test input, enhance existing test generation methods, and automate oracle creation. ML-enhanced test generation has demonstrated improvements in code coverage, test prioritization, and fault detection. However, significant challenges remain, including the need for large and high-quality training datasets, the necessity of retraining models as software evolves, and concerns over scalability and generalization. The study also emphasized the lack of standard benchmarks and replication packages in ML-based test generation

research, recommending that future work focus on increasing reproducibility and comparability across studies.

Paper D examined the interaction between structure-based fitness functions, such as branch coverage, and context-based fitness functions, including exception discovery, execution time, and output diversity. The study assessed the extent to which combining these objectives influenced key aspects of automated test generation, including branch coverage attainment, goal-based fitness fulfillment, fault detection, and test suite characteristics. Instead of balancing these objectives, the study measured their interaction and their impact on generated test suites.

The findings indicate that incorporating goal-based fitness functions alongside branch coverage does not significantly reduce structural coverage. In some cases, multi-objective test generation improved goal attainment and increased the likelihood of fault detection. Specifically, targeting both branch coverage and exception count led to an increase in exceptions discovered, while optimizing for branch coverage and output diversity sometimes accelerated coverage attainment. However, a notable trade-off was the increase in test suite size, which could affect maintainability and execution efficiency. Additionally, the study found that while multi-objective optimization led to improvements in certain metrics, these benefits were often more limited than initially hypothesized. These results provide deeper insights into the effects of structural and contextual objectives in search-based test generation, informing the design of more effective automated testing strategies.

The findings across the four studies demonstrate that integrating search-based and machine learning techniques enhances automated test generation. ML-driven oracles provide automation benefits but require further refinement for broader applicability. Search-based techniques improve test coverage and efficiency but face scalability constraints. Multi-objective optimization offers a balanced approach but necessitates further tuning to optimize trade-offs. These insights contribute to advancing automated software testing by improving adaptability, fault detection, and efficiency.

1.5 Threats to Validity

Ensuring the validity of our research requires careful consideration of potential limitations and biases. This section outlines the primary threats to validity and the measures taken to mitigate them.

1.5.1 External Validity

External validity concerns the generalizability of our findings. Our study focuses on automated test case generation using search-based and machine learning-driven approaches. While we evaluate our methods on a representative selection of test subjects, the applicability of our findings to other software systems may be limited.

We have chosen datasets and benchmarks widely used in search-based software testing research [25], ensuring that our findings align with prior work and facilitate comparison. However, our selection of projects and faults may introduce bias, as some defect types may be underrepresented. To mitigate this, we have included a diverse range of faults and tested multiple configurations to assess the robustness of our approach.

Furthermore, our study focuses on a specific set of test generation tools, primarily based on evolutionary algorithms and machine learning models. While these represent

state-of-the-art techniques, other methods may yield different results. Future work should explore alternative approaches to determine the extent of generalizability.

For **Papers A and C**, which are systematic literature reviews, external validity is primarily influenced by the completeness of the selected studies. Our conclusions rely on the set of studies included in the review, and it is possible that important studies were omitted. Secondary studies do not necessarily capture all relevant research in a field, but the selection protocol (search string, inclusion and exclusion criteria) should ensure an adequate and representative sample. To mitigate this, different search strings were tested, a validation exercise was conducted to verify robustness, and four major databases were used to capture the majority of relevant software engineering publications. Additional snowballing was performed to improve coverage.

1.5.2 Internal Validity

Internal validity pertains to the accuracy of our experimental setup and its ability to establish causal relationships. Given the stochastic nature of search-based and machine learning techniques, the consistency of our results may be affected by randomness in test generation.

To control for this, we conducted multiple experimental runs and aggregated results across different configurations. Each test was repeated with varied random seeds to minimize the influence of randomness on our conclusions [8]. However, budget constraints limited the number of trials conducted, which may affect the statistical power of our findings.

Additionally, hyperparameter tuning plays a significant role in the effectiveness of ML-driven test case generation. While we used parameters validated in prior research, slight modifications could impact performance. Future studies should explore more adaptive tuning techniques to improve reproducibility.

For **Papers A and C**, internal validity is influenced by the reliability of our study selection and classification process. The selection of studies required subjective judgments regarding relevance, categorization, and inclusion. To mitigate bias, article selection and data extraction were conducted using predefined protocols that were reviewed and refined iteratively. Independent verification was performed on a sample of all selection and categorization decisions to reduce bias.

1.5.3 Conclusion Validity

Conclusion validity relates to the reliability of our results and the statistical methods used for analysis. We have employed non-parametric statistical tests due to the non-normal distribution of our data, ensuring that our findings are robust [10]. Descriptive statistics and box plots were used to validate the results before applying hypothesis testing.

A key concern in our study is potential biases introduced during test evaluation. Our effectiveness metrics focus on code coverage and fault detection, which are commonly used in software testing research but do not capture all aspects of test quality. Future research should explore additional metrics, such as maintainability and readability of generated tests.

For **Papers A and C**, the analyses performed were qualitative, requiring inference from the authors. This introduces potential bias in data extraction and categorization. To mitigate this, study selection, categorization, and property extraction were guided by

predefined criteria. Protocols were reviewed by multiple researchers, and verification was performed on a sample of all decisions to ensure consistency.

1.5.4 Construct Validity

Construct validity evaluates whether our study measures what it intends to measure. The fitness functions used in search-based test generation play a critical role in directing test evolution. If the fitness functions are not well-designed, generated test cases may not effectively contribute to fault detection or structural coverage [3].

We have carefully selected and validated our fitness functions based on prior research, but different problem domains may require alternative criteria. Future studies should investigate adaptive fitness functions that dynamically adjust to different software characteristics.

For **Papers A and C**, construct validity is related to the properties used for data extraction and study classification. The selected properties guided the review process and may have been incomplete or misleading. To mitigate this risk, properties were iteratively refined based on a sample of studies before applying them to the entire dataset.

Overall, while we have taken measures to ensure the rigor and reliability of our findings, potential limitations remain. Further validation on diverse datasets, tools, and experimental configurations will strengthen the conclusions drawn from our study.

1.6 Conclusions

Automated test generation has been a long-standing research challenge, with advancements in search-based techniques, machine learning-driven approaches, and hybrid methodologies contributing to more effective and scalable solutions. This thesis has explored various methodologies aimed at enhancing test generation, evaluating their strengths, limitations, and potential improvements.

Multi-objective test generation has shown promising results, as combining coverage-driven and goal-based fitness functions can improve both fault detection rates and goal attainment without significant drawbacks [3]. While increasing the number of objectives can sometimes lead to larger test suites, our findings suggest that multi-objective optimization remains preferable over single-objective approaches, as it enhances fault detection and test robustness [8]. Future research should explore further optimization of fitness functions and consider varying metaheuristic search algorithms to refine multi-objective test generation strategies.

Machine learning has been increasingly integrated into automated test generation, improving various aspects of the process, such as test input selection, oracle generation, and test prioritization. ML-driven approaches have demonstrated effectiveness in generating system, GUI, unit, and performance test cases, as well as in predicting defect-prone areas and improving test efficiency [6]. Reinforcement learning, in particular, has emerged as a powerful tool for generating adaptive test cases, dynamically optimizing the testing process based on execution feedback [8]. However, challenges related to training data availability, scalability, and model interpretability remain open research questions.

Another key aspect of automated test generation is the test oracle problem, which has been addressed through ML-driven oracle inference techniques. Studies have

explored using neural networks, support vector machines, and decision trees to generate verdict, metamorphic relation, and expected output oracles [5]. While these approaches show potential, challenges such as data dependency, retraining requirements, and handling complex software behaviors must be addressed to improve the reliability and applicability of ML-based oracles.

Unit testing remains a fundamental testing practice, yet writing unit tests is often labor-intensive and requires domain expertise. The integration of search-based test generation techniques has significantly improved automation in unit testing, framing test input selection as an optimization problem and applying evolutionary algorithms to find optimal test inputs [2]. Future work should explore refining distance-based fitness functions, human-readable test input generation, and leveraging deep learning for enhanced unit test automation.

In conclusion, the thesis has demonstrated the potential efficacy of multi-objective test generation, ML-assisted test automation, and oracle inference techniques in enhancing software testing. Despite significant advancements, challenges remain in optimizing test generation algorithms, improving ML model generalizability, and ensuring the replicability of findings. Future research should focus on hybridizing search-based and ML-driven techniques, refining fitness functions, and creating standardized benchmarks to foster progress in automated testing.

1.7 Future Work

Building upon the findings presented in this thesis, several directions can be explored to further advance automated test case generation and its applications in software testing. Our research has highlighted key areas for improvement and expansion, offering opportunities for deeper investigation and practical implementation.

The next step is to refine the multi-objective optimization techniques used in test case generation. While our study has demonstrated the benefits of combining code coverage and goal-based fitness functions, additional exploration is needed to identify optimal fitness function combinations. Future research should investigate adaptive mechanisms that dynamically adjust optimization objectives based on software characteristics and testing goals.

Furthermore, the integration of machine learning into test generation presents several promising avenues. One potential direction is to enhance reinforcement learning-based test case generation by incorporating advanced reward mechanisms that better reflect fault detection effectiveness. Additionally, further studies should examine the impact of training data selection and model generalization to improve scalability across different software projects.

Expanding the scope of ML-based test oracles is another possible area of future work. While our research has examined the potential of ML in oracle inference, challenges such as data dependency, retraining frequency, and handling complex software behaviors remain. Future studies should focus on developing robust oracle models that can adapt to evolving software versions while minimizing false positives in test verdicts.

Additionally, we aim to explore the application of hybrid search-based and ML-driven techniques in unit test automation. Investigating how deep learning can enhance test case readability, maintainability, and diversity will contribute to making automated unit test generation more practical for developers. Further work should also analyze the

impact of generated test cases on software maintainability, ensuring that automation contributes positively to long-term software quality.

Finally, we propose the creation of standardized benchmarks and datasets for evaluating automated test generation techniques. Establishing a shared evaluation framework will facilitate more consistent comparisons between methodologies, enabling a more structured progression in the field. Collaborations with industry partners could further enhance the applicability of these techniques in real-world software development environments.

By addressing these challenges and expanding our research in these directions, we aim to contribute to the continuous improvement of automated software testing methodologies, ultimately fostering more reliable and efficient software development processes.

PaperA

Using Machine Learning to Generate Test Oracles: A Systematic Literature Review

Afonso Fontes, Gregory Gay

Proceedings of the 1st International Workshop on Test Oracles (TORACLE'21). Athens, Greece, August 2021.

Abstract

Machine learning may enable the automated generation of test oracles. We have characterized emerging research in this area through a systematic literature review examining oracle types, researcher goals, the ML techniques applied, how the generation process was assessed, and the open research challenges in this emerging field.

Based on 22 relevant studies, we observed that ML algorithms generated test verdict, metamorphic relation, and—most commonly—expected output oracles. Almost all studies employ a supervised or semi-supervised approach, trained on labeled system executions or code metadata—including neural networks, support vector machines, adaptive boosting, and decision trees. Oracles are evaluated using the mutation score, correct classifications, accuracy, and ROC. Work-to-date show great promise, but there are significant open challenges regarding the requirements imposed on training data, the complexity of modeled functions, the ML algorithms employed—and how they are applied—the benchmarks used by researchers, and replicability of the studies. We hope that our findings will serve as a roadmap and inspiration for researchers in this field.

keywords: Test Oracle, Automated Test Generation, Automated Test Oracle Generation, Machine Learning

2.1 Introduction

Software testing is invaluable in ensuring the reliability of the software that powers our society [11]. It is also notoriously difficult and expensive, with severe consequences for productivity, the environment, and human life if not conducted properly [4]. New tools and methodologies are needed to control that cost without reducing the quality of the testing process. Automation has a critical role to play in this effort by controlling testing costs and focusing developer attention on important tasks [26, 27].

Consider test creation, an effort-intensive task that requires the selection of sequences of program input and *oracles* that judge the correctness of the resulting execution [5]. Automated test oracle creation is a topic of particular interest—and has earned the title "the test oracle problem" [5]. In current practice, oracles are often test-specific and require dedicated human effort to create. Advances have been made, but *the test oracle problem remains unsolved*. If oracle creation could be even partially automated, developers' effort and cost savings could be immense.

Advances in the field of machine learning (ML) have shown that algorithms can match or surpass human performance across many problem domains [28]. Machine learning has been used to advance the state-of-the-art in virtually every field. Automated test generation is no exception. We are interested in understanding and characterizing emerging research around the use of ML to generate or to support the creation of test oracles. Specifically, we are interested in understanding the types of oracles generated, the researchers' goals using ML, which specific ML techniques were applied, how such techniques were trained and validated, and how the success of the generation process was assessed. We also seek to identify limitations that must be overcome and open research challenges in this emerging field.

To that end, we have performed a systematic literature review. Following a search of relevant databases and a rigorous filtering process, we have gathered a sample of 22 relevant studies. We have examined each study, gathering the data needed to answer our research questions. The findings of this study include:

- ML has been used to generate test verdict (18%), metamorphic relation (27%), and expected output (55%) oracles.
- ML algorithms train predictive models that serve either as a stand-in for an existing test oracle—predicting a test verdict—or as a way to learn information about a function—either the expected output or metamorphic relations—that can be used as part of issuing a verdict.
- Almost all studies (96%) employ supervised ML, trained on labeled system execution logs or source code metadata and validated based on the accuracy of the trained model.
- 59% of the approaches employed neural network (NN)—including Backpropagation NNs, Multilayer Perceptrons, RBF NNs, probabilistic NNs, and Deep NNs. 23% of approaches adopted support vector machines. 5% adopted decision trees, and another 5% adopted adaptive boosting. The remaining 5% did not specify a technique.
- Results were most often evaluated using the mutation score (55%), followed by number of correct classifications (18%), classification accuracy (18%), and ROC (5%). One study did not perform evaluation.
- The sampled studies show great promise, but there are still significant limitations and open challenges:

```
@Test
public void testPrintMessage() {
    String str = "Test␣Message";
    TransformCase tCase = new TransformCase(str);
    String upperCaseStr = str.toUpperCase();
    assertEquals(upperCaseStr, tCase.getText());
}
```

Figure 2.1: Example of a unit test. The `assertEquals` statement is an oracle, comparing the expected and actual output.

- Oracle generation is limited by the required quantity, quality, and content of training data. Assembling training data may require significant human effort. Models should be retrained over time.
- Applied techniques may be insufficient for modeling complex functions with many possible outputs. Varying degrees of output abstraction should be explored. Deep learning and ensemble techniques, as well as hyperparameter tuning, should be explored.
- Research is limited by overuse of toy examples, the lack of common benchmarks, and the inavailability of code and data. A benchmark should be created for evaluating oracle research, and researchers should be encouraged to provide replication packages and open code.

Our study is the first to summarize this emerging research field. We hope that our findings will serve as a roadmap and inspiration for researchers interested in automated oracle generation

2.2 Background and Related Work

Testing and Test Oracles: Before complex software can be trusted, it is important to verify that the code is functioning as intended. Verification is often performed through the process of *testing*—the application of *input* to the system, and analysis of the resulting *output*, to identify visible failures or other unexpected behaviors [11].

During testing, a *test suite* containing one or more *test cases* is applied to the SUT. A test case consists of a *test sequence (or procedure)*–a series of interactions with the SUT–with *test input* applied to some component of the SUT. Input can range from a method call, to an API call, to an action taken within a graphical interface, depending on the granularity of the testing effort. Then, the test case will validate the output of the called components against a set of encoded expectations—the *test oracle*—to determine whether the test passes or fails [11]. An oracle can be a predefined specification—encoded in a form usable by the test case—the output of another program, a past version of the SUT, or a model, or even manual inspection performed by humans. Most commonly, the oracle is formulated as a series of assertions on the values of output and stateful attributes [5].

An example unit test is shown in Figure 2.1. The test passes a string to the constructor of the `TransformCase` class, then calls its `getText()` method to transform the string to upper-case. An assertion is used as an oracle to check whether the output is an upper-case version of the provided string.

Machine Learning: ML approaches construct models from observed data—and the structure of that data—to make decisions [29]. Instead of being explicitly programmed

with a set of instructions like in traditional software, ML algorithms "learn" from observations using statistical analyses, facilitating the automation of decision-making processes. Learning begins with the search for patterns in a given dataset and, depending on the algorithm employed, may improve through new interactions over time.

ML approaches largely fall into three categories: supervised, unsupervised, and reinforcement learning [29]. In supervised learning, algorithms use previously labeled "training" data to infer a model that makes predictions about newly encountered data. In contrast to supervised methods, unsupervised algorithms do not make use of previously labeled data. Instead, approaches identify patterns based on the similarities and differences between data items. Rather than labeling items, unsupervised approaches are often used to cluster data and detect anomalies. Reinforcement learning algorithms select actions given their estimation of their ability to achieve some in-built goal, using feedback on the effect of the actions taken to improve their estimation of how to maximize achievement of this goal [30]. Such algorithms are often the basis of automated processes, such as game bots or autonomous driving.

Recent "deep learning" (DL) approaches—often supervised–can make complex and highly accurate inferences from massive datasets that would be impossible in traditional ML approaches. This is because DL has an architecture inspired by organic neural networks that attempts to mimic how the human brain works [31] using nonlinear processing layers where one layer's output serves as the successive layer's input. Deep learning requires a computationally intense training process and larger quantities of data than traditional supervised ML, but can learn highly accurate models, extract features and relationships from data automatically, and potentially apply models across applications.

Related Work: To date, we are aware of no other systematic literature reviews dedicated to the use of ML to generate test oracles. However, there are secondary studies that cover overlapping topics. Most relevant is the survey on test oracles by Barr et al. [5]. Their survey thoroughly summarizes research on test oracles up to 2014. They divide test oracles into four broad types, including those specified by human testers, those derived automatically from development artifacts, those that reflect implicit properties of all programs, and those that rely on a human-in-the-loop to judge test results. Approaches based on ML belong to the "derived" category, as they learn automatically from project artifacts to replace or augment human-written oracles. They discuss early approaches to using ML to derive oracles.

Durelli et al. performed a systematic mapping study on the application of ML to software testing [6]. Their scope is broader, but they do note that ML has been applied to support test oracle construction. They find that supervised learning is the most-used family of ML techniques overall software testing topics and that Artificial Neural Networks are the most used algorithm.

Our study differs from the above through its focus specifically on the use of ML in oracle generation. This focus allows detailed analysis of this research area that is absent from broader surveys and mapping studies. Our study is also able to reflect more recent research than that covered in older studies.

2.3 Methodology

Our concern in this work is to understand how researchers have used machine learning (ML) to perform, or otherwise enhance, automated test oracle generation. We have investigated contributions to the literature related to this topic and seek to understand their methodology, results, and insights. To achieve this task, it is necessary to carry out a secondary study—specifically a Systematic Literature Review (SLR) [32]. This section describes how we conducted our SLR.

We are interested in assessing the *effect* of integrating ML into the oracle generation process, understanding the *adoption* of these techniques—how and why they are being integrated, and which specific techniques are being applied, and identifying the potential *impact* and *risks* of this integration. Table 2.1 lists the research questions we are interested in answering, briefly defines why those questions are important, and lists the properties extracted from primary studies to answer them (defined in Section 2.3.3).

Questions 1-3 allow us to understand how ML techniques have enhanced oracle generation, why they were applied, and which specific oracle types were targeted. **RQ2** is motivational, covering the authors' primary objectives. In contrast, **RQ3** expressly is a technical question, examining the specific roles of the included ML techniques, as well as its training and validation processes.

RQ4 examines which ML techniques were used to perform the generation task, as well as *why* that specific method was adopted, if the authors provide such information. **RQ5** focuses on how the oracle generation approach is evaluated. Finally, **RQ6** aims to cover the limitations of the proposed approaches, open issues, and insights that we have uncovered in this area. To answer these questions, we have done the following:

[a] Formed a list of studies (Section 2.3.1).
[b] Filtered this list for relevance (Section 2.3.2).
[c] Extracted data from each study, guided by a set of properties of interest (Section 2.3.3).
[d] Identified trends in the extracted data (Section 2.4).

2.3.1 Initial Study Selection

To locate studies for consideration, a search was conducted using four databases: IEEE Xplore, ACM Digital Library, Science Direct, and Scopus. We created a search string to narrow the results by combining terms of interest regarding automated test generation and machine learning. Note that our search was purposefully broad, intended to capture studies using ML to enhance both input and oracle generation. This approach allowed us to capture a wide range of studies, including those that a narrow search would miss. We then filtered the pool for relevancy. Each database uses a different search engine, and the search options and search formulation slightly vary between them. In general, the search string used was:

(“test case generation” OR “test generation” OR “test oracle” OR “test input”) AND (“machine learning” OR “reinforcement learning” OR “deep learning” OR “neural network”)

These keywords are not guaranteed to capture all existing relevant articles. However, they are designed to capture a sufficiently wide sample to answer our research

questions. Specifically, we combine terms related to test case generation—including specific test components—and terms related to machine learning—including common technologies.

Our focus is specifically on the use of ML in oracle generation, not on any form of automated oracle generation. To obtain a representative sample, we have selected ML-related terms that we expect will capture a wide range of studies. These terms may omit some oracle generation techniques that could be in-scope, but allow us to obtain a representative sample while controlling the number of studies that require manual inspection.

Before exporting the results, we applied an initial filter to the results using the advanced search option in each database, which consists of the following selection criteria: (a) published studies in conferences and journals (excluding grey literature such as pre-prints, technical reports, abstracts, editorials, and book chapters); (b) studies published before November 2020 (when we conducted the search); (c) studies written in the English language. After exporting all results, a total of 1936 studies were identified. This is shown as the first step in Figure 2.2.

To evaluate the search string's effectiveness, we conducted a three-step verification process. First, we randomly sampled ten entries from the 73 studies that remained following the manual filtering. Then we looked in each article for ten citations that might also be in scope, resulting in a list of 100 citations. We checked whether the search string also retrieved the citations in the list, and all 100 were retrieved by the string (pre-filtering). Although this is a small sample, it indicates the robustness of the search string.

After the search, the next step was to identify whether secondary studies already existed on this topic. If so, the need for this SLR would be reduced. We found no previous secondary studies focusing specifically on ML-based oracle generation. However, we identified a small number of related studies. These are discussed in Section 2.2.

2.3.2 Selection Filtering

The initial search resulted in 1,936 studies. It is unlikely that all would be relevant. Therefore, we applied a series of filtering steps to obtain a focused list. Figure 2.2 presents the filtering process and the number of entries after applying each filter. The tag in the center of box 1 represents the 1,936 studies exported from the search and added to the list. The tags in the other boxes represent the number of entries removed in that particular step. The numbers between boxes show the total number of articles that resulted after applying the previous step. Finally, the highlighted box at the end shows the final number of studies used to answer our research questions.

To ensure relevancy, we used a set of keywords to filter the list. We first searched the title and abstract of each study for the keyword "test". This step removed 834 articles. We then searched the resulting list for either "learning" or "neural"—representing the application of machine learning. Every article from IEEE Xplore and Scopus passed these filters. However, the number of articles from the ACM Digital Library and Science Direct was significantly reduced. We merged the filtered lists for both keywords. Some studies contained both keywords in the title or abstract. To remove these, as well as any studies that were returned by multiple databases, we removed all duplicate entries, which resulted in 626 remaining studies. We then removed 22 secondary studies, leaving 604 studies.

Table 2.1: List of research questions, along with motivation for answering the question.

ID	**Research Question**	**Objective**
RQ1	Which oracle types have been generated using ML?	Highlights test oracle types (e.g., information used to issue verdicts) targeted for ML-enhanced oracle generation.
RQ2	What is the goal of using machine learning as part of oracle generation?	To understand the reasons for applying ML techniques to perform or enhance oracle generation (e.g., potential benefits, expected result).
RQ3	How was machine learning integrated into the process of oracle generation?	Identifies how the ML technique was applied as part of the process of oracle generation, and specify its training and validation steps.
RQ4	Which ML techniques were used to perform or enhance oracle generation?	Identify specific ML techniques used in the process, including type, learning method, and selection mechanisms.
RQ5	How is the oracle generation process evaluated?	Describe the evaluation of the oracle generation process, highlighting any artifacts (programs or datasets) they relied on.
RQ6	What are limitations and open challenges in ML-based oracle generation?	Highlights the limitations of oracle generation, such as data dependency, accuracy, or training time, and challenges that must be overcome to apply oracle generation in the field.

Table 2.2: List of properties used to answer the research questions. For each property, we include a name, the research questions the property is associated with, and a short description.

ID	**Property Name**	**RQ**	**Description**
P1	Test Oracle Type	RQ1, RQ2	The specific type of oracle focused on by the approach. It helps to categorize the studies, enabling comparison between contributions.
P2	Proposed Research	RQ2	A short description of the approach proposed or research performed.
P3	Hypotheses and Results	RQ1, RQ3	Highlights the differences between expectations and conclusions of the proposed approach.
P4	ML Integration	RQ3	Covers how ML techniques have been integrated into the oracle generation process. It is essential to understand what aspects of generation are handled or supported by ML.
P5	ML Technique Applied	RQ4	Name, type, and description of the ML technique used in the study.
P6	Reasons for Using the Technique	RQ4	The reasons stated by the authors for choosing this particular ML technique.
P7	ML Training Process	RQ4	How the approach was trained, including the specific data sets or artifacts used to perform this training. Helps us understand how contributions could be replicated or extended.
P8	External Tools or Libraries Used	RQ4	External tools or libraries used to implement the ML technique.
P9	ML Objective and Validation Process	RQ4, RQ5	The objective of the ML technique (i.e., validation metric), and how it is validated, including data, artifacts, and metrics used (if any).
P10	Oracle Creation Evaluation Process	RQ5	Covers how the ML-enhanced oracle generation process, as a whole, is evaluated (i.e., how successful are the generated oracles at detecting faults or meeting some other testing goal?). Allows understanding of the effects of ML on improving the testing process.
P11	Potential Research Threats	RQ6	Notes on the threats to validity that could impact each study.
P12	Strengths and Limitations	RQ6	Used to understand the general strengths and limitations of enhancing oracle creation with ML.
P13	Future Work	RQ6	Any future extensions proposed by the authors, with a particular focus on those that could overcome identified limitations.

We examined the remaining studies manually, removing all not in scope following an inspection of the title and abstract. We removed any studies not related to software test generation or that do not apply ML during the test generation process (i.e., the

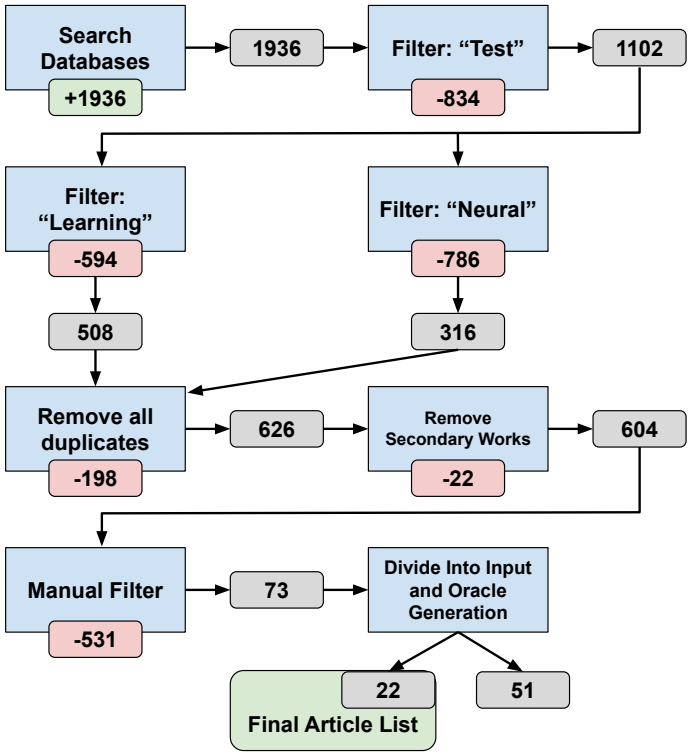


Figure 2.2: Steps taken to determine the final list of studies.

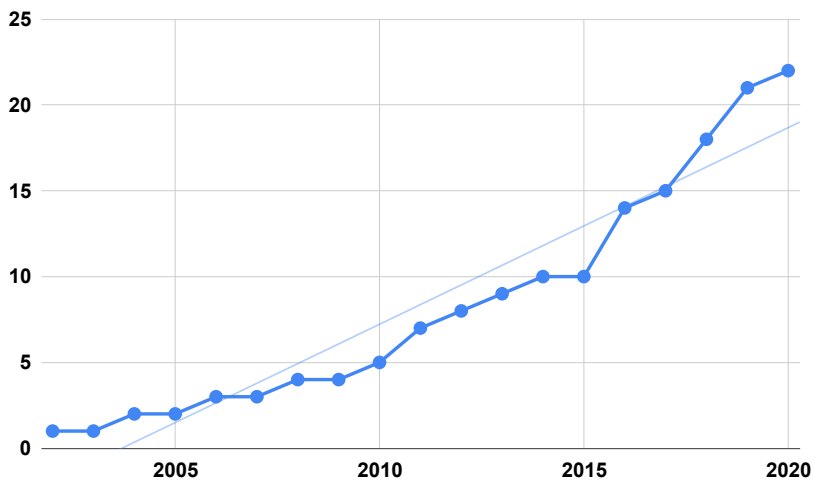


Figure 2.3: Growth in the number of publications in ML-based oracle generation from 2002-2020.

ML element is related to a particular activity such as test suite reduction). This determination was made by first reading the abstract of the paper, then the introduction, then the full paper, until a clear determination could be made of the relevancy of the study. Both authors independently inspected studies during this step to prevent the accidental removal of relevant studies. In cases of disagreement, the authors discussed

and came to a conclusion. This left 73 studies. Finally, we divided these studies into those related to input or oracle generation. This step resulted in a final total of 22 studies related to oracle generation for consideration.

Figure 2.3 shows the rate of growth in this emerging research area. The first study in our sample was published in 2002 and the most recent in 2020. Interest in this topic is growing with the emergence of new and more powerful ML approaches, with over half of the studies having been published since 2016.

2.3.3 Data Extraction

To answer the questions listed in Table 4.1, we have examined each study. We have focused on a set of key properties, identified in Table 2.2. Each property listed in the table is briefly defined and is associated with the research questions that it will help answer. In many cases, several properties are collectively used to answer a RQ. For example, the answer to RQ2, which aims to cover the goals of using ML as part of the automated test generation process, can be extracted from property P2 in many cases. However, P1 is related because it provides context to the research and the particular type of test oracle may dictate how ML is applied. Each property is important in capturing the essential details of the study and how it contributes to answering our RQs.

In reported experiments, the proposed approach either exceeded or failed to meet the initial hypotheses. This is covered by the third property, P3, which could lead to or be part of the answer for RQ1 and RQ3. The fourth property targets RQ3 and notes how the adopted ML technique is integrated into the testing process. To understand how ML techniques can enhance automated test generation, it is important to understand which techniques are applied as well as the motivation behind adopting a specific technique. These aspects are covered by P5 and P6, which are used to answer RQ4. We also note whether the project analyzed is new or the continuation of prior research as part of collecting data for these properties.

The following three properties focus on understanding the application of ML in the study, including a partial assessment of the potential to replicate the research, by covering core characteristics of the ML technique—the training process (P7), external tools used to implement the technique (P8), and the validation process (P9). P7 focuses on the datasets or other information sources used to train the learning technique. Our primary focus with P8 is to cover how external tools, environments, or ML libraries—such as TensorFlow or Keras—are used to train, build, or execute the ML technique. The combination of properties P7, P8, and P9 will answer RQ4, which examines how the ML technique is trained, validated, and assessed as part of its integration. RQ5 examines how the entire oracle generation process is evaluated. P10 is primarily used to answer this research question. However, P9 may also help answer this question.

Research question RQ6 covers open challenges. Properties P11-P13 contribute to answering this question, including limitations and threats to validity—either disclosed by the authors or inferred from our analysis—and future work.

Data extraction was performed primarily by the first author of this study. However, to ensure the accuracy of the extraction process, the second author performed an independent extraction for a randomly-chosen sample of the studies. We compared our findings, and found that we had near-total agreement.

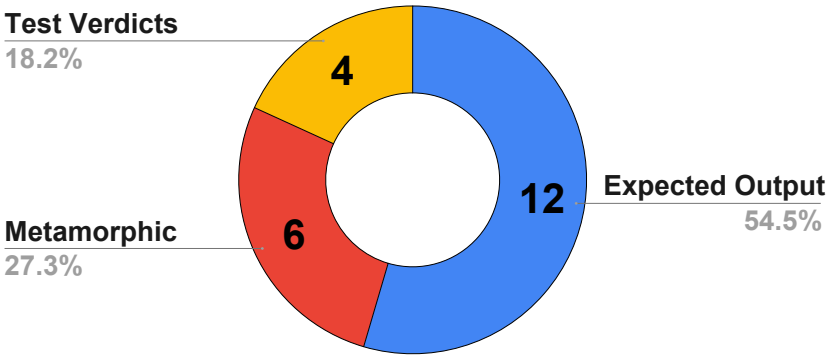


Figure 2.4: The types of oracles generated, and the number of studies where this type of oracle is generated.

2.4 Results and Discussion

We divide the examination of the results as follows: the types of oracles generated using ML and why ML was applied (RQ1-2, Section 2.4.1), how ML was applied in the examined studies (RQ3-5, Section 2.4.2), and the limitations and open challenges in this emerging research field (RQ6, Section 2.4.3).

We divide the examination of the results as follows: the types of oracles generated using ML and why ML was applied (RQ1-2, Section 2.4.1), how ML was applied in the examined studies (RQ3-5, Section 2.4.2), and the limitations and open challenges in this emerging research field (RQ6, Section 2.4.3).

2.4.1 Test Oracle Types and Motivation

Before examining which ML techniques have been integrated into oracle generation, or how they have been integrated, it is first crucial to understand *why* they have been integrated. A test oracle is a broad, high-level concept—simply *some* means to judge the correctness of the system given test input [5]. Therefore, our first two research questions are intended to give an overview of the specific types of oracle that have been the focus of the collected studies (RQ1) and to provide motivation for why ML was applied as part of creating these oracles. Figure 2.4 shows our results. Broadly, three types of oracles have been generated in the examined studies:

- **Test Verdicts:** The approach directly predicts the final test verdict, given provided input. For example, this type of oracle might directly issue a "pass" or "fail" verdict for the test case.
- **Expected Output:** The approach predicts specific system behavior that should result from applying the provided input [33]. The predicted behavior can vary in its level of abstraction, from a concrete output to a broad *class* of output—generally leaning more towards the abstract, given the challenges of making specific predictions for complex systems.
- **Metamorphic Relations:** A metamorphic relation is a necessary property of a function, relating input to the output produced [34]. For example, a metamorphic relation for a $sine$ function is $sin(x) = sin(\pi - x)$. Such relations allow us to infer expected results for different input values to a function, and violations of

such properties identify potential faults. Approaches in this category attempt to learn metamorphic relations for new systems from provided data.

Of the 22 collected studies, a majority—12 approaches—produce expected output oracles. Six produce metamorphic relations, and four produce direct test verdicts.

The goal of ML is to automate or support a decision process. Given an observation, a ML technique can make a prediction. That prediction can either be the final decision to be made, or it can relate to a piece of information needed to make that decision. Test oracles follow a similar conceptual model. Test oracles consist of two core components—the oracle *information*, or a set of facts used to issue the verdict on the test case, and the oracle *procedure*, the actions taken to issue a verdict based on the embedded information and observations of system behavior [35]. Motivationally, we can see that ML offers a natural means to replace either the oracle information—which typically requires human effort to specify—or the oracle as a whole. Test verdict oracles perform the entire decision process, directly issuing a verdict.

The other two oracle types, expected outputs and metamorphic relations, replace human specification of oracle information with a model that predicts that information instead. The procedure can then act on that prediction rather than relying on human-specified facts.

RQ1 (Oracle Types): Machine Learning algorithms have been used to generate test verdict (18%), metamorphic relation (27%), and expected output (55%) oracles.

RQ2 (Goal of ML): ML algorithms train models that serve either as a stand-in for a test oracle or to learn information about a function (e.g., expected output or metamorphic relations) that can be used as part of issuing a verdict.

2.4.2 Application of Machine Learning

Table 2.3 summarizes relevant data gathered from the 22 studies where ML was used to generate test oracles. Immediately, we can see that almost all approaches adopted a supervised approach, where a model is trained and used to make predictions about new input. Unsupervised and reinforcement learning (RL) have been used as part of input generation. These approaches may also be applicable as part of oracle generation—e.g., an oracle modeled as a RL agent could make predictions and get feedback on their accuracy, or an unsupervised clustering approach could be used as part of an oracle that detects anomalies. One study did propose the use of RL-like techniques as part of metamorphic relation generation. However, the focus has been firmly on supervised learning.

The sampled studies train oracles using a set of previously-captured and labeled system executions or metadata about source code features. The model is then used to predict the correctness of new behaviors or to predict the type of behavior that will result from applying the input. We will discuss each oracle type in turn.

Test Verdicts: All studies within this category applied a ML technique to associate patterns in the training data with the resulting test verdict (i.e., they trained a model for the purpose of regression). This approach enables the oracles generated to assert whether a test passes or fails without running the SUT.

Table 2.3: Data on the sampled studies, including the type of ML approach, specific ML technique, training data used, the targeted goal of the ML approach, how the approach was evaluated, and the type of application used in the evaluation.

Ref	Year	Oracle Type	ML Approach	Technique	Training Data	ML Objective	Evaluation Metric	Evaluated On
[36]	2018	Test Verdicts	Supervised	Adaptive Boosting	System Executions	Regression	Mutation Score	Shopping Cart
[37]	2018	Test Verdicts	Supervised	Backpropagation NN	System Executions	Regression	Mutation Score	Embedded Software
[38]	2016	Test Verdicts	Supervised	Multilayer Perceptron	System Executions	Regression	Accuracy	User Creation
[39]	2010	Test Verdicts	Supervised	Backpropagation NN	System Executions	Regression	Mutation Score	Student Registration
[40]	2004	Expected Output	Supervised	Backpropagation NN	System Executions	Regression	Correct Classifications	Triangle Classification
[41]	2016	Expected Output	Supervised	SVM	System Executions	Label Propagation	Mutation Score	Image Processing
[42]	2008	Expected Output	Supervised	Backpropagation NN	System Executions	Regression	Correct Classifications	Triangle Classification
[43]	2019	Expected Output	Supervised	Deep NN	System Executions	Regression	Mutation Score	Mathematical Functions
[44]	2011	Expected Output	Supervised	RBF NN	System Executions	Regression	Correct Classifications	Triangle Classification
[45]	2011	Expected Output	Supervised	Multilayer Perceptron	System Executions	Regression	Mutation Score	Insurance Application
[46]	2012	Expected Output	Supervised	Multilayer Perceptron	System Executions	Regression	Mutation Score	Insurance Application
[47]	2016	Expected Output	Supervised	Backpropagation NN + Cascade	System Executions	Regression	Accuracy	Credit Analysis
[48]	2002	Expected Output	Supervised	Not Specified	System Executions	Regression	Mutation Score	Credit Analysis
[49]	2014	Expected Output	Supervised	Backpropagation NN	System Executions	Regression	Mutation Score	Triangle Classification
[50]	2006	Expected Output	Supervised	Multilayer Perceptron	System Executions	Regression	Mutation Score	Mathematical Functions
[51]	2019	Expected Output	Supervised	Probabilistic NN	System Executions	Regression	Correct Classifications	Prime, Triangle Class
[34]	2018	Metamorphic	Supervised	SVM	Code Features	Label Propagation	Accuracy	Various Functions
[52]	2020	Metamorphic	Reinforcement	Not Specified	System Executions	Discovered Relations	Not Evaluated	Ocean Modeling
[53]	2013	Metamorphic	Supervised	SVM, Decision Trees	Code Features	Regression	Mutation Score	Various Functions
[54]	2016	Metamorphic	Supervised	SVM	Code Features	Regression	Mutation Score	Various Functions
[55]	2019	Metamorphic	Supervised	SVM	Code Features	Label Propagation	ROC	Matrix Calculation
[56]	2017	Metamorphic	Supervised	RBF NN	Code Features	Multi-label Regression	Accuracy	Various Functions

Makondo et al. [38] utilize a Multilayer Perception (MLP) Neural Network (NN)—a basic NN, often constructed with a single hidden layer. Shahamiri et al. [39] and Gholami et al. [37] utilized Feed-forward Backpropagation (BP) NNs to create their test oracles. A BP NN "learns" by reducing error rates by tuning the weights in each neuron after computing the error, making the model more stable. Braga et al. [36] use a classifier based on adaptive boosting.

Braga et al. [36] gather usage data from a shopping website by inserting several specific capture components into the site. The data then goes through a preprocessing step and then is finally used for training the ML. Shahamiri et al. [39] focus on a student registration-verifier application that checks whether a students' records satisfy the minimum requirements for enrollment. Gholami et al. [37] focus on embedded systems in their evaluation. Makondo et al. [38] examined a user creation function. Braga et al. [36], Gholami et al. [37] and Shahamiri et al. [39]) evaluate their approaches using the mutation score. They insert synthetic faults, and measure how many of these faults that the generated oracle can detect. Makondo et al. [38] evaluate using the accuracy of the classification model.

Expected Output: More than half of the studies generate expected output oracles. The approaches train on system executions, and then predict the output given a new input. Often, the level of detail of the output generated is constrained or abstracted to a small set of representative values, rather than attempting to predict highly specific output. For example, rather than yielding a specific integer for integer output, the approach might constrain the output to a limited set of representative values (classifications) and predict one of those values. Otherwise, evaluation is limited to code that issues output from an enumerated set of values. A common application is the "triangle classification problem," also known as TRITYP [40, 44, 44, 49, 51]. The program receives three numbers representing the lengths of a triangle's sides and outputs a classification of the type of triangle as scalene, isosceles, equilateral, or not a triangle. This is a problem that can prove challenging given its branching behavior. However, it still has a limited set of output possibilities. This makes it a reasonable starting point for oracle generation.

Zhang et al. [51] also model a function that judges whether an integer is prime or not. This is an even more straightforward application—a two-class classification problem. Shahamiri et al. [45, 46] generate oracles for a car insurance application, while Singhal et al. [47] and Vanmali et al. [48] generate oracles for a credit analysis at a bank. Ding et al. [41] generate oracles for an image processing function that classifies a type of cell from image sections. All of these applications produce output from an enumerated set of values, easing the difficulty of generating an oracle.

Ye et al. [50] and Monsefi et al. [43] generate oracles for functions with integer output. Some of the cases they examine have a limited range of produced outputs (e.g., a function that predicts the length of a route). Still, the remaining functions offer some indication that deep learning can model more complex functions or predict more detailed expected output.

Ding et al. [41] used a support vector machine (SVM) to perform label propagation. Label propagation is a semi-supervised learning technique, where a mixture of labeled and unlabeled training data is used to train the model, and the algorithm attempts to propagate labels from the labelled data to similar, unlabeled data. This can reduce the quantity of training data needed.

The other approaches follow a more traditional supervised, regression-based learning process, and generally make use of different NNs. Four of the examined studies

adopt a Backpropagation NN [40,42,47,49]. Three other studies employ the Multilayer Perceptron technique [45, 46, 50]. Sangwan et al. uses a Radial Basis Function (RBF) NN [44]. RBF is a specific activation function applied to the inputs of the network. Monsefi et. al [43] adopt a Deep NN, which has more input and output layers than a regular NN, with a fuzzy encoder + decoder. Finally, Zhang et. al adopt a probabilistic NN [51].

In terms of evaluation, five of the studies are focused on the accuracy of the oracle in a set of cases where the ground truth is known—measuring the percentage of correct classifications [40, 42, 44, 51] or the accuracy of the model [47]. The remaining seven used the mutation score as the evaluation metric [41, 43, 45, 46, 48–50].

Metamorphic Relations: Six approaches generate metamorphic relations—properties of a function that explain how particular input links to corresponding output [34]. Such relations allow us to infer expected results for different input values to a function, and violations of such properties identify potential faults.

Several of the examined studies build on the initial ideas of Kanewala et al. [53], where they proposed an approach that (a) converts the source code of functions into control-flow graphs, (b) selects source code elements as features for a data set, (c) train a model that can predict whether a feature exhibits a particular metamorphic relation (selected from a pre-compiled list of relations). This requires a set of training data, where features are labeled with a binary classification based on whether or not they exhibit that particular relation. A SVM and Decision Trees are used to train the predictive model. Kanewala et al. extended this work by adding a graph kernel to the process [54]. Hardin et al. adapted this approach to work with a semi-supervised label propagation algorithm [34]. Finally, Zhang et al. [56] experimented with the use of a RBF NN. They extended the approach to a multi-label classification that can handle multiple metamorphic relations at once instead of predicting one at a time. All four of these studies are evaluated on a variety of functions, from mathematical functions, to data structures, to sorting operations. They were evaluated either using the mutation score or accuracy measurements.

Nair et al. [55] extended this work by demonstrating how data augmentation can enlarge the training dataset by using mutants as the source of the additional training data. They compared the enlarged dataset to the original dataset on a set of 45 matrix calculation functions in terms of the Receiver Operating Characteristic, or the ratio of true positive to false positive classifications.

Hiremath et al. [52] propose an approach for using an ML algorithm to predict metamorphic relations for an ocean modeling application. The approach would post a set of relations, evaluate whether they hold, and attempt to minimize a cost function based on the validity of the set of proposed relations. They do not specify an approach, but this maps to common applications of Reinforcement Learning. They do not evaluate their approach, but plan to develop and evaluate it in future work.

We can answer RQ3-5 as follows:

RQ3 (Integration of ML): Almost all studies (96%) employed a supervised or semi-supervised approach, trained on labeled system execution logs or source code metadata and validated using the accuracy of the trained model.

RQ4 (ML Techniques): 59% of the approaches employed a NN—including Backpropagation NNs (27%), Multilayer Perceptrons (18%), RBF NN (9%), probabilistic NN (5%), and Deep NN (5%). 23% of approaches adopted support vector machines. One also adopted decision trees (5%), and used adaptive boosting (5%). 5% did not specify a technique.

RQ5 (Evaluation of Approach): Results were most often evaluated using the mutation score (55%), followed by number of correct classifications (18%), classification accuracy (18%), and ROC (5%). One study did not perform evaluation.

2.4.3 Limitations and Open Challenges

The sampled studies show great promise. They illustrate the potential for solving the oracle problem. However, we have observed multiple limitations and challenges that must be overcome to transition research into use in real-world software development.

Volume, Contents, and Collection of Training Data: Supervised ML approaches, even semi-supervised approaches, require training data to create the predictive model that serves as the test oracle. There are multiple challenges related to the *required volume* of training data, the *required contents* of the training data, and the *human effort* required to produce that training data.

Regardless of the specific type of oracle, the volume of training data that is needed can be vast. This data is generally attained from labeled system execution logs, which means that the SUT needs to be executed *many* times to gather the information needed to train the model. Approaches based on deep learning could produce highly accurate oracles, but may require thousands of executions to gather the required training data. Some approaches also must preprocess the collected data before training. The time required to produce the training data can be high and must be considered.

This is particularly true for expected value oracles. Even if the output is abstracted into a small pool of representative values, predicting one of several values is a more difficult task than a boolean classification, and requires significant training data for *each* of the values that can result to make accurate classifications. In addition, the training data for expected value oracles must come from passing test cases—i.e., the output must be what was expected—or labels must be hand-applied by humans. A small number of cases based on failing output may be acceptable if the algorithm is resilient to noise in the training data, but training on faulty code can easily result in an inaccurate model. This introduces a significant barrier to automating training by, e.g., generating test input and simply recording the output that results.

Oracles that produce a direct test verdict model a simpler classification problem—is the result a pass or a fail? However, the requirements on the contents of the underlying data are significant. Each entry in the dataset must be assigned a verdict in order to train the model. This requires either existing test oracles—reducing the need for a ML-based oracle in the first place—or human labeling of test results. Humans are limited in their ability to serve as an oracle, as judging test results is time-consuming and can be erroneous as tester becomes fatigued [5, 57]. This makes it difficult to produce a significant volume of training data. Further complicating this problem is the fact that training a test verdict oracle requires the training data to contain a large

number of *failing test cases*. This implies that faults have already been discovered in the system and, presumably, fixed before the oracle is trained. This also will reduce the potential effectiveness of a ML-based oracle.

Metamorphic relation oracles face a similar dilemma. In many of the approaches, the training data consists of source code features labeled with a classification representing whether a particular type of metamorphic relation holds over that feature. This training data must be hand-labeled by a human tester with knowledge of whether these relations hold or not. This requires significant up-front knowledge and effort to establish the ground truth.

Regardless of the oracle type, generating oracles for complex systems will require ML techniques that can extrapolate from limited training data and that can tolerate noise in the training data. Means of generating synthetic training data, like in the work of Nair et al. [55], demonstrate the potential for *data augmentation* to help in overcoming this limitation.

