Empirical Analysis of SBST tools: A taxonomy of coverage gaps

Version of June 27, 2025

Nicolae Rusnac

Empirical Analysis of SBST tools: A taxonomy of coverage gaps

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Nicolae Rusnac

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Empirical Analysis of SBST tools: A taxonomy of coverage gaps

Author: Nicolae Rusnac
Student id: 5257573

Abstract

Search-Based Software Testing (SBST) tools can automatically generate tests to achieve high code coverage; however, a systematic understanding of why they fail in specific situations is necessary. This thesis addresses this gap by developing a comprehensive taxonomy of coverage failures through an empirical analysis of the three most prominent SBST tools: Pynguin (Python), SynTest (JavaScript), and EvoSuite (Java). By classifying and analysing failure patterns across these tools and language paradigms, this research provides a foundational framework to diagnose shortcomings, prioritise future development, and enhance the practical effectiveness of automated test generation.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Olsthoorn, Faculty EEMCS, TU Delft
Committee Member:	Dr. J. Decouchant, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

Testing is an integral part of modern software development, which ensures that systems function reliably, securely, and as intended. As software becomes increasingly integrated into critical areas such as healthcare, finance, and national security, the consequences of defects become more severe, ranging from financial losses to compromised safety. Effective testing identifies and prevents issues early in development, reducing the cost of fixing bugs and protecting user trust. However, the growing complexity of modern software systems and the demands of rapid development cycles make testing more challenging than ever, highlighting the need for advanced automated testing techniques capable of systematically generating test cases to navigate software complexity.

One such technique is SBST (Search-Based Software Testing), which has attracted significant research attention over the past decade, particularly in recent years. SBST typically employs metaheuristic search algorithms, such as evolutionary algorithms, guided by fitness functions, often derived from coverage criteria to explore the input space of the software and automatically generate test cases [10]. Notable tools embracing this approach include Pynguin (for Python), SynTest (for JavaScript), and EvoSuite (for Java), which will be analysed in this thesis. Such tools aim to reduce the manual test implementation effort by automatically generating input to maximise code coverage. In selected benchmarks, such tools achieve significant coverage, typically between 60-90% of code lines [25] [18] [11]. This is in line with what Big Tech companies consider a good number to aim for, with Atlassian considering 80% to be a good target, while Google deems anything between 60-90% as an acceptable target depending on the context [6] [1]. However, reaching these high coverage figures does not eliminate important challenges. Current important difficulties include generating complex inputs, especially for non-trivial data structures or creating coverage for the most critical parts of a project consistently. Yet, the specific reasons behind these challenges and their attributes across different tools and projects are not yet well understood.

1.1 High-Level Problem Statement

Although the high-level challenges previously mentioned are recognised, there is still a lack of deep understanding regarding the specific challenges that SBST tools encounter. While it is true that many research papers experimenting with or developing such tools describe limitations or scenarios in which they are struggling, often there is an absence of concrete evidence for such claims (such as code snippets or references to code projects). Additionally, the information about such inefficiencies is scattered across different research materials, and the information about their limitations is presented in an inconsistent way. Currently, no centralised system exists to categorise the tools' failure scenarios. Lacking such a centralised framework leads to difficulties in analysing their shortcomings comprehensively. It is difficult to understand where, why, and how often they fail, but also how impactful a certain cause of failure is. It's also difficult to understand whether the limitations of a specific tool are inherent to the tool itself, whether it is reflective of the SBST paradigm, or whether they are inherent to a specific programming language.

