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.