Retraining and Feedback: After training, models generated by supervised learning techniques have a fixed error rate and do not learn from new mistakes made after training. In other words, if the training data is insufficient or inaccurate, the generated oracle will remain inaccurate as long as it remains in use. The ability to improve the oracle based on additional feedback after training could help account for limitations in the initial training data.

There are two primary means to overcome this limitation—either retraining the model using an enriched training dataset, or adopting a reinforcement learning approach that can adapt its expectations based on attained feedback on the accuracy of its decisions. Both means carry challenges. Retraining requires (a) establishing a schedule for when to train the updated model, and (b), an active effort on the part of human testers to enrich and curate the training dataset. Enriching this dataset—as well as the use of RL—requires some kind of feedback mechanism to judge the accuracy of the oracle. This is likely to require human feedback on, at least, a subset of the decisions made, reducing the potential cost savings.

RQ6 (Challenges): Oracle generation is limited by the required quantity, quality, and content of training data. Assembling training data may require significant human effort. Models should be retrained over time.

Complexity of Modeled Functionality: Many approaches are demonstrated on highly simplistic functions, with only a few lines of code and a small number of possible outputs. While it is intuitive to *start* with highly simplistic examples to examine the viability of an approach, application of such techniques in the field would require oracle generation for far more complex system functions. If a function is simple, there is likely little need for oracle generation in the first place. It remains to be seen whether generated oracles can predict the output of real-world production code, or even simple code with an unconstrained or lightly constrained output space.

Generation of an expected output oracle that can model any arbitrary function with unconstrained output may be prohibitively difficulty for even the most effective ML techniques available today. Some abstraction should be expected. One possibility to consider is a variable level of abstraction—e.g., a training-time decision to cluster the output into an adjustable number of representative values (i.e., the centroid of each cluster). Training could take place over different settings for this parameter, and an acceptable balance between quality and level-of-detail could be explored.

Variety, Complexity, and Tuning of ML Techniques: Many of the proposed approaches—especially the earlier ones—are based on simple neural networks with only a few hidden layers. These techniques have strict limitations in the complexity of the functions they can model, and have been superseded by newer ML techniques. Deep learning techniques, which may utilize a high number of hidden layers, may be key in building models of more complex functions. One approach to date has utilized deep learning [43], and we would expect more to explore these techniques in the coming years. However, deep learning also introduces steep requirements on the training data that may limit its applicability [58].

Almost all of the proposed approaches are based on a single ML technique. An approach explored in other domains is the use of *ensembles* [59]. In such approaches, models are trained on the same data using a variety of techniques. Each model is asked for a prediction, then the final prediction is based on the consensus of the ensemble. Ensembles are often able to reach stable, accurate conclusions in situations where a single model may be inaccurate. Ensembles may be a way to overcome the fragility of current oracle generation approaches.

Many ML techniques have a number of *hyperparameters* that can be tuned (e.g., the learning rate, number of hidden units, or activation function) [60]. Hyperparameter tuning can have a major impact on model accuracy, and can enable significant improvements in the results of even simple ML techniques. The proposed approaches do not explore the impact of hyperparameter tuning on the trained models. This is an oversight that should be corrected in future work.

RQ6 (Challenges): Applied techniques may be insufficient for modeling complex functions with many possible outputs. Varying degrees of output abstraction should be explored. Deep learning and ensemble techniques, as well as hyperparameter tuning, should be explored.

Lack of a Standard Benchmark: The emergence of bug benchmarks (e.g., [25, 61]) has enabled sophisticated analyses and comparison of approaches to automated input generation and program repair. To date, oracle generation has often been evaluated on case examples—often over-simplistic examples—where code or metadata is unavailable. This makes comparison and replication difficult.

The creation of a benchmark for oracle generation research could advance the state-of-the-art in the field, spur new research advances, and enable replication and extension of proposed approaches. Such a benchmark should contain a variety of code examples from multiple domains and of varying levels of complexity, allowing the field to move beyond over-simplistic examples. Code examples should be paired with the metadata needed to support oracle generation. This would include sample test cases and human-created test oracles, at minimum. Such a benchmark could also include sample training data that could be augmented over time by researchers.

Lack of Replication Package or Open Code: A common dilemma in software engineering research is lack of access to the code built by researchers or the data used to draw conclusions. Often, the paper itself is not sufficient to allow replication or application of the technique in a new context. This applies to research in oracle generation as well. Some studies make use of open-source ML frameworks (e.g., scikit-learn). This is positive, in that the tools are trustworthy and available. However, without the authors' code and data, there may not be enough information to enable

replication. Further, these frameworks themselves evolve over time, and the attained results may differ because the underlying ML technique has changed since the original study was published.

New approaches should include a replication package with the source code written by the authors, execution scripts, and the versions of external dependencies that were used at the time that the study was performed. This should also include data used by the authors in their analyses.

RQ6 (Challenges): Research is limited by overuse of simplistic examples, the lack of common benchmarks, and the unavailability of code and data. A benchmark should be created for evaluating oracle research, and researchers should be encouraged to provide replication packages and open code.

2.5 Threats to Validity

External and Internal Validity: Our conclusions are based on the studies sampled. It is possible that we may have omitted important studies or sampled an inadequate number of studies. This can affect internal validity—the evidence we use to make conclusions—and external validity—the generalizability of our findings. SLRs are not required to reflect all studies from a research field. Rather, their selection protocol (search string, inclusion and exclusion criteria) should be sufficient to ensure an adequate sample of the field. We believe that our selection strategy was appropriate. We tested different search strings, and performed a validation exercise to test the robustness of our string. We have used four databases, covering the majority of relevant software engineering venues. Our final set of studies includes 22 primary studies, which we believe is sufficient to make informed conclusions.

Conclusion Validity: The analyses performed are qualitative, and require inference from the authors. This could introduce bias into our conclusions. For example, subjective judgements are required as part of article selection, data extraction, and coding (e.g., categorizing studies based on the oracle type). To control for bias, protocols were discussed and agreed upon by both authors, and independent verification took place on—at least—a sample of all decisions made by either author.

Construct Validity: We used a set of properties to guide data extraction. These properties may have been incomplete or misleading. However, we have tried to establish properties that were appropriate and directly informed by our research questions. These properties were iteratively refined using a selection of papers.

2.6 Conclusions

Machine learning has the potential to solve the *"test oracle problem"*—the challenge of automatically generating oracles for a function. We have characterized emerging research in this area through a systematic literature review examining oracle types, researcher goals, the ML techniques applied, how the generation process was assessed, and the open research challenges in this emerging field.

Based on 22 relevant studies, we observed that ML algorithms have been used to generate test verdict, metamorphic relation, and—most commonly—expected output

oracles. The ML algorithms train predictive models that serve either as a stand-in for an existing test oracle—predicting a test verdict—or as a way to learn information about a function—either the expected output or metamorphic relations—that can be used as part of issuing a verdict.

Almost all studies employed a supervised or semi-supervised approach, trained on labeled system executions or source code metadata. Of these approaches, many used some type of neural network—including Backpropagation NNs, Multilayer Perceptrons, RBF NNs, probabilistic NNs, and Deep NNs. Others applied include support vector machines, decision trees, and adaptive boosting. Results were most often evaluated using the mutation score, number of correct classifications, classification accuracy, and ROC.

The studies show great promise, but there are significant open challenges. Generation is limited by the required quantity, quality, and content of training data. Models should be retrained over time. Applied techniques may be insufficient for modeling complex functions with many possible outputs. Varying degrees of output abstraction, deep learning and ensemble techniques, and hyperparameter tuning should be explored. In addition, research is limited by overuse of simplistic examples, lack of common benchmarks, and unavailability of code and data. A robust open benchmark should be created, and researchers should provide replication packages.

2.7 Acknowledgments

This research was supported by Vetenskapsrådet grant 2019-05275.

Paper B

Automated Support for Unit Test Generation

Afonso Fontes, Gregory Gay, Francisco Gomes de Oliveria Neto, Robert Feldt

***Book chapter, Optimising the Software Development Process with Artificial Intelligence. Springer, 2022*.**

Abstract

Unit testing is a stage of testing where the smallest segment of code that can be tested in isolation from the rest of the system—often a class—is tested. Unit tests are typically written as executable code, often in a format provided by a unit testing framework such as `pytest` for Python.

Creating unit tests is a time and effort-intensive process with many repetitive, manual elements. To illustrate how AI can support unit testing, this chapter introduces the concept of *search-based unit test generation*. This technique frames the selection of test input as an optimization problem—*we seek a set of test cases that meet some measurable goal of a tester*—and unleashes powerful *metaheuristic* search algorithms to identify the best possible test cases within a restricted timeframe. This chapter introduces two algorithms that can generate `pytest`-formatted unit tests, tuned towards coverage of source code statements. The chapter concludes by discussing more advanced concepts and gives pointers to further reading for how artificial intelligence can support developers and testers when unit testing software.

3.1 Introduction

Unit testing is a stage of testing where the smallest segment of code that can be tested in isolation from the rest of the system—often a class—is tested. Unit tests are typically written as executable code, often in a format provided by a unit testing framework such as `pytest` for Python. Unit testing is a popular practice as it enables test-driven development—where tests are written before the code for a class, and because the tests are often simple, fast to execute, and effective at verifying low-level system functionality. By being executable, they can also be re-run repeatedly as the source code is developed and extended.

However, creating unit tests is a time and effort-intensive process with many repetitive, manual elements. If elements of unit test creation could be automated, the effort and cost of testing could be significantly reduced. Effective automated test generation could also complement manually written test cases and help ensure test suite quality. Artificial intelligence (AI) techniques, including optimization, machine learning, natural language processing, and others, can be used to perform such automation.

To illustrate how AI can support unit testing, we introduce in this chapter the concept of *search-based unit test input generation*. This technique frames the selection of test input as an optimization problem—*we seek a set of test cases that meet some measurable goal of a tester*—and unleashes powerful *metaheuristic* search algorithms to identify the best possible test input within a restricted timeframe. To be concrete, we use metaheuristic search to produce `pytest`-formatted unit tests for Python programs.

This chapter is laid out as follows:

- In Section 3.2, we introduce our running example, a Body Mass Index (BMI) calculator written in Python.
- In Section 3.3, we give an overview of unit testing and test design principles. Even if you have prior experience with unit testing, this section provides an overview of the terminology we use.

- In Section 3.4, we introduce and explain the elements of search-based test generation, including solution representation, fitness (scoring) functions, search algorithms, and the resulting test suites.
- In Section 3.5, we present advanced concepts that build on the foundation laid in this chapter.

To support our explanations, we created a Python project composed of (i) the class that we aim to test, (ii) a set of test cases created manually for that class following good practices in unit test design, and (iii) a simple framework including two search-based techniques that can generate new unit tests for the class.

The code examples are written in Python 3, therefore, you must have Python 3 installed on your local machine in order to execute or extend the code examples. We target the `pytest` unit testing framework for Python.[1] We also make use of the `pytest-cov` plug-in for measuring code coverage of Python programs[2], as well as a small number of additional dependencies. All external dependencies that we rely on in this chapter can be installed using the `pip3` package installer included in the standard Python installation. Instructions on how to download and execute the code examples on your local machine are available in our code repository at `https://github.com/Greg4cr/PythonUnitTestGeneration`.

[1] For more information, see `https://pytest.org`.

[2] See `https://pypi.org/project/pytest-cov/` for more information.

```
class BMICalc:
    def __init__(self, height, weight, age):
        self.height = height
        self.weight = weight
        self.age    = age

    def bmi_value(self):
# The height is stored as an integer in cm. Here we convert it to
# meters (m).
  bmi_value = self.weight / ((self.height / 100.0) ** 2)
      return bmi_value

    def classify_bmi_adults(self):
        if self.age > 19:
            bmi_value = self.bmi_value()
            if bmi_value < 18.5:
                return "Underweight"
            elif bmi_value < 25.0:
                return "Normal weight"
            elif bmi_value < 30.0:
                return "Overweight"
            elif bmi_value < 40.0:
                return "Obese"
            else:
                return "Severely Obese"
        else:
            raise ValueError(
    "Invalid age. The adult BMI classification requires an age "
                "older than 19.")
```

Figure 3.1: An excerpt of the BMICalc class. The snippet includes the constructor for the BMICalc class, the method that calculates the BMI value according to Equation 3.1, and a method that returns the BMI classification for adults.

3.2 Example System—BMI Calculator

$$BMI = \frac{weight}{(height)^2} \tag{3.1}$$

The formula can be adapted to be used with different measurement systems (e.g., pounds and inches). In turn, the BMI classification uses the BMI value to classify individuals based on different threshold values that vary based on the person's age and gender[3].

The BMI thresholds for children and teenagers vary across different age ranges (e.g., from 4 to 19 years old). As a result, the branching options quickly expand. In this example, we focus on the World Health Organization (WHO) BMI thresholds for cisgender[4] women, who are adults older than 19 years old[5], and children/teenagers between 4 and 19 years old[6]. In Figure 3.1, we show an excerpt of the BMICalc class and the method that calculates the BMI value for adults. The complete

[3]Threshold values can also vary depending on different continents or regions.

[4]An individual whose personal identity and gender corresponds with their birth sex.

[5]See https://www.euro.who.int/en/health-topics/disease-prevention/nutrition/a-healthy-lifestyle/body-mass-index-bmi

[6]See https://www.who.int/tools/growth-reference-data-for-5to19-years/indicators/bmi-for-age

```
1   def classify_bmi_teens_and_children(self):
2      if self.age < 2 or self.age > 19:
3      raise ValueError(
4       'Invalid age. The children and teen BMI classification ' +
5       'only works for ages between 2 and 19.')

6       bmi_value = self.bmi_value()
7       if   self.age <= 4:  ...
8       elif self.age <= 7:  ...
9       elif self.age <= 10: ...
10      elif self.age <= 13: ...
11      elif self.age <= 16: ...
12      elif self.age <= 19: ...
```

Figure 3.2: Method for the BMI classification of several age brackets that, in turn, expand further into the branches of each BMI classification. For readability, the actual thresholds were omitted from the excerpt above.

Table 3.1: Threshold values used between different BMI classifications across the various age brackets. The children and teens reference values are for young girls.

Classification	$[2, 4]$	$(4, 7]$	$(7, 10]$	$(10, 13]$	$(13, 16]$	$(16, 19]$	> 19
Underweight	≤ 14	≤ 13.5	≤ 14	≤ 15	≤ 16.5	≤ 17.5	< 18.5
Normal weight	≤ 17.5	≤ 14	≤ 20	≤ 22	≤ 24.5	≤ 26.5	< 25
Overweight	≤ 18.5	≤ 20	≤ 22	≤ 26.5	≤ 29	≤ 31	< 30
Obese	> 18.5	> 20	> 22	> 26.5	> 29	> 31	< 40
Severely obese	—	—	—	—	—	—	≥ 40

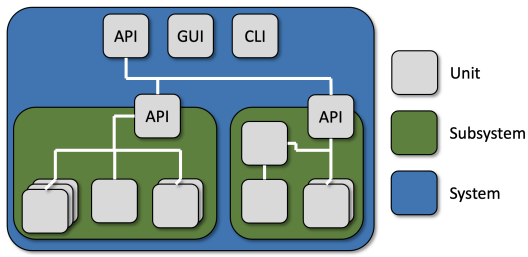


Figure 3.3: Illustration of common levels of granularity in testing. A system is made up of one or more largely-independent subsystems. A subsystem is made up of one or more low-level "units" that can be tested in isolation.

code for the BMICalc class can be found at `https://github.com/Greg4cr/PythonUnitTestGeneration/blob/main/src/example/bmi_calculator.py`.

The BMI classification is particularly interesting case for testing because, (i), it has *numerous branching* statements based on multiple input arguments (age, height, weight, etc.), and (ii), it requires testers to think of specific *combinations of all arguments* to yield BMI values able to cover all possible classifications. Table 3.1 shows all of the different thresholds for the BMI classification used in the BMICalc class.

While the numerous branches add complexity to writing unit tests for our case

example, the use of only integer input simplifies the problem. Modern software requires complex inputs of varying types (e.g., DOM files, arrays, abstract data types) which often need contextual knowledge from different domains such as automotive, web or cloud systems or embedded applications to create. In unit testing, the goal is to test small, isolated units of functionality that are often implemented as a collection of methods that receive primitive types as input. Next, we will discuss the scope of unit testing in detail, along with examples of good unit testing design practices, as applied to our BMI example.

3.3 Unit Testing

Testing can be performed at various levels of granularity, based on how we interact with the system-under-test (SUT) and the type of code structure we focus on. As illustrated in Figure3.3, a system is often architected as a set of one or more cooperating or standalone subsystems, each responsible for a portion of the functionality of the overall system. Each subsystem, then, is made up of one or more "units"—small, largely self-contained pieces of the system that contain a small portion of the overall system functionality. Generally, a unit is a single class when using object-oriented programming languages like Java and Python.

Unit testing is the stage of testing where we focus on each of those individual units and test their functionality in *isolation* from the rest of the system. The goal of this stage is to ensure that these low-level pieces of the system are trustworthy before they are integrated to produce more complex functionality in cooperation. If individual units seem to function correctly in isolation, then failures that emerge at higher levels of granularity are likely to be due to errors in their *integration* rather than faults in the underlying units.

Unit tests are typically written as executable code in the language of the unit-under-test (UUT). Unit testing frameworks exist for many programming languages, such as JUnit for Java, and are integrated into most development environments. Using the structures of the language and functionality offered by the unit testing framework, developers construct *test suites*—collections of test cases—by writing test case code in special test classes within the source code. When the code of the UUT changes, developers can re-execute the test suite to make sure the code still works as expected. One can even write test cases before writing the unit code. Before the unit code is complete, the test cases will fail. Once the code is written, passing test cases can be seen as a sign of successful unit completion.

In our BMI example, the UUT is the `BMICalc` class outlined in the previous section. This example is written in Python. There are multiple unit testing frameworks for Python, with `pytest` being one of the most popular. We will focus on `pytest`-formatted test cases for both our manually-written examples and our automated generation example. Example test cases for the BMI example can be found at `https://github.com/Greg4cr/PythonUnitTestGeneration/blob/main/src/example/test_bmi_calculator_manual.py`, and will be explained below.

Unit tests are typically the majority of tests written for a project. For example, Google recommends that approximately 70% of test cases for Android projects be unit tests [62]. The exact percentage may vary, but this is a reasonable starting point for establishing your expectations. This split is partially, of course, due to the fact that there are more units than subsystem or system-level interfaces in a system and almost

all classes of any importance will be targeted for unit testing. In addition, unit tests carry the following advantages:

- **Useful Early in Development:** Unit testing can take place before development of a "full" version of a system is complete. A single class can typically be executed on its own, although a developer may need to *mock* (fake the results of) its dependencies.
- **Simplicity:** The functionality of a single unit is typically more limited than a subsystem or the system as a whole. Unit tests often require less setup and the results require less interpretation than other levels of testing. Unit tests also often require little maintenance as the system as a whole evolves, as they are focused on small portions of the system.
- **Execute Quickly:** Unit tests typically require few method calls and limited communication between elements of a system. They often can be executed on the developer's computer, even if the system as a whole runs on a specialised device (e.g., in mobile development, system-level tests must run on an emulator or mobile device, while unit tests can be executed directly on the local computer). As a result, unit tests can be executed quickly, and can be re-executed on a regular basis as the code evolves.

When we design unit tests, we typically want to *test all "responsibilities" associated with the unit*. We examine the functionality that the unit is expected to offer, and ensure that it works as expected. If our unit is a single class, each "responsibility" is typically a method call or a short series of method calls. Each broad outcome of performing that responsibility should be tested—e.g., alternative paths through the code that lead to different normal or exceptional outcomes. If a method sequence could be performed out-of-order, this should be attempted as well. We also want to *examine how the "state" of class variables can influence the outcome of method calls*. Classes often have a set of variables where information can be stored. The values of those variables can be considered as the current state of the class. That state can often influence the outcome of calling a method. Tests should place the class in various states and ensure that the proper method outcome is achieved. When designing an individual unit test, there are typically five elements that must be covered in that test case:

- **Initialization (Arrange):** This includes any steps that must be taken before the core body of the test case is executed. This typically includes initializing the UUT, setting its initial state, and performing any other actions needed to execute the tested functionality (e.g., logging into a system or setting up a database connection).
- **Test Input (Act):** The UUT must be forced to take actions through method calls or assignments to class variables. The test input consists of values provided to the parameters of those method calls or assignments.
- **Test Oracle (Assert):** A test oracle, also known as an expected output, is used to validate the output of the called methods and the class variables against a set of encoded expectations in order to issue a verdict—pass or fail—on the test case. In a unit test, the oracle is typically formulated as a series of *assertions* about method output and class attributes. An assertion is a Boolean predicate that

acts as a check for correct behavior of the unit. The evaluation of the predicate determines the verdict (outcome) of the test case.

- **Tear Down (Cleanup):** Any steps that must be taken after executing the core body of the test case in order to prepare for the next test. This might include cleaning up temporary files, rolling back changes to a database, or logging out of a system.
- **Test Steps (Test Sequence, Procedure):** Code written to apply input to the methods, collect output, and compare the output to the expectations embedded in the oracle.

Unit tests are generally written as methods in dedicated classes grouping the unit tests for a particular UUT. The unit test classes are often grouped in a separate folder structure, mirroring the source code folder structure. For instance, the utils.BMICalc class stored in the `src` folder may be tested by a utils.TestBMICalc test class stored in the `tests` folder. The test methods are then executed by invoking the appropriate unit testing framework through the IDE or the command line (e.g., as called by a continuous integration framework). Figure 3.4 shows four examples of test methods for the BMICalc class. Each test method checks a different scenario cover different aspects of good practices in unit test design, as will be detailed below. The test methods and scenarios are:

- `test_BMI_value_valid()`: verifies the correct calculation of the BMI value for valid and typical (*normal*)inputs
- `test_invalid_height()`: checks robustness for invalid values of height using exceptions.
- `test_bmi_adult()`: verifies the correct BMI classification for adults.
- `test_bmi_children_4y()`: checks the correct BMI classification for children up to 4 years old.

Due to the challenges in representing real numbers in binary computing systems, a good practice in unit test design is to allow for an error range when assessing the correct calculations of floating point arithmetic. We use the `approx` method from the pytest framework to automatically verify whether the returned value lies within the 0.1 range of our test oracle. For instance, our first test case would pass if the returned BMI value would be 18.22 or 18.25, however, it would fail for 18.3. Most unit testing frameworks provide a method to assert floating points within specific ranges. Testers should be careful when asserting results from floating point arithmetic because failures in those assertions can represent precision or range limitations in the programming language instead of faults in the source code, such as incorrect calculations. For instance, neglecting to check for float precision is a "test smell" that can lead to flaky test executions [63, 64].[7] If care is not taken some tests might fail when running them on a different computer or when the operating system has been updated.

In addition to asserting the valid behaviour of the UUT (also referred informally to as "happy paths"), unit tests should check the robustness of the implementation. For example, testers should examine how the class handles exceptional behaviour.

[7]Tests are considered flaky if their verdict (pass or fail) changes when no code changes are made. In other words, the tests seems to show random behaviour.

```python
def test_bmi_value_valid():
    bmi_calc = bmi_calculator.BMICalc(150, 41, 18)    # Arrange
    bmi_value = bmi_calc.bmi_value()                  # Act
    # Here, the approx method allows 0.01
    #  differences in floating point errors.
    assert pytest.approx(bmi_value, abs=0.1) == 18.2 # Assert.

# Cases expected to throw exception
def test_invalid_height():
    # 'with' blocks expect exceptions to be thrown, hence
    #   the assertion is checked *after* the constructor call
    with pytest.raises(ValueError) as context:            # Assert
        bmi_calc = bmi_calculator.BMICalc(-150, 41, 18)  # Act

    with pytest.raises(ValueError) as context:            # Assert
        bmi_calc.height = 0                               # Act

def test_bmi_adult():
    bmi_calc = bmi_calculator.BMICalc(160, 65, 21)  # Arrange
    bmi_class = bmi_calc.classify_bmi_adults()       # Act
    assert bmi_class == "Overweight"                # Assert

def test_bmi_children_4y():
    bmi_calc = bmi_calculator.BMICalc(100, 13, 4)
    bmi_class = bmi_calc.classify_bmi_teens_and_children()
    assert bmi_class == "Underweight"
```

Figure 3.4: Examples of test methods for the BMICalc class using the pytest framework.

There are different ways to design unit tests to handle exceptional behaviour, each with its trade-offs. One example is to use exception handling blocks and include *failing assertions* (e.g., assert false) in points past the code that triggers an exception. However, those methods are not effective in checking whether specific types of exceptions have been thrown, such as distinguishing between input/output exceptions for "file not found" or database connection errors versus exceptions thrown due to division by zero or accessing null variables. Those different types of exceptions represent distinct types of error handling situations that testers may choose to cover in their test suites. Therefore, many unit test frameworks have methods to assert whether the UUT raises specific types of exception. Here we use the pytest.raises(...) context manager to capture the exceptions thrown when trying to specify invalid values for height and check whether they are the exceptions that we expected, or whether there are unexpected exceptions. Additionally, testers can include assertions to verify whether the exception includes an expected message.

One of the challenges in writing good unit tests is deciding on the maximum size and scope of a single test case. For instance, in our BMICalc class,the (classifyBMI_teensAndChildren()) method has numerous branches to handle the various BMI thresholds for different age ranges. Creating a single test method that exercises all branches for all age ranges would lead to a very long test method with dozens of assertions. This test case would be hard to read and understand. Moreover, such a test case would hinder debugging efforts because the tester would need to narrow down which specific assertion detected a fault. Therefore, in order to keep our test methods small, we recommend breaking down test coverage of the method (classifyBMI_teensAndChildren()) into a series of small test cases—with each test covering a different age range. In turn, for improved coverage, each of those test cases

should assert all BMI classifications for the corresponding age bracket.

Testers should avoid creating redundant test cases in order to improve the cost-effectiveness of the unit testing process. Redundant tests exercise the same behaviour, and do not bring any value (e.g., increased coverage) to the test suite. For instance, checking invalid height values in the test_bmi_adult () test case would introduce redundancy because those cases are already covered by the test_invalid_height () test case. On the other hand, the (test_bmi_adult ()) test case currently does not attempt to invoke BMI for ages below 19. Therefore, we can improve our unit tests by adding this invocation to the existing test case, or—even better—creating a new method with that invocation (e.g., test_bmi_adult_invalid ()).

3.3.1 Supporting Unit Testing with AI

Conducting rigorous unit testing can be an expensive, effort-intensive task. The effort required to create a single unit test may be negligible over the full life of a project, but this effort adds up as the number of classes increases. If one wants to test thoroughly, they may end up creating hundreds to thousands of tests for a large-scale project. Selecting effective test input and creating detailed assertions for each of those test cases is not a trivial task either. The problem is not simply one of scale. Even if developers and testers have a lot of knowledge and good intentions, they might forget or not have the time needed to think of all important cases. They may also cover some cases more than others, e.g., they might focus on valid inputs, but miss important invalid or boundary cases. The effort spent by developers does not end with test creation. Maintaining test cases as the SUT evolves and deciding how to allocate test execution resources effectively—deciding *which* tests to execute—also require care, attention, and time from human testers.

Ultimately, developers often make compromises if they want to release their product on time and under a reasonable budget. This can be problematic, as insufficient testing can lead to critical failures in the field after the product is released. Automation has a critical role in controlling this cost, and ensuring that both sufficient quality and quantity of testing is achieved. AI techniques—including optimization, machine learning, natural language processing, and other approaches—can be used to partially automate and support aspects of unit test creation, maintenance, and execution. For example,

- Optimization and reinforcement learning can *select test input* suited to meeting measurable testing goals. This can be used to create either new test cases or to amplify the effectiveness of human-created test cases.
- The use of supervised and semi-supervised machine learning approaches has been investigated in order to *infer test oracles* from labeled executions of a system for use in judging the correctness of new system executions.
- Three different families of techniques, powered by optimization, supervised learning, and clustering techniques, are used to make effective use of computing resources when executing test cases:
  - *Test suite minimization* techniques suggest redundant test suites that could be removed or ignored during test execution.
  - *Test case prioritization* techniques order test cases such that the potential for early fault detection or code coverage is maximised.

– *Test case selection* techniques identify the subset of test cases that relate in some way to recent changes to the code, ignoring test cases with little connection to the changes being tested.

If aspects of unit testing—such as test creation or selection of a subset for execution—can be even partially automated, the benefit to developers could be immense. AI has been used to support these, and other, aspects of unit testing. In the remainder of this chapter, we will focus on **test input generation**. In Section 3.5, we will also provide pointers to other areas of unit testing that can be partially automated using AI.

Exhaustively applying all possible input is infeasible due to an enormous number of possibilities for most real-world programs and units we need to test. Therefore, deciding *which* inputs to try becomes an important decision. Test generation techniques can create partial unit tests covering the initialization, input, and tear down stages. The developer can then supply a test oracle or simply execute the generated tests and capture any crashes that occur or exceptions that are thrown. One of the more effective methods of automatically selecting effective test input is *search-based test generation*. We will explain this approach in the following sections.

A word of caution, before we continue—it is our firm stance that AI *cannot* replace human testers. The points above showcase a set of good practices for unit test design. Some of these practices may be more easily achieved by either a human or an intelligent algorithm. For instance, properties such as readability mainly depends on human comprehension. Choosing readable names or defining the ideal size and scope for test cases may be infeasible or difficult to achieve via automation. On the other hand, choosing inputs (values or method calls) that mitigate redundancy can be easily achieved through automation through instrumentation, e.g., the use of code coverage tools.

AI can make unit testing more cost-effective and productive when used to support human efforts. However, there are trade-offs involved when deciding how much to rely on AI versus the potential effort savings involved. AI cannot replace human effort and creativity. However, it can reduce human effort on repetitive tasks, and can focus human testers towards elements of unit testing where their creativity can have the most impact. And over time, as AI-based methods become better and stronger, there is likely to be more areas of unit testing they can support or automate.

3.4 Search-Based Test Generation

Test input selection can naturally be seen as a search problem. When you create test cases, you often have one or more goals. Perhaps that goal is to find violations of a specification, to assess performance, to look for security vulnerabilities, to detect excessive battery usage, to achieve code coverage, or any number of other things that we may have in mind when we design test cases. We cannot try all input—any real-world piece of software with value has a near-infinite number of possible inputs we could try. However, somewhere in that space of possibilities lies a subset of inputs that best meets the goals we have in mind. Out of all of the test cases that could be generated for a UUT, we want to identify—systematically and at a reasonable cost—those that best meet those goals. Search-based test generation is an intuitive AI technique for locating those test cases that maps to the same process we might use ourselves to find a solution to a problem.

Let us consider a situation where you are asked a question. If you do not know the answer, you might make a guess—either be an educated guess or one made completely at random. In either case, you would then get some feedback. *How close were you to reaching the "correct" answer?* If your answer was not correct, you could then make a second guess. Your second guess, if nothing else, should be *closer* to being correct based on the knowledge gained from the feedback on that initial guess. If you are still not correct, you might then make a third, fourth, etc. guess—each time incorporating feedback on the previous guess.

Test input generation can be mapped to the same process. We start with a problem we want to solve. We have some goal that we want to achieve through the creation of unit tests. *If that goal can be measured*, then we can automate input generation. Fortunately, many testing goals can be measured.

- If we are interested in exploring the exceptions that the UUT can throw, then we want the inputs that *trigger the most exceptions*.
- If we are interested in covering all outcomes of a function, then we can divide the output into representative values and identify the inputs that *cover all representative output values*.
- If we are interested in executing all lines of code, then we are searching for the inputs that *cover more of the code structure*.
- If we are interested in executing a wide variety of input, then we want to find a set of inputs with *the highest diversity in their values*.

Attainment of many goals can be measured, whether as a percentage of a known checklist or just a count that we want to maximize. Even if we have a higher-level goal in mind that cannot be directly measured, there may be measurable sub-goals that correlate with that higher-level goal. For example, "find faults" cannot be measured—we do not know what faults are in our goal—but maximizing code coverage or covering diverse outputs may increase the likelihood of detecting a fault.

Once we have a measurable **goal**, we can automate the guess-and-check process outlined above via a metaheuristic optimization algorithm. **Metaheuristics** are strategies to sample and evaluate values during our search. Given a measurable goal, a metaheuristic optimization algorithm can systematically sample the space of possible test input, guided by feedback from one or more **fitness functions**—numeric scoring functions that judge the optimality of the chosen input based on its attainment of our goals. The exact process taken to sample test inputs from that space varies from one metaheuristic to another. However, the core process can be generically described as:

a. Generate one or more initial **solutions** (test suites containing one or more unit tests).

b. While time remains:

   (a) Evaluate each solution using the **fitness functions**.

   (b) Use feedback from the fitness functions and the sampling strategy employed by the **metaheuristic** to improve the solutions.

c. Return the best solution seen during this process.

In other words, we have an optimization problem. We make a guess, get feedback, and then use that additional knowledge to make a *smarter* guess. We keep going until we run out of time, then we work with the best solution we found during that process.

```
[    ]
  [    ]
1    [-1, [246, 680, 2]],
2    [2, [181],
3    [4, []],
4    [1, [466]],
5    [5, []],
6    [4, []],
7    [1, [261],
8    [5, []].
]
]
```

```
import pytest
import bmi_calculator

def test_0():
1cut = bmi_calculator.BMICalc(246,680,2)
2 cut.age = 18
3 cut.classify_bmi_teens_and_children()
4 cut.weight = 466
5 cut.classify_bmi_adults()
6 cut.classify_bmi_teens_and_children()
7 cut.weight = 26
8 cut.classify_bmi_adults()
```

Figure 3.5: The genotype (internal, up) and phenotype (external, down) representations of a solution containing a single test case. Each identifier in the genotype is mapped to a function with a corresponding list of parameters. For instance, `1` maps to setting the `weight`, and `5` maps to calling the method `classify_bmi_adults()`

.

The choice of both metaheuristic and fitness functions is crucial to successfully deploying search-based test generation. Given the existence of a near-infinite space of possible input choices, the order that solutions are tried from that space is the key to efficiently finding a solution. The metaheuristic—guided by feedback from the fitness functions—overcomes the shortcomings of a purely random input selection process by using a deliberate strategy to sample from the input space, gravitating towards "good" input and discarding input sharing properties with previously-seen "bad" solutions. By determining how solutions are evolved and selected over time, the choice of metaheuristic impacts the quality and efficiency of the search process. Metaheuristics are often inspired by natural phenomena, such as swarm behavior or evolution within an ecosystem.

In search-based test generation, the fitness functions represent our goals and guide the search. They are responsible for evaluating the quality of a solution and offering feedback on how to improve the proposed solutions. Through this guidance, the fitness

functions shape the resulting solutions and have a major impact on the quality of those solutions. Functions must be efficient to execute, as they will be calculated thousands of times over a search. Yet, they also must provide enough detail to differentiate candidate solutions and guide the selection of optimal candidates.

Search-based test generation is a powerful approach because it is *scalable* and *flexible*. Metaheuristic search—by strategically sampling from the input space—can scale to larger problems than many other generation algorithms. Even if the "best" solution can not be found within the time limit, search-based approaches typically can return a "good enough" solution. Many goals can be mapped to fitness functions, and search-based approaches have been applied to a wide variety of testing goals and scenarios. Search-based generation often can even achieve higher goal attainment than developer-created tests.

In the following sections, we will explain the highlighted concepts in more detail and explore how they can be applied to generate partial unit tests for Python programs. In Section 3.4.1, we will explain how to represent solutions. Then, in Section 3.4.2, we will explore how to represent two common goals as fitness functions. In Section 3.4.3, we will explain how to use the solution representation and fitness functions as part of two common metaheuristic algorithms. Finally, in Section 3.4.4, we will illustrate the application of this process on our BMI example.

3.4.1 Solution Representation

When solving any problem, we first must define the form the solution to the problem must take. What, exactly, does a solution to a problem "look" like? What are its contents? How can it be manipulated? Answering these questions is crucial before we can define how to identify the "best" solution.

In this case, we are interested in identifying a set of unit tests that maximise attainment of a testing goal. This means that a solution is a **test suite**—a collection of test cases. We can start from this decision, and break it down into the composite elements relevant to our problem.

- A solution is a **test suite**.
- A test suite contains one or more **test cases**, expressed as individual methods of a single test class.
- The solution interacts with a **unit-under-test (UUT)** which is a single, identified Python class with a constructor (optional) and one or more methods.
- Each test case contains an **initialization** of the UUT which is a call to its constructor, if it has one.
- Each test case then contains one or more **actions**, i.e., calls to one of the methods of the UUT or assignments to a class variable.
- The initialization and each action have zero or more **parameters** (input) supplied to that action.

This means that we can think of a test suite as a collection of test cases, and each test case as a single initialization and a collection of actions, with associated parameters. When we generate a solution, we choose a number of test cases to create. For each of those test cases, we choose a number of actions to generate. Different

solutions can differ in size—they can have differing numbers of test cases—and each test case can differ in size—each can contain a differing number of actions.

In search-based test generation, we represent two solutions in two different forms:

- **Phenotype (External) Representation:** The phenotype is the version of the solution that will be presented to an external audience. This is typically in a human-readable form, or a form needed for further processing.

- **Genotype (Internal) Representation:** The genotype is a representation used internally, within the metaheuristic algorithm. This version includes the properties of the solution that are relevant to the search algorithm, e.g., the elements that can be manipulated directly. It is generally a minimal representation that can be easily manipulated by a program.

Figure 3.5 illustrates the two representations of a solution that we have employed for unit test generation in Python. The phenotype representation takes the form of an executable pytest test class. In turn, each test case is a method containing an initialization, followed by a series of method calls or assignments to class variables. This solution contains a single test case, test_0 (). It begins with a call to the constructor of the UUT, BMICalc, supplying a height of 246, a weight of 680, and an age of 2. It then applies a series of actions on the UUT: setting the age to 18, getting a BMI classification from classify_bmi_teens_and_children (), setting the weight to 466, getting further classifications from each method, setting the weight to 26, then getting one last classification from classify_bmi_adults ().

This is our desired external representation because it can be executed at will by a human tester, and it is in a format that a tester can read. However, this representation is not ideal for use by the metaheuristic search algorithm as it cannot be easily manipulated. If we wanted to change one method call to another, we would have to identify which methods were being called. If we wanted to change the value assigned to a variable, we would have to identify (a) which variable was being assigned a value, (b) identify the portion of the line that represents the value, and (c), change that value to another. Internally, we require a representation that can be manipulated quickly and easily.

This is where the genotype representation is required. In this representation, a test suite is a list of test cases. If we want to add a test case, we can simply append it to the list. If we want to access or delete an existing test case, we can simply select an index from the list. Each test case is a list of actions. Similarly, we can simply refer to the index of an action of interest.

Within this representation, each action is a list containing (a) an action identifier, and (b), a list of parameters to that action (or an empty list if there are no parameters). The action identifier is linked to a separate list of actions that the tester supplies, that stores the method or variable name and type of action, i.e., assignment or method call (we will discuss this further in Section 2.4). An identifier of -1 is reserved for the constructor.

The solution illustrated in Figure 3.5 is not a particularly effective one. It consists of a single test case that applies seemingly random values to the class variables (the initial constructor creates what may be the world's largest two-year old). This solution only covers a small set of BMI classifications, and only a tiny portion of the branching behavior of the UUT. However, one could imagine this as a starting solution that could be manipulated over time into a set of highly effective test cases. By making

```
1   def calculate_fitness(metadata, fitness_function,
2           num_tests_penalty,length_test_penalty, solution):
3    fitness = 0.0

4   # Get the statement coverage over the code
5            fitness += statement_fitness(metadata, solution)

6   # Add a penalty to control test suite size
7   fitness -= float(len(solution.test_suite) / num_tests_penalty)

8   # Add a penalty to control the length of individual test cases
9   # Get the average test suite length)
10  total_length = 0
11  total_length = sum([len(test) for test in solution.test_suite])
12                      / len(solution.test_suite)
13  fitness -= float(total_length / length_test_penalty)

14  solution.fitness = fitness
```

Figure 3.6: The high-level calculation of the fitness function.

adjustments to the genotype representation, guided by the score from a fitness function, we can introduce those improvements.

3.4.2 Fitness Function

As previously-mentioned, fitness functions are the cornerstone of search-based test generation. The core concept is simple and flexible—a fitness function is simply a function that takes in a solution candidate and returns a "score" describing the quality of that solution. This gives us the means to differentiate one solution from another, and more importantly, to tell if one solution is *better* than another.

Fitness functions are meant to embody the goals of the tester. They tell us how close a test suite came to meeting those goals. The fitness functions employed determine what properties the final solution produced by the algorithm will have, and shape the evolution of those solutions by providing a target for optimization.

Essentially any function can serve as a fitness function, as long as it returns a numeric score. It is common to use a function that emits either a percentage (e.g., percentage of a checklist completed) or a raw number as a score, then either maximise or minimise that score.

- A fitness function should *not* return a Boolean value. This offers almost no feedback to improve the solution, and the desired outcome may not be located.
- A fitness function should yield (largely) continuous scores. A small change in a solution should not cause a large change (either positive or negative) in the resulting score. Continuity in the scoring offers clearer feedback to the metaheuristic algorithm.
- The best fitness functions offer not just an indication of quality, but a *distance* to the optimal quality. For example, rather than measuring completion of a checklist of items, we might offer some indication of how close a solution came to completing the remaining items on that checklist. In this chapter, we use a simple fitness function to clearly illustrate search-based test generation, but in Section 3.5, we will introduce a distance-based version of that fitness function.

Depending on the algorithm employed, either a single fitness function or multiple fitness functions can be optimised at once. We focus on single-function optimization in this chapter, but in Section 3.5, we will also briefly explain how multi-objective optimization is achieved.

To introduce the concept of a fitness function, we utilise a fitness function based on the **code coverage** attained by the test suite. When testing, developers must judge: (a) whether the produced tests are effective and (b) when they can stop writing additional tests. Coverage criteria provides developers with guidance on both of those elements. As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, these criteria are intended to serve as an approximation of efficacy. If the goals of the chosen criterion are met, then we have put in a measurable testing effort and can decide whether we have tested enough.

There are many coverage criteria, with varying levels of tool support. The most common criteria measure coverage of structural elements of the software, such as individual statements, branches of the software's control flow, and complex Boolean conditional statements. One of the most common, and most intuitive, coverage criteria is **statement coverage**. It simply measure the percentage of executable lines of code that have been triggered at least once by a test suite. The more of the code we have triggered, the more thorough our testing efforts are—and, ideally, the likely we will be to discover a fault. The use of statement coverage as a fitness function encourages the metaheuristic to explore the structure of the source code, reaching deeply into branching elements of that code.

As we are already generating `pytest`-compatible test suites, measuring statement coverage is simple. The pytest plugin pytest–cov measures statement coverage, as well as branch coverage—a measurement of how many branching control points in the UUT (e.g., `if`-statement and loop outcomes) have been executed—as part of executing a `pytest` test class. By making use of this plug-in, statement coverage of a solution can be measured as follows:

[a] Write the phenotype representation of the test suite to a file.

[b] Execute pytest, with the `--cov=<python file to measure coverage over>` command.

[c] Parse the output of this execution, extracting the percentage of coverage attained.

[d] Return that value as the fitness.

This measurement yields a value between 0–100, indicating the percentage of statements executed by the solution. We seek to *maximise* the statement coverage. Therefore, we employ the following formulation to obtain the fitness value of a test suite (shown as code in Figure 3.6):

$$fitness(solution) = statement\_coverage(solution) - bloat\_penalty(solution) \tag{3.2}$$

The *bloat penalty* is a small penalty to the score intended to control the size of the produced solution in two dimensions: the number of test methods, and the number of actions in each test. A massive test suite may attain high code coverage or yield many different outcomes, but it is likely to contain many redundant elements as

well. In addition, it will be more difficult to understand when read by a human. In particular, long sequences of actions may hinder efforts to debug the code and identify a fault. Therefore, we use the bloat penalty to encourage the metaheuristic algorithm to produce *small-but-effective* test suites. The bloat penalty is calculated as follows:

$$\begin{aligned} bloat\_penalty(solution) = &(num\_test\_cases/num\_tests\_penalty) \\ &+ (average\_test\_length/length\_test\_penalty) \end{aligned} \tag{3.3}$$

Where $num\_tests\_penalty$ is 10 and $length\_test\_penalty$ is 30. That is, we divide the number of test cases by 10 and the average length of a single test case (number of actions) by 30. These weights could be adjusted, depending on the severity of the penalty that the tester wishes to apply. It is important to not penalise too heavily, as that will increase the difficulty of the core optimization task—some expansion in the number of tests or length of a test is needed to cover the branching structure of the code. These penalty values allow some exploration while still encouraging the metaheuristic to locate smaller solutions.

```
{
    "file": "bmi_calculator",
    "location": "example/",
    "class": "BMICalc",
    "constructor": {
        "parameters": [
            { "type": "integer", "min": -1 },
            { "type": "integer", "min": -1 },
            { "type": "integer", "min": -1, "max": 150 }
        ] },
    "actions": [
        { "name": "height", "type": "assign", "parameters": [
            { "type": "integer", "min": -1 } ]
        },
        { "name": "weight", "type": "assign", "parameters": [
            { "type": "integer", "min": -1 } ]
        },
        { "name": "age", "type": "assign", "parameters": [
            { "type": "integer", "min": -1, "max": 150 } ]
        },
        { "name": "bmi_value", "type": "method" },
        { "name": "classify_bmi_teens_and_children", "type": "method" },
        { "name": "classify_bmi_adults", "type": "method" }
    ]
}
```

Figure 3.7: Metadata definition for class BMICalc.

3.4.3 Metaheuristic Algorithms

Given a solution representation and a fitness function to measure the quality of solutions, the next step is to design an algorithm capable of producing the best possible solution within the available resources. Any UUT with reasonable complexity has a near-infinite number of possible test inputs that could be applied. We cannot reasonable try them all. Therefore, the role of the metaheuristic is to intelligently sample from that space of possible inputs in order to locate the best solution possible within a strict time limit.

There are many metaheuristic algorithms, each making use of different mechanisms to sample from that space. In this chapter, we present two algorithms:

- **Hill Climber:** A simple algorithm that produces a random initial solution, then attempts to find better solutions by making small changes to that solution—restarting if no better solution can be found.

- **Genetic Algorithm:** A more complex algorithm that models how populations of solutions evolve over time through the introduction of mutations and through the breeding of good solutions.

The Hill Climber is simple, fast, and easy to understand. However, its effectiveness depends strongly on the quality of the initial guess made. We introduce it first to explain core concepts that are built upon by the Genetic Algorithm, which is slower but potentially more robust.

3.4.3.1 Common Elements

Before introducing either algorithm in detail, we will begin by discussing three elements shared by both algorithms—a metadata file that defines the actions available for the UUT, random test generation, and the search budget.

UUT Metadata File: To generate unit tests, the metaheuristic needs to know *how* to interact with the UUT. In particular, it needs to know what methods and class variables are available to interact with, and what the parameters of the methods and constructor are. To provide this information, we define a simple JSON-formatted metadata file. The metadata file for the BMI example is shown in Figure 3.7, and we define the fields of the file as follows:

- **file:** The python file containing the UUT.
- **location:** The path of the file.
- **class:** The name of the UUT.
- **constructor:** Contains information on the parameters of the constructor.
- **actions:** Contains information about each action.
  - **name:** The name of the action (method or variable name).
  - **type:** The type of action (method or assign).
  - **parameters:** Information about each parameter of the action.
    - **type:** Datatype of the parameter. For this example, we only support integer input. However, the example code could be expanded to handle additional datatypes.
    - **min:** An optional minimum value for the parameter. Used to constrain inputs to a defined range.
    - **max:** An optional maximum value for the parameter. Used to constrain inputs to a defined range.

This file not only tells the metaheuristic what actions are available for the UUT, it suggests a starting point for "how" to test the UUT by allowing the user to optionally constrain the range of values. This allows more effective test generation by limiting the range of guesses that can be made to "useful" values. For example, the age of a person cannot be a negative value in the real world, and it is unrealistic that a person would be more than 150 years old. Therefore, we can impose a range of age values that we might try. To test error handling for negative ranges, we might set the minimum value to -1. This allows the metaheuristic to try a negative value, while preventing it from wasting time trying *many* negative values.

In this example, we assume that a tester would create this metadata file—a task that would take only a few minutes for a UUT. However, it would be possible to write code to extract this information as well.

Random Test Generation: Both of the presented metaheuristic algorithms start by making random "guesses"—either generating random test cases or generating entire test suites at random—and will occasionally modify solutions through random generation of additional elements. To control the size of the generated test suites or test cases, there are two user-controllable parameters:

- **Maximum number of test cases:** The largest test suite that can be randomly generated. When a suite is generated, a size is chosen between 1 - max_test_cases , and that number of test cases are generated and added to the suite.
- **Maximum number of actions:** The largest individual test case that can be randomly generated. When a test case is generated, a number of actions between 1 - max_actions is chosen and that many actions are added to the test case (following a constructor call).