Understanding the main cause of the coverage failures is difficult because these tools target different types of languages, primarily statically typed languages (like Java) and dynamically typed ones (like Python and JavaScript). In statically typed languages, the search tools have the advantage of knowing the type of the parameter when generating test cases. This additional information simplifies the search process, as it reduces the amount of guesswork required by the algorithm. Conversely, in a dynamically typed language, there is an additional layer of complexity. Not only do the appropriate values to trigger a certain branch need to be identified, but also the types of the parameters. This significantly increases the search space. However, it is difficult to determine how these types of languages compare and whether they fail similarly or in fundamentally different ways. It is also not clear whether these factors influence the resulting patterns of code coverage, such as potentially biasing the dynamically typed languages toward exploring structurally simpler code regions compared to their static counterparts. This lack of systematic and comparative understanding regarding SBST failures across different language paradigms hinders the research and development progress of such tools.

Specifically, this hindrance manifests in several ways: Without knowing which failure types are most frequent or impactful, researchers and tool developers cannot effectively prioritise their efforts. Without a comparative view, researchers risk wasting effort on low-impact problems or reinventing solutions for dynamically typed languages that already exist for statically typed ones. Without a clear systematic understanding of the failure gaps, it is difficult to determine whether a new SBST tool or algorithm truly overcomes the most significant limitations. While it is possible to evaluate it in terms of line/branch coverage or other metrics, evaluating it in terms of a well-defined system would provide much more valuable insights. Moreover, large-scale practical adoption of such tools is unlikely to happen unless the developers who use them are confident in what they can achieve and have a good understanding of their limitations. Overall, these issues significantly hold back the advancement

and practical application of automated test generation tools, indicating the need for a structured classification of their shortcomings.

To provide this necessary classification, this thesis focuses on the development and validation of a comprehensive taxonomy of SBST coverage failures, based on empirical analysis. This taxonomy is extracted from analysing generated test coverage data across different tools and paradigms: Pynguin (Python), SynTest (JavaScript), and EvoSuite (Java). Such a taxonomy provides a common vocabulary for failures in such tools and offers a framework for their classification, allowing further, more complex analysis. Using the taxonomy provides further insights into the frequency and impact of the categories. Furthermore, applying the taxonomy across different tools allows for comparative analysis across different attributes of the studied languages, most importantly the difference between tools for statically and dynamically typed languages. Based on the aggregated insights from these analyses, it is possible to further understand which problems are worth prioritising in an objective manner. The taxonomy also allows the evaluation of these tools in terms of how they perform in terms of the presented categories in a more complex and granular manner. It also provides a clearer outline of the tools' limitations and capabilities, helping to identify steps needed for practical improvement. Therefore, the specific research objectives and questions guiding the creation and analysis of this taxonomy are presented next.

1.2 Research Objectives

To address the identified lack of systematic understanding, the aim of this thesis is twofold: first, to develop a comprehensive, empirically grounded taxonomy for coverage failures in SBST tools by analysing Pynguin, SynTest, and EvoSuite and second, to use the taxonomy to conduct an analysis of these failures. More precisely, the analysis focuses on their impact, comparative analysis between different tools, types of languages and manually written tests. The goals of the research can be summarised in the following research questions:

1. What distinct categories of failures prevent the studied SBST tools (Pynguin, SynTest, EvoSuite) from achieving code coverage on representative software projects?

2. How can the impact of different coverage failure types be characterised or measured, and what is the relative frequency and estimated impact of each category within the developed taxonomy?

3. How do the observed coverage failure patterns (types, frequency, impact) compare and contrast between statically typed languages (EvoSuite/Java) versus those targeting dynamically typed languages and between two tools targeting dynamically typed languages (Pynguin vs. SynTest)?

4. For the identified coverage failure categories, what potential mitigation strategies or relevant areas of existing research can be identified in the literature?

1.3 Methodology overview

To address the research questions, this thesis engages in an empirical study involving analysing test suites and coverage reports generated by three state-of-the-art SBST tools: Pynguin (Python), SynTest (JavaScript), and EvoSuite (Java). The methodology involved systematic manual analysis of the generated test suites and coverage reports to identify, categorise, and develop the proposed taxonomy of coverage failures. The tests used to generate the coverage were collected from different replication packages used in other research projects for the aforementioned tools. The taxonomy is used as a basis for analysing failure frequency and impact, and for conducting comparative analyses across the different tools and language paradigms. Furthermore, the coverage generated by the tests created by the tools was compared with the coverage generated from manually written tests for the analysed projects. A detailed description of the benchmark projects, experimental setup, data collection procedures, classification methodology, and analysis techniques is provided in Chapter 3.

1.4 Thesis outline

The structure of the thesis is as follows: **Chapter 1: Introduction** - Defines the research context, problem, and objectives and outlines the thesis structure. **Chapter 2: Background and Related Work** - Reviews relevant background literature on SBST principles, the specific tools studied, related work on failure analysis, and already identified faults in such tools. **Chapter 3: Methodology** - Details the empirical methodology used for data collection, taxonomy development, and subsequent quantitative and comparative analyses. **Chapter 4: Taxonomy** - Presents the empirically derived taxonomy of SBST coverage failures, including category definitions and illustrative examples. **Chapter 5: Analysis and Comparative Results** - Reports the quantitative and comparative results concerning failure frequency, impact, and patterns across different tools, language paradigms, and human tests. **Chapter 6: Discussion** - Interprets the findings, discusses their implications and limitations, answers the research questions, and links key failures to potential solutions in the literature. **Chapter 7: Conclusion** - Summarises the main findings and contributions of the thesis and suggests directions for future research.

Chapter 2

Background and Related Work

2.1 Introduction to Automated Software Testing

As described in Chapter 1, the effective testing of software is of utmost importance for ensuring the reliability, security and proper functioning of software systems, most notably with the increasing complexity of modern software systems. The challenges posed by such complexity and the current demands for quick implementation and creation of tests make classical manual test methods expensive and less efficient. This section provides an overview of automated software testing, leading to a detailed overview of SBST (Search-Based Software Testing) and its associated challenges.

Automated software testing refers to the usage of tools and frameworks that generate tests with the goal of maximising some coverage metric. Such tools can focus on different levels of testing: unit, integration or systems testing. This thesis focuses primarily on tests generated at the unit level [1]. In an effort to tackle this challenge, several advanced techniques have been developed, each with its pros and cons:

- **Symbolic Execution:** Symbolic execution is a program analysis technique used to determine what inputs need to be provided in order to trigger certain parts of a program to execute. Instead of providing concrete values, symbolic values are used. During the analysis, the relationships between these symbolic symbols are tracked in a set of mathematical constraints called ”path condition”. In cases where the program's control flow depends on some symbolic values, the exploration engine can explore both possible branches, forking the analysis into two separate paths, each with an updated path condition reflecting the choice that was made. Using a constraint solver to solve a certain path condition, a concrete test case that is guaranteed to follow a specific path through the program can be generated [7].

  The main strengths of this approach lie in its ability to achieve high coverage, effectively find bugs, and target specific execution paths concretely, leading to easier reproduction or identification of bugs. However, this approach's main limitation is the execution time and practical usability. With the size of the

program, the number of executions paths increase exponentially, making exploring all paths an infeasible task in practice. Sometimes, the identified constraints can be tricky and complicated, increasing the probability that the constraint solver would be unable to identify a solution in time. Interaction with external components is also difficult to model, as is reasoning about memory operations for these kinds of tools.

- **Fuzz Testing (Fuzzing):** Fuzz Testing is a dynamic software testing technique that involves providing invalid, unexpected or random data as input for a program to generate anomalous behaviour, such as crashes, failed assertions, or memory errors. Early "black-box" fuzzers generated inputs randomly or through simple mutations, disregarding the program's internal logic. The more modern approach to fuzzing is coverage-guided greybox fuzzing. This technique uses lightweight instrumentation to receive feedback on the parts of the program that some input executes. The fuzzer then tries to prioritise and mutate inputs that discover new code paths, thus increasing the coverage over time and exploring the program's logic more effectively than randomly [20] .