By default, we use 20 as the value for both parameters. This provides a reasonable starting point for covering a range of interesting behaviors, while preventing test suites from growing large enough to hinder debugging. Test suites can then grow or shrink over time through manipulation by the metaheuristic.

Search Budget: This search budget is the time allocated to the metaheuristic. The goal of the metaheuristic is to find the best solution possible within this limitation. This parameter is also user-controlled:

- **Search Budget:** The maximum number of generations of work that can be completed before returning the best solution found.

The search budget is expressed as a number of *generations*—cycles of exploration of the search space of test inputs—that are allocated to the algorithm. This can be set according to the schedule of the tester. By default, we allow 200 generations in this example. However, fewer may still produce acceptable results, while more can be allocated if the tester is not happy with what is returned in that time frame.

```
# Generate an initial random solution, and calculate its fitness
solution_current = Solution()
solution_current.test_suite=generate_test_suite(metadata,
                                    max_test_cases,max_actions)
calculate_fitness(metadata, fitness_function, num_tests_penalty,
                  length_test_penalty, solution_current)

# The initial solution is the best we have seen to date
solution_best = copy.deepcopy(solution_current)

# Continue to evolve until the generation budget is exhausted
# or the number of restarts is exhausted.
gen = 1
restarts = 0

while gen <= max_gen and restarts <= max_restarts:
    tries = 1
    changed = False

    # Try random mutations until we see a better solutions,
    # or until we exhaust the number of tries.
    while tries < max_tries and not changed:
        solution_new = mutate(solution_current)
        calculate_fitness(metadata, fitness_function, num_tests_penalty,
                          length_test_penalty, solution_new)

    # If the solution is an improvement, make it the new solution.
      if solution_new.fitness > solution_current.fitness:
         solution_current = copy.deepcopy(solution_new)
            changed = True

     # If it is the best solution seen so far, then store it.
       if solution_new.fitness > solution_best.fitness:
              solution_best = copy.deepcopy(solution_current)

        tries += 1

    # Reset the search if no better mutant is found within a set number
    # of attempts by generating a new solution at random.
    if not changed:
        restarts += 1
        solution_current = Solution()
solution_current.test_suite = generate_test_suite(metadata,
                                        max_test_cases,max_actions)
        calculate_fitness(metadata, fitness_function, num_tests_penalty,
                          length_test_penalty, solution_current)

    # Increment generation
    gen += 1
# Return the best suite seen
```

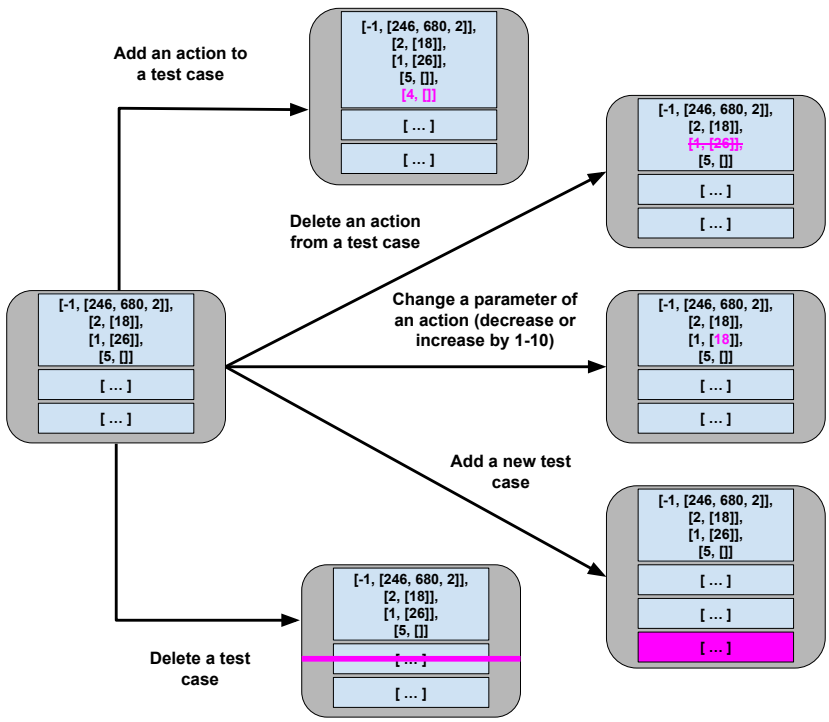
Figure 3.8: The core body of the Hill Climber algorithm.

3.4.3.2 Hill Climber

A Hill Climber is a classic metaheuristic that embodies the "guess-and-check" process we discussed earlier. The algorithm makes an initial guess purely at random, then attempts to improve that guess by making small, iterative changes to it. When it lands on a guess that is better than the last one, it adopts it as the current solution and proceeds to make small changes to that solution. The core body of this algorithm is shown in Figure 3.8. The full code can be found at `https://github.com/Greg4cr/PythonUnitTestGeneration/blob/main/src/hill_climber.py`.

The variable `solution_current` stores the current solution. At first, it is initialised to a random test suite, and we measure the fitness of the solution (lines 2-5). Following this, we start our first generation of evolution. While we have remaining search budget, we then attempt to improve the current solution.

Each generation, we attempt to improve the current solution through the process of *mutation*. During mutation, we introduce a small change to the current solution. Below, we outline the types of change possible during mutation:



After selecting and applying one of these transformations (line 22), we measure the fitness of the mutated solution (line 23). It if it better than the current solution, we make the mutation into the current solution (lines 26-28). If it is better than the best solution seen to date, we also save it as the new best solution (lines 30-31). We then proceed to the next generation.

If the mutation is not better than the current solution, we try a different mutation to see if it is better. The range of transformations results in a very large number of possible transformations. However, even with such a range, we may end up in situations where no improvement is possible, or where it would be prohibitively slow

to locate an improved solution. We refer to these situations at *local optima*—solutions that, while they may not be the best possible, are the best that can be located through incremental changes.

We can think of the landscape of possible solutions as a topographical map, where better fitness scores represent higher levels of elevation in the landscape. This algorithm is called a "Hill Climber" because it attempts to scale that landscape, finding the tallest peak that it can in its local neighborhood.

If we reach a local optima, we need to move to a new "neighborhood" in order to find taller peaks to ascend. In other words, when we become stuck, we restart by replacing the current solution with a new random solution (lines 37-42). Throughout this process, we track the best solution seen to date to return at the end. To control this process, we use two user-controllable parameters.

- **Maximum Number of Tries:** A limit on the number of mutations we are willing to try before restarting (max_tries , line 21). By default, this is set to 200.

- **Maximum Number of Restarts:** A limit of restarts we are willing to try before giving up on the search (max_restarts , line 15). Be default, this is set to 5.

The core process employed by the Hill Climber is simple, but effective. Hill Climbers also tend to be faster than many other metaheuristics. This makes them a popular starting point for search-based automation. Their primary weakness is their reliance on making a good initial guess. A bad initial guess could result in time wasted exploring a relatively "flat" neighborhood in that search landscape. Restarts are essential to overcoming that limitation.

```
#Create initial population.
population = create_population(population_size)

#Initialise best solution as the first member of that population.
solution_best = copy.deepcopy(population[0])

# Continue to evolve until the generation budget is exhausted.
# Stop if no improvement has been seen in some time (stagnation).
gen = 1
stagnation = -1

        while gen <= max_gen and stagnation <= exhaustion:
    # Form a new population.
            new_population = []

        while len(new_population) < len(population):
        # Choose a subset of the population and identify
        # the best solution in that subset (selection).
        offspring1 = selection(population, tournament_size)
        offspring2 = selection(population, tournament_size)

        # Create new children by breeding elements of the best solutions
        # (crossover).
        if random.random() < crossover_probability:
  (offspring1, offspring2) = uniform_crossover(offspring1, offspring2)

        # Introduce a small, random change to the population (mutation).
        if random.random() < mutation_probability:
            offspring1 = mutate(offspring1)
        if random.random() < mutation_probability:
            offspring2 = mutate(offspring2)

        # Add the new members to the population.
        new_population.append(offspring1)
        new_population.append(offspring2)

        # If either offspring is better than the best-seen solution,
        # make it the new best.
        if offspring1.fitness > solution_best.fitness:
            solution_best = copy.deepcopy(offspring1)
            stagnation = -1
        if offspring2.fitness > solution_best.fitness:
            solution_best = copy.deepcopy(offspring2)
            stagnation = -1

    # Set the new population as the current population.
    population = new_population

    # Increment the generation.
    gen += 1
    stagnation += 1
   # Return the best suite seen
```

Figure 3.9: The core body of the Genetic Algorithm.

3.4.3.3 Genetic Algorithm

Genetic Algorithms model the evolution of a population over time. In a population, certain individuals may be "fitter" than others, possessing traits that lead them to thrive—traits that we would like to see passed forward to the next generation through reproduction with other fit individuals. Over time, random mutations introduced into the population may also introduce advantages that are also passed forward to the next generation. Over time, through mutation and reproduction, the overall population will grow stronger and stronger.

As a metaheuristic, a Genetic Algorithm is build on a core generation-based loop like the Hill Climber. However, there are two primary differences:

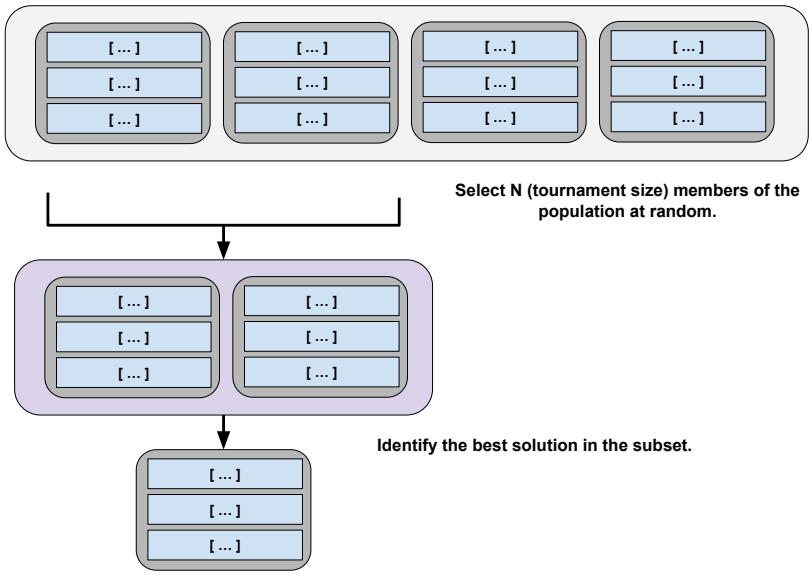
- Rather than evolving a single solution, we simultaneously manage a *population* of different solutions.

- In addition to using mutation to improve solutions, a Genetic Algorithm also makes use of a *selection* process to identify the best individuals in a population, and a *crossover* process that produces new solutions merging the test cases ("genes") of parent solutions ("chromsomes").

The core body of the Genetic Algorithm is listed in Figure 3.9. The full code can be found at `https://github.com/Greg4cr/PythonUnitTestGeneration/blob/main/src/genetic_algorithm.py`.

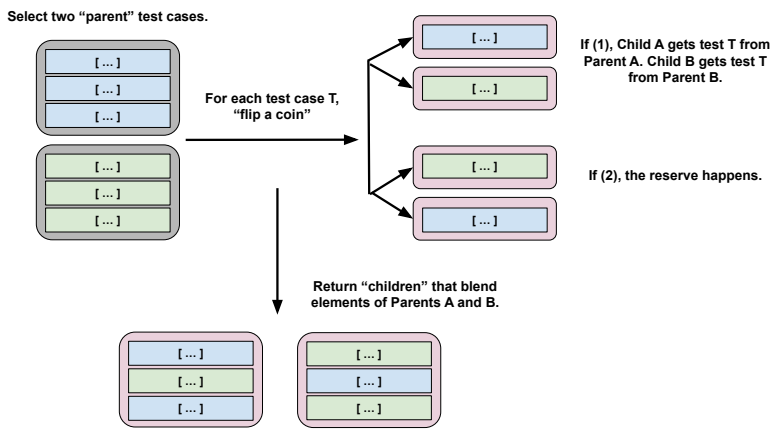
We start by creating an initial population, where each member of the population is a randomly-generated test suite (line 1). We initialise the best solution to the first member of that population (line 5). We then begin the first generation of evolution (line 12).

Each generation, we form a new population by applying a series of actions intended to promote the best "genes" forward. We form the new population by creating two new solutions at a time (line 16). First, we attempt to identify two of the best solutions in a population. If the population is large, this can be an expensive process. To reduce this cost, we perform a *selection* procedure on a randomly-chosen subset of the population (lines 19-20), explained below:



The fitness of the members of the chosen subset is compared in a process called "tournament", and a winner is selected. The winner may not be the best member of the full population, but will be at least somewhat effective, and will be identified at a lower cost than comparing all population members. These two solutions may be carried forward as-is. However, at certain probabilities, we may make further modifications to the chosen solutions.

The first of these is crossover—a transformation that models reproduction. We generate a random number and check whether it is less than a user-set crossover_probability (line 23). If so, we combine individual genes (test cases) of the two solutions using the following process:



If the parents do not contain the same number of test cases, then the remaining cases can be randomly distributed between the children. This form of crossover is known as "uniform crossover". There are other means of performing crossover. For example, in "single-point" crossover, a single index is chosen, and one child gets

all elements from Parent A before that index, and all elements from Parent B from after that index (with the other child getting the reverse). Another form, "discrete recombination", is similar to uniform crossover, except that we make the coin flip for each child instead of once for both children at each index.

We may introduce further mutations to zero, one, or both of the solutions. If a random number is less than a user-set mutation probability (lines 27, 29), we will introduce a single mutation to that solution. We do this independently for both solutions. The mutation process is the same as in the Hill Climber, where we can add, delete, or modify an individual action or add or delete a full test case.

Finally, we add both of the solutions to the new population (line 46). If either solution is better than the best seen to date, we save it to be returned at the end of the process (lines 38-43). Once the new population is complete, we continue to the next generation.

There may be a finite amount of improvement that we can see in a population before it becomes *stagnant*. If the population cannot be improved further, we may wish to terminate early and not waste computational effort. To enable this, we count the number of generations where no improvement has been seen (line 50), and terminate if it passes a user-set *exhaustion* threshold (line 12). If we identify a new "best" solution, we reset this counter (lines 40, 43).

The following parameters of the genetic algorithm can be adjusted:

- **Population Size:** The size of the population of solutions. By default, we set this to 20. This size must be a even number in the example implementation.
- **Tournament Size:** The size of the random population subset compared to identify the fittest population members. By default, this is set to 6.
- **Crossover Probability:** The probability that we apply crossover to generate child solutions. By default, 0.7.
- **Mutation Probability:** The probability that we apply mutation to manipulate a solution. By default, 0.7.
- **Exhaustion Threshold:** The number of generations of stagnation allowed before the search is terminated early. By default, we have set this to 30 generations.

These parameters can have a noticeable effect on the quality of the solutions located. Getting the best solutions quickly may require some experimentation. However, even at default values, this can be a highly effective method of generating test suites.

3.4.4 Examining the Resulting Test Suites

Now that we have all of the required components in place, we can generate test suites and examine the results. To illustrate what these results look like, we will examine test suites generated after executing the Genetic Algorithm for 1000 generations. During these executions, we disabled the exhaustion threshold to see what would happen if the algorithm was given the full search budget to work.

Figure 3.13 illustrates the results of executing the Genetic Algorithm. We can see the change in fitness over time, as well as the change in the number of test cases in the suite and the average number of actions in test cases. Note that fitness is penalised by the bloat penalty, so the actual statement coverage is higher than the final fitness

value. Also note that metaheuristic search algorithms are random. Therefore, each execution of the Hill Climber or Genetic Algorithm will yield different test suites in the end. Multiple executions may be desired in order to detect additional crashes or other issues.

The fitness starts around 63, but quickly climbs until around generation 100, when it hits approximately 86. There are further gains after that point, but progress is slow. At generation 717, it hits a fitness value of 92.79, where it remains until near the very end of the execution. At generation 995, a small improvement is found that leads to the coverage of additional code and a fitness increase to 93.67. Keep in mind, again, that a fitness of "100" is not possible due to the bloat penalty. It is possible that further gains in fitness could be attained with an even higher search budget, but covering the final statements in the code and further trimming the number or length of test cases both become quite difficult at this stage.

The test suite size starts at 13 tests, then sheds excess tests for a quick gain in fitness. However, after that, the number of tests rises slowly as coverage increases. For much of the search, the test suite remains around 20 test cases, then 21. At the end, the final suite has 22 test cases. In general, it seems that additional code coverage is attained by generating new tests and adding them to the suite.

At times, redundant test cases are removed, but instead, we often see redundancy removed through the deletion of actions within individual test cases. The initial test cases are often quite long, with many redundant function calls. Initially, tests have an average of 11 actions. Initially, the number of actions oscillates quite a bit between an average of 8-10 actions. However, over time, the redundant actions are trimmed from test cases. After generation 200, test cases have an average of only three actions until generation 995, when the new test case increases the average length to four actions. With additional time, it is likely that this would shrink back to three. We see that the tendency is to produce a large number of very small test cases. This is good, as short test cases are often easier to understand and make it easier to debug the code to find faults.

More complex fitness functions or algorithms may be able to cover more code, or cover the same code more quickly, but these results show the power of even simple algorithms to generate small, effective test cases. A subset of a final test suite is shown in Figure 3.10.[8]

[8]The full suite can be found at `https://github.com/Greg4cr/PythonUnitTestGeneration/blob/main/src/example/test_bmi_calculator_automated_statement.py`.

```
1 def test_0():
2     cut = bmi_calculator.BMICalc(120,860,13)
3         cut.classify_bmi_teens_and_children()

4 def test_2():
5         cut = bmi_calculator.BMICalc(43,243,59)
6         cut.classify_bmi_adults()
7         cut.height = 526
8         cut.classify_bmi_adults()
9         cut.classify_bmi_adults()

10 def test_5():
11         cut = bmi_calculator.BMICalc(374,343,17)
12         cut.age = 123
13         cut.classify_bmi_adults()
14         cut.age = 18
15         cut.classify_bmi_teens_and_children()
16         cut.weight = 396
17         cut.classify_bmi_teens_and_children()

18 def test_7():
19         cut = bmi_calculator.BMICalc(609,-1,94)

20 def test_11():
21         cut = bmi_calculator.BMICalc(491,712,20)
22         cut.classify_bmi_adults()

23 def test_17():
24         cut = bmi_calculator.BMICalc(608,717,6)
25         cut.classify_bmi_teens_and_children()
26         cut.age = 91
27         cut.classify_bmi_teens_and_children()
28         cut.classify_bmi_teens_and_children()
```

Figure 3.10: A subset of the test suite produced by the Genetic Algorithm, targeting statement coverage.

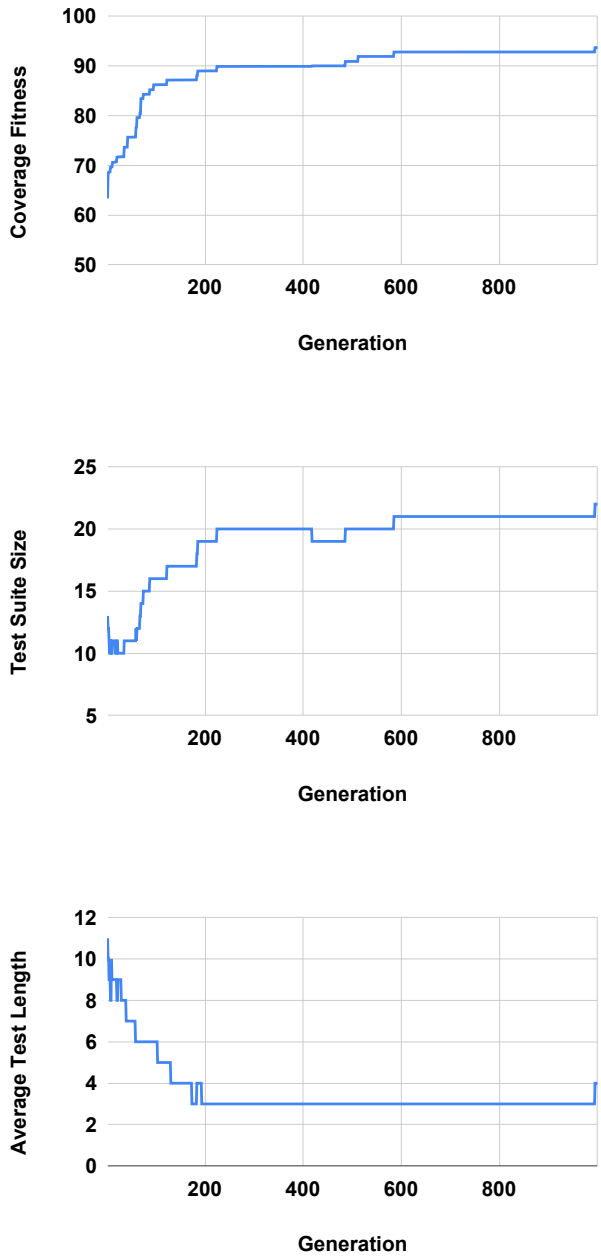


Figure 3.13: Change in fitness, test suite size, and average number of actions in a test case over 1000 generations. Note that fitness includes both coverage and bloat penalty, and can never reach 100.

Some test cases look like `test_0()` and `test_11()` in the example—a constructor call, followed by a BMI classification. Others will adjust the variable assignments, then make calls. For example, `test_5()` covers several paths in the code by making

assignments, then getting classifications, multiple times. test_7 () is an example of one where only a constructor call was needed, as the value supplied—a negative weight, in this case—was sufficient to trigger an exception.

There is still some room for improvement in these test cases. For example, test_2 () and test_17 () both contain redundant calls to a classification method. It is likely that a longer search budget would remove these calls. It would be simple to simply remove all cases where a method is called twice in a row with the same arguments from the suite. However, in other cases, those calls may have different results (e.g., if class state was modified by the calls), and you would want to leave them in place.

Search-based test generation requires a bit of experimentation. Even the Hill Climber has multiple user-selectable parameters. Finding the right search budget, for example, can require some experimentation. It may be worth executing the algorithm once with a very high search budget in order to get an initial idea of the growth in fitness. In this case, a tester could choose to stop much earlier than 1000 generations with little loss in effectiveness. For example, only limited gains are seen after 200 generations, and almost no gain in fitness is seen after 600 generations.

3.4.5 Assertions

It is important to note that this chapter is focused on *test input* generation. These test cases lack assertion statements, which are needed to check the correctness of the program behavior.

These test cases *can* be used as-is. Any exceptions thrown by the UUT, or other crashes detected, when the tests execute will be reported as failures. In some cases, exceptions *should* be thrown. In Figure 3.10, test_17 will trigger an exception when classify_bmi_teens_and_children () is called for a 91-year-old. This exception is the desired behavior. However, in many cases, exceptions are not desired, and these test cases can be used to alert the developer about crash-causing faults.

Otherwise, the generated tests will need assertions to be added. A tester can add assertions manually to these test cases, or a subset of them, to detect incorrect output. Otherwise, researchers and developers have begun to explore the use of AI techniques to generate assertions as well. We will offer pointers to some of this work in Section 3.5.

3.5 Advanced Concepts

The input generation technique introduced in the previous section can be used to generate small, effective test cases for Python classes. This section briefly introduces concepts that build on this foundation, and offers pointers for readers interested in developing more complex AI-enhanced unit test automation tools.

3.5.1 Distance-Based Coverage Fitness Function

This chapter introduced the idea that we can target the maximization of code coverage as a fitness function. We focused on statement coverage—a measurement of the number of lines of code executed. A similar measurement is the *branch coverage*—a measurement of the number of outcomes of control-altering expressions covered by test cases. This criterion is expressed over statements that determine which code will

be executed next in the sequence. For example, in Python, this includes `if`, `for`, and `while` statements. Full branch coverage requires True and False outcomes for the Boolean predicates expressed in each statement.

Branch coverage can be maximised in the same manner that we maximised statement coverage—by simply measuring the attained coverage and favoring higher totals. However, advanced search-based input generation techniques typically use a slightly more complex fitness function based on *how close* a test suite came to covering each of these desired outcomes.

Let's say that we had two test suites—one that attains 50% branch coverage and one that attains 75% coverage. We would favor the one with 75% coverage, of course. However, what if both had 75% coverage? Which is better? The answer is that we want the one that is *closer* to covering the remaining 25%. Perhaps, with only small changes, that one could attain 100% coverage. We cannot know which of those two is better with our simple measurement of coverage. Rather, to make that determination, we divide branch coverage into a set of *goals*, or combinations of an expression we want to reach and an outcome we desire for that expression. Then, for each goal, we measure the *branch distance* as a score ranging from $0 - 1$. The branch distance (dist) is defined as follows:

$$\text{dist}(goal, suite) = \begin{cases} 0 & \text{If the branch is reached and the desired outcome is attained.} \\ \text{dist}_{\text{min}}(goal, suite) & \text{If the branch is reached, but the desired outcome is not attained.} \\ 1 & \text{If the branch has not been reached.} \end{cases} \tag{3.4}$$

Our goal is to *minimise* the branch distance. If we have *reached* the branch of interest and *attained* the desired outcome, then the score is 0. If we have not reached the branch, then the value is 1. If we have reached the branch, but not covered it, then we measure *how close* we came by transforming the Boolean predicate into a numeric function. For example, if we had the expression `if x == 5:` and desired a True outcome, but x was assigned a value of 3 when we executed the expression, we would calculate the branch distance as $abs(x - 5) = abs(3 - 5) = 2$.[9]

We then normalise this value to be between 0 and 1. As this expression may be executed multiple times by the test suite, we take the minimum branch distance as the score. We can then attain a fitness score for the test suite by taking the sum of the branch distances for all goals: $fitness = \sum_{goal \in Goals} distance(goal, suite)$.

The branch distance offers a fine-grained score that is more informative than simply measuring the coverage. Using this measurement allows faster attainment of coverage, and may enable the generation tool to attain more coverage than would otherwise be possible. The trade-off is the increased complexity of the implementation. At minimum, the tool would have to insert logging statements into the program. To avoid introducing side-effects into the behavior of the class-under-test, measuring the branch distance may require complex instrumentation and execution monitoring.

3.5.2 Multiple and Many Objectives

When creating test cases, we typically have many different goals. A critical goal is to cover all important functionality, but we also want few and short test cases, we

[9]For more information on this calculation, and normalization, see the explanations from McMinn, Lukasczyk, and Arcuri: [2, 65, 66].

want tests to be understandable by humans, we want tests to have covered all parts of the system code, and so on. When you think about it, it is not uncommon to come up with five or more goals you have during test creation. If we plan to apply AI and optimisation to help us to create these test cases, we must encode these goals so that they are quantitative and can be automatically and quickly checked. We have the ability to do this through fitness functions. However, if we have multiple goals, we cannot settle for single-objective optimisation and instead have to consider the means to optimise all of these objectives at the same time.

A simple solution to the dilemma is to try to merge all goals together into a single fitness function which can then be optimised, often by adding all functions into a single score—potentially weighting each. For example, if our goals are high code coverage and few test cases, we could normalise the number of uncovered statements and the number of test cases to the same scale, sum them, and attempt to minimise this sum.

However, it is almost inevitable that many of your goals will compete with each other. In this two-objective example, we are punished for adding more test cases, but we are also punished if we do not cover all code. If these two goals were considered equally important, it seems possible that an outcome could be a single, large test case that tries to cover as much of the code as possible. While this might be optimal given the fitness function we formulated, it might not reflect what you really hope to receive from the generation tool. In general, it will be very hard to decide up-front how you want to trade off one objective versus the others. Even if you can in principle set weights for the different elements of the fitness function, when the objectives are fundamentally at odds with each other, there is no single weight assignment that can address all conflicts.

An alternative, and often better, solution is to keep each fitness function separate and attempt to optimise all of them at the same time, balancing optimisation of one with optimisation of each of the others. The outcome of such a multi-objective optimisation is not a single best solution, but a set of solutions that represent good trade-offs between the competing objectives. The set approximates what is known as the Pareto frontier, which is the set of all solutions that are not dominated by any other solution. A solution dominates another one if it is at least as good in all the objectives and better in at least one. This set of solutions represents balancing points, where the solution is the best it can be at some number of goals without losing attainment of the other goals. In our two-objective example of code coverage and test suite size, we might see a number of solutions with high coverage and a low number of test cases along this frontier, with some variation representing different trade-offs between these goals. We could choose the solution that best fits our priorities—perhaps taking a suite with 10 tests and 93% coverage over one with 12 tests and 94% coverage.

One well-known example of using multi-objective optimisation in software testing is the Sapienz test generation developed by Facebook to test Android applications through their graphical user interface [67]. It can generate test sequences of actions that maximise code coverage and the number of crashes, while minimizing the number of actions in the test cases. The system, thus, simultaneously optimises three different objectives. It uses a popular genetic algorithm known as NSGA-II for multi-objective optimisation and returns a set of non-dominated test cases.

When the number of objectives grows larger, some of the more commonly used optimisation algorithms—like NSGA-II—become less effective. Recently, "many-objective" optimisation algorithms that are more suited to such situations have been

proposed. One such algorithm was recently used to select and prioritise test cases for testing software product lines [68]. A total of nine different fitness functions are optimised by the system. In addition to the commonly used test case and test suite sizes, other objectives included are the pairwise coverage of features, dissimilarity of test cases, as well as the number of code changes the test cases cover.

3.5.3 Human-readable Tests

A challenge with automated test generation is that the generated test cases typically do not look similar to test cases that human developers and testers would write. Variable names are typically not informative and the ordering of test case steps might not be natural or logical for a human reading and interpreting them. This can create challenges for using the generated test cases. Much of existing research on test generation has not considered this a problem. A common argument has been that since we can generate so many test cases and then automatically run them there is little need for them to be readable; the humans will not have the time or interest to analyse the many generated test cases anyway. However, in some scenarios we really want to generate and then keep test cases around, for example when generating test cases to reach a higher level of code coverage. Also, when an automatically generated test case fails it is likely that a developer will want to investigate its steps to help identify what leads the system to fail. Automated generation of readable test cases would thus be helpful.

One early result focused on generating XML test inputs that were more comprehensible to human testers [69]. The developed system could take any XSD (XML Schema Definition) file as input and then create a model from which valid XML could then be generated. A combination of several AI techniques were then used to find XML inputs that were complex enough to exercise the system under test enough but not too complex since that would make the generated inputs hard for humans to understand. Three different metrics of complexity was used for each XML inputs (its number of elements, attributes, and text nodes) and the AI technique of Nested Monte-Carlo Search, an algorithm very similar to what was used in the AlphaGO Go playing AI [70], were then used to find good inputs for them. Results were encouraging but it was found that not all metrics were as easily optimised by the chosen technique. Also, for real comprehensibility it will not be enough to only find the right size of test inputs; the specific content and values in them will also be critical.

Other studies have found that readability can be increased by—for example—using real strings instead of random ones (e.g., by pulling string values from documentation), inserting default values for "unimportant" elements (rather than omitting them), and limiting the use and mixture of `null` values with normal values[10] [57, 71].

A more recent trend in automated software engineering is to use techniques from the AI area of natural language processing on source code. For example, GitHub in 2021 released its Co-Pilot system, which can auto-complete source code while a developer is writing it [72]. They used a neural network model previously used for automatically generating text that look like it could have been written by humans. Instead of training it on lots of human-written texts they instead trained it on human-written source code. The model can then be used to propose plausible completions of the source code currently being written by a developer in a code editor. In the future, it is likely that these ideas can and will also be used to generate test code. However, there are many risks with such approaches, and it is not a given that the generated test

[10] Although, of course, some `null` values should be applied to catch common "null pointer" faults.

code will be meaningful or useful in actual testing. For example, it has been shown that Co-Pilot can introduce security risks [73]. Still, by combining these AI techniques with multi-objective optimisation it seems likely that we can automatically generate test cases that are both useful and understandable by humans.

3.5.4 Finding Input Boundaries

One fundamental technique to choose test input is known as boundary value testing/ analysis. This technique aims to identify input values at the *boundary* between different visible program behaviours, as those boundaries often exhibit faults due to—for example—"off-by-one" errors or other minor mistakes. Typically, testers manually identify boundaries by using the software specification to define different *partitions*, i.e., sets of input that exhibit similar behaviours. Consider, for example, the creation of date objects. Testers can expect that valid days mainly lie within the range of 1–27. However, days greater or equal than 28 might reveal different outputs depending on the value chosen for month or year (e.g., February 29th). Therefore, most testers would choose input values between 28–32 as *one of* the boundaries for testing both valid and invalid dates (similarly, boundary values for day between 0–1).

The *program derivative* measures the program's sensitivity to behavioural changes for different sets of input values [74]. Analogous to the mathematical concept of a derivative, the program derivative conveys how function values (output) change when varying an independent variables (input). In other words, we can detect boundary values by detecting notable output differences when using a similar sets of inputs [75]. We quantify the similarities between input and output by applying various distance functions that quantify the similarity between a pair of values. Low distance values indicate that the pair of values are similar to each other. Some of the widely used distance functions are the Jaccard index (strings), Euclidean distance (numerical input) or even the more generic Normalised Compressed Distance (NCD).

The program derivative analyses the ratio between the distances of input and output of a program under test (Equation X). Let a and b be two different input values for program P with corresponding output values $P(a)$ and $P(b)$. We use the distance functions $d_i(a, b)$ and $d_o(P(a), P(b))$ to measure the distance between, respectively, the pair of input and their corresponding output values. The program derivative (PD) is defined as [75]:

$$PDQ_{do,di}(a, b) = \frac{d_o(P(a), P(b))}{d_i(a, b)}, b \neq a \tag{3.5}$$

Note that high derivative values indicate a pair of very dissimilar output (high numerator) with similar inputs (low denominator), hence revealing sets of input values that are more sensitive to changes in the software behaviour. Going back to our Date example, let us consider the d_i and d_o for Dates as the edit distance [11] between the inputs and outputs, respectively, when seen as strings (note that valid dates are just printed back as strings on the output side):

- `i1 = "2021-03-31"; P(i1) = "2021-03-31".`
- `i2 = "2021-04-31"; P(i2) = "Invalid date".`

[11] The edit distance between two strings A and B is the number of operations (add, remove or replace a character) required to turn string A into the string B.

- `i3 = "2021-04-30"; P(i3) = "2021-04-30".`

As a consequence, $d_i(i1, i2) = 1$ as only one character changes between those input, whereas the output distance $d_o(P(i1), P(i2)) = 12$ since there is no overlap between the outputs, resulting in the $PD = 12/1 = 12$. In contrast, the derivative $PD(i1, i3) = 2/2 = 1$ is significantly lower and does not indicate any sudden changes in the output. In other words, the derivative changes significantly for i1 and i2, indicating boundary behavior. Figure 3.14 illustrates the program derivative of our example by varying months and dates for a fixed year value (2021) for a typical Date library. We see that automated boundary value testing can help highlight and, here, visualise boundary values.

Note that the high program derivative values delimits the boundaries for the input on those two dimensions. Therefore, program derivative is a promising candidate to be a fitness function to identify boundary values in the input space. Using our BMI example, note that we can use the program derivative to identify the pairs of height and weight that trigger changes between classifications by comparing the variation in the output distance of similar input values. For instance, the output classifications can change, e.g., from ”Overweight” to ”Obese” by comparing individuals of same height but different weight values.

However, there are still many challenges when automating the generation of boundary values. First, software input is often complex and non-numerical such as objects or standardised files, which introduces the challenge of defining a suitable and accurate distance function able to measure the distances between input/output values. Second, the input space can have many dimensions (e.g., several input arguments) of varied types and constraints such that searching through that space is costly and sometimes infeasible. Last, but not least, boundary values often involve the tester’s domain knowledge or personal experience that are hard to abstract in terms of functions or quantities (e.g., think of the Millennium bug for the date 2000-01-01). More important than fully automating the search of boundaries, testers are encouraged to employ boundary value exploration (BVE) techniques. BVE is a set of techniques (e.g., the visualisation in Figure 3.14) that propose sets of candidate boundary values to help testers in refining their knowledge of boundaries in their own programs under test [75].

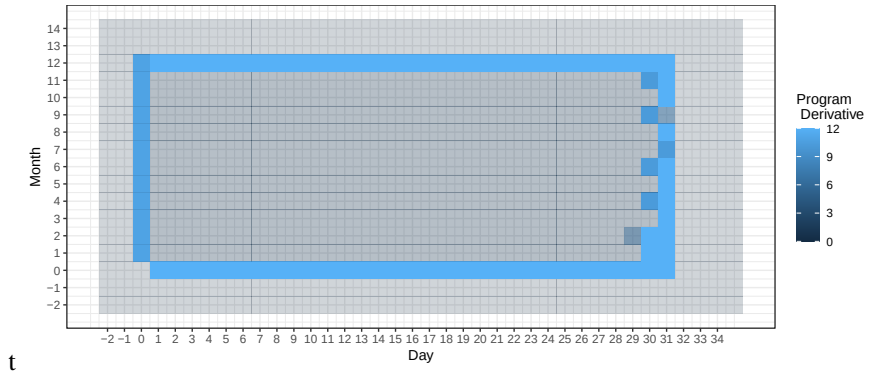


Figure 3.14: Plot of the program derivative by varying days and month for the year 2021. Higher opacity and brighter colors indicate higher derivative values. The derivative indicates the values that divide the boundary between valid and invalid combinations of days and months. Figure adapted from [75].

3.5.5 Finding Diverse Test Suites

A natural intuition that we have as software testers is that the tests we write and run need to differ from each other for the system to be properly tested. If we repeatedly rerun the same test case, or some set of very similar test cases, they are unlikely to uncover unique behaviours of the tested system. All of these similar tests will tend to pass or fail at the same time. Many AI-based techniques—including search-based approaches—have been proposed to select a good and complementary set of test cases, i.e. a diverse test suite. For example, recent research uses reinforcement learning to adapt the strategy employed by a search-based algorithm to generate more diverse tests for particular classes-under-test [76]. A study comparing many different test techniques found that techniques focused on diversity were among the best possible in selecting a small set of test cases [77].

A key problem in applying AI to find diverse test suites is how to quantify diversity. There are many ways in which we can measure how different test cases are, i.e. such as their length, which methods of the tested system they call, which inputs they provide etc. A general solution is to use general metrics from the area of Information Theory that can be used regardless of the type of data, length or specifics of the test cases we want to analyse. One study showed how metrics based on compression were very useful in quantifying test case diversity [78]. Their experiments also showed that test sets comprised of more diverse test cases had better coverage and found more faults.

A potential downside of these methods are that they can be expensive in terms of computations; many test cases and sets need to be considered to find the most diverse ones. Later research have proposed ways to speed diversity calculations up. One study used locality-sensitive hashing to speed up the diversity calculations [79]. Another study used the pair-wise distance values of all test cases as input to a dimensionality reduction algorithm so that a two-dimensional (2D) visual “map” of industrial test suites could be provided to software engineers [80].

3.5.6 Oracle Generation and Specification Mining

This chapter has focused on automatically generating the test inputs and test actions of good test cases. This excludes a key element of any test case: how to judge if the behavior of the system under test is correct. Can AI techniques help us also in generating oracles that make these judges? Or more generally, can we find or extract, i.e. mine, a specification of the SUT from actual executions of it?

Oracle generation is notoriously difficult and likely cannot be solved once and for all. While attempts have been made to "learn" a full oracle using supervised learning techniques, they are typically only viable on small and simple code examples.[12] Still, some researchers have proposed that AI can at least partly help [82]. For example, one study used the Deep AI technique of neural embeddings to summarise and cluster the execution traces of test cases [83]. Their experiments showed that the embeddings were helpful in classifying test case executions as either passing and failing. While this cannot directly be used as an oracle it can be used to select test cases to show to a human tester which can then more easily judge if the behavior is correct or not. Such interactive use of optimisation and AI in software testing has previously been shown to be effective [84].

3.5.7 Other AI Techniques

Many other AI and Machine Learning techniques beyond those that we have described in this chapter have been used to support unit testing tasks, from input generation, to test augmentation, to test selection during execution. The trend is also that the number of such applications grows strongly year by year. Below we provide a few additional examples.

Researchers have proposed the use of Reinforcement Learning when generating test inputs [85]. They implemented the same test data generation framework that had been previously used with traditional search-based, meta-heuristics [86] as well as with Nested Monte-Carlo Search [87] but instead used Reinforcement Learning to generate new test cases. A neural net was used to model the optimal choices when generating test inputs for testing a system through its API. Initial results showed that technique could reach higher coverage for larger APIs where more complex scenarios are needed for successful testing. Another early study showed how Deep Reinforcement Learning could develop its own search-based algorithm that achieves full branch coverage on a training function and that the trained neural network could then achieve high coverage also on unseen tested functions [88]. This indicates that modern AI techniques can be used to learn transferable testing skills.

Reinforcement learning has also been used *within* search-based test generation frameworks to adapt the test generation strategy to particular systems or problems. For example, it has been applied to automatically tune parameters of the metaheuristic [89], to select fitness functions in multi-objective search in service of optimising a high-level goal (e.g., selecting fitness functions that cause a class to throw more exceptions) [76], and to transform test cases by substituting individual actions for alternatives that may assist in testing inheritance in class hierarchies or covering private code [90]

Other researchers have proposed the use of supervised machine learning to generate test input (e.g., [17, 91]). In such approaches, a set of existing test input and results of executing that input (either the output or some other result, such as the code coverage)

[12] An overview of attempts to use machine learning to derive oracles is offered by Fontes and Gay: [81].

are used to train a model. Then, the model is used to guide the selection of new input that attains a particular outcome or interest (e.g., coverage of a particular code element or a new output). It has been suggested that such approaches could be useful for boundary identification—Budnik et al. propose an exploration phase where an adversarial approach is used to identify small changes to input that lead to large differences in output, indicating boundary areas in the input space where faults are more likely to emerge [91]. They also suggest comparing the model prediction with the real outcome of executing the input, and using misclassifications to indicate the need to re-train the model. Such models may also be useful for increasing input diversity as well, as prediction uncertainty indicates parts of the input space that have only been weakly tested [17].

3.6 Conclusion

Unit testing is a popular testing practice where the smallest segment of code that can be tested in isolation from the rest of the system—often a class—is tested. Unit tests are typically written as executable code, often in a format provided by a unit testing framework such as `pytest` for Python.

Creating unit tests is a time and effort-intensive process with many repetitive, manual elements. Automation of elements of unit test creation can lead to cost savings and can complement manually-written test cases. To illustrate how AI can support unit testing, we introduced the concept of search-based unit test input generation. This technique frames the selection of test input as an optimization problem—*we seek a set of test cases that meet some measurable goal of a tester*—and unleashes powerful metaheuristic search algorithms to identify the best possible test input within a restricted timeframe.

Readers interested in the concepts explored in this chapter are recommended to read further on the advanced concepts, such as distance-based fitness functions, multi-objective optimization, generating human-readable input, finding input boundaries, increasing suite diversity, oracle generation, and the use of other AI techniques—such as machine learning—to generate test input.

Paper C

The Integration of Machine Learning into Automated Test Generation: A Systematic Mapping Study

Afonso Fontes, Gregory Gay

Software Testing, Verification and Reliability (STVR), 2023.

Abstract

Context: Machine learning (ML) may enable effective automated test generation.
Objectives: We characterize emerging research, examining testing practices, researcher goals, ML techniques applied, evaluation, and challenges.
Methods: We perform a systematic mapping study on a sample of 124 publications.
Results: ML generates input for system, GUI, unit, performance, and combinatorial testing or improves the performance of existing generation methods. ML is also used to generate test verdicts, property-based, and expected output oracles. Supervised learning—often based on neural networks—and reinforcement learning—often based on Q-learning—are common, and some publications also employ unsupervised or semi-supervised learning. (Semi-/Un-)Supervised approaches are evaluated using both traditional testing metrics and ML-related metrics (e.g., accuracy), while reinforcement learning is often evaluated using testing metrics tied to the reward function.
Conclusion: Work-to-date shows great promise, but there are open challenges regarding training data, retraining, scalability, evaluation complexity, ML algorithms employed—and how they are applied—benchmarks, and replicability. Our findings can serve as a roadmap and inspiration for researchers in this field.
keywords: Automated Test Generation, Test Case Generation, Test Input Generation, Test Oracle Generation, Machine Learning

4.1 Introduction

Software testing is invaluable in ensuring the reliability of the software that powers our society [12]. It is also notoriously difficult and expensive, with severe consequences for productivity, the environment, and human life if not conducted properly. New tools and methodologies are needed to control that cost without reducing the quality of the testing process.

Automation has a critical role in controlling costs and focusing developer attention [27]. Consider test generation—an effort-intensive task where sequences of program *input* and *oracles* that judge the correctness of the resulting execution are crafted for a system-under-test (SUT) [12]. Effective automated test generation could lead to immense effort and cost savings.

Automated test generation is a popular research topic, and outstanding achievements have been made in the area [27]. Still, there are critical limitations to current approaches. Major among these is that generation frameworks are applied in a *general* manner—techniques target simple universal heuristics, and those heuristics are applied in a static manner to all systems equally. Parameters of test generation can be tuned by a developer, but this requires advanced knowledge and is still based on the same universal heuristics. Current generation frameworks are largely unable to adapt their approach to a particular SUT, even though such projects offer rich information content in their documentation, metadata, source code, or execution logs [6]. Such static application limits the potential effectiveness of automated test generation.

Machine learning (ML) algorithms make predictions by analyzing and extrapolating from sets of observations [6]. Advances in ML have shown that automation can match or surpass human performance across many problem domains. ML has advanced the state-of-the-art in virtually every field. Automated test generation is no exception. Recently, researchers have begun to use ML either to *directly* generate input or oracles [88] or to *enhance* the effectiveness or efficiency of existing test generation frameworks [19]. ML offers the potential means to adapt test generation to a SUT, and to enable automation to optimize its approach without human intervention.

We are interested in understanding and characterizing emerging research around the integration of ML into automated test generation[1]. Specifically, we are interested in which testing practices have been addressed by integrating ML into test generation, the goals of the researchers using ML, how ML is integrated into the generation process, which specific ML techniques are applied, how such techniques are trained and validated, and how the whole test generation process is evaluated. We are also interested in identifying the emerging field's limitations and open research challenges. To that end, we have performed a systematic mapping study. Following a search of relevant databases and a rigorous filtering process, we have examined 124 relevant studies, gathering the data needed to answer our research questions.

We observed that ML supports generation of input and oracles for a variety of testing practices (e.g., system or GUI testing) and oracle types (e.g., expected test verdicts and expected output values). During input generation, ML either directly generates input or improves the efficiency or effectiveness of existing generation methods. The most common types of ML are supervised and reinforcement learning. A small number of publications also employ unsupervised or semi-supervised/adversarial

[1]We focus specifically on the use of ML to enhance test generation, as part of the broader field of AI-for-Software Engineering (AI4SE). There has also been research in automated test generation for ML-based systems (SE4AI). These studies are out of the scope of our review.

learning.

Supervised learning is the most common type for system testing, Combinatorial Interaction Testing, and all forms of oracle generation. Neural networks are the most common supervised techniques, and techniques are evaluated using both traditional testing metrics (e.g., coverage) and ML metrics (e.g., accuracy). Reinforcement learning is the most common ML type for GUI, unit, and performance testing. It is effective for practices with scoring functions and when testing requires a sequence of input steps. It is also effective at tuning generation tools. Reinforcement learning techniques are generally based on Q-Learning, and are generally evaluated using testing metrics tied to the reward function. Finally, unsupervised learning is effective for filtering tasks such as discarding similar test cases.

The publications show great promise, but there are significant open challenges. Learning is limited by the required quantity, quality, and contents of training data. Models should be retrained over time. Whether techniques will scale to real-world systems is not clear. Researchers rarely justify the choice of ML technique or compare alternatives. Research is limited by the overuse of simplistic examples, the lack of standard benchmarks, and the unavailability of code and data. Researchers should be encouraged to use common benchmarks and provide replication packages and code. In addition, new benchmarks could be created for ML challenges (e.g., oracle generation).

Our study is the first to thoroughly summarize and characterize this emerging research field[2] We hope that our findings will serve as a roadmap for both researchers and practitioners interested in the use of ML in test generation and that it will inspire new advances in the field.

4.2 Background and Related work

4.2.1 Software Testing

It is essential to verify that software functions as intended. This verification process usually involves *testing*—the application of *input*, and analysis of the resulting *output*, to identify unexpected behaviors in the system-under-test (SUT) [12].

During testing, a *test suite* containing one or more *test cases* is applied to the SUT. A test case consists of a *test sequence (or procedure)*–a series of interactions with the SUT–with *test input* applied to some SUT component. Depending on the granularity of testing, the input can range from method calls, to API calls, to actions within a graphical interface. Then, the test case will validate the output against a set of encoded expectations—the *test oracle*—to determine whether the test passes or fails [12]. An oracle can be a predefined specification (e.g., an assertion), output from a past version, a model, or even manual inspection by humans [12].

An example of a test case, written in the JUnit notation, is shown in Figure 4.1. The test input is a string passed to the constructor of the `TransformCase` class, then a call to `getText()`. An assertion then checks whether the output matches the expected output—an upper-case version of the input.

Testing can be performed at different granularity levels, using tests written in code or applied by humans. The lowest granularity is unit testing, which focuses on isolated

[2]This publication extends an initial systematic literature review [81] that focused only on test oracle generation. Our extended study also includes publications on test input generation and an expanded set of publications for oracle generation. We also include additional and extended analyses and discussion.

```
@Test
public void testPrintMessage() {
    String str = "Test␣Message";
    TransformCase tCase = new TransformCase(str);
    String upperCaseStr = str.toUpperCase();
    assertEquals(upperCaseStr, tCase.getText());
}
```

Figure 4.1: Example of a unit test case written using the JUnit notation for Java.

code modules (generally classes). Module interactions are tested during integration testing. Then, during system testing, the SUT is tested through one of its defined interfaces—a programmable interface, a command-line interface, a graphical user interface, or another external interface. Human-driven testing, such as exploratory testing, is out of the scope of this study, as it is often not amenable to automation.

4.2.2 Machine Learning

Machine learning (ML) constructs models from observations of data to make predictions [6]. Instead of being explicitly programmed like in traditional software, ML algorithms "learn" from observations using statistical analyses, facilitating the automation of decision making. ML has enabled many new applications in the past decade. As computational power and data availability increase, such approaches will increase in their capabilities and accuracy.

ML approaches largely fall into four categories—supervised, semi-supervised, unsupervised, and reinforcement learning—as presented in Figure 4.2. In supervised learning, algorithms infer a model from the training data that makes predictions about newly encountered data. Such algorithms are typically used for classification—prediction of a label from a finite set—or regression—predictions in an unrestricted format, e.g., a continuous value. For example, a model may be trained from image data with the task of classifying whether an animal depicted in a new image is a cat. If a sufficiently large training dataset with a low level of noise is used, an accurate model can often be trained quickly. However, a model is generally static once trained and cannot be improved without re-training.

Semi-supervised algorithms are a form of supervised learning where feedback mechanisms are employed to automatically retrain models. For example, adversarial networks refine accuracy by augmenting the training data with new input by putting two supervised algorithms in competition. One of the algorithms creates new inputs that mimic training data, while the second predicts whether these are part of the training data or impostors. The first refines its ability to create convincing fakes, while the second tries to separate fakes from the originals. Semi-supervised approaches require a longer training time, but can achieve more optimal models, often with a smaller initial training set.

Unsupervised algorithms do not use previously-labeled data. Instead, approaches identify patterns in data based on the similarities and differences between items. They model the data indirectly, with little-to-no human input. Rather than making predictions, unsupervised techniques aid in understanding data by, e.g., clustering related items, extracting interesting features, or detecting anomalies. As an example, a clustering algorithm could take a set of images and cluster them into groups based on the similarity of the images. Such an algorithm could not predict whether a specific

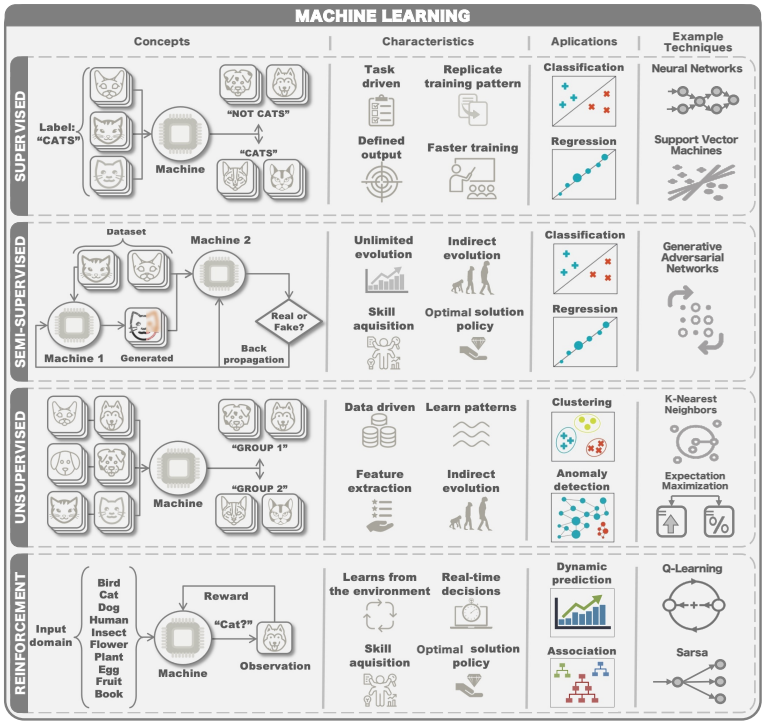


Figure 4.2: Types of ML and their concepts, characteristics, and applications.

image had a cat in it—as was done in supervised learning—but would likely place many of the cat-containing images in the same cluster.

Reinforcement learning algorithms select actions based on an estimation of their effectiveness towards achieving a measurable goal [19]. Reinforcement learning often does not require training data, instead learning through sequences of interactions with its environment. Reinforcement learning "agents" use feedback on the effect of actions taken to improve their estimation of the actions most likely to maximize achievement of their goal (their "policy"). Feedback is provided by a reward function—a numeric scoring function. For example, an agent may predict which animal is contained in an image. It would then get a score based on how close its guess was to being correct—e.g., if the image contained a cat, then a guess of "dog" would get a higher score than a guess of "spider". The agent can also adapt to a changing environment, as estimations are refined each time an action is taken. Such algorithms are often the basis of automated processes, such as autonomous driving, and are effective in situations where sequences of predictions are required.

Recent research often focuses on "deep learning". Deep approaches make complex and highly accurate inferences from massive datasets. Many DL approaches are based on complex many-layered neural networks—networks that attempts to mimic how the human brain works [92]. Such neural networks employ a cascade of nonlinear processing layers where one layer's output serves as the successive layer's input. Deep learning requires a computationally intense training process and larger datasets than traditional ML, but can learn highly accurate models, extract features and relationships

from data automatically, and potentially apply models across applications. “Deep” approaches exist for all four of the ML types discussed above.

4.2.3 Common Test Generation Techniques

Many techniques have been used to generate test input. In this subsection, we briefly introduce four common approaches: (a) random test generation, (b) search-based test generation, (c) symbolic execution, and (d), model-based test generation.

In random test generation, input is generated purely at random and applied to the system-under-test with the aim of triggering a failure. Random input generation is one of the most fundamental, simple, and easy-to-implement generation techniques [7]. Unfortunately, while random testing is often very efficient, most software has too large of an input space to exhaustively cover. Therefore, a weakness of random testing is that the generated input may only span a small, and uneven, portion of that input space. Therefore, many *adaptive* random testing techniques have been proposed. In adaptive random testing, mechanisms are employed to partition the input space, and input is generated for each partition [17]. This ensures an even distribution across the input space.

Search-based test generation formulates input generation as an optimization problem [19]. Of that near-infinite set of inputs for an SUT, we want to identify—systematically and at a reasonable cost—those that meet our goals. Given scoring functions that measure closeness to the attainment of those goals—fitness functions—metaheuristic optimization algorithms can automate that search by selecting input and measuring their fitness. Metaheuristic algorithms are often inspired by natural phenomena. For example, genetic algorithms evolve a population of candidate solutions over many generations by promoting, mutating, and breeding fit solutions. Such techniques retain many of the benefits of random testing, including scalability, and are often better able to identify failure-inducing input [3], or input with other properties of interest [93].

Symbolic execution is a program analysis technique where symbolic input is used instead of concrete input to “execute” the program [16]. The values of program variables are represented by symbolic expressions over these inputs. Then, at any point during a symbolic execution, the program’s state can be represented by these symbolic values of program variables and a Boolean formula containing the collected constraints that must be satisfied for that path through the program to have been taken, also known as the path constraint. By identifying concrete input that satisfies a path constraint, we can ensure that particular paths through the program are covered by test cases. Constraint solvers can be used to identify such input automatically. Recent approaches often are based on dynamic symbolic execution (or concolic execution), where the symbolic execution is combined with concrete random execution to ease the difficulty of solving complex path constraints [7].

Finally, in model-based testing, lightweight models representing aspects of interest of an SUT are used to derive test cases [18]. Often, such models take the form of a state machine. The internal behavior of the SUT is represented by its state. Transitions are triggered by applying input to the SUT. The model describes—at a chosen level of abstraction—the expected SUT behavior over a sequence of input actions. Test cases can then be derived from this model by choosing relevant subsets of input sequences [7].

4.2.4 Related Work

Other secondary studies overlap with ours in scope. We briefly discuss these publications below. Our SLR is the first focused specifically on the application of ML to automated test generation, including both input and oracle generation, and no related study overlaps in full with our research questions. We have also examined a larger and more recent sample of publications.

Durelli et al. performed a systematic mapping study on the application of ML to software testing [6]. Their scope is broad, examining how ML has been applied to any aspect of the testing process. They mapped 48 publications to testing activities, study types, and ML algorithms employed. They observe that ML has been used to generate input and oracles. They note that supervised algorithms are used more often than other ML types and that Artificial Neural Networks are the most used algorithm. Jha and Popli also conducted a short review of literature applying ML to testing activities [21], and note that ML has been used for both input and oracle generation.

Ioannides and Eder conducted a survey on the use of AI techniques to generate test cases targeting code coverage—known as "white box" test generation [22]. Their survey focuses on optimization techniques, such as genetic algorithms, but they note that ML has been used to generate test input.

Barr et al. performed a survey on test oracles [12]. They divide test oracles into four types, including those specified by humans, those derived automatically, those that reflect implicit properties of programs, and those that rely on a human-in-the-loop. Approaches based on ML fall into the "derived" category, as they learn automatically from project artifacts to replace or augment human-written oracles. They discuss early approaches to using ML to derive oracles.

Balera et al. conducted a systematic mapping study on hyper-heuristics in search-based test generation [23]. Search-based test generation applies optimization algorithms to generate test input. A hyper-heuristic is a secondary optimization performed to tune the primary search strategy, e.g., a hyper-heuristic could adapt test generation to the current SUT. A hyper-heuristic can apply ML, especially RL, but can also be guided by other algorithms. We also observe the use of ML-based hyper-heuristics.

ID	**Research Question**	**Objective**
RQ1	Which testing practices have been supported by integrating ML into the generation process?	Highlights testing scenarios and systems types targeted for ML-enhanced test generation.
RQ2	What is the goal of using machine learning as part of automated test generation?	To understand the reasons for applying ML techniques to perform or enhance test generation.
RQ3	What types of ML have been used to perform or enhance automated test generation?	Identifies the type of ML applied, how it was integrated into the generation process, and how it was trained and validated.
RQ4	Which specific ML techniques were used to perform or enhance automated test generation?	Identify specific ML techniques used in the process, including type, learning method, and selection mechanisms.
RQ5	How is the test generation process evaluated?	Describe the evaluation of the ML-enhanced test generation process, highlighting common metrics and artifacts (programs or datasets) used.
RQ6	What are the limitations and open challenges in integrating ML into test generation?	Highlights the limitations of enhancing test generation with and future research directions.

Table 4.1: List of research questions, along with motivation for answering the question.

4.3 Methodology

Our aim is to understand how researchers have integrated ML into automated test generation, including generation of input and oracles. We have investigated publications related to this topic and seek to understand their methodology, results, and insights. To gain this understanding, we performed a systematic mapping study according to the guidelines of Petersen et al. [94].

We are interested in assessing the *effect* of integrating ML into the test generation process, understanding the *adoption* of these techniques—how and why they are being integrated, and which specific techniques are being applied, and identifying the potential *impact* and *risks* of this integration. Table 4.1 lists the research questions we are interested in answering and clarifies the purpose of asking such questions.

The first three questions are high-level questions that clarify how ML has enabled or enhanced test generation, why ML was applied, and which specific testing scenarios were targeted by the enhanced generation techniques. **RQ1** enables us to categorize publications in terms of specific testing practices. By “testing practices”, we refer either to the code or interface level that testing is aimed at (e.g., unit or GUI testing) or to specialized forms of testing (e.g., performance testing). To answer this question, we divide the sampled publications into categories based on the specific goals and targets of test generation. We did not start with pre-decided categories but analyzed and thematically grouped publications. **RQ2** is motivational, covering the authors’

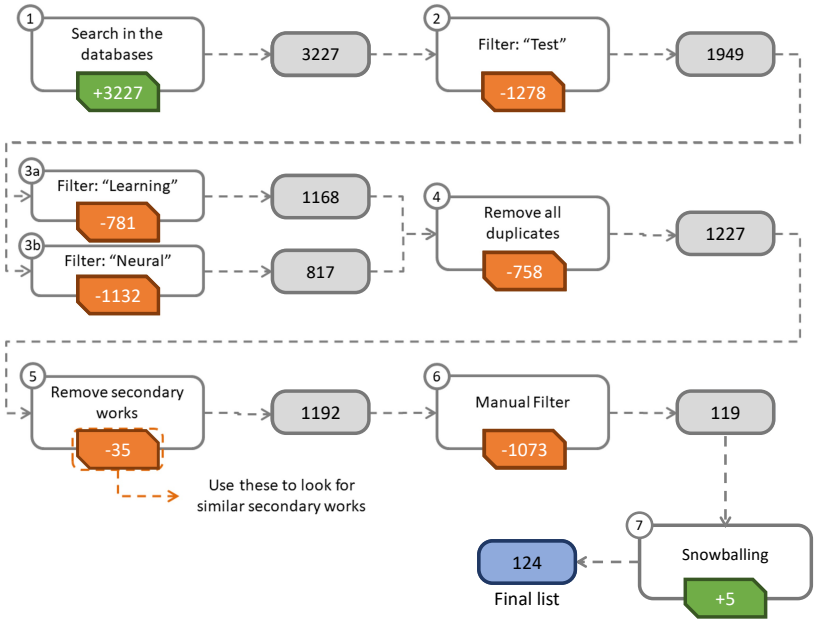


Figure 4.3: Steps taken to determine the final list of publications to analyze.

primary objectives. We are interesting in how the authors intended to use ML—e.g., to directly generate input, to enhance an existing test generation technique, or to identify weaknesses in a testing strategy.

In contrast, **RQ3-5** are technical questions. **RQ3** examines the broad category of ML technique (i.e., supervised, unsupervised, semi-supervised, or reinforcement learning), as well as its training and validation processes. **RQ4** examined which specific ML techniques (e.g., backpropagation neural networks) were used to perform the generation task. **RQ5** focuses on how the test generation approach is evaluated, including metrics and types of systems tested. This can include both the generation framework as a whole, or the specific ML aspect of the framework. Finally, the last research question covers the limitations of the proposed approaches and open research challenges (**RQ6**).

To answer these questions, we have performed the following tasks:

[a] Formed a list of publications by querying publication databases (Section 4.3.1).

[b] Filtered this list for relevance (Section 4.3.2).

[c] Extracted and classified data from each study, guided by properties of interest (Section 4.3.3).

[d] Identified trends in the extracted data to answer each research question (described along with results in Section 4.4).

4.3.1 Initial Study Selection

To locate publications for consideration, a search was conducted using four databases: IEEE Xplore, ACM Digital Library, Science Direct, and Scopus. To narrow the results,

we created a search string by combining terms of interest on test generation and machine learning. The search string used was:

(“test case generation” OR “test generation” OR “test oracle” OR “test input”) AND (“machine learning” OR “reinforcement learning” OR “deep learning” OR “neural network”)

These keywords are not guaranteed to capture all publications on ML in test generation. However, they are intended to attain a relevant sample. Specifically, we combine terms related to test generation and terms related to machine learning, including common technologies. Our focus is not on any particular form of test generation. To obtain a representative sample, we have selected ML terms that we expect will capture a wide range of publications. These terms may omit some in-scope ML techniques, but attain a relevant sample while constraining the amount of manual inspection.

We limited our search to peer-reviewed publications in English. The search string was applied to the full text of articles. Our set of articles was gathered in March 2023, containing an initial total of 3227 articles. This is shown as the first step in Figure 2.2.

To evaluate the search string’s effectiveness, we conducted a verification process. First, we randomly sampled ten entries from the final publication list. Then we looked in each article for ten citations that were in scope, resulting in 100 citations. We checked whether the search string also retrieved these citations, and all 100 were retrieved. Although this is a small sample, it indicates the robustness of the string.

4.3.2 Selection Filtering

We next applied a series of filtering steps to obtain a focused sample. Figure 2.2 presents the filtering process and the number of entries after applying each filter. The number in box 1 represents the initial number of articles. The numbers in the other boxes represent the number of entries removed in that particular step. The numbers between the steps show the total number of articles after applying the previous step.

To ensure that publications are relevant, we used keywords to filter the list. We first searched the title and abstract of each study for the keyword “test” (including, e.g., “testing”). We then searched the title and abstract of the remaining publications for either “learning” or “neural”—representing application of ML. We merged the filtered lists, and removed all duplicate entries. We then removed all secondary studies. This left 1192 publications.

We examined the remaining publications manually, removing all publications not in scope. Publications were **included** if they met the following conditions:

- The publication must be written in English.
- The publication must have appeared in a peer-reviewed venue. This includes journals, conferences, and workshops.
- The publication is a primary study (i.e., reporting original research).
- The research reported relates to test generation—including test inputs, oracles, or full test cases—**and** applies any machine learning technique as part of the generation process.

Articles were **excluded** under the following conditions:

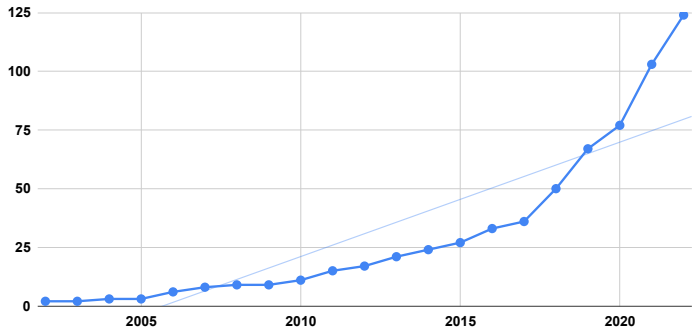


Figure 4.4: Growth of the use of ML in test generation since 2002.

- The publication was not written in English.
- The publication was not in a peer-reviewed venue (e.g., book chapters, letters, white or grey literature).
- The publication is a secondary study (e.g., systematic literature review or mapping study). However, such studies were considered for discussion in Section 4.2.4.
- The reported research does not relate to test generation, or the research is not discussed in the context of test generation[3].
- The reported research does not apply ML as part of test generation (i.e., a non-ML technique is applied or ML is applied to an activity unrelated to test generation).
- The reported research relates to testing *of* ML-based systems rather than test generation.

This determination was made by first reading the title, abstract, and introduction. Then, if the publication seemed in scope, we proceeded to read the entire study. In a small number of cases, publications were deemed out-of-scope only after inspection of the full article. Both authors independently inspected publications during this step to prevent the accidental removal of relevant publications. In cases of disagreement, the authors discussed the study.

This process resulted in a sample of 119 articles. We then performed snowballing by inspecting the bibliography of each publication and adding any additional publications that met our inclusion criteria stated above. The snowballing process added five additional publications, resulting in a final sample of 124 publications.

The publications are listed in Section 4.4.2, associated with the specific testing practice addressed. Figure 4.4 shows the growth of interest in this topic since 2002 (only one study in this sample, from 1993, was published before this date). We can see modest, but growing, interest until 2010. The advancements in ML in the past decade

[3]For example, ML could be applied as part of test suite reduction. Test suite reduction can be applied as part of a broader test generation framework, or it can be applied as a standalone testing technique. If the research was presented explicitly as part of test generation, we retained the publication. If the research was presented in a standalone context, then it was discarded.

ID	**Property Name**	**RQ**	**Description**
P1	Testing Practices Addressed	RQ1, RQ2	The specific type of testing scenarios or application domain focused on by the approach. It helps to categorize the publications, enabling comparison between contributions.
P2	Proposed Research	RQ2	A short description of the approach proposed or research performed.
P3	Hypotheses and Results	RQ1, RQ3	Highlights the differences between expectations and conclusions of the proposed approach.
P4	ML Integration	RQ3	Covers how ML techniques have been integrated into the test generation process. It is essential to understand what aspects of generation are handled or supported by ML.
P5	ML Technique Applied	RQ4	Name, type, and description of the ML technique used in the study.
P6	Reasons for Using the Specific ML Technique	RQ4	The reasons stated by the authors for choosing this ML technique.
P7	ML Training Process	RQ4	How the approach was trained, including the specific data sets or artifacts used to perform this training. This property helps us understand how each contribution could be replicated or extended.
P8	External Tools or Libraries Used	RQ4	External tools or libraries used to implement the ML technique.
P9	ML Objective and Validation Process	RQ4, RQ5	This attribute covers the objective of the ML technique (e.g., reward function or validation metric), and how it is validated, including data, artifacts, and metrics used (if any).
P10	Test Generation Evaluation Process	RQ5	Covers how the ML-enhanced oracle generation process, as a whole, is evaluated (i.e., how successful are the generated input at triggering faults or meeting some other testing goal?). Allows understanding of the effects of ML on improving the testing process.
P11	Potential Research Threats	RQ6	Notes on the threats to validity that could impact each study.
P12	Strengths and Limitations	RQ6	This property is used to understand the general strengths and limitations of enhancing a generation process with ML by collecting and synthesizing these aspects for both the ML techniques and entire test generation approaches.
P13	Future Work	RQ6	Any future extensions proposed by the authors, with a particular focus on those that could overcome the identified limitations.

Table 4.2: List of properties used to answer the research questions. For each property, we include a name, the research questions the property is associated with, and a short description.

have resulted in significantly more use of ML in test generation, especially starting in 2018. Over 70% of the publications in our sample were published in the past five years alone—with 38% in the past two years. This is an area of growing interest and maturity, and we expect the number of publications to increase significantly in the next few years.

4.3.3 Data Extraction and Classification

To answer the questions in Table 4.1, we have extracted a set of key properties from each study, identified in Table 4.2. Each property listed in the table is briefly defined and is associated with the research questions. Several properties may collectively answer a RQ. For example, RQ2—covering the goals of using ML—can be answered using property P2. However, P1 provides context and the testing practice addressed may dictate how ML is applied.

Data extraction was performed primarily by the first author of this study. However, to ensure the accuracy of the extraction process, the second author performed a full independent extraction for a sample of ten randomly-chosen publications. We compared our findings, and found that we had near-total agreement on all properties. The second author then performed a lightweight verification of the findings of the first author for the remaining publications. A small number of corrections were discussed between the authors, but the data extraction was generally found to be accurate.

Systematic mapping studies generally address research questions by grouping publications into different *classifications*, then analyzing trends in the publications in each group. We likewise group publications in the following ways:

- The testing practice addressed. We first divide the research into input and oracle generation, then a specific input granularity (e.g., unit or system-level input generation) or input/test type (e.g., performance testing) or specific oracle type (e.g., test verdicts, expected output).

- The type of ML applied (e.g., supervised or reinforcement learning).

- The specific ML technique applied (e.g., backpropagation neural network).

- The type of training data used, if applicable (e.g., previous system executions).

- The objective of applying ML. This includes both the type of prediction being made (e.g., classification or regression) and the purpose of the prediction (e.g., predicting the input that will cover a path in the code). For reinforcement learning, this includes the reward functions used.

- The evaluation metrics used to assess the proposed research. This includes both traditional test generation evaluation metrics (e.g., number of faults detected) and ML-related metrics (e.g., accuracy).

- The type of example systems used in the evaluation.

For the ML approach, we use the four primary categories of ML described in Section 4.2 to classify publications—supervised, semi-supervised, unsupervised, and reinforcement learning. For the other categories, we did not begin with pre-determined classifications. Rather, we performed thematic analysis of the articles to identify natural groupings in the publication sample. In all cases, our goal was to avoid oversimplification—we favored a large number of specialized classes over a small number of over-arching classes. We describe classification more concretely in Section 4.4.

4.4 Results and Discussion

In this section, first, we identify the testing practices addressed by ML-enhanced test generation (RQ1, Section 4.4.1). We then note observations regarding research related to individual testing practices (Section 4.4.2). Finally, we present answers to RQ2-6 (Sections 4.4.3–4.4.7).

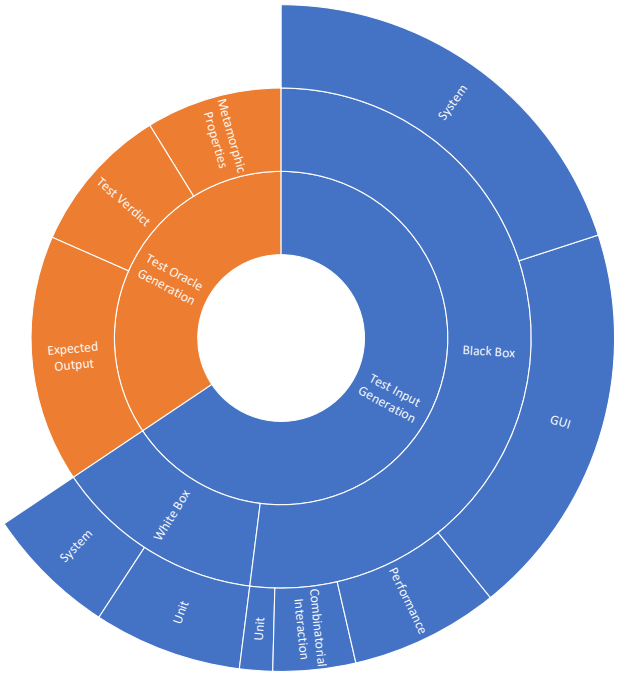
4.4.1 RQ1: Testing Practices Addressed

The purpose of RQ1 is to give an overview of which testing practices have been targeted by the publications to help structure our examination of the sampled articles. Our categorization is shown in Figure 4.5. In this chart, we divide articles into layers, with each layer representing finer levels of granularity. The total number of publications in each category is reported below.

The specific formulation of a test case depends on the product domain and technologies utilized by the SUT [1]. However, broadly, a test case is defined by a set of input steps and test oracles [12], both of which can be the target of automated generation. Therefore, we decided that *input* and *oracles* constitute our first division.

A majority of articles focus on input generation (67% of the sample). Automated input generation has become a major research topic in software testing over the past 20 years [27], and many different forms of automated generation have been proposed, using approaches ranging from symbolic execution [16] to optimization [19]. Oracle generation has long been seen as a major challenge for test automation research [12,27]. However, ML is a realistic route to achieve automated oracle generation [81], and a significant number of publications have started to appear on this topic (33%).

Figure 4.6(a) shows the growth in both topics since 2002. Both show a similar trajectory until 2017, with a sharp increase in input generation after. New ML technologies, such as deep learning, and the growing maturity of open-source learning frameworks, such as PyTorch, Keras, and OpenAI Gym, have potentially contributed to this increase.



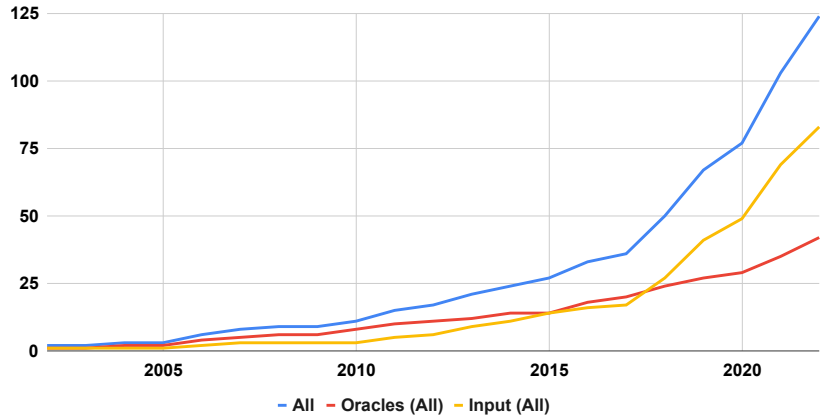
Layer #1	
Test Input Generation	82
Test Oracle Generation	42

Layer #2	
Black Box	65
White Box	17
Expected Output	20
Test Verdict	12
Metamorphic Properties	10

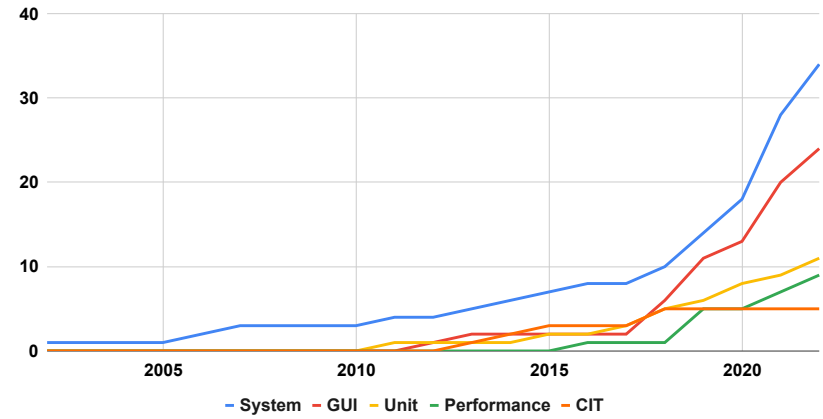
Layer #3	
System Test Generation (Black Box)	25
GUI Test Generation	24
Performance Test Generation	9
Combinatorial Interaction Testing	5
Unit Test Generation (Black Box)	2
Unit Test Generation (White Box)	9
System Test Generation (White Box)	8

Figure 4.5: Testing practices addressed by test generation approaches incorporating ML.

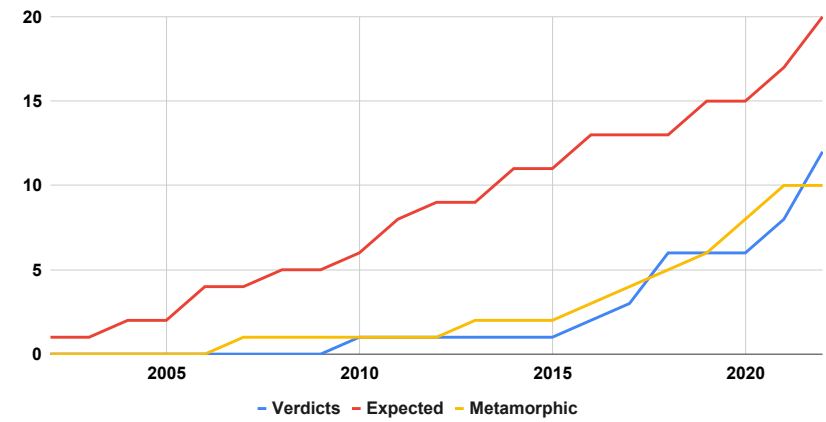
Figure 4.6(a) shows the growth in both topics since 2002. Both show a similar trajectory until 2017, with a sharp increase in input generation after. New ML technologies, such as deep learning, and the growing maturity of open-source learning frameworks, such as PyTorch, Keras, and OpenAI Gym, have potentially contributed to this increase.



(a) Input and Oracle Generation



(b) Forms of Input Generation



(c) Forms of Oracle Generation

Figure 4.6: Grown in the use of ML in test generation since 2002.

4.4.1.1 Test Input Generation

In the second layer of Figure 4.5, we further divided test input generation by the source of information used to create test input:

- **Black Box Testing:** Also known as **functional testing** [1], approaches use information about the program gleaned from documentation, requirements, and other project artifacts to create test inputs.
- **White Box Testing:** Also known as **structural testing** [1], approaches use the source code to select test inputs (e.g., generating input that covers a particular outcome for an `if`-statement). Approaches do not require domain knowledge.

Of the 82 publications addressing input generation, 65 propose Black Box and 17 propose White Box approaches. White Box approaches are traditionally common in input generation, as the "coverage criteria"—checklists of goals [95]—that are the focus of White Box testing offer measurable test generation targets [19]. Such approaches benefit from the inclusion of ML [19]. However, ML may have great potential to enhance Black Box testing. Black Box approaches are based on external data about how the system should behave. ML can be used to automate analyses of that data—enabling new approaches to test generation—as shown by 80% of input generation publications proposing Black Box approaches.
In the third layer of Figure 4.5, we further subdivided approaches based on either the level of granularity that generated inputs were applied at or by the specialized form of input generated:

- **System Test Generation (33 publications):** A practice where tests target a subsystem or full system through a defined interface (e.g., API or CLI) and verify high-level functionality through that interface.
- **GUI Test Generation (24 publications):** A specialized form of system testing where tests target a GUI to identify incorrect functionality or usability/accessibility issues [96]. We also incorporate game testing (when conducted through a GUI) into this category [97].
- **Unit Test Generation (11 publications):** A practice where test cases target a single class and exercise its functionality in isolation from other classes.
- **Performance Test Generation (9 publications):** Tests are generated to assess whether the SUT meets non-functional requirements (e.g., speed, scalability, or resource usage requirements) [98]
- **Combinatorial Interaction Testing (5 publications):** A system-level practice that attempts to produce a small set of tests that cover important interactions between input variables [89].

System-level testing is the most common category (41% of input generation), followed by GUI (29%), then unit testing (13%). GUI, performance (11%) and combinatorial interaction testing (CIT) (6%) represent specialized forms of system testing.

Figure 4.6(b) shows the growth in each area of input generation. We see a particularly strong growth in system and GUI testing since 2017. In addition to the emergence of open-source ML frameworks, we also hypothesize that this is partially driven by

the emergence of mobile and web applications and autonomous vehicles. Mobile applications are tested primarily through a GUI, as are many web applications—leading to increased interest in GUI testing. Only two GUI test generation articles predate 2017, and of the post-2017 articles, 86% relate to testing of either mobile or web applications (see Table 4.5). Other web applications are tested through REST APIs, and are included in system testing. Autonomous vehicles also require new approaches, as they are tested in complex simulators [99]. Since 2017, web and autonomous vehicle testing constitute the two largest dedicated domains for system testing, with 15% and 19% of post-2017 publications, respectively (see Tables 4.3-4.4).

4.4.1.2 Test Oracle Generation

The second layer under test oracle generation in Figure 4.5 divides approaches based on the *type* of test oracle produced:

- **Expected Output (20 publications):** The oracle predicts concrete output behavior that should result from an input. Often, this will be abstracted (i.e., a *class* of output).
- **Test Verdicts (12 publications):** The oracle predicts the final test verdict for a given input (i.e., a "pass" or "fail").
- **Metamorphic Relations and Other Properties of Program Behavior (10 publications):** A metamorphic relation is a property relating input to expected output [34]—e.g., $sin(x) = sin(\pi - x)$. Such properties, as well as other property types that specify the expected behavior of a SUT, can be applied to many inputs. Violations of such properties identify potential faults.

ML supports decision processes. A ML technique makes a prediction, which can either be a decision or information that supports making a decision. Test oracles follow a similar model, consisting of *information* used to issue a verdict and a *procedure* to arrive at a verdict [33]. ML offers a natural means to replace either component. Test verdict oracles replace the procedure, while expected output and property oracles support arriving at a verdict. Figure 4.6(c) shows steady growth for all types.

RQ1 (Testing Practices): ML supports the generation of both test input and oracles, with a greater focus on input generation (67% of the sample). Input generation research targets system testing, specialized types of system testing (GUI, performance, CIT), and unit testing. The majority of these are Black Box approaches, with White Box approaches primarily restricted to unit testing. There has been an increase in system and GUI input generation since 2017, potentially related to the emergence of web and mobile applications and autonomous driving, as well as to the availability of robust, open-source ML and deep learning frameworks. ML supports generation of test verdict, metamorphic (and other property-based), and expected output oracles.

4.4.2 Examining Specific Practices

Before answering the remaining research questions, we examine concretely how ML has supported test generation.

4.4.2.1 System Test Generation

A total of 33 publications target system testing. Table 4.3 outline Black Box approaches, while Table 4.4 outlines White Box approaches. Each table is sorted by ML approach, then by the first author's name. When discussing the objective, we indicate both type of prediction and the purpose of the prediction.

Input Generation (Supervised, Semi-Supervised): Supervised approaches generally train models that associate particular SUT input with targeted qualities. Multiple authors use supervised learning to infer a model from execution logs containing inputs and resulting output [91, 100–102]. The model is used to predict input leading to output of interest. For example, Budnik et al. identify small changes in input that lead to large differences in output, indicating boundary areas where faults are likely to emerge [91]. Both Bergadano and Budnik et al. suggest comparing predictions with real output and using misclassifications to indicate the need to re-train [91, 100].

Another concern is achieving code coverage. Majma et al. use supervised learning for both input and oracle generation [95]. A model associates inputs with paths through the source code, then generates new inputs that execute uncovered paths. Similarly, Feldmeier and Fraser generate test inputs for games by targeting code segments, then training neural networks to predict the player actions in particular game states that will cover the associated lines of code [103]. Utting et al. cluster log files—gathered from customer reports—then compare clusters to logs from executing existing test cases to identify weakly-tested areas of the SUT [104]. Supervised learning is used to fill in these gaps. These logs are formatted as vectors of actions, and the model predicts the next input in the sequence.

Others train models to predict which input will fail. Kirac et al. train a model to identify usage behaviors likely to lead to failures using past test cases [105]. Eidenbenz et al. randomly generate a set of inputs, execute them, label the execution based on whether they failed, and then cluster failing instances to enhance accuracy [106]. They train a model using several algorithms, then compare their ability to predict failing input. They propose an iterative process where more training data is added over time, and predictions are verified by developers.

Several authors generate input using models inferred from behavioral specifications (e.g., requirements). The generated input can then show that these specifications are met. Kikuma et al. create a dataset where requirements are tagged with output that should appear if the requirement is met. Their model associates input actions, conditions, and outputs in the requirements, then generates new tests with inputs, conditions, and expected output [107]. Ueda et al. transform specifications, written in natural language, into a structured abstract test recipe that can be concretely instantiated with different input [108]. Meinke et al. model use cases in a constraint language and generate input from the model inferred from the constraints [109].

In addition, both Deng et al. and Zhang et al. generate input for autonomous vehicles intended to violate properties written by human testers [110, 111]. They present adversarial scenarios where multiple neural networks manipulate image data used as input to an autonomous driving system. Collectively, these models predict which input will violate properties by, e.g., changing day to night or adding rain, and use feedback on their actions to retrain.

Finally, multiple authors generate complex inputs for particular system types. For example, Shrestha [18] train a model to generate valid Simulink models—a visual language for modeling and simulation—for testing tool-chains based on the language.

Table 4.3: Publications under **System Test Generation (Black Box)** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate. NN = Neural Network.

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[112]	2016	Reinforcement	Q-Learning	N/A	Reward (Plan Coverage)	Code Coverage,Assertion Coverage	Robotic Systems
[99]	2021	Reinforcement	Q-Learning	N/A	Reward (Criticality)	Faults Detected	Autonomous Vehicles
[113]	2013	Reinforcement	Delayed Q-Learning	N/A	Reward (Test Improvement)	% of Runs Where Requirements Met	Ship Logistics
[114]	2021	Reinforcement	Q-Learning	N/A	Reward (Code Coverage)	Code Coverage	Triangle Classification, Nesting Structure, Complex Conditions
[85]	2020	Reinforcement	Asynchronous Advantage Actor Critic	N/A	Reward (Transition Coverage)	Not Evaluated	OpenAPI APIs
[115]	2020	Reinforcement	Monte Carlo Control	N/A	Reward (Input Diversity)	Input Diversity, Code Coverage	XML, JavaScript Parsing
[116]	2022	Reinforcement	Deep Q-Network	N/A	Reward (Transition Coverage)	Efficiency,Sensitivity,Transition Cov.	State Machine Benchmark
[117]	2006	Reinforcement	Markov Decision Process	N/A	Reward (State Coverage)	State Coverage	Recycling Robot
[110]	2021	Semi-supervised	Generative Adversarial Network, Convolutional NN	Image Input	Regression (Speed)	Faults Detected	Autonomous Vehicles
[111]	2018	Semi-supervised	Generative Adversarial Network	Image Input	Regression (Steering Angle)	Input Validity, Faults Detected	Autonomous Vehicles
[100]	1993	Supervised	Not Specified	System Executions	Regression (Output)	Not Evaluated	N/A
[91]	2018	Supervised	Backpropagation NN	System Executions	Regression (Output)	Output Coverage	Train Controller
[106]	2021	Supervised	Gaussian Process, Decision Trees, AdaBoostedTree, Random Forest, Support Vector Machine, Artificial NN	System Executions	Regression (Output)	Accuracy	Power Grid Control
[118]	2019	Supervised	Long Short-Term Memory NN	Existing Inputs	Regression (Valid Input)	Accuracy, Code Coverage	FTP Programs
[107]	2019	Supervised	Conditional Random Fields	Test Descriptions	Regression (Requirement Associations)	Accuracy	Telecom Systems
[105]	2019	Supervised	Long Short-Term Memory NN	Existing Inputs	Regression (Failing Input)	Faults Detected, Efficiency	Smart TV
[109]	2021	Supervised	Parallel Distributed Processing	System Executions	Regression (Output)	Efficiency, Faults Detected, Model Size	Autonomous Vehicles
[119]	2021	Supervised	Multilayer Perceptron	System Executions	Classification (Input Validity)	Accuracy	REST APIs (GitHub, LanguageTool, Stripe, Yelp, YouTube)
[102]	2022	Supervised	Regression Tree, Feedforward NN	System Executions	Regression (Output)	Efficiency, Faults Detected	Numeric Functions
[18]	2020	Supervised	Long Short-Term Memory NN	Simulink models	Regression (Validity Rules)	Input Validity, Faults Detected	Simulink tools
[108]	2021	Supervised	Conditional Random Fields	Specifications	Regression (Requirement Associations)	Accuracy	Unspecified
[104]	2020	Supervised, Unsupervised	Decision Trees, Gradient Boosting, K-Nearest Neighbor, MeanShift	System Executions	Regression (Validity Rules), Clustering (Covered Input)	Num. Clusters, Accuracy, Event Coverage	Bus System,Supply Chain
[120]	2007	Supervised	Backpropagation NN	System Executions	Regression (Output)	Accuracy, Efficiency	Fault Tolerant System, Arc Length
[121]	2022	Supervised	Shallow NN	RNG Seeds, System Executions	Regression (Prob. Traffic Violation)	Adaptivity, Faults Detected, Sensitivity	Autonomous Vehicles
[122]	2019	Supervised	Support Vector Machine	Existing Inputs	Regression (Validity Rules)	Tests Generated, Tests Executed, Test Size, Faults Detected	Domain-Specific Compiler

For compilers, an input is a full program, resulting in a large space of inputs. Zhu et al. restrict the range of inputs to avoid wasted effort [122]. They focus on domain-specific compilers and generate input appropriate for those domains. They extract features from the code, such as number of loops or matrix operations, then train a model to predict whether a new test case belongs to that domain. Test cases not belonging are discarded. Protocols require textual input that conforms to a specified format. Often, determining conformance requires manual construction of a grammar. Gao et al. generate protocol test input without a pre-defined grammar [118]. Their model learns the probability distributions of every character of a message, enabling the generation of new valid text sequences.

Input Generation (Reinforcement Learning): Araiza-Illan et al. [112], Huurman et al. [85], Shu et al. [116], and Veanes et al. [117] use reinforcement learning to generate input to cover states or transitions of a model. Araiza-Illan et al. generate input for robots [112]. The agent explores the robot's environment, using coverage of plan models as the reward function. Huurman et al. model APIs as stateful systems—where requests trigger transitions—and generate API calls intended to cover all transitions in the model [85]. Shu et al. [116] and Veanes et al. [117] use reinforcement learning to choose input actions for state-based system models. These tests can then be applied to the real system.

Baumann et al. use reinforcement learning to select input for autonomous driving that violates critical requirements [99]. The reward function encapsulates headway time, time-to-collision, and required longitudinal acceleration to avoid a collision. Reddy et al. use reinforcement learning to generate valid complex inputs (e.g., structured documents) [115]. The reward function favors both unique and valid input. As uniqueness depends on previously-generated input, this is not a problem that can easily be solved with supervised learning.

Enhancing Test Generation: Rather than fully replacing existing test generation methods, ML can also be used to improve their efficiency or effectiveness. A common target for improvement are Genetic Algorithms. A Genetic Algorithm is a search-based method that generates test cases intended to maximize or minimize a fitness function—a domain-specific scoring function, like the reward function in reinforcement learning.

Buzdalov and Buzdalova use reinforcement learning to modify the fitness function, adding and tuning sub-objectives that assist in optimizing the core objective of the search [113]. Zhao and Lv replace the fitness function with a model that predicts which input will cover unseen output behaviors [120]. Liu et al., Mishra et al., and Shihao ey al. also replace the fitness function, training a model to predict which code will be covered by input [125–127]. These models would be used when there is no tool support to measure coverage, or in cases where measuring coverage would be expensive.

Esnaashari and Damia use reinforcement learning to manipulate tests within the population generated by the Genetic Algorithm by modifying their input [114]. Paduraru et al. similarly use reinforcement learning to improve the effectiveness of a random testing tool by taking generated input and modifying it to raise its coverage or execution path length [124]. Zhong et al. bias input selection in a fuzzing tool for autonomous driving simulators by learning which seeds for the random number generator are more likely to lead to traffic violations in the simulation [121]. Sharma et al. replace random input generation in property-based testing with a model inferred from system executions [102]. The model is submitted, along with properties of interest, to a SMT solver to find input potentially violating the properties.

Table 4.4: Publications under **System Test Generation (White Box)** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate. NN = Neural Network.

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[123]	2021	Reinforcement	ReLU Q-Learning	Constraints	Reward (Solving Cost)	Code Coverage, Queries Solved	GNU coreutils
[124]	2021	Reinforcement	Deep Q-Network	N/A	Reward (Code Coverage, Path Length)	Code Coverage	Sorting
[103]	2022	Supervised	Artificial NN	Game States	Regression (Input Action)	Code Coverage, Mutation Score	Scratch Games
[125]	2022	Supervised	Radial-Basis Function NN	Existing Inputs, Code Coverage	Regression (Code Coverge)	Code Coverage	Numeric Functions
[16]	2021	Supervised	Long Short-Term Memory NN, Tree-LSTM, K-Nearest Neighbour	Constraints	Regression (Solving Time)	Accuracy, Constraint Solving Time	GNU coreutils, Busybox utils, SMT-COMP
[95]	2014	Supervised	Backpropagation NN	Existing Inputs, Code Coverage	Regression (Code Coverage)	Code Coverage	Binary Search, Sorting, Median, GCD, Triangle Class.
[126]	2011	Supervised	Backpropagation NN	Existing Inputs, Code Coverage	Regression (Code Coverage)	Code Coverage	Triangle Classification
[127]	2022	Supervised	Backpropagation NN	Existing Inputs, Code Coverage	Regression (Code Coverage)	Code Coverage, Efficiency	Numeric Functions

Mirabella et al. train a model to predict input validity, allowing a generation framework to filter invalid input before applying it [119]. Luo et al. [16] and Chen et al. [123] both enhance constraint solving in symbolic execution. Normally, a fixed timeout is used. Luo et al. instead train a model using multiple methods to predict the time needed to solve a constraint [16]. Chen et al. use reinforcement learning to identify the optimal solving strategy for a constraint [123].

4.4.2.2 GUI Test Generation

Table 4.5 details the 24 GUI testing publications. GUI test generation often focuses on a state-based interface model that formulates display changes as transitions taken following input. Fifteen publications generated input covering this model.

Almost all publications adopted reinforcement learning, as it can learn from feedback after applying an action to the GUI, and many GUIs require a sequence of actions to access certain elements. The main difference between publications lies in the reward function. Many base the reward on coverage of the states of the interface model (e.g., [128]), while incorporating additional information to bias state selection. Additional factors include magnitude of the state change [129], usage specifications [130, 131], unique code functions called [132], curiosity—favoring exploration of new elements [133–135]—coverage of interaction methods (e.g. click, drag) [136], validity of the resulting state [134], and avoidance of navigation loops [137].

Rather than state coverage, Koroglu and Sen base reward on finding violations of specifications [131]. Ariyurek et al. also apply reinforcement learning to select input for grid-based 2D games [96]. The game state is represented as a graph, and "test goals" are synthesized from the graph. The reward emphasizes test goal coverage. Li et al. use supervised learning, training a model to mimic patterns from interaction logs [92]. Their model associates GUI elements with a probability of usage—using probabilities to bias action selection. Kamal et al. filter redundant test cases as part of enhancing a search-based test generation framework [138]. Their model associates input and output, then uses the predicted output to decide if tests are redundant.

Rather than state coverage, Koroglu and Sen base reward on finding violations of specifications [131]. Ariyurek et al. also apply reinforcement learning to select input for grid-based 2D games [96]. The game state is represented as a graph, and "test goals" are synthesized from the graph. The reward emphasizes test goal coverage. Li et al. use supervised learning, training a model to mimic patterns from interaction logs [92]. Their model associates GUI elements with a probability of usage—using probabilities to bias action selection. Kamal et al. filter redundant test cases as part of enhancing a search-based test generation framework [138]. Their model associates input and output, then uses the predicted output to decide if tests are redundant.