  These tools are effective for discovering vulnerabilities, security-related bugs, and memory-related vulnerabilities. Generally, they scale well and are quite fast. However, similarly to SBST-based approaches, they struggle with constant values, with triggering more complex bugs, or with identifying bugs that do not lead to a crash.

- **Search-Based Software Testing (SBST):** SBST is an automated testing technique that formulates test case generation as a search or optimisation problem. It typically employs metaheuristic search algorithms, usually evolutionary algorithms, guided by fitness functions based on different coverage metrics, with the goal of exploring the input space and generating test cases. The aim is to maximise the aforementioned coverage metric by generating relevant tests.

  This type of test generation has received significant research attention, and tools have been developed that achieved significant coverage around 60-90% [10]. This type of testing often struggles to generate non-trivial data structures or generally reach certain parts of the input space. It also relies heavily on well-defined fitness functions.

- **Model-Based Testing (MBT):** Model-Based Testing (MBT) is a software testing technique where test cases are generated based on a model that represents the expected behaviour of the System Under Test (SUT). Instead of designing test cases manually, testers create a model using state machines, flowcharts, and UML diagrams. The model captures the system's requirements and logic. An MBT tool can use this model to generate test cases by applying test selection criteria, such as covering all states or transitions in the model. [29]

  The main benefit of the approach is that it allows for bug discovery during the design stage when the model is created. Test maintenance is easier since,

when the requirements for the system change, only the model needs to be updated, not all tests. The main problem with his approach is the manual overhead. The model needs to be created and designed well, with a high initial cost. The model itself might be incorrectly designed or might not conform to specifications. There is also the question of scalability as manual effort increases with the complexity of the system.

- **Large Language Models (LLMs) for Testing:** This technique involves leveraging large language models like OpenAI's GPT series, Google Gemini, or Claude to perform or assist in generating software tests. Unlike more traditional tools previously mentioned in this list, LLMs use their training on source code and tests to contextually understand the code and reason for generating tests. Typically, such tools can be used through IDE-integrated tools such as GitHub Copilot or the Cursor IDE. To use these tools, developers typically create a prompt referencing the relevant portions in the code that need to be tested in combination with a simple text command. [31]

  These tools are generally easy to use and very accessible, as their use relies on natural language. They are also good at understanding context and generating tests that are more "human-like" and that conform to the practices used in the codebase. They are also not limited to only unit tests, and they generate them with generally high coverage. However, there are also risks associated with such tools. They can often hallucinate and generate erroneous code or code that does not test the intended logic. Sometimes, they can also generate superficial tests but avoid more complex scenarios. They are also "black-box" probabilistic models, meaning they don't behave in a deterministic way.

In this thesis, the focus will be placed on the SBST approaches.

2.2 Search-Based Software Testing (SBST)

As mentioned in the previous section 2.1, Search-Based Software Testing reformulates test generation as an optimisation problem. This section will now provide a more detailed examination of the technique which forms the foundation of this thesis. The core principles that enable SBST will be explored, its demonstrated advantages and current status in the research community will be reviewed, and the high-level challenges that motivate the need for a systematic failure taxonomy will be discussed.

2.2.1 High-Level Challenges in SBST

Despite its potential as an effective test automation framework, Search-Based Software Testing encounters several fundamental and practical difficulties that pervade its various applications. At its technical core lie three deeply interconnected challenges.

First, the test oracle problem represents the difficulty of automatically determining if a test's output is correct remains an important bottleneck that limits full test

automation by requiring costly manual verification [8]. Second, the performance of the entire algorithm depends on fitness function design, as crafting functions that accurately guide the search through complex software landscapes is a critical and non-trivial task [22]. Third, the test case representation, or how inputs are generated for the search, is a critical factor that creates considerable challenges, especially when handling complex data types or the specifics of different programming languages [14]. For the context of this thesis, the main focus will be on the second and third ones.