Santiago et al. use supervised learning to generate sequences of interactions [139, 140]. They trained using human-written interaction sequences spanning several web pages. Their second study extends the approach to interact with forms by extracting feedback messages from the forms [140]. The framework learns constraints for form input, and a constraint solver creates input that meets those constraints. A model is used to classify page components. This helps control how different component types are processed. Their approach requires a complex training phase and a large human-created dataset. However, models can be used for multiple websites, decreasing the training burden.

Khaliq et al. employ multiple forms of supervised learning to generate test cases for Android apps [141] and web apps [142] They use an object recognition model to detect UI elements, then use extracted data from the UI elements to prompt a transformer model for a natural language test description. They then use a parser to translate this description into an executable test case. Similarly, Yazdani et al. have trained a model to identify the UI elements relevant to a particular input action, and use the identified elements to focus random test generation for Android apps [143]. They also demonstrated that the model can be effectively transferred to web apps.

Zheng et al. use both search-based test generation and a deep reinforcement learning technique to generate input for games [97]. Test input is generated by the search algorithm. However, reinforcement learning is used to select policies that control the generation framework.

4.4.2.3 Unit Test Generation

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[144]	2018	Reinforcement	Q-Learning	N/A	Reward (State Cov.)	State Coverage	F-Droid
[96]	2021	Reinforcement	Monte Carlo Tree Search, Sarsa	N/A	Reward (Test Goal Coverage)	Faults Detected	2D Games
[145]	2021	Reinforcement	Q-Learning	N/A	Reward (State Cov.)	Qualitative	Resource Planning
[137]	2013	Reinforcement	Own Technique	N/A	Reward (State Coverage, Loop Interactions)	State Coverage	F-Droid
[129]	2021	Reinforcement	Deep Q-Network	N/A	Reward (State Change Magnitude)	Code Coverage, Faults Detected	F-Droid
[136]	2019	Reinforcement	Q-Learning	N/A	Reward (State Cov., Element Interaction)	State Coverage	F-Droid
[146]	2022	Reinforcement	Sarsa	N/A	Reward (Event Cov.)	Code Coverage	F-Droid
[131]	2020	Reinforcement	Double Q-Learning	N/A	Reward (State Cov., Specifications)	State Coverage	F-Droid
[131]	2021	Reinforcement	Double Q-Learning	N/A	Reward (Specifications)	Faults Detected	F-Droid
[130]	2018	Reinforcement	Q-Learning	N/A	Reward (State Cov., Specifications)	State Coverage	F-Droid
[132]	2012	Reinforcement	Q-Learning	N/A	Reward (State Cov., Calls)	State Coverage	Password Manager, PDF Reader, Task List, Budgeting
[133]	2020	Reinforcement	Q-Learning + Long Short-Term Memory	N/A	Reward (State Cov., Curiosity)	State Coverage	F-Droid, Other Android Apps
[134]	2022	Reinforcement	Q-Learning	N/A	Reward (Curiosity, Validity of Resulting State)	Code Coverage, Faults Detected, Input Diversity, State Coverage	Web Apps
[147]	2018	Reinforcement	Q-Learning	N/A	Reward (State Cov.)	State Coverage	Android Apps
[128]	2021	Reinforcement	Q-Learning	N/A	Reward (State Cov.)	Code Coverage, Faults Detected	F-Droid
[135]	2021	Reinforcement	Q-Learning	N/A	Reward (State Cov., Curiosity)	Code Coverage, Faults Detected, Scalability	Web Apps (Research, Real-World, Industrial)
[97]	2019	Reinforcement	Advantage Actor-Critic	N/A	Reward (Game-Specific)	Faults Detected, State Coverage, Code Coverage	Games
[138]	2019	Supervised	Feedforward NN	Generated Inputs	Regression (Output)	State Coverage	Login Web App
[141]	2022	Supervised	Residual NN, Transformer	UI Screenshots, Natural Language	Classification (UI Elements), Regression (Natural Language Test)	Accuracy, Flakiness, Input Validity	Android Apps
[142]	2022	Supervised	Residual NN, Transformer	UI Screenshots, Natural Language	Classification (UI Elements), Regression (Natural Language Test)	Accuracy, Efficiency, Flakiness, Input Validity	Web Apps
[92]	2019	Supervised	Deep NN	System Executions	Regression (Action Probability)	State Coverage	Android Apps
[139]	2018	Supervised	Recurrent NN	Existing Inputs	Regression (Test Flows)	State Coverage	Unspecified Web App
[140]	2019	Supervised	Random Forest	Web Pages	Classification (Page Elements)	Mutation Score	Task List, Job Recruiting Web Apps
[143]	2021	Supervised	UNet	Screenshots	Regression (Relevant Screen Areas)	Adaptivity, Code Coverage	Androtest, Web Apps

Table 4.5: Publications under **GUI Test Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate. NN = Neural Network.

Because unit testing focuses on individual classes—making domain concerns less applicable—the majority of publications in Table 4.6 are “White Box” approaches and are not tied to particular system types.

Groce [149] and Kim et al. [88] use reinforcement learning to generate input, with code coverage as the reward. Groce applies reinforcement learning to generate input directly [149]. In contrast, Kim et al. use reinforcement learning to generate

Ref	Year	Test Gen. Approach	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[17]	2017	Black	Supervised	Query Strategy Framework	System Executions	Regression (Output)	Mutation Score	Math Library, Time Library
[148]	2018	Black	Unsupervised	Backpropagation NN	Existing Inputs	Clustering (Input Similarity)	Not Evaluated	N/A
[24]	2020	White	Reinforcement	Upper Confidence Bound, Differential Semi-Gradient Sarsa	N/A	Reward (Input Diversity)	Input Diversity, Faults Detected	JSON Parser
[19]	2020	White	Reinforcement	Upper Confidence Bound, Differential Semi-Gradient Sarsa	N/A	Reward (Num. Exceptions)	Num. Exceptions, Faults Detected	Defects4J
[20]	2022	White	Reinforcement	Upper Confidence Bound, Differential Semi-Gradient Sarsa	N/A	Reward (Num. Exceptions, Input Diversity, Strong Mutation)	Num. Exceptions, Input Diversity, Mutation Score, Faults Detected	Defects4J
[149]	2011	White	Reinforcement	Not Specified	N/A	Reward (Code Coverage)	Code Coverage	Data Structures
[90]	2015	White	Reinforcement	Q-Learning	N/A	Reward (Code Coverage)	Code Coverage	Data Structures, Collection Library, Primitives Library, Java/XML Parsers
[88]	2018	White	Reinforcement	Double Deep Q-Network	N/A	Reward (Code Coverage)	Code Coverage, Efficiency	GCD, EXP, Remainder
[150]	2022	White	Supervised	Naive Bayes, Random Forest, SVM, J48	Existing Inputs	Classification (Code Coverage)	Accuracy, Code Coverage, Mutation Score, Efficiency	Numeric Functions, Wheel Brake, Rendering, Mine Control, Notification, XML Parser, Siemens Benchmark
[151]	2021	White	Supervised	Gradient Boosting	Code Metrics	Classification (Fault Prediction)	Accuracy, Faults Detected	Compression, Imaging Library, Math Library, NLP, String Library
[152]	2019	White	Supervised	Backpropagation NN	Existing Inputs	Regression (Code Coverage)	Not Evaluated	N/A

Table 4.6: Publications under **Unit Test Generation** with publication date, generation approach, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate. NN = Neural Network.

optimization-based input generation algorithms [88]. The agent manipulates heuristics controlling the search algorithms.

Walkinshaw and Fraser use a supervised approach to generate input for system parts that have only been weakly tested [17]. A model is trained to predict the output. The model will have more confidence in prediction accuracy for input similar to the training data. Input with low certainty is retained, as they are likely to test parts of the system ignored in the training data. These inputs can later be used to re-train the model, shifting focus to other parts of the system.

Many authors use ML to enhance existing test generation approaches—often based on Genetic Algorithms. Almulla and Gay use reinforcement learning to select which fitness functions will be optimized in service of a higher-level testing goal [19, 20, 24]. For example, the agent can learn which combinations of fitness functions best trigger exceptions [19], increase input diversity [24], or increase Strong Mutation Coverage [20]. He et al. use reinforcement learning to improve coverage of private and inherited methods by augmenting generated tests [90]. The agent can make two types of changes—it can replace a method call with one whose return type is a subclass of the original method's, and it can replace a call to a public method with a call to a method that calls a private method. The reward is focused on private method coverage. Chen et al. employ supervised learning to improve the effectiveness of random generation and concolic testing [150]. They generate classification models for particular branches in the code, and use predictions about whether a test will cover a branch to ease the constraint-solving process.

Hershkovich et al. predict whether a class is likely to be faulty [151]. This can improve generation efficiency by determining which classes to target. They train a model—using an ensemble of methods—using source code metrics, labeled on whether a class had faults. Ji et al. use supervised learning to replace a fitness evaluation in a Genetic Algorithm [152]. They focus on data-flow coverage, which is very expensive to calculate. The model replaces the need to actually measure coverage. Hooda et al. train a model to cluster test cases [148]. When new tests are generated, those too close to a cluster centroid are rejected, improving generation efficiency.

4.4.2.4 Performance Test Generation

Performance test generation refers to the generation of test cases for the purpose of assessing whether the SUT meets non-functional requirements, such as speed, response time, scalability, or resource usage requirements [98]. Such tests are often generated to identify and eliminate performance bottlenecks. Table 4.7 details the performance testing publications. Performance can be measured, which offers feedback for subsequent rounds of generation. Thus, the majority of approaches are based on iterative processes, including reinforcement [98, 153–155], rule [156], and adversarial learning [157].

Ahmad et al. generate input intended to expose performance bottlenecks, with reward based on maximized execution time [98]. They note room for improvement by integrating other performance indicators into the reward. Rather than generating explicit program input, Moghadam et al. apply reinforcement learning to control the execution environment [154, 155, 158]. They identify resource configurations (CPU, memory, disk) where timing requirements are violated, with reward based on response time deviation and resource usage. These environmental factors constitute "implicit" input that can change the behavior of the SUT.

Sedaghatbaf et al. generate input violating performance requirements using two competing neural networks [157]. The generator produces input, and the discriminator classifies whether input violates requirements. This feedback improves the generator. Chen et al. also employ adversarial learning to generate input for resource-constrained neural networks intended to expose performance bottlenecks for such networks [159].

Luo et al. use the RIPPER rule learner to identify input classes that trigger intensive computations [156]. When tests are executed, executions are clustered based on execution time. RIPPER learns and iteratively refines rules differentiating the clusters, which are then used to generate new input.

Schulz et al. generate workloads for load testing [160]. The model generates realistic load levels on a system at various times and scenarios. Past session logs are clustered, and a multivariate time series is applied to predict system load during a scenario. Finally, Koo et al. use reinforcement learning to improve symbolic execution during stress testing [153]. They identify input that triggers worst-case execution time, defined as inputs that trigger a long execution path. The agent controls the exploration policy used by symbolic execution so that long paths are favored. The reward is based on path length and the feasibility of generating input for that path.

4.4.2.5 Combinatorial Interaction Testing

Table 4.8 shows the publications that use ML as part of Combinatorial Interaction Testing (CIT). Mudarakola et al. [161, 162] and Patil and Prakash [163] use neural networks to generate covering arrays—minimal sets of tests that cover all pairwise interactions between input variables. Patil and Prakash [163] predict the interactions covered by an input. They use this model to identify a covering array. Mudarakola et al [162] map each hidden layer of a neural network to a variable and each node to a value class. The values are connected by their connection to other variables. They do not use the network for prediction, but as a structuring mechanism to generate a covering array. Code coverage is used to prune redundant test cases. In a follow-up study [161], they manually construct a network using requirements, linking outputs to input values, with each input node mapping to an input variable, hidden layers linked to conditions from the requirements, and output nodes linked to predicted SUT output. The network again provides structure—a covering array is generated based on paths through the network.

Jia et al. use reinforcement learning to tune the generation strategy of a search-based generation framework using Simulated Annealing [89]. The agent selects how Simulated Annealing mutates a covering array. The reward is based on the change in coverage of combinations after imposing a mutation. Their framework recognizes and exploits policies that improve coverage.

CIT assumes that input values are divided into classes. Division is generally done manually, but identifying divisions is non-trivial. Duy Nguyen and Tonella use clustering to identify value classes, based on executed code lines (and how many times lines were executed) [164].

4.4.2.6 Test Oracle Generation

Tables 4.9-4.11 summarize the 42 oracle generation publications. Almost all approaches adopt supervised learning. These approaches train models, which stand in for traditional oracles, using previous system executions, screenshots, or metadata

Table 4.7: Publications under **Performance Test Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate. NN = Neural Network.

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[98]	2019	Reinforcement	Dueling Deep Q-Network	N/A	Reward (Execution Time)	Identified Bottlenecks	Auction Website
[155]	2019	Reinforcement	Q-Learning	N/A	Reward (Response Time Deviation)	Not Evaluated	N/A
[153]	2019	Reinforcement	Q-Learning	N/A	Reward (Path Length, Feasibility)	Paths Explored, Efficiency	Biological Computation, Parser, Sorting, Data Structures
[154]	2019	Reinforcement	Q-Learning	N/A	Reward (Response Time Deviation)	Not Evaluated	N/A
[158]	2022	Reinforcement	Q-Learning + Fuzzy Logic	N/A	Reward (Response Time Deviation, Resource Usage)	Efficiency, Adaptivity	Resource-Sensitive Programs (e.g., compression)
[159]	2022	Semi-Supervised	Generative Adversarial Network	System Executions	Regression (Performance), Classification (Input Distribution)	Identified Bottlenecks, Efficiency, State Coverage, Model Sensitivity, Input Quality	Object Recognition
[157]	2021	Semi-Supervised	Conditional Generative Adversarial Network	System Executions	Regression (Perf. Requirements), Classification (Test Realism)	Identified Bottlenecks, Accuracy, Labelling and Training Effort	Auction Website
[156]	2016	Supervised	RIPPER	System Executions	Regression (Rule Learning)	Identified Bottlenecks	Insurance, Online Stores, Project Management
[160]	2021	Supervised	Multivariate Time Series	Session Logs	Regression (Load)	Accuracy	Student Information

Table 4.8: Publications under **Combinatorial Interaction Testing** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate. NN = Neural Network.

Ref	Year	ML Approach	Technique	Training Data	e ML Objective	Evaluation Metrics	Evaluated On
[89]	2015	Reinforcement	SOFTMAX	N/A	Reward (Input Combinations)	Covering Array Size, Efficiency	Misc. Synthetic, Real Systems
[161]	2018	Supervised	Artificial NN	Specifications	Other (Structure Input Space), Regression (Output)	Covering Array Size	Temperature Monitoring
[162]	2014	Supervised	Artificial NN	Pairwise Input Combinations	Other (Structure Input Space)	Covering Array Size	Web Apps
[163]	2018	Supervised	Artificial NN	Pairwise Input Combinations	Regression (Input Coverage)	Covering Array Size, Efficiency	Unspecified
[164]	2013	Unsupervised	Expectation-Maximization	System Executions	Clustering (Code Coverage)	Qualitative Analysis	Bubble Sort, Math Functions, HTTP Processing, Banking

about source code features. The model predicts the correctness of output or properties of the expected output.

Test Verdicts: The majority of studies employ neural networks to train models that directly predict whether a test should pass or fail [37–39, 83, 165–167]. Most are simple, traditional neural networks for simple programs. However, Ibrahimzada et al. and Tsimpourlas et al. have recently explored how deep learning can train models for complex programs [83, 165, 167]. Braga et al. also are able to generate models for a complex application using an ensemble technique [36].

Test Verdicts: The majority of studies employ neural networks to train models that directly predict whether a test should pass or fail [37–39, 83, 165–167]. Most are simple, traditional neural networks for simple programs. However, Ibrahimzada et al. and Tsimpourlas et al. have recently explored how deep learning can train models for complex programs [83, 165, 167]. Braga et al. also are able to generate models for a complex application using an ensemble technique [36].

Chen et al. train a model to identify rendering errors in video games by training on screenshots of previous faults [168].

Table 4.9: Publications under **Test Verdicts Test Oracle Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate. NN = Neural Network.

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metric	Evaluated On
[36]	2018	Supervised	Adaptive Boosting	System Executions	Classification (Verdict)	Mutation Score	Shopping Cart
[168]	2021	Supervised	Convolutional NN	Screenshots	Classification (Verdict)	Accuracy, Faults Detected	Games (Android, iOS)
[37]	2018	Supervised	Backpropagation NN	System Executions	Classification (Verdict)	Mutation Score	Embedded Software
[165]	2022	Supervised	Recurrent NN	Source/Test Code, System Executions	Classification (Verdict)	Accuracy, Efficiency, Faults Detected, Mutation Score	Defects4J
[166]	2022	Supervised	Artificial NN	System Executions	Classification (Verdict)	Correct Classifications	Not Specified
[169]	2018	Supervised	L*	System Executions	Classification (Verdict)	Faults Detected, Efficiency	Platoon Simulator
[170]	2017	Supervised	Not Specified	System Executions	Classification (Verdict)	Faults Detected	Automotive Applications
[38]	2016	Supervised	Multilayer Perceptron	System Executions	Classification (Verdict)	Accuracy	User Creation
[171]	2022	Supervised	Convolutional NN, Multilayer Perceptron	Screenshots	Regression(Deviation from Correctness)	Accuracy	Augmented Reality Apps
[39]	2010	Supervised	Backpropagation NN	System Executions	Classification (Verdict)	Mutation Score	Student Registration
[167]	2021	Supervised	Multilayer Perceptron, Long Short-Term Memory NN	System Executions	Classification (Verdict)	Accuracy, Training Data Size	Blockchain Module, Deep Learning Module, Encryption Library, Stream Editor
[83]	2022	Supervised	Multilayer Perceptron, Long Short-Term Memory NN	System Executions	Classification (Verdict)	Accuracy, Adaptivity, Training Data Size	Blockchain Module, Deep Learning Module, Encryption Library, Network Protocols, Stream Editor, String Library

Rafi et al. apply a similar process to identify object-placement errors in augmented reality apps [171]. They do not predict a concrete pass/fail verdict, as users may perceive object placement differently. Instead, when labelling data, they asked multiple humans to offer verdicts, then labeled examples with the percentage that responded with a pass verdict. The model, then, predicts the percentage of users that would see a placement as correct in a new screenshot.

Khosrowjerdi et al. combine supervised learning and model checking [170]. A model is learned from system executions that predicts output. Given the model and specifications, a model checker assesses whether each specification is met, yielding a verdict. For each violation, a test is generated that can be executed to confirm the fault. If the fault is not real, the test and its outcome can be used to retrain the model. In a follow-up study [169], they demonstrate their technique on systems-of-systems.

Expected Output: The approaches generally train on system executions, and then predict the specific output expected for a new input. Output is often abstracted to representative values or limited to functions with enumerated values, rather than specific output. For example, a common application is "triangle classification"—a classification of a triangle as scalene, isosceles, equilateral, or not-a-triangle. This function is often used as an initial demonstration for test generation algorithms because it has branching behavior. Because it has limited outputs, it is also a common target for demonstrating the potential of oracle generation. Zhang et al. model a function that judges whether an integer is prime—a binary classification problem [51]. Many others also generate oracles for applications with a limited range of output [41, 45–48, 176]. However, some authors have generated oracles for functions with unconstrained—e.g., integer—output [43, 50, 172, 174, 175, 177].

The majority of approaches used some form of neural network [40, 42–47, 49–51, 95, 176, 177]. Ding and Zhang [41] also used label propagation—a technique where labeled and unlabeled training data are used, and the algorithm propagates labels to similar, unlabeled data—to reduce the quantity of labeling to create the training data.

Recently, Dinella et al. [173] and Yu et al. [178] demonstrated the use of language-generating transformer models for test oracle creation. Rather than inferring a model from system executions, a model is trained instead on source and test code, then given the code-under test and/or a partial unit test, the model directly produces assertions predicted to be appropriate for the prompt. Such models are trained on large datasets of code from many projects, and can potentially be applied generally.

Metamorphic and Properties: Several publications build on the research of Kanewala and Bieman [53], whose approach (a) converts code into control-flow graphs, (b) selects code elements as features for a data set, and (c), trains a model that predicts whether a feature exhibits a particular metamorphic relation from a list. This requires training data where features are labeled with a classification based on whether or not they exhibit a particular relation. Kanewala et al. extended this work by adding a graph kernel [54]. Hardin and Kanawala adapted this approach for label propagation [34] . Zhang et al. extended the approach to a multi-label classification that can handle multiple metamorphic relations at once [56]. Finally, Nair et al. demonstrated how data augmentation can enlarge the training dataset using mutants as the source of additional training data [55].

Korkmaz and Yilmaz predict the conditions on screen transitions in a GUI [181]. Their model is trained using past system execution and potential guard conditions. Shu and Lee use supervised learning to assess security properties of protocols [182]. A protocol is specified using a state machine, and message confidentiality is assessed on

Table 4.10: Publications under **Expected Output Test Oracle Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate. NN = Neural Network.

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metric	Evaluated On
[40]	2004	Supervised	Backpropagation NN	System Executions	Classification (Output)	Correct Classifications	Triangle Classification
[172]	2021	Supervised	Regression Tree, Support Vector Machine, Ensemble, RGP, Stepwise Regression	System Executions	Regression (Time)	Accuracy	Elevator
[173]	2022	Supervised	Transformer	Source/Test Code	Regression (Assertions)	Faults Detected, Accuracy	Defects4J
[41]	2016	Supervised	Support Vector Machine	System Executions	Classification (Output)	Mutation Score	Image Processing
[174]	2021	Supervised	Regression Tree, Support Vector Machine, Ensemble, TRGP, Stepwise Regression	System Executions	Regression (Time)	Accuracy	Elevator
[175]	2022	Supervised	Regression Tree, Support Vector Machine, Ensemble, RGP, Stepwise Regression	System Executions	Regression (Waiting, Execution Time)	Mutation Score, Accuracy	Elevator
[42]	2008	Supervised	Backpropagation NN	System Executions	Classification (Output)	Correct Classifications	Triangle Classification
[95]	2014	Supervised	Backpropagation NN	System Executions	Classification (Output)	Faults Detected	Static Analysis
[43]	2019	Supervised	Deep NN	System Executions	Regression (Output)	Mutation Score	Mathematical Functions
[44]	2011	Supervised	Radial-Basis Function NN	System Executions	Regression (Output)	Correct Classifications	Triangle Classification
[45]	2011	Supervised	Multilayer Perceptron	System Executions	Classification (Output)	Mutation Score	Insurance Application
[46]	2012	Supervised	Multilayer Perceptron	System Executions	Classification (Output)	Mutation Score	Insurance Application
[176]	2010	Supervised	Artificial NN	System Executions	Classification (Output)	Mutation Score, Accuracy, Precision, Correct Classifications	Student Registration
[47]	2016	Supervised	Backpropagation NN + Cascade	System Executions	Classification (Output)	Accuracy	Credit Analysis
[48]	2002	Supervised	Not Specified	System Executions	Classification (Output)	Mutation Score	Credit Analysis
[49]	2014	Supervised	Backpropagation NN, Decision Tree	System Executions	Classification (Output)	Mutation Score	Triangle Classification
[177]	2006	Supervised	Backpropagation NN	System Executions	Regression (Output)	Precision	Mathematical Functions
[50]	2006	Supervised	Multilayer Perceptron	System Executions	Regression (Output)	Mutation Score	Mathematical Functions
[178]	2022	Supervised	Transformer	Source/Test Code	Regression (Assertions)	Accuracy	Misc. Open-Source Projects
[51]	2019	Supervised	Probabilistic NN	System Executions	Classification (Output)	Correct Classifications	Prime, Triangle Class

Table 4.11: Publications under **Metamorphic (And Other Properties) Test Oracle Generation** with publication date, ML type, ML technique, training data, objective of the ML, evaluation metrics, and applications used to evaluate. NN = Neural Network.

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metric	Evaluated On
[52]	2020	Reinforcement	Not Specified	N/A	Reward (Relations)	Not Evaluated	Ocean Modeling
[179]	2021	Reinforcement	Not Specified	N/A	Reward (Relations)	Not Evaluated	Ocean Modeling
[180]	2020	Reinforcement	Contextual Bandit	N/A	Reward (Faults Detected)	Faults Detected	Object Detection
[34]	2018	Supervised	Support Vector Machine	Code Features	Classification (Property)	Accuracy	Misc. Functions
[53]	2013	Supervised	Support Vector Machine, Decision Trees	Code Features	Classification (Property)	Mutation Score	Misc. Functions
[54]	2016	Supervised	Support Vector Machine	Code Features	Classification (Property)	Mutation Score	Misc. Functions
[181]	2021	Supervised	Decision Trees	System Executions	Regression (Conditions)	Accuracy	Android Apps
[55]	2019	Supervised	Support Vector Machine	Code Features	Classification (Property)	ROC	Matrix Calculation
[182]	2007	Supervised	L*	System Executions	Classification (Violation)	Training Data Size	Handshake Protocols
[56]	2017	Supervised	Radial-Basis Function NN	Code Features	Classification (Property)	Accuracy	Misc. Functions

Type of Goal	Goal	# Pubs.	Publications
Generate Input	Maximize Coverage	32	[85, 96, 103, 112, 129, 136, 137, 141, 142, 144, 145, 149] [88, 92, 95, 130–133, 146, 161–163] [104, 116, 117, 128, 134, 135, 139, 140, 147]
	Expose Performance Bottlenecks	8	[98, 154–160]
	Show Conformance to (or Violation of) Specifications	7	[99, 107–111, 131]
	Generate Complex Inputs	4	[18, 115, 118, 122]
	Improve Input or Output Diversity	4	[17, 91, 100, 164]
	Predict Failing Input	2	[105, 106]
Generate Oracle	Predict Output	20	[40–46, 95, 172–175] [47–51, 176–178]
	Predict Test Verdict	12	[36–39, 83, 165–171]
	Predict Properties of Output	10	[34, 52–56, 179–182]
Enhance Existing Method	Improve Effectiveness	15	[19, 20, 24, 89, 90, 102, 113, 114, 124, 143, 150, 153] [97, 120, 121]
	Improve Efficiency	10	[16, 119, 123, 125–127, 138, 148, 151, 152]

Table 4.12: ML goals and the number of publications pursuing each goal.

message reachability. A model is inferred, then assessed for violations. If a violation is found, input is produced to check against the implementation. If the violation is false, the test helps retrain the model.

Hiremath et al. predict metamorphic relations for ocean modeling [52, 179]. The reinforcement learning approach poses relations, evaluates whether they hold, and attempts to minimize a cost function based on the validity of the set of proposed relations. Spieker and Gotlieb use reinforcement learning to *select* metamorphic relations from a superset of potentially-applicable relations [180]. Their approach evaluates whether selected relations can discover faults in an image classification algorithm.

4.4.3 RQ2: Goals of Applying ML

Table 4.12 lists the goals of authors in adopting ML, sorted into three broad categories. In the first two, ML is used directly to generate input or an oracle. As previously discussed, oracle generation uses ML to predict output, to properties of output, or a test verdict. Regarding input generation, the most common goal is to use ML to increase coverage of some criterion associated with effective testing. This includes coverage of code, states or transitions of models, or input interactions. Other uses of ML include generating input that exposes performance bottlenecks, demonstrates conformance to—or violation of—specifications, or increases input/output diversity. Others generate input for a complex data type or input likely to fail.

In the final category, ML tunes the performance or effectiveness of a generation framework—often search-based of Symbolic Execution-based approaches. To improve efficiency, ML clusters redundant tests, replaces expensive calculations with predictions, chooses generation targets, or checks input validity. To improve effectiveness, ML manipulates test cases (e.g., replaces method calls) or tunes the generation strategy (e.g., selects fitness functions, mutation heuristics, or timeouts).

RQ2 (Goal of ML): ML generates input (47%)—particularly to maximize some form of coverage—or oracles (33%)—particularly that predict an expected

output. It is also used to improve efficiency or effectiveness of existing generation methods (20%).

4.4.4 RQ3: Integration into Test Generation

RQ3 highlights where and how ML has been integrated into the testing process. This includes types of ML applied, training data, and how ML was used (regression, classification, reward functions).

Supervised techniques were the first applied to input and oracle generation, and remain the most common. Supervised techniques are—by far—the most common for oracle generation. They are also the most common for system and combinatorial interaction testing. The predictions made by models are either from pre-determined options (classification) or open (regression). Classification is often used in oracle generation, e.g., to produce a verdict (pass/fail) or output from a limited range. Regression is common in input generation, where complex predictions must be made.

Both training time and quantity of training data need to be accounted for when considering a supervised technique. After being trained, a model will not learn from new interactions, unlike with reinforcement learning. A model must be retrained with new training data to improve its accuracy. Therefore, it is important that supervised methods be supplied with sufficient quantity and quality of training data. Supervised techniques generally learn from past system executions, labeled with a measurement of interest. If the label can be automatically recorded, then gathering sufficient data is often not a major concern. However, if the SUT is computationally inefficient or information is not easily collectible (e.g., a human must label data), it can be difficult to use supervised ML.

Adversarial learning may help overcome data challenges. This strategy forces models to compete, creating a feedback loop where performance is improved without the need for human input. Multiple publications adopted adversarial networks, generally in cases where input was associated with a numeric quality (performance, vehicle speed—e.g., [110, 111]). Neither case requires human labeling, so models can be automatically retrained. Other recent deep learning approaches—often trained on many systems—show promise in their ability to adapt to unseen systems (e.g., [83, 121]).

Reinforcement learning is the second most common type of ML. Reinforcement learning was even used more often than supervised in 2020, and almost as often in 2021. Reinforcement learning has been used in all input generation problems and is the most common technique for GUI, unit, and performance generation.

Reinforcement learning is appealing because it does not require pre-training and automatically improves accuracy through interactions. Reinforcement learning is most applicable when effectiveness can be judged using a numeric metric, i.e., where a measurable assessment already exists. This includes performance measurements—e.g., resource usage—or code coverage. Reinforcement learning is also effective when the SUT has branching or stateful behavior—e.g., in GUI testing, where a *sequence* of input may be required. Similarly, performance bottlenecks often emerge as the consequence of a sequence of actions, and code coverage may require multiple setup steps. Reinforcement learning is effective in such situations because it can learn from the outcome of taking an action. Therefore, it is effective at constructing sequences of input steps that ultimately achieve some goal of interest. Many supervised approaches

are not equipped to learn from each individual action, and must attempt to predict the full sequence of steps at once.

Outside of individual tests, reinforcement learning is also effective at enhancing test generation algorithms. Genetic Algorithms, for example, evolve test suites over a series of subsequent generations. Reinforcement learning can tune aspects of this evolution, in some cases guided by feedback from the same fitness functions targeted by the optimization. If a test suite attains high fitness, reinforcement learning may be able to improve that score by manipulating the test cases of the algorithm parameters. Reinforcement learning can, of course, generate input effectively in a similar manner to an optimization algorithm. However, it also can often improve the algorithm such that it produces even better tests.

Authors of sampled publications applied unsupervised learning to cluster test cases to improve generation efficiency or to identify weakly tested areas of the SUT. While clustering has not been used often in the sampled publications, clustering is common in other testing practices (e.g., to identify tests to execute [6]). Therefore, it may have potential for use in filtering tasks during generation, especially to improve efficiency. Future work should further consider how clustering could be applied as part of test generation.

RQ3 (Integration of ML): The most common ML types are supervised (61%) and reinforcement learning (34%). Some publications also employ unsupervised (2%) or semi-supervised (3%) learning. (Semi-)Supervised learning is the most common ML for system testing, CIT, and all forms of oracle. Reinforcement learning is the most common technique for GUI, unit, and performance testing, and is used where testing goals often have measurable scores, a sequence of input is required, or existing generation tools can be tuned. Clustering was also used for filtering, e.g., discarding similar test cases.

4.4.5 RQ4: ML Techniques Applied

RQ4 examines specific ML techniques. Table 4.13 lists techniques employed, divided by ML type. Neural networks are the most common techniques in supervised learning. Support vector machines are also employed often, as are forms of decision trees.

In particular, backpropagation neural networks are used most (11%). Backpropagation neural networks are a classic technique where a network is composed of multiple layers [183]. In each layer, a weight value for each node is calculated. In such networks, information is fed *forward*—there are no cyclic connections to earlier layers. However, the backpropagation feature propagates error backward, allowing earlier nodes to adjust weights if necessary. This leads to less complexity and faster learning rates. In recent years, more complex neural networks have continued to implement backpropagation as one (of many) features.

Recently, neural networks utilizing Long Short-Term Memory have also become quite common. Unlike traditional feedforward neural networks, Long Short-Term Memory has feedback connections [184]. This creates loops in the network, allowing information to persist. This adaptation allows such networks to process not just single data points, but sequences where one data point depends on earlier points. Long Short-Term Memory networks and deep neural networks are likely to become more common in the next few years as more researchers adopt deep learning techniques.

Type	Family	Technique	# Pubs.
Supervised	Neural Networks	Backpropagation NN	14
		Multi-Layer Perceptron	8
		Artificial NN	7
		Long Short-Term Memory NN	6
		Transformer	4
		Radial-Basis Function NN	3
		Convolutional NN, Deep NN, Feedforward NN, Recurrent NN, Residual NN	2
		Backpropagation NN + Cascade, Probabilistic NN, Shallow NN, UNet	1
	Trees	Decision Tree	5
		Random Forest	3
		Gradient Boosting, Regression Tree	2
		Ada-Boosted Tree, C4.5, J48, Tree-LSTM	1
	Others	Support Vector Machine	11
		L*,Conditional Random Fields, Ensemble, K-Nearest Neighbors, Regression Gaussian Process,Stepwise Regression	2
		Adaptive Boosting, Gaussian Process, Multivariate Time Series, Naive Bayes, Parallel Distributed Processing, Query Strategy Framework, RIPPER	1
Reinforcement	Q-Learning	Q-Learning	16
		Deep Q-Network	3
		Double Q-Learning	2
		Delayed Q-Learning, Dueling Deep Q-Network, Double Deep Q-Network, Q-Learning + Fuzzy Logic, Q-Learning + Long Short-Term Memory,ReLU Q-Learning	1
	Others	Differential Semi-Gradient Sarsa, Upper Confidence Bound	3
		Sarsa	2
		Advantage Actor-Critic, Asynchronous Advantage Actor Critic, Contextual Bandit, Markov Decision Process, Monte Carlo Control, Monte Carlo Tree Search, SOFTMAX	1
Semi-Supervised		Generative Adversarial Network	3
		Convolutional NN, Conditional Generative Adversarial Network	1
Unsupervised		Backpropagation NN, Expectation-Maximization, MeanShift	1

Table 4.13: ML techniques adopted—divided by ML type and family of ML techniques—ordered by number of publications where the technique is adopted. NN = Neural Network.

The emergence of transformer models—complex neural networks that learn from, and generate, natural language [173]—is promising for both test and oracle generation. Transformers make use of a mechanism called "self-attention" that uses backpropagation to infer the relationship between words in a phrase [185]. This mechanism enables automated context-extraction and summarization of text, which in term enables the model to produce complex textual output as well.

Reinforcement learning is dominated by forms of Q-Learning—Q-Learning and its variants are used in 22% of publications. Q-Learning is a prototypical form of off-policy reinforcement learning, meaning that it can choose either to take an action guided by the current "best" policy—maximizing expected reward—or it can choose to take a random action to refine the policy [30]. Many other reinforcement learning techniques are also off-policy, and follow a similar process, with various differences (e.g., calculating reward or action decisions in a different manner).

Some authors have chosen specific techniques because they worked well in previ-

ous work (e.g., [54, 131]). Others saw certain techniques work on similar problems outside of test generation (e.g., [89]), or chose techniques thought to represent the state-of-the-art for a problem class (e.g., [105]). However, most authors do not justify their choice of technique, nor do they often compare alternatives.

In recent years, open-source ML frameworks have emerged that accelerate the pace and effectiveness of research by making robust algorithms available. The authors of 51 publications (41% of the sample) explicitly made use of existing frameworks. The most common ML frameworks used in the sampled publications include keras-rl (e.g, [88]), Matlab (e.g., [172]), OpenAI Gym (e.g., [85]), PyTorch (e.g., [180]), scikit-learn (e.g., [34]), TensorFlow (e.g., [91]), and WEKA (e.g., [156]). In the other 73 publications—especially older publications—authors either implemented ML algorithms or adapted unspecified implementations. The use of a framework constrains technique choice. However, all of these frameworks offer many techniques, and may allow researchers to compare results across techniques. This could lead to more informed and robust implementations.

RQ4 (ML Techniques): Neural networks, especially backpropagation neural networks, are the most common supervised techniques. Reinforcement learning is generally based on Q-Learning. Technique choice is often not explained, but may be inspired by insights from previous or related work, an algorithm having performed well on a similar problem, or algorithms available in open-source frameworks (e.g., OpenAI Gym or WEKA).

4.4.6 RQ5: Evaluation of the Test Generation Framework

RQ5 examines how authors have evaluated their work—in particular, how ML affects evaluation. The metrics adopted by the authors are listed in Table 4.14. We group similar metrics (e.g., coverage metrics, notions of fault detection, etc.). In most cases, these metrics are used to evaluate the quality of the input or oracle generation approach.

In most cases, the entire framework is evaluated. Almost all of these evaluations employ standard metrics for test generation. Some metrics are specific to a testing practice (e.g., covering array size) or aspect of generation (e.g., number of queries solved), while others are applied across testing practices (e.g., fault detection). Naturally—whether ML is incorporated or not—a generation framework must be evaluated on its effectiveness.

Many authors also evaluate the ML component separately. Supervised approaches were often evaluated using some notion of model accuracy—using various accuracy measurements, correct classification rate, and ROC. Approaches have also been evaluated on the quantity of required training data, whether a model can be applied to unknown systems, and the sensitivity of model predictions to small changes in the input or model parameters. In addition, one study used the size of the trained model to help explain the results of applying the technique, rather than using it to measure solution quality. Semi-supervised approaches were also evaluated using accuracy, the required labeling/training effort, and sensitivity. Finally, one study employing an unsupervised approach used the number of clusters produced to analyze the results of applying their approach.

Reinforcement learning approaches were generally not evaluated using ML-specific metrics, except for a study that examined their adaptivity and sensitivity.

This is reasonable, as reinforcement learning learns how to maximize a numeric function. The reward is based on the goals of the overall generation framework. Rather than evaluating using an absolute notion of accuracy, the success of reinforcement learning can be seen in improved reward measurements, attainment of a checklist of goals, or metrics such as fault detection.

RQ5 (Evaluation): The full generation framework is generally evaluated by traditional testing metrics (e.g., fault detection). However, the ML components are also evaluated—especially in supervised learning—using accuracy, adaptivity, quantity of training data needed, labeling/training effort, prediction sensitivity, and other ML metrics. Reinforcement learning is generally evaluated using testing metrics tied to the reward.

Type	Metric	# Pubs.
Supervised	**Prediction Accuracy (e.g., correct classifications, ROC)**	37
	Faults Detected (including mutants and performance issues)	33
	Efficiency (e.g., scalability, # tests generated/executed, time), Coverage Attained (e.g, code, state)	12
	Test Size (e.g., size of test cases, suite, or covering array)	4
	Adaptivity (whether a model can be transferred to a new system), Validity of Generated Inputs, **Quantity of Training Data Required**	3
	Flakiness of Generated Tests	2
	Input/Output Diversity, **Model Size, Sensitivity of Predictions**	1
Reinforcement	Coverage	25
	Faults Detected	13
	Efficiency	6
	Input/Output Diversity	4
	# Exceptions Discovered	2
	Adaptivity, Qualitative Analysis, # Queries Solved, # Requirements Met, **Sensitivity**, Test Size	1
Semi-Supervised	Faults Detected	4
	Prediction Accuracy, Coverage, Efficiency, Quality of Generated Inputs, Validity of Generated Inputs, **Required Labeling and Training Effort, Sensitivity of Predictions**	1
Unsupervised	**# Clusters Produced,** Qualitative Analysis	1

Table 4.14: Evaluation metrics adopted (similar metrics are grouped), divided by ML approach, and ordered by number of publications using each metric. Metrics in bold are related to ML.

4.4.7 RQ6: Limitations and Open Challenges

The sampled publications show great potential. However, we have observed multiple challenges that must be overcome to transition research into real-world use.

Volume, Contents, and Collection of Training Data: (Semi-)Supervised ML requires training data to create a model. There are multiple challenges related to the *required volume* of training data, the *required contents* of the training data, and *human effort* required to produce that training data.

Regardless of the testing practice addressed, the volume of required training data can be vast. This data is generally attained from labeled execution logs, which means that the SUT needs to be executed *many* times to gather the information needed to train the model. Approaches based on deep learning could produce highly accurate models but may require thousands of executions to gather required training data. Some approaches also must preprocess the collected data. While it may be possible to

automatically gather training data, the time required to produce the dataset can still be high and must be considered.

This is particularly true for cases where a regression is performed rather than a classification—e.g., an expected value oracle [172] or complex test input [18]. Producing a complex continuous value is more difficult than a simple classification, and requires significant training data—with a range of outcomes—to make accurate predictions.

In addition, the contents of the training data must be considered. If generating input, the training data must contain a wide range of input scenarios with diverse outcomes that reflect the specific problem of interest and its different branching possibilities. Consider code coverage prediction (e.g., [95, 152]). If one wishes to predict the input that will cover a particular element, then the training data must contain sufficient information content to describe how to cover that element. That requires a diverse training set.

Models based on output behavior—e.g., expected value oracles or models that predict input based on particular output values [91, 100, 101]—suffer from a related issue. The training data for expected value oracles must either come from passing test cases—that is, the output must be correct—or labels must be applied by humans. A small number of cases accidentally based on failing output may be acceptable if the algorithm is resilient to noise in the training data, but training on faulty code can result in an inaccurate model. This introduces a significant barrier to automating training by, e.g., generating input and simply recording the output that results.

Similarly, models that make predictions based on failures—e.g., test verdict oracles or models that produce input predicted to trigger a failure [105] or performance issue [156]—require training data that contains a large number of *failing test cases*. This implies that faults have already been discovered and, presumably, fixed before the model is trained. This introduces a paradox. There may be remaining failures to discover. However, the more training data that is needed, the less the need for—or impact of—the model.

In some cases, training data must be labeled (or even collected) by a human. Again, oracles suffer heavily from this problem. Test verdict oracles require training data where each entry is assigned a verdict. This requires either existing test oracles—reducing the need for a ML-based oracle—or human labeling of test results. Judging test results is time-consuming and can be erroneous as testers become fatigued [12], making it difficult to produce a significant volume of training data. Generation of metamorphic relation oracles requires overcoming a similar dilemma, where training data must be labeled based on whether a particular metamorphic relation holds. This requires labeling by a tester with significant knowledge of the source code.

For some problems, these issues can be avoided by employing reinforcement learning instead. Reinforcement learning will learn while interacting with the SUT. In cases where the effectiveness of ML can be measured automatically—e.g., code coverage and performance bottlenecks—reinforcement learning is a viable solution. However, cases where ground truth is required—e.g., oracles—are not as amenable to reinforcement learning. Reinforcement learning also requires many executions of the SUT, which can be an issue if the SUT is computationally expensive or otherwise difficult to execute and monitor, such as when specialized hardware is required for execution.

Otherwise, techniques are required that (1) can enhance training data, (2) can extrapolate from limited training data, and (3), can tolerate noise in the training

data. Means of generating synthetic training data, like in the work of Nair et al. [55], demonstrate the potential for data augmentation to help overcome this limitation. Adversarial learning also offers a way to improve the accuracy of a model—reducing the need for a large training dataset. Again, however, such approaches are of limited use in cases where human involvement is required. In addition, deep learning approaches—such as transformers—can often be trained on data from many different projects, potentially yielding models that are also effective on projects not in their training set (e.g., [178]).

RQ6 (Challenges): Supervised learning is limited by the required quantity, quality, and contents of training data—especially when human effort is required. Oracles particularly suffer from these issues. Reinforcement learning and adversarial learning are viable alternatives when data collection and labeling can be automated.

Retraining and Feedback: After training, models have a fixed error rate and do not learn from new mistakes made. If the training data is insufficient or inaccurate, the generated model will be inaccurate. The ability to improve the model based on additional feedback could help account for limitations in the initial training data.

There are two primary means to overcome this limitation—either retraining the model using an enriched training dataset or adopting a reinforcement learning approach that can adapt its expectations based on feedback. Both means carry challenges. Retraining requires (a) establishing a schedule for when to train the updated model, and (b), an active effort on the part of human testers to enrich and curate the training dataset. Adversarial learning offers an automated means to retrain the model. However, there are still limitations on when it can be applied.

Enriching the dataset—as well as the use of reinforcement learning—requires some kind of feedback mechanism to judge the effectiveness of the predictions made. This can be difficult in some cases, such as test oracles, where human feedback may be required. Human feedback, even on a subset of the decisions made, reduces the cost savings of automation.

RQ6 (Challenges): Models should be retrained over time. How often retraining occurs depends, partially, on the cost to gather and label additional data or on the amount of human feedback required.

Complexity of Studied Systems: Regardless of ML type, many of the proposed approaches are evaluated on highly simplistic systems. 44% of the publications evaluate using toy examples, with only a few lines of code or possible function outcomes. While it is intuitive to *start* with simplistic examples to examine the viability of an ML approach, the real-world application requires accurate predictions for complex functions and systems with many branching code paths. If a function is simple, there is likely little need for a predictive model in the first place. Several recent studies feature thorough evaluations of complex systems (e.g., [20, 123, 135]), even on industrial systems (e.g., [89, 119]). However, many studies evaluate on only a single example or a handful of examples, and many of those examples are still not

very complex. It largely remains to be seen whether many proposed techniques can be used on real-world production code.

The generation of models for arbitrary systems with unconstrained output may be prohibitively difficult even for sophisticated ML techniques. This is particularly the case for expected value oracles. In such cases, some abstraction should be expected—either a simplification of the core logic of the system or a partition of inputs or outputs into symbolic values. One possibility to consider is a variable level of abstraction—e.g., a training-time decision to cluster output predictions into an adjustable number of representative values (such as the centroids of clusters of outputs). Training could take place over different settings for this parameter, and the balance between accuracy and abstraction could be explored.

In any evaluation, a variety of systems should be considered. The complexity of the systems should vary. This enables the assessment of scalability of the proposed techniques. Researchers should examine how prediction accuracy, training data requirements (for supervised learning), and time to convergence on an optimal policy (for reinforcement learning) scale as the complexity of the system increases. This would enable a better understanding of the limitations and applicability of ML-based techniques in test generation for real-world systems.

RQ6 (Challenges): Scalability of ML techniques to real-world systems is not clear. When modeling complex functions, varying degrees of abstraction could be explored if techniques are unable to scale. In evaluations, a range of systems should be considered, and explicit analyses of scalability (e.g., accuracy, training, learning rate) should be performed.

Variety, Complexity, and Tuning of ML Techniques: Authors rarely explain or justify their choice of ML algorithm—often stating that an algorithm worked well previously or that it is "state-of-the-art", if any rationale is offered. It is even rarer that multiple algorithms are compared to determine which is best for a particular task. As the purpose of many research studies is to demonstrate the viability of an idea, the choice of algorithm is not always critically important. However, this choice still has implications, as it may give a false impression of the applicability of an approach and unnecessarily introduce a performance ceiling that could be overcome through the consideration of alternative techniques.

One reason for this limitation may be that testing researchers are generally ML *users*, not ML experts. They may lack the expertise to know which algorithms to apply. Collaboration with ML researchers may help overcome this challenge. The use of open-source ML frameworks can also ease this challenge by removing the need for researchers to develop their own algorithms. Rather than needing to understand each algorithm, they could instead compare the performance of available alternatives. This comparison would also lead to a richer evaluation and discussion.

Many of the proposed approaches—especially earlier ones—are based on simple neural networks with few layers. These techniques have strict limitations in the complexity of the data they can model and have been replaced by more sophisticated techniques. Deep learning, which may utilize many hidden layers, may be essential in making accurate predictions for complex systems. Few approaches to date have utilized deep learning, but such approaches are starting to appear, and we would expect

more to explore these techniques in the coming years. However, deep learning also introduces steep requirements on the training data that may limit its applicability.

Almost all of the proposed approaches utilize a single ML technique. An approach explored in many domains is an *ensemble* [106]. In such approaches, models are trained on the same data using a variety of techniques. Each model is asked for a prediction, and then the final prediction is based on the consensus of the ensemble. Ensembles are often able to reach stable, accurate conclusions in situations where a single model may be inaccurate. A small number of studies have applied ensembles [36, 106, 151, 172, 174], but such techniques are rare.