Beyond these main issues, SBST faces significant practical barriers. These include the difficulties of extending the ideas of the approach beyond unit testing to be applicable to handle system-level and integration testing, ensuring scalability for large, complex systems, and managing the engineering overhead associated with tooling, maintenance, and language portability [14]. Together, these technical and practical hurdles contribute to the challenge of overcoming the barriers to widespread industrial adoption, where issues of consistency, integration, and demonstrating a clear return on investment are paramount.

2.3 Overview of SBST Tools Under Study

To connect the theoretical discussion of SBST in practice, this section introduces the three state-of-the-art tools whose outputs form the basis of the empirical analysis in this thesis: EvoSuite, the SynTest Framework, and Pynguin. These specific tools were selected because they are well-researched and together provide a strong basis for a multi-faceted comparative analysis. The selection facilitates a comparison across different language paradigms, as EvoSuite targets Java (a statically typed language), while Pynguin and SynTest target Python and JavaScript (dynamically typed languages). Furthermore, the inclusion of both Pynguin and SynTest allows for an insightful analysis within the dynamically typed paradigm. The following subsections will detail the relevant characteristics and configurations of each of these tools as used in this study.

EvoSuite is a prominent and mature tool in the SBST landscape, specifically dedicated to the automatic generation of JUnit test suites for Java classes. Its extensive ecosystem facilitates adoption across diverse development workflows, offering multiple usage capabilities including a command-line interface, Docker support, and integration with popular IDEs like Eclipse and IntelliJ IDEA, as well as a dependency manager such as Maven. Its reputation as the most popular and developed test generation tool for Java is supported by a history of significant development and validation against large-scale benchmarks like the SF110 Corpus. Essentially, EvoSuite employs a highly refined steady-state Genetic Algorithm that focuses on ”whole test suite” optimisation. This approach evolves entire test suites based on their collective ability to satisfy multiple coverage goals simultaneously, being a strategy apparently superior to targeting coverage goals one by one. The search is guided by a fitness function that primarily optimises for structural coverage, using metrics like approach

level and branch distance, which are calculated based on data gathered from bytecode instrumentation of the Java classes under test. To address the test oracle problem, EvoSuite's primary assertion strategy is to capture the current behaviour of the code, creating effective regression oracles. While this does not ensure that the logic intended to be implemented in the code is correct, it relies on the assumption that the written code is successful in implementing the logic. It also offers a more advanced mutation-based assertion generation strategy to improve fault-finding capabilities. Furthermore, EvoSuite tackles the challenge of complex dependencies through a sophisticated mocking system, utilising both internal mocks for non-deterministic components and integration with the popular Mockito framework. A key innovation is its ability to make the configuration of these mock objects part of the evolutionary search itself, allowing the tool to learn how to set up dependencies to reach difficult parts of the code.

The SynTest Framework is a modular and extensible ecosystem designed with the ambitious goal of democratizing access to SBST techniques across a variety of programming languages. Its defining characteristic is a language-agnostic core architecture that encapsulates fundamental SBST components, such as implementations of various metaheuristic search algorithms and abstract representations for test cases. Language-specific modules then build upon this common core to provide support for target languages; currently, there are notable implementations for JavaScript/-TypeScript and Solidity, the language of Ethereum smart contracts. This modular design promotes reusability and aims to accelerate the development of SBST support for new languages. In this thesis, the focus is placed primarily on the JavaScript language. A key strength of the framework is its built-in support for multiple sophisticated, many-objective search algorithms, including NSGA-II, MOSA, and DynaMOSA, providing users and researchers with significant flexibility. One of the most significant innovations within the SynTest ecosystem is its direct approach to handling the challenges of dynamic languages. For JavaScript, the team has developed and integrated an unsupervised probabilistic type inference mechanism that operates within the test generation process. This allows the tool to make "guesses" about the appropriate data types for variables and parameters, guiding the search more effectively and limiting the high rate of runtime errors that would result from simple random type selection.

Pynguin is a test generation framework focused specifically on Python, being one of the first fully automated tools to produce unit tests for general-purpose Python code. Distributed as a standard Python package, Pynguin is an extensible command-line tool presented as a research prototype, not yet tailored for critical production use. Reflecting the risks of dynamically executing potentially untrusted code, its developers strongly recommend running it in an isolated environment like a Docker container, due to potentially dangerous interactions with external systems. A key feature of Pynguin is its implementation of a diverse suite of search algorithms, with the effective DynaMOSA as its default, alongside other strategies like MIO, MOSA, and an EvoSuite-like "WHOLE_SUITE" approach. Pynguin's primary contribution lies in its multi-faceted strategy for addressing the significant hurdles of Python's dy-

namic typing. It heavily relies on optional static type hints (PEP 484) when present, as they provide invaluable information to the test generator. For code lacking these annotations, it employs dynamic type tracing to observe runtime types and make more ”guesses” for subsequent test generation. A flexible system of weighted type sources further enhances this. Pynguin is also equipped with several sophisticated assertion generation strategies, moving beyond simple checks to include a mechanism based on mutation analysis and an approach that uses ”checked coverage” to create more concise and relevant assertion sets.

2.4 Existing Work on Failure Analysis and Limitations in Automated Test Generation

A review of existing research reveals numerous granular challenges directly related to the generation of test inputs and the setup of test conditions. However, it is worth mentioning that these faults are scattered across multiple bodies of research and not backed by empirical data, but rather by observation during the performance of experiments.

A recurring and important problem is the failure of search algorithms to properly construct complex input objects with the specific states required to cover certain code paths [14]. This often requires generating valid and meaningful sequences of method calls to instantiate objects and manipulate them into a necessary prerequisite state, which is a major hurdle [22]. The difficulty is amplified in languages like C/C++ when handling pointers and dynamic data structures, as these create a variable-sized search space that is hard for traditional SBST techniques to navigate [17]. Specific Language features also present major hurdles. In dynamically-typed languages like Python, the absence of static type information can lead to an exponential increase in possible inputs and frequent runtime errors [14]. Specific constructs in statically-typed languages, such as loop-assigned flag variables, are notoriously problematic as they create ineffective fitness landscapes that offer no guidance to the search [17]. Furthermore, generating valid inputs for specialised domains is a key challenge, from creating realistic GUI event sequences out of an explosive set of possibilities [23] to finding specific thread interleavings to test concurrent programs [21]. Ultimately, the literature indicates that these input generation failures often stem from two deeper, foundational issues: a Test Case Representation that cannot effectively generate the required complex inputs, and a Fitness Function that provides poor or no guidance for constructing them [22].

2.5 Identifying the Gap: The Need for a Comprehensive Taxonomy of SBST Failures

A review of the existing work indicated that the topic of identifying coverage gaps had been approached in the past, finding valuable but quite specialised failure modes.

Harman et al. [16] investigated the directions of research and the lack of Search-Based Software Testing. Although the insights presented were valuable, the paper focused more on analysing non-functional properties, such as execution time, security, and energy use. The authors identified that more work needs to be done on multi-objective optimisation and the automatic identification of test strategies. Although not directly related, this paper focuses more on classifying the research areas within SBST rather than directly on the faults of the tools themselves, focusing on a higher level.

Another relevant piece of research by Fraser & Arcuri [12] focuses on the weak points of Evosuite. Previously, the authors analysed 100 real-world open-source projects, classifying the issues found with the code into different categories. The paper focuses on different challenges, such as Usability and Engineering challenges. While they identify multiple categories, such as environmental problems, Non-determinism, Java peculiarities, or Resource Limits, it is not exactly clear how the analysis was performed, whether the identified categories are exhaustive, or whether they are truly exclusive. It also does not perform any sort of analysis on the results or provide clear criteria on how to classify such faults. It primarily serves as a descriptive report rather than a comprehensive taxonomy.