Many ML techniques have parameters that can be tuned (e.g., learning rate, number of hidden units, or activation function). Parameter tuning can significantly impact prediction accuracy and enable significant improvements in the results of even simple ML techniques. The sampled publications do not explore the impact of such tuning. This is an oversight that should be corrected in future work.

RQ6 (Challenges): Researchers rarely justify the choice of ML technique or compare alternatives. The use of open-source ML frameworks can ease comparison. Deep learning and ensemble techniques, as well as hyperparameter tuning, should also be explored more widely.

Lack of Standard Benchmarks: Research benchmarks have enabled sophisticated analyses and comparison of approaches for automated test generation. Such benchmarks usually contain a set of systems prepared for a particular type of evaluation. Bug benchmarks, in particular, contain real faults curated from a variety of systems, along with metadata on those faults. Such benchmarks ease comparison with past research, remove bias from system selection and demonstrate the effectiveness of techniques. Only a small subset of the sampled publications make use of existing research benchmarks. The most common, by far, is the F-Droid Android benchmark (e.g., [129, 136]). Others made use of examples commonly used in research such as the Defects4J (e.g., [20, 173]) or the RUBiS web app example (e.g., [157]). However, the majority of studies do not use benchmarks or open-source evaluation targets.

Some studies require their own particular evaluation. However, in cases where evaluation is over-simplistic, or where code or metadata is unavailable, this makes comparison and replication difficult. Benchmarks are typically tied to particular system types or testing practices. In cases where benchmarks exist—unit, web app, mobile app, and performance testing in particular—we would encourage researchers to use these benchmarks to enable comparison to past work or to allow researchers to make comparisons with their work.

In other cases, the creation of benchmarks specifically for ML-enhanced test generation research could advance the state-of-the-art in the field, spur new research advances, and enable replication and extension of proposed approaches. In particular, we recommend the creation of such a benchmark for oracle generation. Such a benchmark should contain a variety of code examples from multiple domains and of varying levels of complexity. Code examples should be paired with the metadata needed to support oracle generation. This would include sample test cases and human-created test oracles, at minimum. Such a benchmark could also include sample training data that could be augmented over time by researchers.

Lack of Replication Package or Open Code: A common dilemma is lack of access to research code and data. Often, a publication is not sufficient to allow replication or application in a new context. This applies to research in ML-enhanced test generation as well, as only 33% of the publications in our sample provided open-source code or replication packages.

Outside of this 33%, some publications made use of open-source ML frameworks. This is positive, in that the specific ML techniques are trustworthy and available. Potentially, experimental results could be replicated in such cases by applying the same techniques to the same settings. However, there still may not be enough information in the paper to enable replication, such as specific parameter settings. Further, these frameworks evolve over time, and the results may differ because the underlying ML technique has changed since the original study was published.

Researchers should include a replication package with their source code, execution scripts, and the versions of external dependencies used when the study was performed. This package should also include training data and the gathered experiment observations used by the authors in their analyses.

RQ6 (Challenges): Research is limited by the overuse of simplistic examples, the lack of common benchmarks, and the unavailability of code and data. Researchers should be encouraged to use available benchmarks, and provide replication packages and open code. New benchmarks could be created for ML challenges (e.g., oracle generation).

4.5 Threats to Valididy

External and Internal Validity: Our conclusions are based on the publications sampled. It is possible that we may have omitted important publications. This can affect internal validity—the evidence we use to make conclusions—and external validity—the generalizability of our findings. Secondary studies can be valuable even if they do not capture all publications from a research field as long as their selection protocol (search string, inclusion/exclusion criteria, snowballing) ensures an adequate sample to infer similar findings to a complete set of relevant publications. We believe that our selection strategy was appropriate. We tested different search strings and performed a validation exercise to test the robustness of our string. We have used four databases, covering the majority of relevant venues, and performed additional snowballing. Our final set of publications includes 124 primary publications, which we believe is sufficient to make informed conclusions.

Conclusion Validity: Subjective judgments are part of article selection, data extraction, and categorizing publications. To control for bias, protocols were discussed and agreed upon by both authors, and independent verification took place on—at least—a sample of all decisions made by either author.

Construct Validity: We used a set of properties to guide data extraction. These properties may have been incomplete or misleading. However, we have tried to establish properties that were informed by our research questions. These properties were iteratively refined, and we believe they have allowed us to thoroughly answer the questions.

4.6 Conclusions

Automated test generation is a well-studied research topic, but there are critical limitations to overcome. Recently, researchers have begun to use ML to enhance automated test generation. We have characterized emerging research on this topic through a systematic mapping study examining testing practices that have been addressed, the goals of using ML, how ML is integrated into the generation process, which specific ML techniques are applied, how the full test generation process is evaluated, and open research challenges.

We observed that ML generates input for system, GUI, unit, performance, and combinatorial testing or improves the performance of existing generation methods. ML is also used to generate test verdicts, property-based, and expected output oracles. Supervised learning—often based on neural networks—and reinforcement learning—often based on Q-learning—are common, and some publications also employ unsupervised or semi-supervised learning. (Semi-/Un-)Supervised approaches are evaluated using both traditional testing metrics and ML-related metrics (e.g., accuracy), while reinforcement learning is often evaluated using testing metrics tied to the reward function.

The work-to-date shows great promise, but there are open challenges regarding training data, retraining, scalability, evaluation complexity, ML algorithms employed—and how they are applied—benchmarks, and replicability. Our findings can serve as a roadmap for both researchers and practitioners interested in the use of ML as part of test generation.

4.7 Acknowledgments

This research was supported by Vetenskapsrådet grant 2019-05275.

Paper D

Exploring the Interaction of Code Coverage and Non-Coverage Objectives in Search-Based Test Generation

Afonso Fontes, Gregory Gay, Robert Feldt

In submission to Software Testing, Verification, and Reliability (STVR).

Abstract

Context: Search-based test generation typically targets structural coverage of source code. Past research suggests that targeting coverage alone is insufficient to yield tests that achieve common testing goals (e.g., discovering situations where a class-under-test throws exceptions) or detect faults. A suggested alternative is to perform multi-objective optimization targeting both coverage and additional objectives directly related to the goals of interest. However, it is not fully clear how coverage and goal-based objectives interact during the generation process and what effects this interaction will have on the generated test suites.
Objectives: We assess five hypotheses about multi-objective test generation and the relationships between coverage-based and goal-based objectives, focusing on the effects on coverage, goal attainment, fault detection, test suite size, test case length, and the impact of the search budget.
Methods: We generate test suites using the EvoSuite framework targeting Branch Coverage, three testing goals—Exception Count, Output Coverage, and Execution Time—and combinations of coverage and goal-based objectives.
Results: Targeting multiple objectives does not reduce code coverage and can—in some situations—increase goal attainment and detect more faults compared to single-target configurations. It also produces larger test suites, but test case length is not significantly increased. The benefits of multi-objective optimization are often more limited than hypothesized in past research, but are still sufficient to recommend multi-objective optimization over targeting coverage or testing goals alone.
Conclusion: Our study offers insights and guidance into how coverage and goal-based objectives interact during multi-objective test generation.

Keywords: Automated Test Generation, Search-Based Test Generation, Coverage Criteria, Adequacy Criteria, Branch Coverage

5.1 Introduction

Structural coverage criteria measure the percentage of the source code that has been executed according to a set of criterion-specific rules regarding (a) which code structures should be executed, and (b) how those structures should be executed [4, 9, 186]. Two of the most common criteria include Statement Coverage—which mandates that all code statements be executed, but places no constraints on how they are executed—and Branch Coverage—which mandates that all control-diverting statements (e.g., `if`, `case`, and loop conditions) evaluate to each of their possible outcomes [187].

Coverage measurement is a common advisory activity for testers [9]. The current percentage of coverage attained can serve as an approximation of "how much testing" has been conducted, and missed coverage goals can serve as the targets of additional test cases. Because the attainment of most coverage criteria can be automatically measured through program instrumentation and execution analysis, such criteria have also become the de facto basis of automated test generation—especially techniques such as search-based test generation, fuzzing, and symbolic or concolic execution [2, 7].

Consider, for example, search-based test generation. In search-based test generation, metaheuristic optimization algorithms sample from the space of possible test inputs to identify those that maximize or minimize *fitness functions*—numeric scoring functions representing properties of interest [2]. Coverage criteria serve as natural fitness functions, often associating each code structure of interest with a score representing how close an execution came to executing that structure in the manner prescribed by the criterion—e.g., how much x would need to change for the condition $(x == 0)$ to evaluate to `true` within a particular control structure [66].

Coverage-directed testing is ubiquitous in automated test generation because structural coverage is easy to measure, easy to translate into an optimization target, and is hypothesized to have a correlation to the probability of fault detection [187]. However, concerns have been raised about its use as the primary target of automated generation [4, 188, 189]. We have previously conducted large-scale case studies on coverage-directed test generation, focusing on model and search-based test generation [3, 4, 190]. These studies have yielded important observations about the efficiency and effectiveness of coverage-directed test generation—at least, in the manner it is generally employed.

First, we have observed that achieving structural coverage is a reasonable *starting point* for effective automated test generation. For example, we observed that coverage was the single strongest predictor of the likelihood of fault discovery [3]. That is, if we want to detect potential faults, *we must execute the code*. The same basic observation holds for many other goals a tester may have. If we want to expose situations where the code can crash, *we must execute the code*. If we want to show that performance or reliability targets are met, *we must execute the code*. Other testing goals—e.g., diversity, exposing interaction faults, and more—similarly benefit from exploration of the codebase. Targeting code coverage during search-based test generation is an effective and efficient method of exploring a wide range of program behaviors [3]. Therefore, even if a testers' goals lie beyond code coverage, *coverage is generally required to achieve those goals*.

However, we also observed that code coverage *alone* is a poor basis for producing test suites that meet these goals. In our past work, coverage only had a moderate correlation to the likelihood of fault detection [3], and was often weaker than random generation at detecting code mutations [4]. *Many different inputs can generally cover*

the same coverage goals. While some coverage criteria are stricter than others, the majority impose few or no constraints on how code is executed [3, 4, 13, 191].

"How" is important. Testers rarely design tests for the sole purpose of attaining coverage [188, 192]. In practice, tests are designed around specifications and coverage is used to identify clear weaknesses in the suite [9]. That is—coverage serves an advisory role for testers, rather than the primary basis of test design. If we want to expose crashing code, we select input with a high probability of triggering a crash. If we want to violate performance requirements, we select input with a high probability of slowing program execution. If there are multiple bugs in a branch, we typically need diverse inputs to uncover them all, as well as to cover the specification [78, 193]. In other words, while research in automated test generation has predominately focused on code coverage, *coverage alone is not enough*.

Search-based test generation already offers a solution for moving "beyond coverage" through multi-objective optimization—where multiple fitness functions are simultaneously optimized. To simultaneously gain the benefits of code coverage and produce robust tests that meet actual testing goals, one could target the combination of structural coverage and additional non-coverage fitness functions reflecting these goals of interest.

Our past research suggests that such a pairing can lead to better test suites than targeting coverage *or* the goal of interest alone [3, 190]. Consider a common testing goal—identifying situations where the system-under-test (SUT) throws an exception. This is a non-trivial goal, as we rarely know up-front which exceptions could be thrown. Targeting coverage may not satisfy this goal, as exception-triggering input will only be chosen if it uniquely enhances coverage. We could alternatively try to directly maximize the number of exceptions thrown. However, this count offers no feedback, more exceptions will only be discovered by random chance. We observed situations where targeting both offered feedback missing when targeting either alone—with the exception count biasing the input used to attain coverage, and branch coverage offering a means to explore the code base.

These observations suggest the potential benefit of blending code coverage and goal-based fitness functions. While multi-objective generation has been previously studied (e.g., [3, 194, 195]), how these objectives interact—and, in particular, the interaction between coverage and goal-based fitness functions—has not been studied in depth. Therefore, in this study, we assess and explore five hypotheses about this interaction:

Hypothesis 1: The inclusion of goal-based fitness functions as additional generation targets will have an impact on the attainment of code coverage, as compared to targeting coverage alone.

That is, targeting multiple objectives could affect the evolution of code coverage during the test generation process—raising or lowering the final quantity of coverage attained, changing the specific coverage goals covered, or affecting the rate at which coverage is attained during the generation process.

Hypothesis 2: Targeting both coverage and a goal-based fitness function will have an impact on the attainment of goal-based fitness functions, as compared to targeting coverage or a goal-based fitness function alone.

Similar to Hypothesis 1, targeting multiple objectives could affect the final fitness values of the goal-based objectives—raising or lowering goal attainment when compared to targeting a goal-based or a coverage-based objective alone.

Hypothesis 3: Targeting both coverage and a goal-based fitness function will have an impact on the fault detection of generated test suites, as compared to targeting coverage or a goal-based fitness function alone.

That is, targeting multiple objectives could increase or decrease the likelihood that the generated test suites detect faults or the number of tests that fail when a fault is detected.

Hypothesis 4: Targeting both coverage and a goal-based fitness function will have an impact on the size of the test suite and the average test length, as compared to targeting coverage or a goal-based fitness function alone.

That is, targeting multiple objectives could increase the number of test cases in the generated suites or increase the number of interactions in individual test cases, as each targeted objective adds additional obligations that the test suite must achieve. These obligations each may require distinct test input and setup to achieve, leading to the need for more or longer test cases.

Hypothesis 5: An increase in the search budget may lead to increased attainment of each objective, but will not change the fundamental relationships assessed in the previous hypotheses.

That is, we hypothesize that the effects that we observe when exploring the previous hypotheses will hold at higher search budgets. For example, if multi-objective optimization leads to higher fault detection at a limited search budget than single-objective optimization, we hypothesize that it will also do so at a higher search budget.

To assess these hypotheses, we target Branch Coverage—the most common structural coverage criterion [9]—as well as three specific testing goals:

- We further explore the goal of discovering situations where the SUT can crash.
- The discovery of situations that could violate performance goals—based on the maximization of execution time [158].
- Ensuring that test suites maximize coverage of diverse behaviors [193], specifically output diversity of the tested functions, which has been hypothesized to lead to faster coverage attainment and higher likelihood of fault detection [196].

Our study offers insight into how coverage and goal-based objectives interact during multi-objective test generation, with a focus on how this interaction affects code coverage, goal attainment, fault detection, the size of the test suite, and the length

```
@Test
public void testPrintMessage() {
    String str = "Test␣Message";
    StringUtils tCase = new StringUtils(str);
    tCase.removeWhitespace();
    assertEquals("TestMessage", tCase.getString());
}
```

Figure 5.1: Example of a unit test case written using the JUnit notation for Java.

of test cases. This research offers a starting point for exploring how search-based test generation can be adapted for particular goals, product domains, execution scenarios, or code structures, enables guidance on how to use test generation to meet tester goals, and can influence the creation of more efficient and effective test generation techniques and tools.

5.2 Background and Related work

5.2.1 Unit Testing

Testing can be performed at various levels of granularity. In this research, we focus on *unit testing*, where test cases target small segments of code that can be tested in isolation [1]. Unit tests are written as executable code, which can be re-executed on demand by developers. We refer to a purposefully grouped set of test cases as a *test suite*. Unit testing frameworks exist for many programming languages, such as JUnit for Java, and are integrated into most development environments.

An example of a unit test, written in JUnit, is shown in Figure 5.1. A unit test consists of a *test sequence (or procedure)*–a series of method calls to the class-under-test (CUT)–with *test input* provided to each method. Then, the test case will validate the output of the called methods and the class variables against a set of encoded expectations—the *test oracle*—to determine whether the test passes or fails. In a unit test, the oracle is typically formulated as a series of assertions on the values of method output and class attributes [5]. In the example in Figure 5.1, the *test input* consists of passing a string to the constructor of the `StringUtils` class, then calling its `removeWhitespace()` and `getString()` methods. We use an assertion to ensure that a space is removed from the input.

5.2.2 Adequacy (Coverage) Criteria

When testing, developers must judge both whether the tests they have written are effective and whether they have created enough test cases. Adequacy criteria have been developed to provide developers with guidance regarding these topics [9].

Each adequacy criterion prescribes, for a given program, a set of goals—referred to as test obligations. If tests fulfill the test obligations, than testing is deemed "adequate" with respect to faults that manifest through the structures of interest to the criterion. Most adequacy criteria measure coverage of structural elements of the software—such as individual statements, branches of the software's control flow, and complex boolean conditional statements—during the execution of a test suite [1, 4]. However, there are also adequacy criteria based, e.g., on coverage of formal requirements [197].

Adequacy criteria have seen widespread use in software development. Code coverage is routinely measured as part of automated build processes [13][1], and is mandated by safety standards in critical domains such as automotive [198] and avionics [199]. It is easy to understand the appeal of adequacy criteria. They offer clear checklists of testing goals that can be objectively evaluated and automatically measured through program instrumentation and execution analysis [13]. These same qualities make adequacy criteria ideal for use as automated test generation targets [66].

5.2.2.1 Branch Coverage

A branch refers to an outcome of any program statement that can cause program execution to diverge down a particular control flow path, such as the conditions in `if`, `case`, or loop definitions. Branch coverage requires that all outcomes of all control-diverging statements are executed at least once by the test suite under assessment.

To give an example, consider the `removeWhitespace()` method being tested in Figure 5.1, whose code is depicted in Figure 5.3. In this method, there are two program statements that affect the control flow—the loop condition on line 7 and the `if`-condition on line 9. To achieve branch coverage over this method, both conditions must evaluate to true and false at least once when the test suite is executed. In other words, there are four test obligations that must be fulfilled.

By default, coverage obligations are formulated over the source code. However, in Java, test obligations are often instead formulated and measured over the bytecode representation as this form is easier and more efficient to instrument and monitor. The bytecode representation of `removeWhitespace()` is shown in Figure 5.2. The same control-altering expressions are present, on lines 11 and 19. Branch coverage requires that both lines evaluate to true and false.

Branch Coverage is arguably the most commonly used coverage criterion, with ample tool support and industrial adoption [200]. For example, branch coverage measurement is built into the popular IntelliJ IDEA development environment. Therefore, we focus on Branch Coverage in this study as a representation of structural coverage criteria.

By default, coverage obligations are formulated over the source code. However, in Java, test obligations are often instead formulated and measured over the bytecode representation as this form is easier and more efficient to instrument and monitor. The bytecode representation of `removeWhitespace()` is shown in Figure 5.2. The same control-altering expressions are present, on lines 11 and 19. Branch coverage requires that both lines evaluate to true and false.

Branch Coverage is arguably the most commonly used coverage criterion, with ample tool support and industrial adoption [200]. For example, branch coverage measurement is built into the popular IntelliJ IDEA development environment. Therefore, we focus on Branch Coverage in this study as a representation of structural coverage criteria.

[1]For example, see `https://codecov.io/`.

```
public void removeWhitespace();
  Code:
     0: ldc           #7              // String
     2: astore_1
     3: iconst_0
     4: istore_2
     5: iload_2
     6: aload_0
     7: getfield      #9              // Field str:Ljava/lang/String;
    10: invokevirtual #15             // Method java/lang/String.length:()I
    13: if_icmpge     46
    16: aload_0
    17: getfield      #9              // Field str:Ljava/lang/String;
    20: iload_2
    21: invokevirtual #21             // Method java/lang/String.charAt:(I)C
    24: istore_3
    25: iload_3
    26: invokestatic  #25             // Method java/lang/Character.isWhitespace:(C)Z
    29: ifne          40
    32: aload_1
    33: iload_3
    34: invokedynamic #31,  0         // InvokeDynamic #0:makeConcatWithConstants:
                                      // (Ljava/lang/String;C)Ljava/lang/String;
    39: astore_1
    40: iinc          2, 1
    43: goto          5
    46: aload_0
    47: aload_1
    48: putfield      #9              // Field str:Ljava/lang/String;
    51: return
```

Figure 5.2: Java Bytecode of the `removeWhitespace()` method from Figure 5.3.

```
public class StringUtils{
    private String str;
    ...
    public void removeWhitespace(){
        String modified = "";

        for (int index = 0; index < this.str.length(); index++) {
            char ch = this.str.charAt(index);
            if (!Character.isWhitespace(ch)) {
                modified += ch;
            }
        }

        this.str = modified;
    }
    ...
}
```

Figure 5.3: Subset of the class-under-test in Figure 5.1.

5.2.3 Search-Based Test Generation

Manual creation of a large volume of test cases can be tedious and expensive. Automation of aspects of test creation, such as test input selection, can reduce and focus the required manual effort [7]. Search-based test generation frames input selection as a search problem, where metaheuristic optimization algorithms attempt to identify test input that best embody properties that testers seek in their test cases [2, 7].

These properties are assessed using one or more fitness functions—numeric scoring functions. The metaheuristic embeds a strategy for sampling solutions from the space of possible inputs, often based on a natural process such as evolution or swarm

behavior [201]. In test generation, a "solution" is often either a single test case or a full test suite. The metaheuristic uses the selected fitness functions to assess solution quality, offering feedback to guide the selection and improvement of solutions over a series of generations. Search-based test generation has proven to be a flexible [10], scalable [202], and competitive [3, 194] method of automated test generation.

The most common metaheuristics for search-based test generation are genetic algorithms, which are modeled after the natural evolution of a population [203]. While specific aspects vary, a "typical" test generation follows these steps:

- An initial population of solutions is randomly generated. Each solution represents a test suite, containing test cases.
- Each generation, the fitness score of each solution is calculated and a new population is created. This population is formed through four sources of solutions:
  - One of the best solutions may be carried over to the new population intact.
  - At a certain probability, elements of two solutions will be combined to create two "children" (crossover). For example, the children may blend test cases from the parents.
  - At a certain probability, a solution can be mutated—e.g., a test case may be modified.
  - At a certain probability, a new randomly generated solution will be added to the population to maintain diversity.
- When the search budget—typically expressed in time or number of generations—expires, the best solution is returned.

5.2.4 Related Work

Multi and many-objective optimization algorithms have become increasingly common in search-based test generation [204]. Even if the goal of the test generation process is solely code coverage, coverage can quickly be gained by representing each test obligation as an independent objective and applying multi or many-objective optimization [8]. Multiple studies have compared different algorithms for multi and many-objective optimization in terms of coverage achieved (e.g., [65, 205–207]). However, these studies focused solely on coverage-based fitness and have not examined the interaction between coverage-based and goal-based fitness functions.

The number of exceptions or crashes discovered is a common secondary objective in search-based test generation, optimized in conjunction with coverage-based fitness functions (e.g., [3, 93, 208, 209]). Others have explored combinations of coverage criteria with non-functional criteria during test generation or test suite minimization, such as memory consumption [210] or execution time [211]. While these represent multi-objective optimization of coverage and goal-based fitness functions, these studies do not examine how these fitness functions interact, e.g., how the combinations affect coverage or fault detection.

Rojas et al. examined multi-objective optimization of Line Coverage—a structural coverage criterion—and additional fitness functions [93]. Relevant to our work, they also include two of the same goal-based objectives that we focus on, exception count and output diversity. They found that adding additional fitness functions led to only a minimal loss in the final percentage of Line Coverage achieved. They also found that coverage of secondary criteria increased over when Line Coverage was targeted alone.

Therefore, there is a partial overlap in our focus. However, they only examined the final level of coverage and focused on different aspects of test generation. We address a broader set of hypotheses and examine coverage attainment more deeply.

Palomba et al. examine optimization of Branch Coverage and fitness functions intended to improve test quality based on the cohesion and coupling of test cases [195]. They found that targeting these quality objectives could increase code coverage over targeting coverage alone.

Weiglhofer et al. showed that coverage-directed test generation can be used to complement test generation based on testing goals [212]. In their approach, humans develop "test purposes", specifications used in conjunction with formal models to generate test cases. Coverage-directed testing is then used to generate tests for parts of systems not covered by the test purposes. They do not apply multi-objective optimization, but the core concept is similar.

We previously examined the likelihood of fault detection of test suites generated targeting various fitness functions [3, 190]. Much of this research focused on single-objective optimization. However, we did find that some combinations of objectives, such as Branch Coverage and the exception count, had a higher likelihood of fault detection than targeting Branch Coverage or exception count alone [3]. This past work partially addresses one of our hypotheses, but we examine that hypothesis more closely in this study.

Zhou et al. propose an approach, "smart selection", for selecting a subset of test obligations when targeting multiple coverage-based fitness functions for test generation [213]. Their approach reduces redundancy between fitness functions and eases optimization difficulty. McMinn et al. have also proposed using search techniques to evolve new coverage criteria that combine features of existing criteria [214]. In previous work, we also used reinforcement learning to dynamically select the fitness functions targeted during multi-objective test generation [20]. We demonstrated that fitness functions could be identified that increased attainment of common testing goals for particular classes-under-test. However, these studies do not examine the interaction between coverage and goal-based fitness functions.

5.3 Methods

Our aim in this research is to examine the interaction between coverage-based and goal-based fitness functions during multi-objective test generation. In Section 2.1, we raised five hypotheses about how these objectives could interact. We assess those hypotheses by addressing the following specific research questions:

- **RQ1:** How is the Branch Coverage of generated test suites influenced by targeting additional goal-based fitness functions, compared to targeting Branch alone?
  - **RQ1.1:** How is the final percentage of attained Branch Coverage influenced?
  - **RQ1.2:** How is the set of satisfied Branch Coverage obligations influenced?
  - **RQ1.3:** How is the evolution of coverage attainment influenced?

- **RQ2:** How is the attainment of testing goals by generated test suites influenced by targeting Branch Coverage in addition to a goal-based fitness function, compared to targeting coverage or a goal-based fitness function alone?
- **RQ3:** How is the fault detection of generated test suites influenced by targeting Branch Coverage in addition to a goal-based fitness function, compared to targeting coverage or a goal-based fitness function alone?
- **RQ4:** How is the suite size and test case length of generated test suites influenced by targeting Branch Coverage in addition to a goal-based fitness function, compared to targeting coverage or a goal-based fitness function alone?
- **RQ5:** What influence does the search budget have on Branch Coverage, goal attainment, fault detection, test suite size, and test case length attained by suites targeting different fitness function configurations?

As discussed in Section 2.1, we focus on three concrete testing goals: (1) discovery of scenarios where exceptions are thrown ("Exception Count"), (2) discovery of scenarios where the execution time may violate performance goals ("Execution Time"), and (3), maximization of output diversity ("Output Coverage"). To address these research questions, we have performed the following experiment, targeting Branch Coverage, these three goals, and combinations of Branch Coverage with each goal:

[a] **Collected Case Examples:** We have selected 93 case examples from the Defects4J fault dataset, from 14 Java projects (Section 5.3.1).

[b] **Defined Test Generation Configurations:** We selected three single-objective configurations (Branch Coverage, Exception Count, Output Coverage) and three multi-objective configurations (Branch Coverage plus each testing goal listed above) and two search budgets (180 and 300 seconds) to target in our experiments (Section 5.3.2).

[c] **Generated Test Suites:** For each class modified by each case example, fitness function configuration, and search budget, we generated 10 test suites using EvoSuite. We target the fixed version of each class-under-test (CUT) (Section 5.3.2).

[d] **Monitored Coverage Evolution:** We monitor how satisfaction of Branch Coverage obligations changes over the course of each invocation of EvoSuite (Section 5.3.3).

[e] **Recorded Generation Statistics:** For each suite generated, at the end of the generation process, we record information on Branch Coverage obligation satisfaction, fitness values for each targeted function, test suite size, and test case length (Section 5.3.3).

[f] **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are removed (Section 5.3.3).

[g] **Assessed Fault-finding Effectiveness:** We measure the number of faults detected, the proportion of test suites that detect each fault to the number generated (likelihood of fault detection), and number of failing tests in each suite (Section 5.3.3).

Project	Faults Selected	Total
Chart	7, 6, 10, 8, 3, 5	6
Cli	27, 7, 29, 28, 1, 10, 3, 40, 2, 5	10
Closure	161, 74, 19, 154, 162, 164, 37, 55, 41, 70, 12, 71, 5	13
Codec	7, 6, 17, 1, 2	5
Collections	25, 26	2
Compress	2, 47, 46	3
Csv	1	1
Gson	3, 4, 5, 6	4
JacksonCore	11	1
JacksonDatabind	62, 93, 111, 112	3
Jsoup	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 55, 60, 77	16
Lang	4, 5, 6, 8, 9, 10, 11, 12, 41, 55, 64, 65	12
Math	95, 11, 87, 81, 100, 39, 90, 41, 3, 49, 40, 2	12
Mockito	6, 8, 37, 15, 2	5
	Total	**93**

Table 5.1: Subset of Defects4J faults selected for this study.

[h] **Analyzed the Collected Data:** We address the research questions using the data gathered above (Section 5.3.4).

We make a replication package available containing the data collected in this experiment: `https://doi.org/10.5281/zenodo.11047567`
We also make available our modified version of EvoSuite:

- Code: `https://github.com/afonsohfontes/evosuite`
- Executable: `https://github.com/afonsohfontes/defects4j/tree/master/framework/lib/test_generation/generation`

5.3.1 Case Example Selection

Defects4J is an extensible database of real faults extracted from Java projects [215][2]. The current dataset, version 2.0.1, consists of 835 faults from 17 Java projects. To control experiment costs, in this study, we aimed to select a sample of approximately 100 faults, chosen to reflect the proportion of faults-per-project in the full dataset. To select this sample, we initially selected **206** faults at random, sampled based on the number of faults-per-project in the full dataset. We then generated test suites targeting Branch Coverage and the three multi-objective configurations following the procedure described in Section 5.3.2, and omitted faults where either:

- Errors prevented the completion of 10 valid trials for all configurations, where a test suite was generated and all data collection completed successfully.
- Where the average Branch Coverage was below 5%—we judged that the research questions could not be reliably answered without a minimal level of coverage being reached over the classes-under-test.

This filtering process ultimately resulted in a set of **93** faults used in this study, listed in Table 5.1.

[2]Available from `http://defects4j.org`

For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the fault, and a list of classes and lines of code modified by the patch that fixes the fault.

Each fault is required to meet three properties. First, a pair of code versions must exist that differ only by the minimum changes required to address the fault. The "fixed" version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix must be isolated from unrelated code changes such as refactorings.

5.3.2 Test Generation Configuration

In this study, we make use of the EvoSuite unit test generation framework for Java [216]. EvoSuite is mature, actively maintained, and has been successfully applied to a wide variety of projects [3, 194]—even winning multiple tool competitions (e.g., [217]). Specifically, we make use of a modified version of EvoSuite version 1.2.1, where we have added an additional fitness function—Execution Time—as well as additional monitoring and data collection capabilities.

Test Generation Algorithm We make use of EvoSuite's "whole test suite generation" algorithm, based on a monotonic Genetic Algorithm [216][3]. In this implementation of whole test suite generation, each solution represents a full test suite—in contrast to approaches where a solution represents a single test case. Then, rather than targeting one obligation (sub-goal) of each fitness function one-by-one, fitness is measured over all obligations of each fitness function at the same time.

In traditional multi-objective optimization algorithms, such as NSGA-II [218], an attempt is made to balance fitness function attainment, and each fitness function is treated as independent. In contrast, in this implementation of whole test suite generation, a single aggregate fitness score is calculated. The fitness for a test suite T over the class-under-test C is:

$$\text{fitness}(T, C) = \sum_{f \in F} \hat{f}(T, C) \tag{5.1}$$

That is, the aggregate fitness is the sum of the normalized score of each fitness function. EvoSuite treats all optimizations as *minimization* problems, where lower fitness scores represent better solutions.

Fitness Function Configurations We execute EvoSuite for each case example utilizing six fitness function configurations, representing three single-objective configurations (Branch Coverage, Exception Count, and Output Coverage) and three multi-objective configurations (Branch Coverage with Exception Count, Output Coverage, and Execution Time)[4]. The fitness functions are defined as follows:

[3]This implementation of whole test suite generation has been replaced as the default optimization algorithm in EvoSuite by DynaMOSA, a many-objective optimization algorithm [8]. While DynaMOSA has been shown to achieve better coverage than whole test suite generation in some experiments [206], we use whole test suite generation to enable clearer comparison to our past research [3, 190].

[4]Due to technical details of its implementation, we are unable to target Execution Time without also targeting Branch Coverage. Therefore, Execution Time cannot currently be targeted as a single-objective configuration.

- **Branch Coverage**: As defined in Section 2.2, Branch Coverage requires that all outcomes of all control-diverging statements are executed at least once by a test suite. For search-based test generation to be most effective, a fitness score should offer feedback to help guide the identification of better solutions. To that end, the most effective fitness functions tend to encode information about the *distance* to satisfying any unsatisfied goals. Therefore, rather than simply measuring whether each test obligation is covered or not, the fitness calculation for Branch Coverage instead embeds information—for each test obligation—on *how close execution came to satisfying that obligation*.

  The branch coverage fitness function is a minimization of the following, where T refers to the test suite and B represents the set of test obligations. Each test obligation, $b \in B$, represents a control-diverging program statement and a desired outcome for that statement (`true` or `false`).

$$\text{fitness}(T, B) = \sum_{b \in B} d(T, b) \tag{5.2}$$

  where $d(T, b)$ is defined as:

$$d(T, b) = \begin{cases} 0 & \text{if } b \text{ has been satisfied} \\ \overline{d_{min}(t \in T, b)} & \text{if } b \text{ has been evaluated at least twice} \\ 1 & \text{otherwise} \end{cases} \tag{5.3}$$

  In the case where an obligation has not been satisfied, $\overline{d_{min}(t \in T, b)}$ represents the *branch distance*—the magnitude of change in execution that would be needed to achieve the targeted outcome for that control-diverging statement. The branch distance is determined based on how the condition has been formulated, following a standard set of formulae [66]. In this case, the minimal observed value of the branch distance is used in the fitness calculation, and is normalized to be between 0–1.

- **Exception Count:** This fitness function represents the goal of causing the class-under-test to throw as many exceptions as possible—either declared or undeclared. The fitness function is a minimization of the following formula, where T refers to the test suite, $E_{discovered}$ represents the number of exceptions discovered during the current generation process, and E_{thrown} represents the number thrown by T.

$$\text{fitness}(T) = 1 - \frac{E_{thrown}}{E_{discovered}} \tag{5.4}$$

  As the number of possible exceptions that a class can throw cannot be known ahead of time, the number of test obligations may change each time EvoSuite is executed on a CUT.

- **Execution Time:** This fitness function represents a scenario where we seek test suites that could uncover potential violations of performance requirements. We have added a new fitness function to EvoSuite for this purpose, which is calculated as follows:

$$\text{fitness}(T) = 1 - \frac{Time_{current}}{Time_{max}} + penalty \tag{5.5}$$

Where $Time_{current}$ represents the execution time of the solution currently under assessment and $Time_{max}$ is the largest execution time discovered during that search to date.

One avenue to generate test suites with long execution times is simply to generate excessively long test cases that call many methods. Therefore, to prevent the generation of overly bloated test cases, the fitness calculation applies a penalty based on the average test case length within the suite:

$$penalty = 0.1 \times \frac{Length_{current}}{Length_{max}} \tag{5.6}$$

- **Output Coverage:** This configuration represents the goal of generating test suites that cover many different types of outcomes of the methods of the CUT. A tester may seek such diversity for two reasons. First, increased output coverage is hypothesized to lead to earlier and potentially higher code coverage [193, 219], and second, to potentially increase fault detection over pure white-box techniques [196].

  Output coverage rewards diversity in the method output by mapping return types to a list of abstract values—Alshahwan and Harman provide a detailed explanation, including fitness formulae [196]. A test suite satisfies output coverage if, for each public method in the CUT that returns a data type covered by the fitness function, at least one test yields a concrete return value matching each abstract value. For numeric data types, distance functions similar to the branch distance offer feedback using the difference between the chosen value and the targeted abstract values.

Search Budgets Two search budgets were used—180 seconds and 300 seconds per class. This allows us to examine how an increased search budget affects the test suites produced by each single and multi-objective configuration.

Generation Procedure Test suites are generated individually for each of the classes modified to fix each fault chosen from Defects4J. We repeat generation a fixed number of times for each class, fitness function configuration, and search budget.

Test suites are generated targeting the fixed version of each CUT and applied to the faulty version to eliminate the oracle problem. EvoSuite generates assertion-based oracles. Generating oracles based on the fixed version of the class means that we can confirm that the fault is actually detected, and not just that there are coincidental differences in program output. This translates to a regression testing scenario, where tests are generated using a version of the system understood to be "correct" in order to guard against future issues. Tests that fail on the faulty version detect behavioral differences between the two versions.

Test suite generation and execution were performed on virtual machines, each configured with 4 vCPUs, 8GB of RAM, and 20GB of storage, running a server version of Ubuntu 18.04.4 LTS. Each virtual machine was dedicated to executing experiments for a specific subset of faults and fitness function configurations, ensuring that experiments remained isolated and independent to ensure result reliability.

To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. We also disable test suite reduction, an optional procedure that removes redundant test cases at the end of the generation process. We do this to

maintain traceability between intermediate and final test suites during suite evolution. All other settings were kept at their default values. As results may vary, we performed 10 trials for each CUT, fitness function configuration, and search budget. This resulted in the generation of 11160 test suites (two budgets, ten trials, six configurations, 93 faults).

Generation tools may generate flaky (unstable) tests [194]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, all non-compiling test suites are removed. Then, each remaining test suite is executed on the fixed version of the CUT. If the test results are inconsistent, the test case is removed. This process is repeated until all tests pass five times in a row. On average, less than 1% of test cases were removed from each suite.

5.3.3 Data Collection

To answer our research questions, we capture the following data during and after generation:

- **Final Fitness Function Values:** For each test suite, we record the final fitness values for all four fitness functions considered in this experiment (Branch Coverage, Exception Count, Execution Time, and Output Coverage).
- **Branch Coverage Obligation Satisfaction:** Given a CUT, achieving Branch Coverage requires satisfying a set of test obligations, as defined in Section 2.2. We record information on the satisfaction of Branch Coverage obligations, including:
  - **Number of Test Obligations:** For each class-under-test, we record the number of Branch Coverage obligations.
  - **Percentage of Obligations Satisfied:** For each final test suite, we record the percentage of Branch Coverage obligations satisfied.
  - **Specific Obligations Satisfied:** For each final test suite, we record the specific obligations satisfied.
  - **Evolution of Branch Coverage During Generation:** To understand the dynamic evolution of coverage over the course of each invocation of EvoSuite, we tracked the percentage of obligations and specific obligations covered by the best test suite in the population once per second during the generation process.
- **Fault Detection:** To evaluate the fault-finding effectiveness of the generated test suites, we execute each test suite against the faulty version of each CUT. We then record the following:
  - **Likelihood of Fault Detection:** Across all trials for a particular fault, fitness function configuration, and search budget, we record the proportion of trials where the fault was detected to the total number of trials for that configuration.
  - **Number of Failing Tests:** For each test suite, we record the number of test cases that detect that fault (pass on the fixed version and fail on the faulty version).

- **Test Suite Size:** We recorded the number of tests in each test suite.
- **Average Test Case Length:** Each test consists of one or more method calls, variable initializations, and assertions. We record the average number of lines in each test case.

5.3.4 Data Analysis

We answer each research question using the data gathered, comparing results attained by each fitness function configuration, split based on the search budget. To analyze the data, we employ a combination of descriptive statistics, distribution comparisons, and effect size tests when distributions are found to differ. Further explanation is provided in Section 2.4. Here, we provide a general overview of the data analysis procedure.

Descriptive Statistics Descriptive statistics provide an initial overview of the collected data.

[a] **Data Analysis**: Basic statistical measures such as the average, median, standard deviation, and percentiles are calculated for data appropriate for answering each research question. This provides an initial understanding of the data distribution and central tendencies [220].

[b] **Data Visualization**: We utilize box plots as a graphical representation to offer a visual insight into the result distribution across different configurations [221].

Distribution Comparisons We are interested in assessing whether the observed differences between two fitness function configurations at a particular search are significantly different.

For each research question, we select data relevant to that question (e.g., Branch Coverage attainment in RQ1). Then, for each pair of fitness function configurations, we formulate a hypothesis and null hypothesis in the following format:

- H: Generated test suites have different distributions of X depending on the targeted fitness function configuration.
- $H0$: Observations of X for both configurations are drawn from the same distribution.

Our observations for each of the collected data items defined above are drawn from an unknown distribution. To evaluate the null hypothesis without any assumptions on distribution, we use the Wilcoxon rank-sum test [222], a non-parametric test. We apply the test with $\alpha = 0.05$. A p-value less than α indicates a statistically significant difference [223].

To mitigate the risk of Type I errors, a Bonferroni correction was be applied in analyzing RQ2 and RQ3 [224]. The Bonferroni correction is a technique used to counteract the problem of multiple comparisons [225]. When multiple statistical tests are performed, the likelihood of observing at least one significant result just by chance (a false positive) increases. The Bonferroni correction addresses this issue by adjusting the threshold for statistical significance.

Specifically, it divides the desired overall p-value threshold (0.05) by the number of comparisons being made. For example, if five comparisons are performed, the

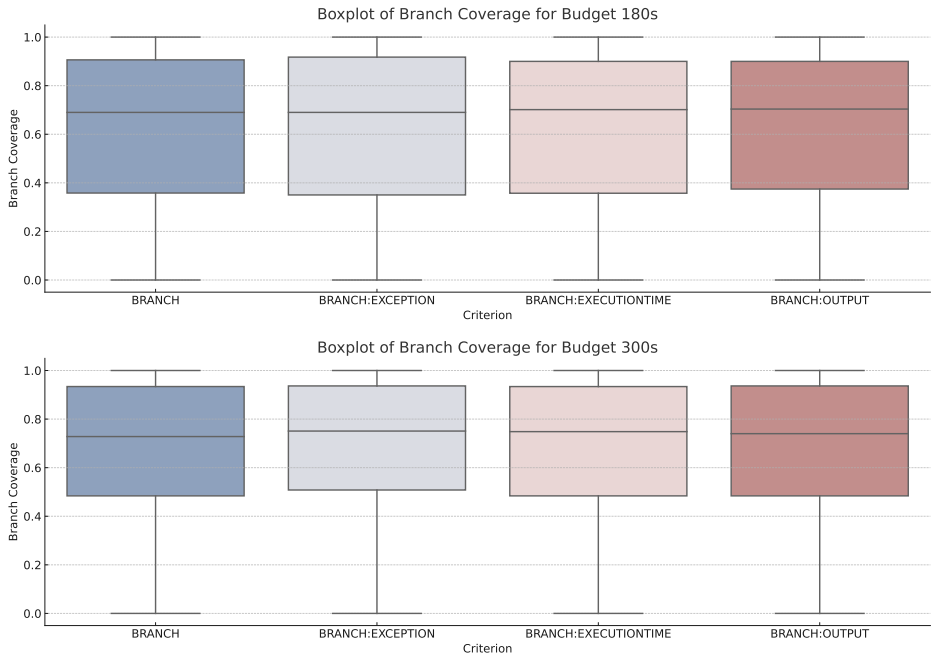


Figure 5.4: Boxplots of the Branch Coverage attained by test suites, divided by budget.

Bonferroni correction would adjust the significance level to 0.01 for each test. This makes it harder for any single comparison to reach the significance threshold, thereby reducing the chance of false positives.

Effect Size The Wilcoxon test determines if there are significant differences between the distributions of two configurations. We use Cohen's d to measure the effect size of these differences, providing a clearer understanding of their magnitude and practical significance [226].

We apply the standard interpretation of Cohen's d:

- $d > 0$ indicates that observations of configuration A will have a significantly higher mean value than configuration B.
- $d < 0$ indicates that an observation of configuration A will have a significantly lower value than configuration B.
- The absolute value of d is categorized as follows for further interpretation:
  - $|d| < 0.2$: Negligible effect
  - $0.2 \leq |d| < 0.5$: Small effect
  - $0.5 \leq |d| < 0.8$: Medium effect
  - $|d| \geq 0.8$: Large effect

5.4 Results

5.4.1 Effect on Structural Coverage (RQ1)

In this section, we address the following hypothesis:

> **Hypothesis 1:** The inclusion of goal-based fitness functions as additional generation targets will have an impact on the attainment of code coverage, as compared to targeting coverage alone.

Often, targeting multiple objectives can have *some* effect on each individual objective targeted, as compared to targeting a single objective on its own. If objectives are contradictory, or if too many objectives are targeted at once, then the final attainment of each may be lowered [3,93]. However, targeting one objective may also offer feedback that enhances attainment of another [20,190]. Therefore, we wish to assess—first—the impact that targeting additional goal-based objectives has on code coverage-based objectives. We examine three aspects of coverage: (1) the final percentage of coverage attained, (2) the specific coverage obligations covered by the final test suites, and (3), the evolution of coverage attainment over evolution. For this evaluation, we compare suite generated targeting Branch Coverage alone to suite targeting Branch Coverage and an additional goal-based fitness function.

Attained Branch Coverage Table 5.4 offers descriptive statistics on the final attainment of Branch Coverage by generated test suites. Figures 5.4 and 5.5 also depict the attained Branch Coverage overall, and by project from Defects4J, respectively.

Table 5.4 and Figure 5.4 do not demonstrate any clear differences between configuration, with regard to the final attained Branch Coverage. The distribution of results is visually similar for each configuration, and the mean and median Branch Coverage attained by each configuration is within a narrow range. An increase in search budget yields an increase in the average Branch Coverage, as well as less variance in the final results—seen in a rise in the 25th percentile. However, this improvement seems largely consistent across configurations.

In Figure 5.5, we do see some differences between configurations for particular projects. However, there are few clear trends and only a small number of bugs were drawn from many of these projects. Still, we note some observations from the projects with over 10 included bugs. First, for project Cli, we see a higher median Branch Coverage for the combination of Branch Coverage and Execution Time at both search budgets. For the project JSoup, we see that the combination of Branch and Output Coverage yields a slightly higher median coverage at both search budgets. We will investigate both of these observations—as well as potential differences for other projects—more closely in future work.

To confirm our initial inspection, we performed a Wilcoxon Rank-Sum test to assess pairwise comparisons between different test generation configurations for the two search budgets and, where needed, applied the Bonferroni correction to reduce the likelihood of Type I errors. The results of this test are shown in Table 5.2, where we see that no comparison demonstrated statistically significant differences—that is, no p-value was below 0.05.

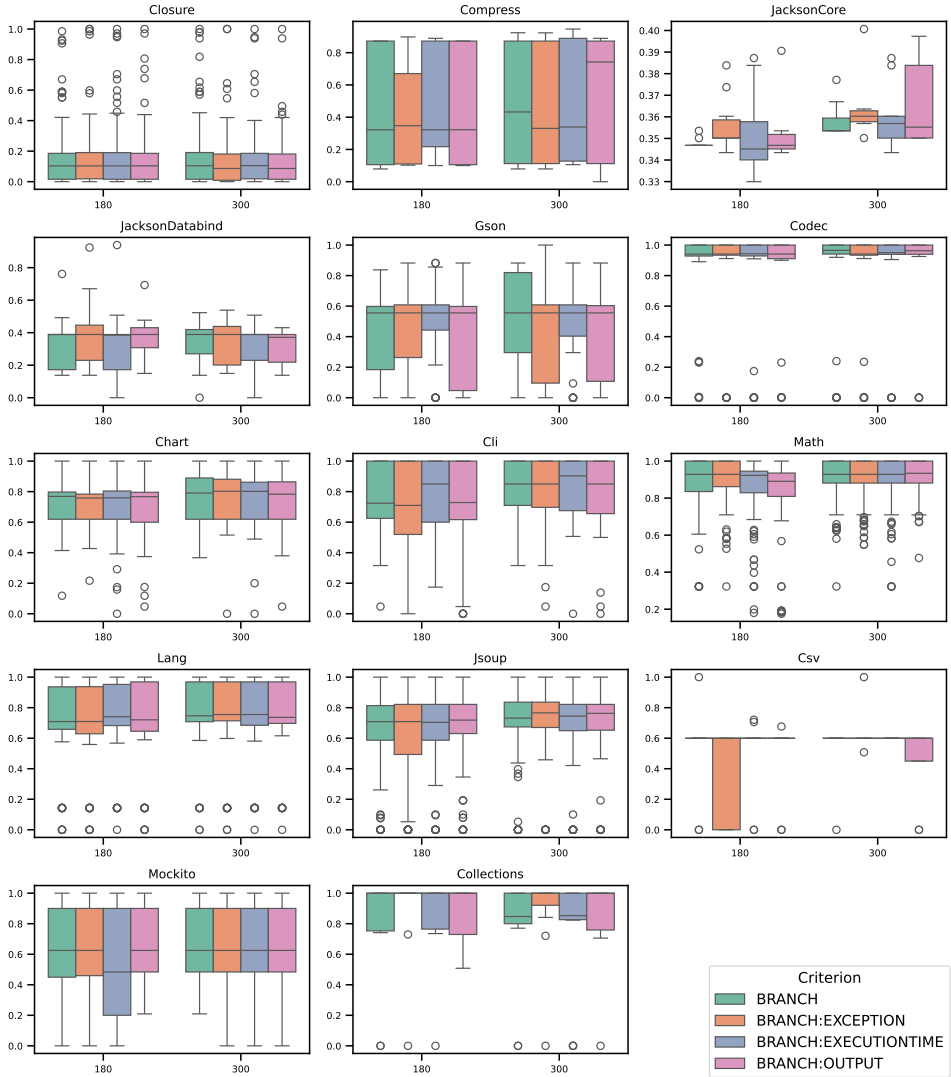


Figure 5.5: Boxplots of the Branch Coverage, divided by both budget and project from Defects4J. The X-axis reports the budget and the Y-axis the Branch coverage.