In another paper by Lukasczyk [19], some problems inherent to Penguin are discussed. Unlike most other papers focusing on the weak points of SBST, this one focuses on a dynamically typed language. Some of the identified faults are the challenge of dynamic typing, high reliance on type hints, and ineffective fallback mechanisms when type hints are not available. Again, while highlighting some of the problems, the paper focuses more of a descriptive overview of the tool and its interactions with the Python language, including its weak points and limitations, rather than a taxonomy. The same pattern is observed where the focus is placed on intuitive observations rather on ones backed by empirical data.

While the main goal of the paper "1600 Faults in 100 Projects" [13] was evaluating a fault-finding technique, the analysis also involved creating a number of categories for failures, similar to the goal of the current thesis. The classification is mostly done on the basis of exceptions, such as NullPointerExceptions, IllegalArgumentException or ClassCastException. However, a distinction that is made in the paper is between actual faults and violations of implicit preconditions. This type of categorisation can be considered an attempt at categorising failures by their root cause. The paper also delves deeper into the causes for the aforementioned exceptions, mentioning their causes. However, the classification is qualitative and is not defined by any well-defined criteria.

Based on the existing body of work, it is clear that identifying such coverage gaps is indeed an area of research that has garnered attention and one that has been previously explored to some extent. It is also clear that most accounts of these failures are anecdotal, based on observations rather than empirical analysis. Moreover, these descriptive categories were not designed to be mutually exclusive or to have clear classification criteria. No analysis was conducted based on their impact or frequency. All these analyses were performed on a single tool, lacking the generalisation of

insights that come with a more comprehensive study. All these characteristics of the studies indicate that, indeed, a comprehensive classification if warranted.

Therefore, to address this critical gap, this thesis undertakes the development and empirical validation of a comprehensive taxonomy of SBST coverage failures. Such a framework is essential for providing the common vocabulary and structured analysis needed to move beyond scattered observations. By systematically categorising failures, this work aims to enable a more data-driven understanding of where and why SBST tools fall short, creating the way for more targeted improvements and a clearer assessment of future advancements in the field.

Chapter 3

Methodology

This chapter of the thesis focuses on the methodology used to research and answer the questions presented in Chapter 1 regarding SBST coverage failures. As indicated previously, an empirical study is conducted in this thesis, centred around developing a taxonomy created by analysing specific SBST tools. This chapter proceeds as follows: **Section 3.1** discusses the analysed tools and the benchmark software projects studied. **Section 3.2** Outlines the data collection and management procedures used throughout the taxonomy development and analysis. **Section 3.3** Explains the Taxonomy Development procedures. **Section 3.4** Describes the types of analyses that are performed and the way in which they are carried out. **Section 3.5** Addresses the limitations of the methodology.

3.1 Research Context: Tools and Subjects

This section explains the choices of the SBST tools and the benchmark projects selected for the empirical analysis. The data used in the analysis is sourced from replication packages containing tests generated by the chosen tools from already existing research studies. These selections ensured the study could effectively address the research questions regarding taxonomy development and analysis.

3.1.1 SBST Tools Studied

The data generated by three SBST tools, SynTest (JavaScript), Pynguin (Python), and EvoSuite (Java), were used in the analysis performed in this thesis. These tools are well-researched and provide a good basis for comparative analysis. SynTest and Pynguin both focus on dynamically typed languages, allowing for comparative analysis between tools for languages within the same paradigm, while EvoSuite allows for comparison across different paradigms.

Pynguin is an extensible framework for test generation designed for generating high-coverage test suites for Python. The data analysed in this thesis originates from replication packages from prior bodies of research work. The data in the replication packages uses Pynguin with the DynaMoSA multi-objective search algorithm and

with Python type hints enabled, being the best performing configuration at the time of writing [19]. The datasets utilised in the analysis use Pynguin versions 0.25.2 [19] and 0.27.0 [15]. Although minor differences exist between the source experiments [2], the analysis focuses on the optimal configuration of the algorithm as representative of this target state-of-the-art configuration documented in the respective source experiments. Overall, including Pynguin data generated under these conditions provides a good base for performing analysis of the tool's behaviour.