RQ1.1 (Attained Branch Coverage): Optimizing a second goal-based fitness function does not have a significant impact on the final Branch Coverage attained by test suites.

Attained Coverage Obligations During and after the test generation process, we collected information on which specific Branch Coverage obligations were covered by generated test suites. To assess whether different configurations tend to cover distinct test obligations, we calculated the average coverage of each obligation for

Comparison	Budget	P-value
Branch vs Branch & Exception	180	0.659
Branch vs Branch & Execution Time	180	0.357
Branch vs Branch & Output	180	0.275
Branch vs Branch & Exception	300	0.967
Branch vs Branch & Execution Time	300	0.485
Branch vs Branch & Output	300	0.157

Table 5.2: Calculated p-values from comparisons of attained Branch Coverage by different configurations, split by search budget.

Criterion	Budget	Mean	Std	Min	25%	50%	75%	Max
Branch & Exception	180	0.0033	0.0373	-0.0625	-0.0010	0.0000	0.0038	0.3194
Branch & Execution Time	180	-0.0015	0.0286	-0.1427	-0.0048	0.0000	0.0039	0.1226
Branch & Output	180	-0.0008	0.0527	-0.3875	-0.0043	0.0000	0.0058	0.2097
Branch & Exception	300	-0.0003	0.0232	-0.1111	-0.0051	0.0000	0.0048	0.0482
Branch & Execution Time	300	-0.0030	0.0311	-0.1936	-0.0063	0.0000	0.0073	0.1014
Branch & Output	300	-0.0011	0.0310	-0.1660	-0.0076	0.0000	0.0073	0.0871

Table 5.3: Descriptive statistics of the difference in the average coverage of each obligation between branch and another configuration, split by configuration and budget.

Configuration	Budget	Mean	Std Dev	Min	25th %	Median	75th %	Max
Branch	180	0.618	0.326	0.000	0.383	0.690	0.906	1.000
Branch & Exception	180	0.614	0.333	0.000	0.367	0.690	0.912	1.000
Branch & Execution Time	180	0.621	0.324	0.000	0.389	0.704	0.900	1.000
Branch & Output	180	0.620	0.325	0.000	0.383	0.704	0.900	1.000
Branch	300	0.665	0.317	0.000	0.484	0.726	0.929	1.000
Branch & Exception	300	0.665	0.325	0.000	0.507	0.749	0.936	1.000
Branch & Execution Time	300	0.659	0.322	0.000	0.484	0.742	0.932	1.000
Branch & Output	300	0.653	0.324	0.000	0.475	0.740	0.934	1.000

Table 5.4: Descriptive statistics of Branch Coverage across different test generation configurations and search budgets.

each class-under-test across all trials conducted for each configuration, search budget, and bug. For example, if four of the ten trials targeting Branch Coverage and Output Coverage for bug Chart-3 covered the first Branch Coverage obligation for the targeted class, then the average coverage of that obligation would be 0.40.

The resulting averages were then used to compare targeting Branch Coverage alone to targeting Branch Coverage and a second goal-based fitness function. For example, if there were four coverage obligations for a class:

- When targeting Branch Coverage alone, the average coverage of each obligation was 0.4, 0.7, 0.8, and 0.4.
- When targeting Branch and Exception Count, the average coverage of each obligation was 0.5, 0.7, 0.8, and 0.3.
- The resulting difference between the two would be -0.1, 0.0, 0.0, and 0.1.

Figure 5.7 visualizes the result of this comparison for one class from one bug and search budget, as an example of the differences that can emerge. In this example,

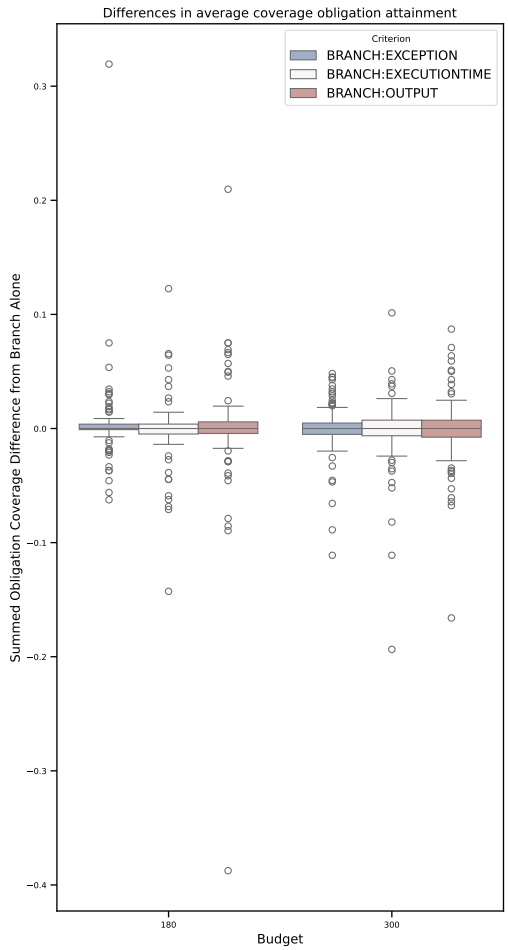


Figure 5.6: Boxplots of the differences in covered obligations between targeting Branch Coverage alone versus Branch and goal-based fitness function.

positive spikes show cases where targeting Branch alone performed better for a particular obligation versus the compared configuration and vice versa.

To generalize the assessment across all bugs, we calculated the sum of these differences for each class for each bug. Figure 5.6 plots the difference between each configuration across all bugs, split by search budget. Table 5.3 includes descriptive statistics on the difference in the average coverage of each obligation.

Figure 5.6 shows the vast majority of the summed differences are close to zero, with a median of 0.00 for all comparisons. This means that—in most cases—there

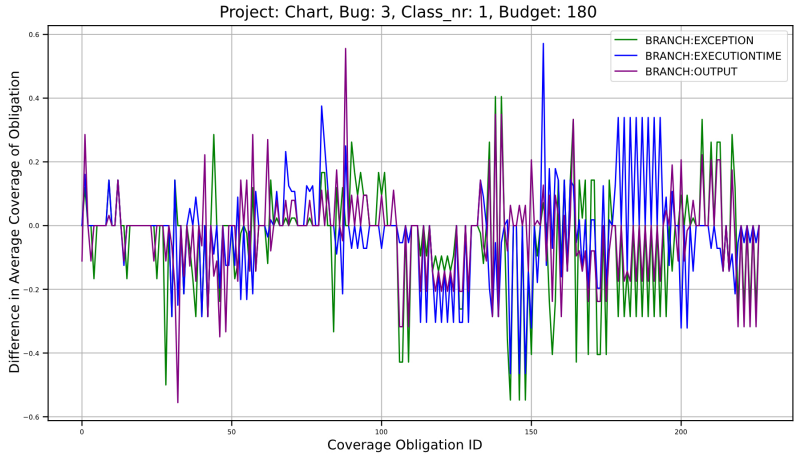


Figure 5.7: Difference in the average coverage of each obligation by Branch alone against Branch & Exception (green), Branch & Execution Time (Blue), and Branch & Output (purple) for Chart-3, Class 1, when the budget is set to 180 seconds.

are few major differences in the obligations covered by each configuration. There are differences in the 25th and 75th percentiles between configurations, but relatively narrow ones.

RQ1.2 (Obligations Covered): In the majority of cases, the addition of a goal-based fitness function does not change the likelihood of covering particular test obligations. Almost all differences in the average coverage of individual obligations are within 10% of when Branch Coverage is targeted alone.

Figure 5.6 shows that there are a number of outliers. Most outliers are clustered within -0.1 to 0.1, i.e., within 10% difference in average coverage. However, it is possible that some of these outliers offer information that could improve the results of test generation. In particular, negative outliers are interesting, as they suggest cases where the addition of a goal-based fitness function improved Branch Coverage.

We inspected the negative outliers, with a particular focus on the six most extreme cases—Math-81 at the 180 second budget for both Branch and Execution Time and Branch and Output Coverage, JSoup-9 for Branch and Exception Count and Branch and Execution Time at the 300 second budget, Math-11 for Branch and Execution Time at the 300 second budget, and Closure-164 for Branch and Output Coverage at the 300 second budget. Overall, there were few clear and actionable conclusions that we could draw from these outliers. However, we share some interesting observations.

The class modified in Math-81 for Branch and Output Coverage at the 180 second budget was the most extreme outlier, with an average difference of 38.75% in coverage. The explanation for this difference is relatively straightforward. Many of the methods of the class-under-test have numeric return values. It is likely that Output Coverage, by placing emphasis on returning diverse results for these methods, helped to steer test generation towards covering Branch Coverage obligations in these methods and

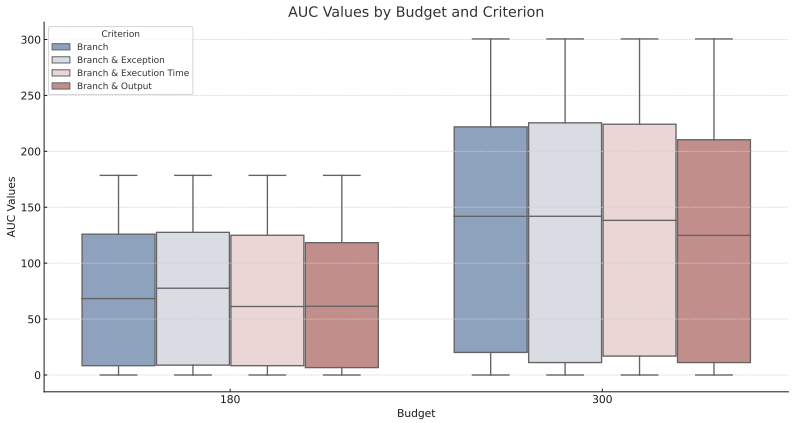


Figure 5.8: Boxplot for Area Under the Curve (AUC) of the branch coverage evolution, split by budget and Criterion.

Criterion	Budget	Average	Median
Branch	180	74.06	68.32
Branch & Exception	180	75.44	77.59
Branch & Execution Time	180	70.73	61.25
Branch & Output	180	70.30	61.48
Branch	300	134.19	141.89
Branch & Exception	300	135.65	141.93
Branch & Execution Time	300	130.55	138.27
Branch & Output	300	127.60	124.85

Table 5.5: Average and median values for AUC of coverage evolution for different configurations, split by budget.

in methods indirectly called through these methods. This improvement disappears at the 300 second search budget, suggesting that Branch Coverage alone is eventually effective. However, the addition of Output Coverage speeds coverage attainment.

A similar observation can be made for Branch and Output Coverage for the class-under-test in Closure-164, where there was an average coverage difference of 16.60%. Most methods in this class return Boolean values. It is possible the Output Coverage helped to encourage Branch Coverage by ensuring that the methods returned both possible values. Here, this difference increased with the search budget.

A potential explanation for the outliers for Branch and Execution Time is that the Execution Time fitness function encourages the generation of longer test cases, with more program interactions. This function penalizes test cases that are too long, but the average test case length is still higher than when Branch Coverage is targeted alone. This may encourage improvement in coverage as well, in a small number of cases.

Evolution of Coverage Attainment We collected the evolution of coverage during the test generation process, based on the Branch Coverage achieved by the best test suite in the evolving population. A snapshot of coverage is captured each second during the generation process. This allows us to calculate the AUC (Area Under the Curve) and the time the search took to achieve 25%, 50%, and 75% coverage during

AUC Comparison	Budget	P-value	Effect Size	Category
Branch vs Branch & Exception	180	0.007	-0.05	Negligible
Branch vs Branch & Execution Time	180	0.950	-	-
Branch vs Branch & Output	180	0.923	-	-
Branch vs Branch & Exception	300	0.281	-	-
Branch vs Branch & Execution Time	300	0.978	-	-
Branch vs Branch & Output	300	0.506	-	-

Table 5.6: P-values and effect size (when significant differences occur) on pairwise comparisons of AUC by different configurations, split by search budget.

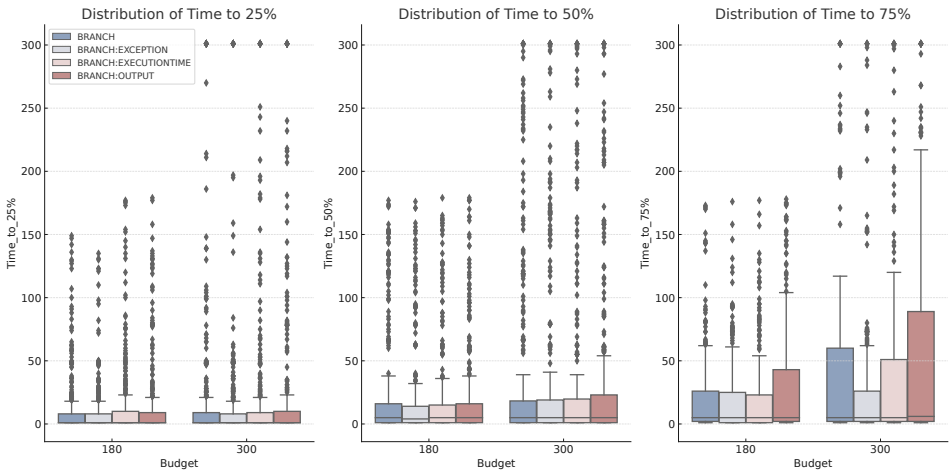


Figure 5.9: Boxplots of the time taken to achieve 25%, 50%, and 75% Branch Coverage for each configuration, split by budget.

each trial for each configuration.

Figure 5.8 and Table 5.5 shows the statistics for AUC for each configuration and search budget. Higher AUC values indicate that Branch Coverage evolved early while lower means the search took more time to achieve coverage.

We observe that the median AUC is lower for both search budgets for Branch and Execution Time as well as for Branch and Output Coverage than for Branch alone, potentially indicating slightly slower coverage attainment. However, the 25th and 75th percentiles are similar. We also observe that the median AUC is slightly higher for Branch and Exception Coverage than for Branch alone at the 180 second budget, potentially indicating a slight improvement in the rate of coverage attainment.

In Table 5.6, we show the results of statistical testing on the AUC. A significant difference was found when comparing Branch alone versus Branch and Exception Coverage when the budget was set to 180 seconds. However, the effect size was negligible. All other p-values are considerably above the 0.05 threshold.

Figure 5.9 and Table 5.7 report the time needed to reach 25, 50, and 75% Branch Coverage. The median time to reach each landmark is very similar across all configurations, regardless of search budget. The largest differences between configurations can be seen in the 75th percentile for each configuration. Here, we often see a higher 75th percentile for Branch and Output, as compared to the higher configurations, indicating again that coverage attainment may be slightly slowed with the inclusion of Output

Criterion	Budget	Count	Average	Median
Time to 25%				
Branch	180	544	10.33	1.00
Branch & Exception	180	527	9.12	1.00
Branch & Execution Time	180	501	12.06	1.00
Branch & Output	180	512	12.45	1.00
Branch	300	551	19.00	1.00
Branch & Exception	300	513	17.37	1.00
Branch & Execution Time	300	479	19.76	1.00
Branch & Output	300	492	21.34	1.00
Time to 50%				
Branch	180	433	22.55	5.00
Branch & Exception	180	416	18.70	4.00
Branch & Execution Time	180	376	19.67	5.00
Branch & Output	180	391	20.74	5.00
Branch	300	448	35.64	5.00
Branch & Exception	300	417	34.98	5.00
Branch & Execution Time	300	378	33.36	5.00
Branch & Output	300	391	37.39	5.00
Time to 75%				
Branch	180	233	24.29	5.00
Branch & Exception	180	236	22.08	5.00
Branch & Execution Time	180	209	23.15	5.00
Branch & Output	180	217	33.29	5.00
Branch	300	244	41.32	5.00
Branch & Exception	300	231	35.94	5.00
Branch & Execution Time	300	222	43.28	5.00
Branch & Output	300	229	57.98	6.00

Table 5.7: Descriptive statistics on the time (in seconds) to reach coverage thresholds, split by configuration and budget. "Count" indicates the number of trials that reached this threshold.

Coverage as a goal. However, there is no evidence that this effect is significant.

While the median remains relatively consistent across search budgets, the 75% percentile is often higher at the 300 second budget, especially at the 75% coverage threshold. The average also raises across search budgets. This is due to a small number of additional test suites reaching these thresholds later in the generation process under the higher budget. Again, the median is relatively consistent across all budgets and configurations.

RQ1.3 (Coverage Evolution): There are almost no significant differences between configurations with regard to the rate of attainment of Branch Coverage. Branch and Exception Coverage at a 180 second budget shows statistically significant improvement, but with only a negligible effect size.

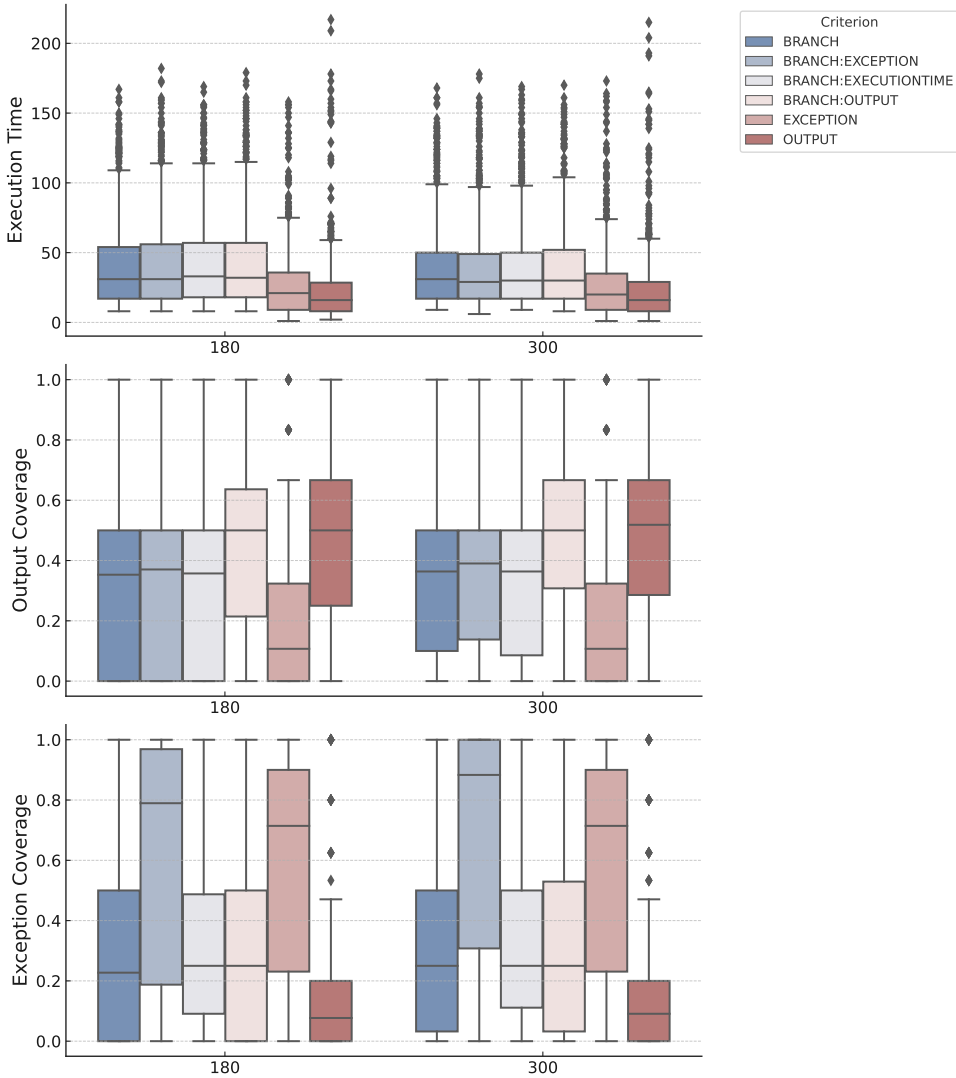


Figure 5.10: Boxplots for Execution time, Exception Count, and Output Coverage divided by budget and configuration.

5.4.2 Impact on Goal-Based Objectives (RQ2)

Our second hypothesis was the following:

> **Hypothesis 2:** Targeting both coverage and a goal-based fitness function will have an impact on the fault detection of generated test suites, as compared to targeting coverage or a goal-based fitness function alone.

Similar to the first hypothesis, targeting multiple objectives could affect the final fitness values of the goal-based fitness functions—e.g., raising or lowering goal attainment when compared to targeting a goal-based or a structure-based fitness

Criterion	Budget	Execution Time		Output Coverage		Exception Coverage	
		Avg	Median	Avg	Median	Avg	Median
Branch	180	41.73	31.00	0.31	0.35	0.29	0.22
Branch & Exception	180	42.69	31.00	0.32	0.37	0.62	0.79
Branch & Execution Time	180	42.90	33.00	0.31	0.36	0.31	0.25
Branch & Output	180	42.89	31.00	0.43	0.49	0.30	0.25
Exception	180	27.48	21.00	0.19	0.11	0.59	0.71
Output	180	23.13	16.00	0.45	0.50	0.13	0.08
Branch	300	39.83	30.00	0.33	0.36	0.30	0.25
Branch & Exception	300	39.11	28.00	0.35	0.39	0.68	0.88
Branch & Execution Time	300	40.37	30.00	0.33	0.36	0.32	0.25
Branch & Output	300	40.38	30.00	0.46	0.50	0.32	0.25
Exception	300	26.87	20.00	0.19	0.11	0.59	0.71
Output	300	23.57	16.00	0.47	0.52	0.14	0.09

Table 5.8: Averages and median values for Execution Time, Output Coverage, and Exception Count, by configuration and budget.

Comparison	Budget	P-value	Effect Size (Cohen's d)	Category
Branch vs Branch & Exception	180	2.03×10^{-68}	-0.99	Large
Branch vs Exception	180	3.85×10^{-80}	-0.98	Large
Branch & Exception vs Exception	180	0.057	-	-
Branch vs Branch & Exception	300	5.59×10^{-79}	-1.14	Large
Branch vs Exception	300	2.53×10^{-69}	-0.91	Large
Branch & Exception vs Exception	300	3.32×10^{-10}	0.24	Small

Table 5.9: Significance tests and effect size (when significant) for comparisons of **Exception Count** between single and multi-objective configurations across different budgets.

function alone.

In addition to examining this hypothesis directly, there is a secondary hypothesis of interest. One of the reasons for the prevalence of structural coverage in search-based test generation is that structural coverage can be translated effectively into distance-based fitness functions, e.g., the Branch Distance used for optimizing Branch Coverage [66]. This means that tests can be efficiently generated that widely explore the codebase. Goal-based fitness functions often lack distance-based fitness functions [20]. Consequently, they may offer less feedback to the optimization process. As a result, targeting both coverage and goal-based objectives could potentially result in higher attainment of goal-based fitness by offering an additional feedback mechanism [3, 20]. Past research has not assessed this hypothesis. In this experiment, the Exception Count is one such example. In contrast, Output Coverage does have a distance-based fitness function, so such benefits may not be observed in this case.

During the experiment, we recorded the final attainment of each goal-based fitness function for all generated test suites. Note that the Execution Time fitness function cannot be executed without also targeting Branch Coverage, so we were unable to generate test suites targeting Execution Time alone. In addition, note that the Exception Count is normalized between 0–1 for all bugs, based on the largest number of exceptions seen in any trial for that bug, as the number of possible exceptions differs between bugs.

Figure 5.10 shows boxplots for Exception Count, Output Coverage, and Execution Time for each fitness function configuration and search budget. Average and median values are reported in Table 5.8. Finally, Tables 5.9–5.11 report p-values and effect

Comparison	Budget	P-value	Effect Size (Cohen's d)	Category
Branch vs Branch & Output	180	2.41×10^{-25}	-0.47	Small
Branch vs Output	180	1.67×10^{-37}	-0.56	Medium
Branch & Output vs Output	180	0.016	-	-
Branch vs Branch & Output	300	9.47×10^{-26}	-0.50	Medium
Branch vs Output	300	2.12×10^{-34}	-0.54	Medium
Branch & Output vs Output	300	0.2	-	-

Table 5.10: Significance tests and effect size (when significant) for comparisons of **Output Coverage** between single and multi-objective configurations across different budgets.

Comparison	Budget	P-value	Effect Size (Cohen's d)	Category
Branch vs Branch & Execution Time	180	0.304	-	-
Branch vs Branch & Execution Time	300	0.882	-	-

Table 5.11: Significance tests and effect size (when significant) for comparisons of **Execution Time** between single and multi-objective configurations across different budgets.

sizes (when appropriate) for comparisons between single-objective generation versus multi-objective optimization.

First, we observe that no goal-based fitness function can serve as a proxy for another goal-based fitness function. Targeting Output Coverage yields a low Exception Count and Execution Time. Similarly, targeting Exception Coverage yields low Output Coverage and Execution Time. If one targets a goal-based fitness function *alone*, they should not expect high attainment of goals other than the one that function was designed for.

Targeting Branch Coverage *alone* yields better performance at each goal than targeting a fitness function designed for a different goal, suggesting that coverage of the code base will always lead to *some* degree of goal attainment. However, these suites are also significantly worse at attaining Output Coverage or Exception Count than targeting either goal directly or targeting multiple objectives. In other words, code coverage is also a weak proxy for a goal-based fitness function—as noted in Section 2.1, coverage alone is not enough to ensure goal attainment.

RQ2 (Goal Coverage): Targeting code coverage *alone* leads to worse goal attainment than directly targeting a goal-based objective.

Table 5.10 shows that, at both budgets, there is no significant difference in Output Coverage between targeting Output Coverage alone and targeting both Branch and Output Coverage. Similarly, Table 5.9 shows that, at the 180 second budget, targeting both Branch Coverage and the Exception Count performs no worse than targeting the Exception Count alone.

However, at the 300 second budget, there is a small improvement in Exception Count when targeting both Branch Coverage and the Exception Count. This confirms our prior observation that there are situations where both Branch Coverage and the Exception Count offer the other missing feedback—with the Exception Count steering Branch Coverage towards input that triggers exceptions and Branch Coverage offering

feedback on how to further explore the code base [3]. These situations are not universal, but—at the higher search budget—they do seem to exist.

We were unable to generate suites targeting Execution Time alone due to limitations in the implementation. However, from Table 5.8, we can see that there is a slight improvement in the average (at both budgets) and the median (at the 180 second budget) Execution Time from targeting Branch and Execution Time simultaneously. However, there are no statistically significant differences between targeting both objectives versus targeting Branch Coverage alone (Table 5.11). No configuration was significantly better at yielding tests with high execution times. It is possible that the examples chosen from Defects4J had few or no performance issues that could be exposed through unit testing.

RQ2 (Goal Coverage): Targeting code coverage and a goal-based objective simultaneously results in no reduction in goal-based fitness compared to targeting a goal-based objective alone, and can lead to improvements in some situations—as witnessed with Exception Count.

5.4.3 Impact on Fault Detection (RQ3)

The third hypothesis that we raised was the following:

Hypothesis 3: Targeting both coverage and a goal-based fitness function will have an impact on the fault detection of generated test suites, as compared to targeting coverage or a goal-based fitness function alone.

Regardless of the impact on the code coverage or attainment of non-coverage testing goals, targeting multiple objectives could change the specific inputs applied to the class-under-test. As a result, there could be a change to the fault-revealing power of those test suites—either increased due to a change in the versatility of the test suite [3, 93, 190] or, even, a potential decrease.

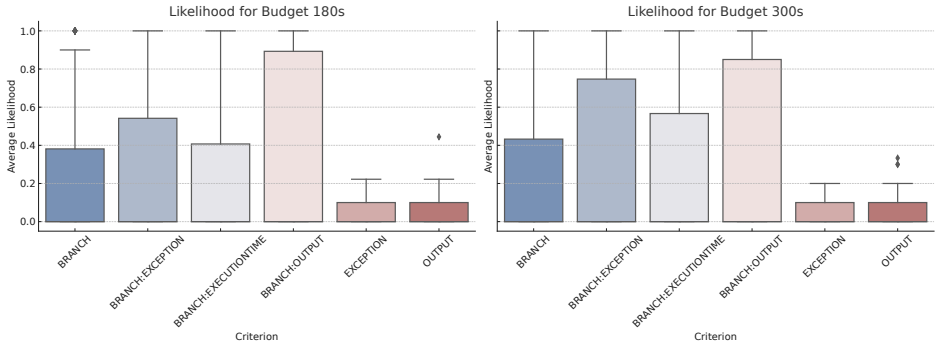


Figure 5.11: Boxplots of the **likelihood of fault detection**, divided by budget and configuration.

Comparison	Budget	p-values	Effect Size (Cohen's d)	Category
Branch vs Branch & Exception	180	0.657	-	-
Branch vs Branch & Execution Time	180	0.895	-	-
Branch vs Branch & Output	180	0.457	-	-
Branch vs Exception	180	6.39×10^{-5}	0.668	Medium
Branch vs Output	180	1.72×10^{-4}	0.625	Medium
Branch & Exception vs Exception	180	8.32×10^{-5}	0.792	Medium
Branch & Output vs Output	180	5.60×10^{-5}	0.810	Large
Branch vs Branch & Exception	300	0.384	-	-
Branch vs Branch & Execution Time	300	0.612	-	-
Branch vs Branch & Output	300	0.461	-	-
Branch vs Exception	300	1.41×10^{-5}	0.729	Medium
Branch vs Output	300	6.07×10^{-5}	0.669	Medium
Branch & Exception vs Exception	300	6.28×10^{-6}	0.937	Large
Branch & Output vs Output	300	2.92×10^{-5}	0.838	Large

Table 5.12: Significance tests and effect size (when significant) for comparisons of **likelihood of fault detection** between single and multi-objective configurations across different budgets.

Criterion	Budget	Min	25%	50%	75%	Max	Avg	# of bugs detected
Branch	180	0.00	0.00	0.00	0.38	1.00	0.24	28
Branch & Exception	180	0.00	0.00	0.00	0.54	1.00	0.27	30
Branch & Execution Time	180	0.00	0.00	0.00	0.41	1.00	0.25	34
Branch & Output	180	0.00	0.00	0.00	0.89	1.00	0.28	35
Exception	180	0.00	0.00	0.00	0.10	0.22	0.04	23
Output	180	0.00	0.00	0.00	0.10	0.44	0.09	27
Branch	300	0.00	0.00	0.00	0.43	1.00	0.26	32
Branch & Exception	300	0.00	0.00	0.00	0.75	1.00	0.30	35
Branch & Execution Time	300	0.00	0.00	0.00	0.57	1.00	0.28	39
Branch & Output	300	0.00	0.00	0.00	0.85	1.00	0.31	33
Exception	300	0.00	0.00	0.00	0.10	0.20	0.06	28
Output	300	0.00	0.00	0.00	0.10	0.33	0.13	31

Table 5.13: Descriptive statistics on the **likelihood of fault detection**, split by budget and configuration.

To assess this hypothesis, we consider two aspects of fault detection. First, the *likelihood of fault detection*—for each fault, the proportion of suites that detect the

fault to those generated. Second, we consider the *number of failing tests*—how many test cases detect the fault when it is detected. We consider both so that we can examine both how likely a fault is to be detected and how much information exists to understand and debug the fault. If two configurations have the same likelihood of detection, one may offer more failing tests to use in the debugging process.

Figure 5.11 illustrates the likelihood of fault detection, and Table 5.13 offers descriptive statistics for each configuration and search budget. Table 5.13 also lists the number of faults detected over the set of 93 considered in this experiment. Immediately, we can see that the multi-objective configurations detect *more* faults—with Branch and Output detecting the most at the 180 second budget and Branch and Execution Time detecting the most at the 300 second budget. The multi-objective configurations are followed by Branch Coverage, then Output Coverage, then the Exception Count.

RQ3 (Fault Detection): Suites targeting multi-objective configurations detected more faults than single-objective configurations. Suites targeting Branch Coverage alone detected more faults than suites targeting goal-based objectives.

However, as the majority of faults are never detected, the median likelihood of fault detection is also zero for all configurations. The average, skewed by cases where faults are detected, is more informative. Targeting Branch and Output Coverage yields the highest average likelihood of fault detection at both search budgets (28 and 31%), followed at both budgets by Branch and Exception Count (27 and 30%) and Branch and Execution Time (25 and 28%). This same ordering can be seen in the 75th percentile in Figure 5.11. Again, targeting the goal-based functions alone yields the lowest average likelihood of fault detection—with the worst performance from targeting the Exception Count alone.

Table 5.12 includes significance tests and effect sizes (when distributions are found to be significantly different) for the likelihood of fault detection. At both budgets, the multi-objective configurations do *not* yield significantly different results from targeting Branch Coverage alone in the likelihood of fault detection. However, targeting Branch Coverage yields better results (with medium effect size) than targeting a goal-based fitness function. Targeting Branch and Exception Count simultaneously yields better results than targeting Exception Count alone, with medium effect size at the 180 second budget and large effect size at the 300 second budget. Finally, targeting Branch Coverage and Output Coverage yields better results than targeting Output Coverage alone, with large effect sizes at both budgets.

RQ3 (Fault Detection): Suites targeting Branch Coverage alone or a multi-objective configuration outperform suites targeting a goal-based objective in the likelihood of fault detection with medium–large effect size.

RQ3 (Fault Detection): Suites targeting a multi-objective configuration fail to outperform suites targeting Branch Coverage alone with significance in the likelihood of fault detection. However, targeting a multi-objective configuration does increase the average and 75th percentile performance.

Criterion	Budget	Avg	Min	25%	50%	75%	Max
Branch	180	0.89	0.00	0.00	0.00	0.00	29.00
Branch & Exception	180	1.13	0.00	0.00	0.00	1.00	30.00
Branch & Execution Time	180	1.00	0.00	0.00	0.00	0.00	33.00
Branch & Output	180	0.88	0.00	0.00	0.00	1.00	37.00
Exception	180	0.04	0.00	0.00	0.00	0.00	1.00
Output	180	0.09	0.00	0.00	0.00	0.00	1.00
Branch	300	1.08	0.00	0.00	0.00	1.00	41.00
Branch & Exception	300	1.28	0.00	0.00	0.00	1.00	39.00
Branch & Execution Time	300	1.04	0.00	0.00	0.00	1.00	39.00
Branch & Output	300	0.98	0.00	0.00	0.00	1.00	38.00
Exception	300	0.06	0.00	0.00	0.00	0.00	1.00
Output	300	0.13	0.00	0.00	0.00	0.00	1.00

Table 5.14: Descriptive statistics on **number of failing tests**, split by budget and configuration.

Figure 5.12 illustrates the number of failing tests, and Table 5.14 offers descriptive statistics for each configuration and search budget. Table 5.18 includes significance tests and effect sizes, when significance is found.

Here we see largely similar trends to the likelihood of fault detection, with the median number of failing tests being 0 for all configurations. Targeting Branch and Exception Count yields the largest average number of failing tests at both budgets (1.13 and 1.28), followed by Branch and Execution Time (1.00) at the 180 second budget and Branch alone (1.08) at the 300 second budget. However, these results are in a relatively narrow range, and no multi-objective configuration is an outlier in terms of the number of tests that fail when a fault is detected. As shown in Table 5.18, targeting Branch Coverage alone or a multi-objective configuration yields a larger number of failing tests than targeting a goal-based objective alone, with small effect size.

RQ3 (Fault Detection): Suites targeting Branch Coverage alone or a multi-objective configuration outperform suites targeting a goal-based objective in the number of failing tests with small effect size.

RQ3 (Fault Detection): Suites targeting a multi-objective configuration fail to outperform suites targeting Branch Coverage alone with significance in the number of failing tests.

5.4.4 Impact on Test Suite Contents (RQ4)

Our fourth hypotheses was that:

Hypothesis 4: Targeting both coverage and a goal-based fitness function will have an impact on the size of the test suite and the average test length, as compared to targeting coverage or a goal-based fitness function alone.

In general, we would expect that targeting multiple objectives would increase the suite size or average test case length. Each unit test case contains one or more interactions with the class-under-test. Each targeted objective imposes a set of obligations

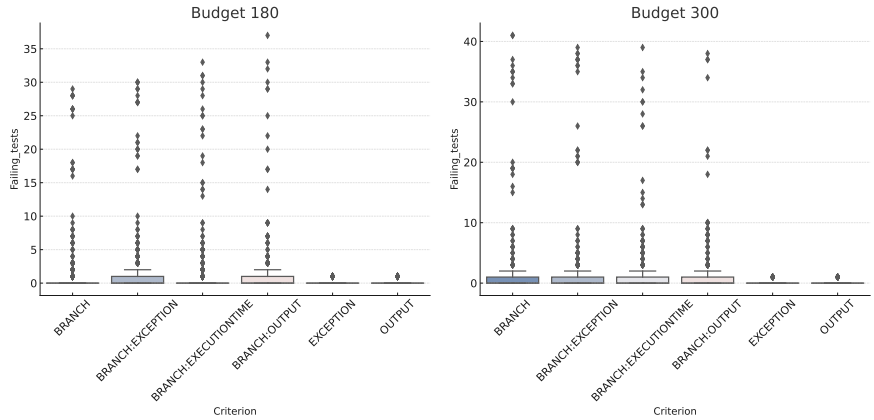


Figure 5.12: Boxplots of the **number of failing tests**, divided by budget and configuration.

Criterion	Budget	Suite Size		Test Case Length	
		Avg	Median	Avg	Median
Branch	180	18.36	14.00	34.51	21.43
Branch & Exception	180	23.45	19.00	33.48	21.38
Branch & Execution Time	180	19.76	15.00	35.12	21.82
Branch & Output	180	22.89	18.00	35.90	21.67
Exception	180	9.92	6.00	20.35	20.00
Output	180	10.23	5.00	21.76	19.75
Branch	300	20.23	14.00	32.35	21.29
Branch & Exception	300	25.64	19.00	31.06	21.19
Branch & Execution Time	300	22.41	16.00	34.88	21.73
Branch & Output	300	25.96	18.00	34.76	21.54
Exception	300	9.98	6.00	19.85	20.00
Output	300	10.65	6.00	20.45	19.75

Table 5.15: Average and median test suite size and average test case length, divided by budget and configuration.

that must be covered in those interactions, and only particular input will ensure those obligations are met. Naturally, then, multi-objective optimization imposes a larger set of obligations than single-objective optimization.

Each test case can cover obligations of multiple criteria, meaning that one should not expect a linear increase in suite size or test case length during multi-objective optimization compared to single-objective optimization. However, it is unlikely that there will be *no* increase. Some obligations require highly specific test input or setup, necessitating additional specialized test cases or an increased number of program interactions. Therefore, some increase in suite size, test length, or both is likely.

Figure 5.13 shows boxplots for the test suite size and average test case length, with Table 5.15 reporting median and average values for both. Tables 5.16 and 5.17 report the results of significance tests and effect sizes (when significance is found) for both measurements, comparing single and multi-objective configurations.

From Figure 5.13 and Table 5.16, we can immediately see that the distributions of test suite sizes vary significantly between configurations. Targeting code coverage and a goal-based fitness function simultaneously results in larger test suites than target-

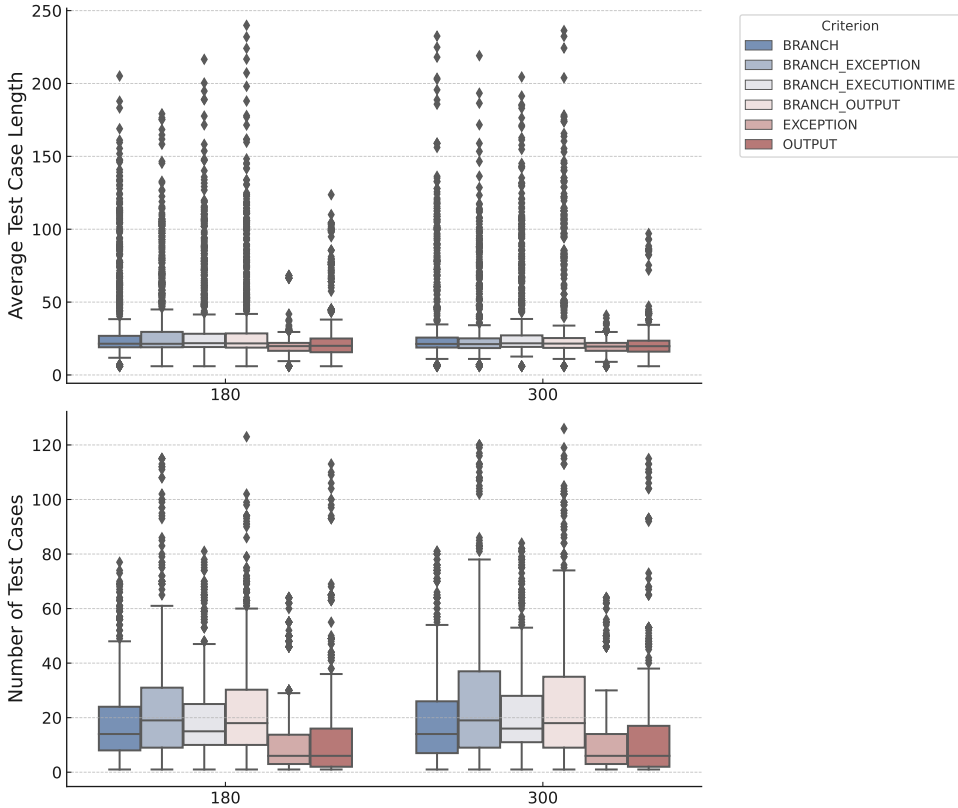


Figure 5.13: Boxplots for the test suite size and the average test case length, divided by budget and configuration.

ing either alone, with medium–large effect size compared to targeting a goal-based objective and negligible–small effect size compared to targeting Branch Coverage alone.

Figure 5.13 and Table 5.17 also show that the average test case length tends to increase with multi-objective optimization compared to targeting a goal-based objective alone, with small–medium effect size. However, the test length does not increase compared to targeting Branch Coverage alone—with only a negligible increase when targeting Branch and Execution Time.

Comparison	Budget	P-value	Effect Size (Cohen's d)	Category
Branch vs Branch & Exception	180	1.41×10^{-7}	-0.29	Small
Branch vs Branch & Output	180	1.95×10^{-6}	-0.26	Small
Branch vs Branch & Execution Time	180	0.011	-	-
Branch & Exception vs Exception	180	4.97×10^{-69}	0.84	Large
Branch & Output vs Output	180	2.38×10^{-54}	0.68	Medium
Branch vs Branch & Exception	300	4.35×10^{-7}	-0.27	Small
Branch vs Branch & Output	300	1.80×10^{-6}	-0.27	Small
Branch vs Branch & Execution Time	300	0.001	-0.13	Negligible
Branch & Exception vs Exception	300	9.21×10^{-67}	0.92	Large
Branch & Output vs Output	300	1.96×10^{-51}	0.73	Medium

Table 5.16: Significance comparisons and effect sizes (when significant) for **test suite size** between single and multi-objective configurations across different budgets.

Comparison	Budget	P-value	Effect Size (Cohen's d)	Category
Branch vs Branch & Exception	180	0.891	-	-
Branch vs Branch & Output	180	0.374	-	-
Branch vs Branch & Execution Time	180	0.490	-	-
Branch & Exception vs Exception	180	4.97×10^{-30}	0.63	Medium
Branch & Output vs Output	180	2.18×10^{-24}	0.45	Small
Branch vs Branch & Exception	300	0.691	-	-
Branch vs Branch & Output	300	0.3	-	-
Branch vs Branch & Execution Time	300	0.002	-0.09	Negligible
Branch & Exception vs Exception	300	4.30×10^{-26}	0.60	Medium
Branch & Output vs Output	300	6.61×10^{-21}	0.52	Medium

Table 5.17: Significance comparisons and effect sizes (when significant) for **average test case length** between single and multi-objective configurations across different budgets.

Comparison	Budget	p-values	Effect Size (Cohen's d)	Category
Branch vs Branch & Exception	180	0.043	-	-
Branch vs Branch & Execution Time	180	0.518	-	-
Branch vs Branch & Output	180	0.053	-	-
Branch vs Exception	180	2.96×10^{-37}	0.35	Small
Branch vs Output	180	7.43×10^{-31}	0.35	Small
Branch & Exception vs Exception	180	1.90×10^{-36}	0.41	Small
Branch & Output vs Output	180	4.16×10^{-35}	0.36	Small
Branch vs Branch & Exception	300	0.014	-	-
Branch vs Branch & Execution Time	300	0.124	-	-
Branch vs Branch & Output	300	0.012	-	-
Branch vs Exception	300	2.90×10^{-45}	0.33	Small
Branch vs Output	300	2.62×10^{-37}	0.33	Small
Branch & Exception vs Exception	300	2.29×10^{-38}	0.40	Small
Branch & Output vs Output	300	3.34×10^{-38}	0.41	Small

Table 5.18: Significance tests and effect size (when significant) for comparisons of **number of failing tests** between single and multi-objective configurations across different budgets.

Branch Coverage tends to have more obligations to cover than the Exception Count or Output Coverage, as a program will generally have more branches in control flow than output partitions or thrown exceptions. Covering the obligations of Branch Coverage requires more interactions with the class-under-test and requires that a larger number of specialized scenarios be set up and executed compared to a goal-based

objective alone, increasing both suite size and test length.

There is a larger increase between Branch and Exception Count and Exception Count alone in both suite size and test length than between Branch and Output Coverage and Output Coverage alone. This is because the Exception Count depends on the number of exceptions discovered, which—in almost all cases—will be fewer than the number of required output partitions for the methods of the class-under-test. Further, tests that trigger exceptions may not achieve high coverage, as the execution path will end when the exception is triggered. If an exception is triggered early in the execution of a particular method, few coverage obligations will be achieved.

RQ4 (Test Suite Contents): Both the test suite size and average test length increase with multi-objective optimization compared to when a goal-based criterion is targeted alone. Branch Coverage tends to impose more obligations than goal-based objectives, leading to the increase.

There is only a small increase in test suite size—and no increase in average test length—between Exception Count and Branch Coverage and Output Coverage and Branch Coverage versus Branch Coverage alone. As discussed above, the number of obligations for the goal-based objectives is small compared to the number for Branch Coverage, so only a small increase in suite size would be expected.

RQ4 (Test Suite Contents): Targeting Exception or Output Coverage in addition to Branch Coverage leads to a small increase in test suite size compared to targeting Branch Coverage alone. However, there is no increase in test case length.

We see no or negligible increase in suite size and test length between Branch Coverage and Branch and Execution Time. The Execution Time fitness function differed from the others in that it had no "obligations". Rather, the goal was simply to find the maximum execution time for a test suite during the generation process. Therefore, one would not expect a significant impact on the test suite size. Some impact on test case length would be reasonable, however, as increasing the number of interactions will increase the execution time. That said, the fitness function imposed a high penalty on test case length to prevent the generation of bloated test cases. Further, as shown in Table 5.11, actual attainment of the Execution Time goal was limited.

RQ4 (Test Suite Contents): Targeting Execution Time in addition to Branch Coverage leads to no or negligible change in suite size or test case length compared to targeting Branch Coverage alone.

5.4.5 Impact of Search Budget (RQ5)

Our final hypothesis was the following:

Hypothesis 5: An increase in the search budget may lead to increased attainment of each objective, but will not change the fundamental relationships assessed in the previous hypotheses.

We would expect that an increased search budget would increase the resulting attainment of each targeted objective. However, we also hypothesized that the relative relationships between single and multi-objective optimization will not fundamentally differ. This hypothesis largely held true.

With regard to attained Branch Coverage, the increased search budget led to higher coverage attainment (Table 5.4) and more suites reaching particular coverage thresholds (Table 5.7). However, this increase is approximately consistent across configurations, regardless of the targeted fitness functions. The same trends between configurations generally held at both search budgets with regard to total attained coverage, the particular obligations covered, and the rate of coverage attainment.

With regard to coverage of goal-based objectives, an increased search budget led to slightly higher median attainment of Exception Count and Output Coverage (Table 5.8. However, again, the same general trends were witnessed in comparisons of multi-objective and single-objective generation at both budgets for the most part. Two exceptions emerged (Tables 5.9 and 5.10). First, at a higher budget, targeting Branch and Exception yielded significantly better Exception Count than targeting Branch alone (when there was no significant difference at the lower budget). Second, a negligible difference at the lower search budget between targeting Branch and Output and targeting Output Coverage alone in terms of the achieved Output Coverage disappeared at a higher budget.

With regard to fault detection, the number of faults detected increased with the search budget. In addition, we see that the average and 75th percentile likelihood of fault detection also increased with the search budget (Table 5.13)—as well as the average number of failing tests (Table 5.14). The general relationships between single and multi-objective configurations held, except that some effect sizes increased at the higher budget (Table 5.12).

Finally, an increased search budget generally led to little-to-no change in the median test suite size or test case length—however, there was a minor increase in the average suite size (Table 5.15). The observations with regard to single versus multi-objective generation held across budgets, with small amplifications at the larger search budget (e.g., an increased effect size for Branch and Output versus Output alone at a 300 second budget for the average test case length).

RQ5 (Search Budget): An increased search budget leads to increased Branch Coverage, goal attainment, and fault detection, but does not substantially affect test suite size and average test length.

RQ5 (Search Budget): An increased search budget generally does not fundamentally change—but may increase the effect size of—relationships between single and multi-objective optimization.

5.5 Discussion

5.5.1 Assessment of Hypotheses

Our study assessed five hypotheses about the relationships between coverage-directed test generation, goal-directed test generation, and multi-objective optimization targeting both coverage and testing goals. Here, we summarize our findings with regard to these hypotheses.

> **Hypothesis 1:** The inclusion of goal-based fitness functions as additional generation targets will have an impact on the attainment of code coverage, as compared to targeting coverage alone.

Ultimately, our observations **refute this hypothesis**. We found that adding a second goal-based fitness function does not have a significant impact on the final Branch Coverage attained by test suites. Further, in the majority of cases, the addition of a goal-based fitness function does not change the likelihood of covering particular test obligations. Almost all differences in the average coverage of individual obligations are within 10% of when Branch Coverage is targeted alone. Finally, there are almost no significant differences between configurations with regard to the rate of attainment of Branch Coverage.

> **Hypothesis 2:** Targeting both coverage and a goal-based fitness function will have an impact on the attainment of goal-based fitness functions, as compared to targeting coverage or a goal-based fitness function alone.

Our observations **partially confirm this hypothesis**. We observed that targeting code coverage *alone* leads to worse goal attainment than directly targeting a goal-based objective, adding evidence to our previous observations [3] that coverage is a prerequisite for goal attainment but does not guarantee attainment.

We observed that targeting code coverage and a goal-based objective simultaneously results in no reduction in goal-based fitness compared to targeting a goal-based objective alone, and can lead to improvements in some situations—as witnessed with Exception Count. As suggested in our previous work [3, 190], the addition of Branch Coverage can offer feedback that leads to the discovery of more exceptions. Similar benefits are not offered to Output Coverage, as this function has a more distance-based fitness function.

> **Hypothesis 3:** Targeting both coverage and a goal-based fitness function will have an impact on the fault detection of generated test suites, as compared to targeting coverage or a goal-based fitness function alone.

Our observations **partially confirm this hypothesis**. Suites targeting multi-objective configurations detected more faults than single-objective configurations. Suites targeting Branch Coverage alone detected fewer faults than multi-objective configurations, but they did detect more faults than suites targeting goal-based objectives.

In addition, suites targeting Branch Coverage alone or a multi-objective configuration outperform suites targeting a goal-based objective in the likelihood of fault

detection with medium–large effect size. However, suites targeting a multi-objective configuration fail to outperform suites targeting Branch Coverage alone with significance in the likelihood of fault detection. Targeting a multi-objective configuration does increase the average and 75th percentile performance.

These findings reinforce our previous observations [3, 190] that coverage is needed to discover faults but does not guarantee the selection of the specific inputs needed to trigger a failure. Targeting coverage yields more failures than only targeting testing goals. Targeting a goal *in addition* to coverage resulted in the discovery of more faults than coverage alone by biasing the test input used in the generated suite. However, many faults still remain undetected. Future research should consider additional goal-based fitness functions, and aim to discover which functions can best shape coverage towards an increased likelihood of fault detection.

Hypothesis 4: Targeting both coverage and a goal-based fitness function will have an impact on the size of the test suite and the average test length, as compared to targeting coverage or a goal-based fitness function alone.

Our observations **partially confirm this hypothesis**. Both the test suite size and average test length increase with multi-objective optimization compared to when a goal-based criterion is targeted alone. Branch Coverage tends to impose more obligations than goal-based objectives, leading to the increase.

Targeting Exception or Output Coverage in addition to Branch Coverage leads to a small increase in test suite size compared to targeting Branch Coverage alone. However, there is no increase in test case length. Targeting Execution Time in addition to Branch Coverage leads to no or negligible change in suite size or test case length compared to targeting Branch Coverage alone, as the Execution Time fitness function does not have multiple distinct test obligations.

Hypothesis 5: An increase in the search budget may lead to increased attainment of each objective, but will not change the fundamental relationships assessed in the previous hypotheses.

Our observations **confirm this hypothesis**. An increased search budget leads to increased Branch Coverage, goal attainment, and fault detection, but does not substantially affect test suite size and average test length. At the same time, an increased search budget does not fundamentally change—but may amplify—relationships between single and multi-objective optimization.

5.6 Threats to Validity

External Validity For this study, we focused on case examples of real faults from the Defects4J dataset. The use of this dataset introduces certain threats to external validity. First, the faults used in the study represent only 14 Java projects. This is a relatively small number of projects. Nevertheless, we believe that Defects4J offers enough case examples that our results are generalizable to, at minimum, other small to medium-sized Java projects. Further, as Defects4J is used extensively in search-based test generation research [25], the use of Defects4J examples enables comparisons of our results with other research, and eases replication.

The set of specific faults used from the Defects4J dataset may also introduce selection bias, as certain types of faults or certain projects may be overrepresented or underrepresented. While we lacked experimental resources to consider all faults, we worked to ensure that we drew a proportional sample. We initially selected 206 faults, then retained 93 faults in the final experiment. This set remains large enough to offer a broad range of case examples and we do not believe that any single project is overrepresented.

We have based our research on a single test generation framework, EvoSuite. There are many search-based methods of generating tests and these methods may yield different results. Unfortunately, no other generation framework offers the same variety of fitness functions, particularly goal-based fitness functions. Therefore, a more thorough comparison of tools cannot be made at this time. In addition, by focusing on a single generation framework, we ensure that all test suites are compared in a controlled and fair manner.

Within EvoSuite, we also only employed one multi-objective algorithm, whole suite generation. Other algorithms may yield different results, as search objectives may be targeted through different mechanisms. We chose this algorithm to enable comparison to past research, and chose to focus on a single algorithm to perform a focused and detailed analysis of the data collected. We believe that the general trends observed would hold regardless of algorithm, even if specific results varied. In future work, we will consider the influence of algorithm more closely.

Internal Validity Evolutionary algorithms inherently introduce randomness, affecting result consistency. To mitigate this, we conducted multiple trials, aiming to average out randomness and stabilize outcomes. To control experiment cost, we only generated ten test suites for each class, budget, and fitness function configuration. A larger number of repetitions may yield different results. However, given the consistency of our results, we believe that this is a sufficient number of trials to draw stable conclusions from.

Conclusion Validity Conclusion validity depends on our choice of statistical tests and the assumptions underlying those tests. Data segmentation allowed for targeted analysis of different budget and fitness function configurations, with descriptive statistics and box plots providing an initial overview that could be used to validate the results of statistical analyses. We have favored non-parametric methods, as distribution characteristics were not known a priori, and normality cannot be assumed.

5.7 Conclusion

Past research has suggested the potential benefit of blending code coverage and goal-based fitness functions. While multi-objective generation has been previously studied, how these objectives interact—and, in particular, the interaction between coverage and goal-based fitness functions—has not been studied in depth. Therefore, in this study, we assessed and explored five hypotheses about this interaction and its effects on code coverage, goal attainment, fault detection, the size of the test suite, the length of test cases, and the impact of the search budget.

Ultimately, our observations suggest that there are more benefits than drawbacks in targeting multiple objectives over a single objective. Targeting multiple objectives

does not reduce code coverage, and—in some cases—can increase goal attainment. At the same time, targeting multiple objectives can lead to the detection of more faults and a higher average likelihood of fault detection. Multi-objective optimization does lead to larger test suites, but imposes only a small increase over suites targeting code coverage alone, and test case length is not significantly increased.

The benefits of multi-objective optimization are often more limited than hypothesized in past research, but are still sufficient to recommend multi-objective optimization over targeting coverage or testing goals alone. Our study offers insight into how coverage and goal-based objectives interact during multi-objective test generation, offering guidance to researchers and testers and a starting point for future research on multi-objective test generation.

In future work, we would like to continue to explore these—and other—hypotheses with an expanded scope and and consideration of additional experimental variables. We will target a wider variety of projects and faults, and will also vary the metaheuristic algorithms used to perform multi-objective generation (e.g., contrasting whole suite generation and DynaMOSA). In addition, we will consider combinations of more than two fitness functions. Our past research found that EvoSuite's default combination of eight fitness functions performed worse at fault detection than simply targeting Branch Coverage under the same budget, as competing objectives and overhead of calculating fitness limited test suite evolution. However, a subset of more than two and less than eight functions may yield highly effective results.

5.8 Acknowledgments

This research was supported by Vetenskapsrådet grants 2019-05275 and 2020-05272. Computing resources were provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by Vetenskapsrådet grant agreement 2022-06725.

Bibliography

[1] M. Pezze and M. Young, *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.

[2] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, pp. 105–156, 2004.

[3] A. Salahirad, H. Almulla, and G. Gay, "Choosing the fitness function for the job: Automated generation of test suites that detect real faults," *Software Testing, Verification and Reliability*, vol. 29, no. 4-5, p. e1701, 2019, e1701 stvr.1701. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1701

[4] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "The risks of coverage-directed test case generation," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, 2015.

[5] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[6] V. H. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. Dias, and M. P. Guimaraes, "Machine learning applied to software testing: A systematic mapping study," *IEEE Transactions on Reliability*, vol. 68, no. 3, pp. 1189–1212, 2019.

[7] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[8] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[9] M. Aniche, *Effective Software Testing: A developer's guide*. Simon and Schuster.

[10] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *Software Engineering, IEEE Transactions on*, vol. 36, no. 6, pp. 742–762, 2010.

[11] K. Naik and P. Tripathy, *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[12] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[13] A. Groce, M. A. Alipour, and R. Gopinath, "Coverage and its discontents," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward!'14. New York, NY, USA: ACM, 2014, pp. 255–268. [Online]. Available: http://doi.acm.org/10.1145/2661136.2661157

[14] K. Li and Y. Zhou, "Large language models as test case generators: Performance evaluation and enhancement," *arXiv preprint arXiv:2404.13340*, 2024.

[15] W. C. Ouédraogo, K. Kaboré, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyandé, "Large-scale, independent and comprehensive study of the power of llms for test case generation," *arXiv preprint arXiv:2407.00225*, 2024.

[16] S. Luo, H. Xu, Y. Bi, X. Wang, and Y. Zhou, "Boosting symbolic execution via constraint solving time prediction (experience paper)," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA. New York, NY, USA: Association for Computing Machinery, p. 336–347.

[17] N. Walkinshaw and G. Fraser, "Uncertainty-driven black-box test data generation," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 253–263.

[18] S. L. Shrestha, "Automatic generation of simulink models to find bugs in a cyber-physical system tool chain using deep learning," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 110–112. [Online]. Available: https://doi.org/10.1145/3377812.3382163

[19] H. Almulla and G. Gay, "Learning how to search: Generating exception-triggering tests through adaptive fitness function selection," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, Oct 2020, pp. 63–73.

[20] H. Almulla and G. Gay, "Learning how to search: generating effective test cases through adaptive fitness function selection," vol. 27, no. 2, p. 38, 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10048-8

[21] N. Jha and R. Popli, "Artificial intelligence for software testing-perspectives and practices," in *International Conference on Computational Intelligence and Communication Technologies (CCICT)*, pp. 377–382.

[22] C. Ioannides and K. I. Eder, "Coverage-directed test generation automated by machine learning – a review," vol. 17, no. 1.

[23] J. M. Balera and V. A. de Santiago Júnior, "A systematic mapping addressing hyper-heuristics within search-based software testing," *Information and Software Technology*, vol. 114, pp. 176–189, 2019.

[24] H. Almulla and G. Gay, "Generating diverse test suites for gson through adaptive fitness function selection," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12420 LNCS, pp. 246–252, 2020, cited By 0. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85092933212&doi=10.1007%2f978-3-030-59762-7_18&partnerID=40&md5=f1ae2eee34d85dd191295cd2ed4ee57a

[25] G. Gay and R. Just, "Defects4j as a challenge case for the search-based software engineering community," in *Search-Based Software Engineering*, A. Aleti and A. Panichella, Eds. Cham: Springer International Publishing, 2020, pp. 255–261.

[26] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)—Software Engineering in Practice Track (SEIP)*, ser. ICSE 2017. New York, NY, USA: ACM, 2017.

[27] A. Orso and G. Rothermel, "Software testing: A research travelogue (2000–2014)," in *Proceedings of the on Future of Software Engineering*, ser. FOSE 2014. New York, NY, USA: ACM, 2014, pp. 117–132. [Online]. Available: http://doi.acm.org/10.1145/2593882.2593885

[28] V. Dunjko and H. J. Briegel, "Machine learning & artificial intelligence in the quantum domain: a review of recent progress," *Reports on Progress in Physics*, vol. 81, no. 7, p. 074001, 2018.

[29] E. Alpaydin, *Introduction to machine learning*. MIT press, 2020.

[30] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning, Second Edition An Introduction*, 2018.

[31] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[32] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[33] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "Automated oracle data selection support," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[34] B. Hardin and U. Kanewala, "Using semi-supervised learning for predicting metamorphic relations," in *Proceedings of the 3rd International Workshop on Metamorphic Testing*, ser. MET '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 14–17. [Online]. Available: https://doi.org/10.1145/3193977.3193985

[35] D. J. Richardson, S. L. Aha, and T. O'Malley, "Specification-based test oracles for reactive systems," in *Proc. of the 14th Int'l Conf. on Software Engineering*. Springer, May 1992, pp. 105–118.

[36] R. Braga, P. S. Neto, R. Rabêlo, J. Santiago, and M. Souza, "A machine learning approach to generate test oracles," in *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, ser. SBES '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 142–151. [Online]. Available: https://doi.org/10.1145/3266237.3266273

[37] F. Gholami, N. Attar, H. Haghighi, M. V. Asl, M. Valueian, and S. Mohamadyari, "A classifier-based test oracle for embedded software," in *2018 Real-Time and Embedded Systems and Technologies (RTEST)*, May 2018, pp. 104–111.

[38] W. Makondo, R. Nallanthighal, I. Mapanga, and P. Kadebu, "Exploratory test oracle using multi-layer perceptron neural network," in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Sep. 2016, pp. 1166–1171.

[39] S. Shahamiri, W. Wan Kadir, and S. Bin Ibrahim, "An automated oracle approach to test decision-making structures," vol. 5, 2010, pp. 30–34, cited By 10. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-77958584496&doi=10.1109%2fICCSIT.2010.5563989&partnerID=40&md5=732483e9576df4cfb81151cf2f666730

[40] K. K. Aggarwal, Y. Singh, A. Kaur, and O. P. Sangwan, "A neural net based approach to test oracle," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 3, p. 1–6, May 2004. [Online]. Available: https://doi.org/10.1145/986710.986725

[41] J. Ding and D. Zhang, "A machine learning approach for developing test oracles for testing scientific software," vol. 2016-January, 2016, pp. 390–395, cited By 4. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84988431007&doi=10.18293%2fSEKE2016-137&partnerID=40&md5=ab9dd29e1f7369a3a2e41933a169d76e

[42] H. Jin, Y. Wang, N. Chen, Z. Gou, and S. Wang, "Artificial neural network for automatic test oracles generation," in *2008 International Conference on Computer Science and Software Engineering*, vol. 2, Dec 2008, pp. 727–730.

[43] A. Monsefi, B. Zakeri, S. Samsam, and M. Khashehchi, "Performing software test oracle based on deep neural network with fuzzy inference system," *Communications in Computer and Information Science*, vol. 891, pp. 406–417, 2019, cited By 0. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85075817713&doi=10.1007%2f978-3-030-33495-6_31&partnerID=40&md5=e74f5939bdd085a046c2f882260287fa

[44] O. P. Sangwan, P. K. Bhatia, and Y. Singh, "Radial basis function neural network based approach to test oracle," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, p. 1–5, Sep. 2011. [Online]. Available: https://doi.org/10.1145/2020976.2020992

[45] S. Shahamiri, W. Kadir, S. Ibrahim, and S. Hashim, "An automated framework for software test oracle," *Information and Software Technology*, vol. 53, no. 7, pp. 774–788, 2011, cited By 31. [Online]. Available: https://www.scopus.com/

inward/record.uri?eid=2-s2.0-79955055107&doi=10.1016%2fj.infsof.2011.02.006&partnerID=40&md5=a86d6be1213fba799db8e24df632367c

[46] S. Shahamiri, W. Wan-Kadir, S. Ibrahim, and S. Hashim, "Artificial neural networks as multi-networks automated test oracle," *Automated Software Engineering*, vol. 19, no. 3, pp. 303–334, 2012, cited By 23. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84863642080&doi=10.1007%2fs10515-011-0094-z&partnerID=40&md5=92dcec011b1a06b3275252dd2d1449cf

[47] A. Singhal, A. Bansal, and A. Kumar, "An approach to design test oracle for aspect oriented software systems using soft computing approach," *International Journal of Systems Assurance Engineering and Management*, vol. 7, no. 1, pp. 1–5, 2016, cited By 1. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84957956401&doi=10.1007%2fs13198-015-0402-2&partnerID=40&md5=99e4f57c672249b63483b7fc06ded311

[48] M. Vanmali, M. Last, and A. Kandel, "Using a neural network in the software testing process," *International Journal of Intelligent Systems*, vol. 17, no. 1, pp. 45–62, 2002, cited By 63. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0036157249&doi=10.1002%2fint.1002&partnerID=40&md5=81a29cc673c51e8f659a1676551ec393

[49] Vineeta, A. Singhal, and A. Bansal, "Generation of test oracles using neural network and decision tree model," in *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, Sep. 2014, pp. 313–318.

[50] M. Ye, B. Feng, L. Zhu, and Y. Lin, "Neural networks based automated test oracle for software testing," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4234 LNCS - III, pp. 498–507, 2006, cited By 9. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-33750708220&partnerID=40&md5=858633bde3ccbdac0195bd004eb4141e

[51] R. Zhang, Y.-W. Wang, and M.-Z. Zhang, "Automatic test oracle based on probabilistic neural networks," *Advances in Intelligent Systems and Computing*, vol. 752, pp. 437–445, 2019, cited By 0. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85053258403&doi=10.1007%2f978-981-10-8944-2_50&partnerID=40&md5=81b1448d0ff5e3381e84522956705caa

[52] D. J. Hiremath, M. Claus, W. Hasselbring, and W. Rath, "Automated identification of metamorphic test scenarios for an ocean-modeling application," in *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, Aug 2020, pp. 62–63.

[53] U. Kanewala and J. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," 2013, pp. 1–10, cited By 30. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84893326644&doi=10.1109%2fISSRE.2013.6698899&partnerID=40&md5=947a8b49ea54dd44a9656fa5110480fa

[54] U. Kanewala, J. Bieman, and A. Ben-Hur, "Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels," *Software Testing Verification and Reliability*, vol. 26, no. 3, pp. 245–269, 2016, cited By 35. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84963615021&doi=10.1002%2fstvr.1594&partnerID=40&md5=89589cb8ce1bf35471174edb2e7e6df5

[55] A. Nair, K. Meinke, and S. Eldh, "Leveraging mutants for automatic prediction of metamorphic relations using machine learning," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, ser. MaLTeSQuE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–6. [Online]. Available: https://doi.org/10.1145/3340482.3342741

[56] P. Zhang, X. Zhou, P. Pelliccione, and H. Leung, "Rbf-mlmr: A multi-label metamorphic relation prediction approach using rbf neural network," *IEEE Access*, vol. 5, pp. 21 791–21 805, 2017.

[57] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proceedings of the First International Workshop on Software Test Output Validation*, ser. STOV '10. New York, NY, USA: ACM, 2010, pp. 1–4. [Online]. Available: http://doi.acm.org/10.1145/1868048.1868049

[58] L. Taylor and G. Nitschke, "Improving deep learning using generic data augmentation," 2017.

[59] E. Kocaguneli, T. Menzies, and J. W. Keung, "On the value of ensemble effort estimation," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1403–1416, 2012.

[60] L. L. Minku, "A novel online supervised hyperparameter tuning procedure applied to cross-company software effort estimation," *Empirical Software Engineering*, vol. 24, no. 5, pp. 3153–3204, 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09686-w

[61] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 10–13. [Online]. Available: https://doi.org/10.1145/3196398.3196473

[62] A. Developers, "Fundamentals of testing," https://developer.android.com/training/testing/fundamentals.

[63] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 643–653. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920

[64] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint*

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 830–840.

[65] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for python," vol. 28, no. 2.

[66] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability*, vol. 23, no. 2, pp. 119–147, 2013.

[67] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931054

[68] R. Hierons, "Comparing test sets and criteria in the presence of test hypotheses and fault domains," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 4, p. 448, 2002.

[69] S. Poulding and R. Feldt, "The automated generation of humancomprehensible xml test sets," in *Proc. 1st North American Search Based Software Engineering Symposium (NasBASE)*, 2015.

[70] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[71] A. Alsharif, G. M. Kapfhammer, and P. McMinn, "What factors make sql test cases understandable for testers? a human study of automatic test data generation techniques," in *International Conference on Software Maintenance and Evolution (ICSME 2019)*, pp. 437–448.

[72] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[73] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "An empirical cybersecurity evaluation of github copilot's code contributions," *arXiv preprint arXiv:2108.09293*, 2021.

[74] R. Feldt and F. Dobslaw, "Towards automated boundary value testing with program derivatives and search," in *Search-Based Software Engineering*, S. Nejati and G. Gay, Eds. Springer International Publishing, pp. 155–163.

[75] F. Dobslaw, F. G. de Oliveira Neto, and R. Feldt, "Boundary value exploration for software analysis," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 346–353.

[76] H. Almulla and G. Gay, "Learning how to search: Generating effective test cases through adaptive fitness function selection," vol. abs/2102.04822. [Online]. Available: https://arxiv.org/abs/2102.04822

[77] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, pp. 523–534.

[78] R. Feldt, S. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, pp. 223–233.

[79] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, pp. 222–232.

[80] F. G. D. O. Neto, R. Feldt, L. Erlenhov, and J. B. D. S. Nunes, "Visualizing test diversity to support test optimisation," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, pp. 149–158.

[81] A. Fontes and G. Gay, "Using machine learning to generate test oracles: A systematic literature review," in *International Workshop on Test Oracles*, ser. TORACLE. New York, NY, USA: Association for Computing Machinery, p. 1–10.

[82] W. B. Langdon, S. Yoo, and M. Harman, "Inferring automatic test oracles," in *Proceedings of the 10th International Workshop on Search-Based Software Testing*, ser. SBST '17. IEEE Press, 2017, p. 5–6.

[83] F. Tsimpourlas, G. Rooijackers, A. Rajan, and M. Allamanis, "Embedding and classifying test execution traces using neural networks," vol. 16, no. 3, pp. 301–316. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/sfw2.12038

[84] B. Marculescu, R. Feldt, R. Torkar, and S. Poulding, "An initial industrial evaluation of interactive search-based testing for embedded software," vol. 29, pp. 26–39.

[85] S. Huurman, X. Bai, and T. Hirtz, "Generating api test data using deep reinforcement learning," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 541–544. [Online]. Available: https://doi.org/10.1145/3387940.3392214

[86] R. Feldt and S. Poulding, "Finding test data with specific properties via metaheuristic search," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, pp. 350–359.

[87] S. Poulding and R. Feldt, "The automated generation of humancomprehensible xml test sets," in *Proc. 1st North American Search Based Software Engineering Symposium (NasBASE)*.

[88] J. Kim, M. Kwon, and S. Yoo, "Generating test input with deep reinforcement learning," in *Proceedings of the 11th International Workshop on Search-Based Software Testing*, ser. SBST '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 51–58. [Online]. Available: https://doi.org/10.1145/3194718.3194720

[89] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 540–550.

[90] W. He, R. Zhao, and Q. Zhu, "Integrating evolutionary testing with reinforcement learning for automated test generation of object-oriented software," *Chinese Journal of Electronics*, vol. 24, no. 1, pp. 38–45, 2015.

[91] C. Budnik, M. Gario, G. Markov, and Z. Wang, "Guided test case generation through ai enabled output space exploration," in *Proceedings of the 13th International Workshop on Automation of Software Test*, ser. AST '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 53–56. [Online]. Available: https://doi.org/10.1145/3194733.3194740

[92] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 1070–1073. [Online]. Available: https://doi.org/10.1109/ASE.2019.00104

[93] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *Search-Based Software Engineering*, ser. Lecture Notes in Computer Science, M. Barros and Y. Labiche, Eds. Springer International Publishing, 2015, vol. 9275, pp. 93–108. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-22183-0_7

[94] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, pp. 1–18, 2015.

[95] N. Majma and S. M. Babamir, "Software test case generation test oracle design using neural network," in *2014 22nd Iranian Conference on Electrical Engineering (ICEE)*, May 2014, pp. 1168–1173.

[96] S. Ariyurek, A. Betin-Can, and E. Surer, "Automated video game testing using synthetic and humanlike agents," vol. 13, no. 1, pp. 50–67.

[97] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 772–784.

[98] T. Ahmad, A. Ashraf, D. Truscan, and I. Porres, "Exploratory performance testing using reinforcement learning," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug 2019, pp. 156–163.

[99] D. Baumann, R. Pfeffer, and E. Sax, "Automatic generation of critical test cases for the development of highly automated driving functions," in *IEEE Vehicular Technology Conference (VTC)*, pp. 1–5.

[100] F. Bergadano, "Test case generation by means of learning techniques," *SIGSOFT Softw. Eng. Notes*, vol. 18, no. 5, p. 149–162, Dec. 1993. [Online]. Available: https://doi.org/10.1145/167049.167074

[101] P. Papadopoulos and N. Walkinshaw, "Black-box test generation from inferred models," in *Proceedings of the Fourth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, ser. RAISE '15. IEEE Press, 2015, p. 19–24.

[102] A. Sharma, V. Melnikov, E. Hüllermeier, and H. Wehrheim, "Property-driven testing of black-box functions," in *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*, ser. FormaliSE '22. New York, NY, USA: Association for Computing Machinery, p. 113–123. [Online]. Available: https://doi.org/10.1145/3524482.3527657

[103] P. Feldmeier and G. Fraser, "Neuroevolution-based generation of tests and oracles for games," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556939

[104] M. Utting, B. Legeard, F. Dadeau, F. Tamagnan, and F. Bouquet, "Identifying and generating missing tests using machine learning on execution traces," in *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, Aug 2020, pp. 83–90.

[105] M. Kıraç, B. Aktemur, H. Sözer, and C. Gebizli, "Automatically learning usage behavior and generating event sequences for black-box testing of reactive systems," *Software Quality Journal*, vol. 27, no. 2, pp. 861–883, 2019, cited By 0. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85059878995&doi=10.1007%2fs11219-018-9439-1&partnerID=40&md5=291bd6c82252e9f7c1b0f230e23ec10c

[106] R. Eidenbenz, C. Franke, T. Sivanthi, and S. Schoenborn, "Boosting exploratory testing of industrial automation systems with ai," pp. 362–371.

[107] K. Kikuma, T. Yamada, K. Sato, and K. Ueda, "Preparation method in automated test case generation using machine learning," in *Proceedings of the Tenth International Symposium on Information and Communication Technology*, ser. SoICT 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 393–398. [Online]. Available: https://doi.org/10.1145/3368926.3369679

[108] K. Ueda and H. Tsukada, "Accuracy improvement by training data selection in automatic test cases generation method," pp. 438–442.

[109] K. Meinke and H. Khosrowjerdi, "Use case testing: A constrained active machine learning approach," vol. 12740 LNCS, pp. 3–21.

[110] Y. Deng, G. Lou, X. Zheng, T. Zhang, M. Kim, H. Liu, C. Wang, and T. Y. Chen, "Bmt: Behavior driven development-based metamorphic testing for autonomous driving models," in *International Workshop on Metamorphic Testing (MET)*, pp. 32–36.

[111] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 132–142.

[112] D. Araiza-Illan, A. G. Pipe, and K. Eder, "Intelligent agent-based stimulation for testing robotic software in human-robot interactions," in *Proceedings of the 3rd Workshop on Model-Driven Robot Software Engineering*, ser. MORSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 9–16. [Online]. Available: https://doi.org/10.1145/3022099.3022101

[113] M. Buzdalov and A. Buzdalova, "Adaptive selection of helper-objectives for test case generation," in *2013 IEEE Congress on Evolutionary Computation*, June 2013, pp. 2245–2250.

[114] M. Esnaashari and A. H. Damia, "Automation of software test data generation using genetic algorithm and reinforcement learning," vol. 183, p. 115446.

[115] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1410–1421. [Online]. Available: https://doi.org/10.1145/3377811.3380399

[116] T. Shu, C. Wu, and Z. Ding, "Boosting input data sequences generation for testing efsm-specified systems using deep reinforcement learning," vol. 155, p. 107114. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922002233

[117] M. Veanes, P. Roy, and C. Campbell, "Online testing with reinforcement learning," in *Proceedings of the First Combined International Conference on Formal Approaches to Software Testing and Runtime Verification*, ser. FATES'06/RV'06. Berlin, Heidelberg: Springer-Verlag, p. 240–253. [Online]. Available: https://doi.org/10.1007/11940197_16

[118] Z. Gao, W. Dong, R. Chang, and C. Ai, "The stacked seq2seq-attention model for protocol fuzzing," in *2019 IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT)*, Oct 2019, pp. 126–130.

[119] A. G. Mirabella, A. Martin-Lopez, S. Segura, L. Valencia-Cabrera, and A. Ruiz-Cortés, "Deep learning-based prediction of test input validity for restful apis," in *International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*, pp. 9–16.

[120] R. Zhao and S. Lv, "Neural-network based test cases generation using genetic algorithm," in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, Dec 2007, pp. 97–100.

[121] Z. Zhong, G. Kaiser, and B. Ray, "Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles," pp. 1–15.

[122] J. Zhu, L. Wang, Y. Gu, and X. Lin, "Learning to restrict test range for compiler test," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2019, pp. 272–274.

[123] Z. Chen, Z. Chen, Z. Shuai, G. Zhang, W. Pan, Y. Zhang, and J. Wang, "Synthesize solving strategy for symbolic execution," pp. 348–360.

[124] C. Paduraru, M. Paduraru, and A. Stefanescu, "Riverfuzzrl - an open-source tool to experiment with reinforcement learning for fuzzing," pp. 430–435.

[125] Z. Liu, X. Yang, S. Zhang, Y. Liu, Y. Zhao, and W. Zheng, "Automatic generation of test cases based on genetic algorithm and rbf neural network," vol. 2022, p. 1489063, 2022. [Online]. Available: https://doi.org/10.1155/2022/1489063

[126] K. Mishra, S. Tiwari, and A. Misra, "Combining non revisiting genetic algorithm and neural network to generate test cases for white box testing," *Advances in Intelligent and Soft Computing*, vol. 124, pp. 373–380, 2011, cited By 1. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84855228800&doi=10.1007%2f978-3-642-25658-5_46&partnerID=40&md5=0f21d2455ee273d6f3954f310f7d02a3

[127] S. Pan, H. Zhang, X. Zuo, and H. Deng, "Method of generating program path test cases based on neural network," in *International Conference on Optoelectronic Information and Computer Engineering (OICE 2022)*, Y. Yang, Ed., vol. 12308, International Society for Optics and Photonics. SPIE, p. 123080D. [Online]. Available: https://doi.org/10.1117/12.2647709

[128] H. Yasin, S. Hamid, and R. Yusof, "Droidbotx: Test case generation tool for android applications using q-learning," vol. 13, no. 2, pp. 1–30.

[129] E. Collins, A. Neto, A. Vincenzi, and J. Maldonado, "Deep reinforcement learning based android application gui testing," in *Brazilian Symposium on Software Engineering*, ser. SBES. New York, NY, USA: Association for Computing Machinery, p. 186–194.

[130] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "Qbe: Qlearning-based exploration of android applications," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, April 2018, pp. 105–115.

[131] Y. Koroglu and A. Sen, "Functional test generation from ui test scenarios using reinforcement learning for android applications," *Software Testing Verification and Reliability*, 2020, cited By 0. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85092015479&doi=10.1002%2fstvr.1752&partnerID=40&md5=f43056f29da7195d72ff0e731caf5989

[132] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "Autoblacktest: Automatic black-box testing of interactive applications," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 81–90.

[133] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 153–164. [Online]. Available: https://doi.org/10.1145/3395363.3397354

[134] S. Sherin, A. Muqeet, M. U. Khan, and M. Z. Iqbal, "Qexplore: An exploration strategy for dynamic web applications using guided search," vol. 195, p. 111512. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222001881

[135] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," pp. 423–435.

[136] C. Degott, N. P. Borges Jr., and A. Zeller, "Learning user interface element interactions," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 296–306. [Online]. Available: https://doi.org/10.1145/3293882.3330569

[137] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 623–639, 2013, cited By 95. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84888802373&doi=10.1145%2f2544173.2509552&partnerID=40&md5=476dda06dd32a8e6289ecf3a83dc01b1

[138] M. M. Kamal, S. M. Darwish, and A. Elfatatry, "Enhancing the automation of gui testing," in *Proceedings of the 2019 8th International Conference on Software and Information Engineering*, ser. ICSIE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 66–70. [Online]. Available: https://doi.org/10.1145/3328833.3328842

[139] D. Santiago, P. J. Clarke, P. Alt, and T. M. King, "Abstract flow learning for web application test generation," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 49–55. [Online]. Available: https://doi.org/10.1145/3278186.3278194

[140] D. Santiago, J. Phillips, P. Alt, B. Muras, T. King, and P. Clarke, "Machine learning and constraint solving for automated form testing," vol. 2019-October, 2019, pp. 217–227, cited By 0. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85081113091&doi=10.1109%2fISSRE.2019.00030&partnerID=40&md5=1be86067a371b94f0bd32074f2005a34

[141] Z. Khaliq, S. U. Farooq, and D. A. Khan, "A deep learning-based automated framework for functional user interface testing," vol. 150, p. 106969. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922001070

[142] Z. Khaliq, D. A. Khan, and S. U. Farooq, "Using deep learning for selenium web ui functional tests: A case-study with e-commerce applications," vol. 117, p. 105446. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197622004365

[143] F. Yazdani, B. Daragh, and S. Malek, "Deep gui: Black-box gui input generation with deep learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 905–916.

[144] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 2–8. [Online]. Available: https://doi.org/10.1145/3278186.3278187

[145] M. Brunetto, G. Denaro, L. Mariani, and M. Pezzè, "On introducing automatic test case generation in practice: A success story and lessons learned," vol. 176, p. 110933.

[146] M. K. Khan and R. Bryce, "Android gui test generation with sarsa," in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0487–0493.

[147] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 31–37. [Online]. Available: https://doi.org/10.1145/3278186.3278191

[148] I. Hooda and R. Chhillar, "Test case optimization and redundancy reduction using ga and neural networks," *International Journal of Electrical and Computer Engineering*, vol. 8, no. 6, pp. 5449–5456, 2018, cited By 3. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85066159186&doi=10.11591%2fijece.v8i6.pp5449-5456&partnerID=40&md5=1f455167b829969e3bc6e626b3b5d65d

[149] A. Groce, "Coverage rewarded: Test input generation via adaptation-based programming," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. USA: IEEE Computer Society, 2011, p. 380–383. [Online]. Available: https://doi.org/10.1109/ASE.2011.6100077

[150] B. Chen, Y. Liu, X. Peng, Y. Wu, and S. Qin, "Baton: symphony of random testing and concolic testing through machine learning and taint analysis," vol. 66, no. 3, p. 132101, 2022. [Online]. Available: https://doi.org/10.1007/s11432-020-3403-2

[151] E. Hershkovich, R. Stern, R. Abreu, and A. Elmishali, "Prioritized test generation guided by software fault prediction," in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 218–225.

[152] S. Ji, Q. Chen, and P. Zhang, "Neural network based test case generation for data-flow oriented testing," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, April 2019, pp. 35–36.

[153] J. Koo, C. Saumya, M. Kulkarni, and S. Bagchi, "Pyse: Automatic worst-case test generation by reinforcement learning," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, April 2019, pp. 136–147.

[154] M. H. Moghadam, "Machine learning-assisted performance testing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1187–1189. [Online]. Available: https://doi.org/10.1145/3338906.3342484

[155] M. Helali Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, "Machine learning to guide performance testing: An autonomous test framework," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2019, pp. 164–167.

[156] Q. Luo, D. Poshyvanyk, A. Nair, and M. Grechanik, "Forepost: A tool for detecting performance problems with feedback-driven learning software testing," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 593–596. [Online]. Available: https://doi.org/10.1145/2889160.2889164

[157] A. Sedaghatbaf, M. H. Moghadam, and M. Saadatmand, "Automated performance testing based on active deep learning," in *IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 11–19.

[158] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, "An autonomous performance testing framework using self-adaptive fuzzy reinforcement learning," vol. 30, no. 1, pp. 127–159, 2022. [Online]. Available: https://doi.org/10.1007/s11219-020-09532-z

[159] S. Chen, M. Haque, C. Liu, and W. Yang, "Deepperform: An efficient approach for performance testing of resource-constrained neural networks," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3561158

[160] H. Schulz, D. Okanović, A. van Hoorn, and P. Tůma, "Context-tailored workload model generation for continuous representative load testing," in *ACM/SPEC International Conference on Performance Engineering*, ser. ICPE. New York, NY, USA: Association for Computing Machinery, p. 21–32.

[161] L. Mudarakola and J. Sastry, "A neural network based strategy (nnbs) for automated construction of test cases for testing an embedded system using combinatorial techniques," *International Journal of Engineering and Technology(UAE)*, vol. 7, no. 1.3, pp. 74–81, 2018, cited By 6. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85067270771&partnerID=40&md5=397c33d415fed6913c48d4bb4751dc7a

[162] L. Mudarakola, J. Sastry, and C. Vudatha, "Generating test cases for testing web sites through neural networks and input pairs," *International Journal of Applied Engineering Research*, vol. 9, no. 22, pp. 11 819–11 831, 2014, cited By 7. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84925945679&partnerID=40&md5=b04e683a64af28552e91ef9f0ae6fce7

[163] R. Patil and V. Prakash, "Neural network based approach for improving combinatorial coverage in combinatorial testing approach," *Journal of Theoretical and Applied Information Technology*, vol. 96, no. 20, pp. 6677–6687, 2018, cited By 2. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85056233628&partnerID=40&md5=4f989bf779060baabae4e8302a603f91

[164] C. Duy Nguyen and P. Tonella, "Automated inference of classifications and dependencies for combinatorial testing," 2013, pp. 622–627, cited By 1. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84893575078&doi=10.1109%2fASE.2013.6693123&partnerID=40&md5=102d724421c89c7ae1f70e2a20020355

[165] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand, "Perfect is the enemy of test oracle," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, p. 70–81. [Online]. Available: https://doi.org/10.1145/3540250.3549086

[166] K. Kamaraj, B. Lanitha, S. Karthic, P. N. S. Prakash, and R. Mahaveerakannan, "A hybridized artificial neural network for automated software test oracle," vol. 45, no. 2, pp. 1837–1850. [Online]. Available: http://www.techscience.com/csse/v45n2/50392

[167] F. Tsimpourlas, A. Rajan, and M. Allamanis, "Supervised learning over test executions as a test oracle," in *ACM Symposium on Applied Computing*, ser. SAC. New York, NY, USA: Association for Computing Machinery, 2021, p. 1521–1531.

[168] K. Chen, Y. Li, Y. Chen, C. Fan, Z. Hu, and W. Yang, "Glib: Towards automated test oracle for graphically-rich applications," in *ACM Joint Meeting of European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE. New York, NY, USA: Association for Computing Machinery, p. 1093–1104.

[169] H. Khosrowjerdi and K. Meinke, "Learning-based testing for autonomous systems using spatial and temporal requirements," in *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, ser. MASES 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 6–15. [Online]. Available: https://doi.org/10.1145/3243127.3243129

[170] H. Khosrowjerdi, K. Meinke, and A. Rasmusson, "Learning-based testing for safety critical automotive applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10437 LNCS, pp. 197–211, 2017, cited By 7. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85029520480&doi=10.1007%2f978-3-319-64119-5_13&partnerID=40&md5=a74bc1966cd142e83070da2e7cc1bb37

[171] T. Rafi, X. Zhang, and X. Wang, "Predart: Towards automatic oracle prediction of object placements in augmented reality testing," in *Proceedings of the 37th*

IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3561160

[172] A. Arrieta, J. Ayerdi, M. Illarramendi, A. Agirre, G. Sagardui, and M. Arratibel, "Using machine learning to build test oracles: an industrial case study on elevators dispatching algorithms," in *IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 30–39.

[173] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2130–2141. [Online]. Available: https://doi.org/10.1145/3510003.3510141

[174] A. Gartziandia, A. Arrieta, A. Agirre, G. Sagardui, and M. Arratibel, "Using regression learners to predict performance problems on software updates: A case study on elevators dispatching algorithms," in *ACM Symposium on Applied Computing*, ser. SAC. New York, NY, USA: Association for Computing Machinery, 2021, p. 135–144.

[175] A. Gartziandia, A. Arrieta, J. Ayerdi, M. Illarramendi, A. Agirre, G. Sagardui, and M. Arratibel, "Machine learning-based test oracles for performance testing of cyber-physical systems: An industrial case study on elevators dispatching algorithms," vol. 34, no. 11, p. e2465. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2465

[176] S. R. Shahamiri, W. M. N. Wan Kadir, and S. Ibrahim, "A single-network ann-based oracle to verify logical software modules," in *2010 2nd International Conference on Software Technology and Engineering*, vol. 2, Oct 2010, pp. V2–272–V2–276.

[177] M. Ye, B. Feng, L. Zhu, and Y. Lin, "Automated test oracle based on neural networks," in *2006 5th IEEE International Conference on Cognitive Informatics*, vol. 1, pp. 517–522.

[178] H. Yu, Y. Lou, K. Sun, D. Ran, T. Xie, D. Hao, Y. Li, G. Li, and Q. Wang, "Automated assertion generation via information retrieval and its integration with deep learning," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 163–174. [Online]. Available: https://doi.org/10.1145/3510003.3510149

[179] D. J Hiremath, M. Claus, W. Hasselbring, and W. Rath, "Towards automated metamorphic test identification for ocean system models," in *IEEE/ACM International Workshop on Metamorphic Testing (MET)*, pp. 42–46.

[180] H. Spieker and A. Gotlieb, "Adaptive metamorphic testing with contextual bandits," *Journal of Systems and Software*, vol. 165, p. 110574, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121220300558

[181] O. Korkmaz and C. Yilmaz, "Sysmodis: A systematic model discovery approach," 2021, pp. 67–76.

[182] G. Shu and D. Lee, "Testing security properties of protocol implementations - a machine learning based approach," in *27th International Conference on Distributed Computing Systems (ICDCS '07)*, June 2007, pp. 25–25.

[183] R. HECHT-NIELSEN, "Iii.3 - theory of the backpropagation neural network**based on "nonindent" by robert hecht-nielsen, which appeared in proceedings of the international joint conference on neural networks 1, 593–611, june 1989. © 1989 ieee." in *Neural Networks for Perception*, H. Wechsler, Ed. Academic Press, pp. 65–93. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780127412528500108

[184] A. Graves, *Long Short-Term Memory*. Springer Berlin Heidelberg, pp. 37–45. [Online]. Available: https://doi.org/10.1007/978-3-642-24797-2_4

[185] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[186] Y. T. Chen, R. Gopinath, A. Tadakamalla, M. D. Ernst, R. Holmes, G. Fraser, P. Ammann, and R. Just, "Revisiting the relationship between fault detection, test adequacy criteria, and test set size," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, pp. 237–249.

[187] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, pp. 597–608.

[188] H. Hemmati, "How effective are code coverage criteria?" in *2015 IEEE International Conference on Software Quality, Reliability and Security*, pp. 151–156.

[189] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 435–445. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568271

[190] G. Gay, "Generating effective test suites by combining coverage criteria," in *Proceedings of the Symposium on Search-Based Software Engineering*, ser. SSBSE 2017. Springer Verlag, 2017.

[191] Y. Meng, G. Gay, and M. Whalen, "Ensuring the observability of structural test obligations," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018, available at http://greggay.com/pdf/18omcdc.pdf.

[192] D. Istanbuly, M. Zimmer, and G. Gay, "How do different types of testing goals affect test case design?" in *IFIP International Conference on Testing Software and Systems*. Springer, pp. 97–114.

[193] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, "Searching for cognitively diverse tests: Towards universal test diversity metrics," in *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. IEEE, pp. 178–186.

[194] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE 2015. New York, NY, USA: ACM, 2015.

[195] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 130–141.

[196] N. Alshahwan and M. Harman, "Coverage and fault detection of the output-uniqueness test selection criteria," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610413

[197] M. Staats, M. W. Whalen, A. Rajan, and M. P. Heimdahl, "Coverage metrics for requirements-based testing: Evaluation of effectiveness," in *Proceedings of the Second NASA Formal Methods Symposium*. NASA, April 2010.

[198] O. I. de Normalización, *ISO 26262: Road Vehicles : Functional Safety*. ISO. [Online]. Available: https://books.google.se/books?id=3gcAjwEACAAJ

[199] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate, "Testing or formal verification: Do-178c alternatives and industrial experience," *IEEE Software*, vol. 30, no. 3, pp. 50–57, May 2013.

[200] A. Parsai and S. Demeyer, "Comparing mutation coverage against branch coverage in an industrial setting," vol. 22, no. 4, pp. 365–388.

[201] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[202] J. Malburg and G. Fraser, "Combining search-based and constraint-based testing," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 436–439. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100092

[203] R. Feldt and S. Poulding, "Broadening the search in search-based software testing: It need not be evolutionary," in *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*, May 2015, pp. 1–7.

[204] A. Ramirez, J. R. Romero, and S. Ventura, "A survey of many-objective optimisation in search-based software engineering," vol. 149, pp. 382–395.

[205] A. Arcuri, "Test suite generation with the many independent objective (mio) algorithm," vol. 104, pp. 195–206. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584917304822

[206] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," vol. 104, pp. 207–235.

[207] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," vol. 104, pp. 236–256.

[208] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "Deploying search based software engineering with sapienz at facebook," in *Search-Based Software Engineering*. Cham: Springer International Publishing, 2018, pp. 3–45.

[209] M. Kechagia, X. Devroey, A. Panichella, G. Gousios, and A. van Deursen, "Effective and efficient api misuse detection via exception propagation and search-based testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2019, p. 192–203. [Online]. Available: https://doi.org/10.1145/3293882.3330552

[210] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1098–1105. [Online]. Available: http://doi.acm.org/10.1145/1276958.1277175

[211] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software*, vol. 83, no. 4, pp. 689 – 701, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121209003069

[212] M. Weiglhofer, G. Fraser, and F. Wotawa, "Using coverage to automate and improve test purpose based testing," vol. 51, no. 11, pp. 1601–1617.

[213] Z. Zhou, Y. Zhou, C. Fang, Z. Chen, and Y. Tang, "Selectively combining multiple coverage goals in search-based unit test generation," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556902

[214] P. McMinn, M. Harman, G. Fraser, and G. M. Kapfhammer, "Automated search for good coverage criteria: Moving from code coverage to fault coverage through search-based software engineering," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, ser. SBST '16. New York, NY, USA: ACM, pp. 43–44. [Online]. Available: http://doi.acm.org/10.1145/2897010.2897013

[215] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the*

2014 International Symposium on Software Testing and Analysis, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[216] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Software Engineering*, vol. 22, no. 2, pp. 852–893, Apr 2017. [Online]. Available: https://doi.org/10.1007/s10664-015-9424-2

[217] S. Vogl, S. Schweikl, G. Fraser, A. Arcuri, J. Campos, and A. Panichella, "Evosuite at the sbst 2021 tool competition," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, pp. 28–29.

[218] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," vol. 6, no. 2, pp. 182–197, publisher: IEEE.

[219] R. Feldt, S. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 223–233.

[220] J. W. Tukey *et al.*, *Exploratory data analysis*. Reading, MA, vol. 2.

[221] R. McGill, J. W. Tukey, and W. A. Larsen, "Variations of box plots," vol. 32, no. 1, pp. 12–16.

[222] F. Wilcoxon, "Individual comparisons by ranking methods," vol. 1, no. 6, pp. 80–83.

[223] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," pp. 50–60.

[224] O. J. Dunn, "Multiple comparisons among means," vol. 56, no. 293, pp. 52–64.

[225] F. Bretz, T. Hothorn, and P. Westfall, *Multiple comparisons using R*. Chapman and Hall/CRC.

[226] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press.