SynTest-Javascript is part of the broader SynTest framework. It is an automated test generation tool focusing on server-side JavaScript. The main functioning principle is similar to Pynguin, as it uses search-based algorithms in the process of generating unit tests. For determining the types when generating tests, the tool utilises unsupervised probabilistic type inference [27]. The analysed data is based on the replication package referenced in the paper [25]. This data is generated as a result of running the tool on the 0.1.0 version with the DynaMOSA algorithm using the elitist variation, by running the algorithm for 60 seconds for each module. SynTest is an interesting tool to evaluate, considering it provides insights in an alternative dynamically typed language. Additionally, it uses a different type of inference algorithm, in contrast to Pynguin, allowing for more insights to emerge when comparing the tools.

EvoSuite is an SBST tool for the Java language. Similarly to the other tools, it uses evolutionary algorithms, like DynaMOSA, to optimise test suites for code coverage [11]. The EvoSuite data analysed in this thesis was obtained from replication packages associated with two prior studies [24, 15]. In both sources, the data was generated using a time budget of 120 seconds per target class with the default DynaMOSA configuration. However, the tool versions differ: data from the ASE'20 paper [24] used EvoSuite version 1.0.7, while data from the ICSE'24 paper [15] used version 1.2.0. Using different versions might impact the analysis based on the created taxonomy; this potential limitation is discussed further in Section 3.6. The inclusion of EvoSuite is important as it allows for comparative analysis between SBST tool performance on a statically typed language (Java) versus dynamically typed languages (Python, JavaScript), directly addressing RQ3.

3.1.2 Benchmark Software Projects

The projects that form the basis of the analysis are the ones drawn from the replication packages for the tools used in this thesis: Pynguin, SynTest and EvoSuite. The selection of the projects is inherited from the original studies corresponding to the replication packages [15] [19] [25] [24]. For most replication packages, all projects were considered, with the exception of the dataset originating from Gruber [15]. In this specific instance, a subset of projects was used with a specific filter, as detailed below, to better suit the goal of identifying and categorising coverage failures. This filter only considered projects with more than 10 files, with at least one file with coverage lower than 100% and projects where not all files had 0% coverage, to avoid outliers or test failures. The number of selected projects was done on an ad-hoc basis,

according to the time left for completing the project. There is still a large number of projects available, which can be used to further extend and refine the taxonomy.

In total 42 projects were selected. This number assumes that a high enough diversity was reached for the purposes of the thesis, taking into consideration the time and manual effort required to conduct the analysis. It is important to highlight that not every file in a project was evaluated, but only those for which tests were present in their corresponding replication packages.

3.2 Data Acquisition from Replication Packages

Identifying suitable replication packages containing pre-computed test suites for the target tools formed the first step of the data acquisition. Replication packages not containing both the generated tests and the source projects were excluded due to time constraints. An extensive search was conducted, focusing on replication packages from Zenodo and GitHub, using the names of the tools in the search process. While multiple replication packages were found, many would only contain logs, test statistics or the projects on which the tests were to be generated. In other cases, multiple replication packages would share the same benchmark of projects. In other cases, replication packages referenced in research papers would no longer be publicly available. Therefore, the selection process primarily focused on identifying packages that met the following criteria:

1. **Tool Relevance** - Replication packages related to the tools used in the study
2. **Data Completeness** - Packages that include both the generated unit tests and the original projects on which they were generated
3. **Accessibility** - The package was publicly available

Given the limited number of replication packages meeting the criteria, most were included in this thesis to gather sufficient data for the planned taxonomy development and comparative analysis. The coverage data later used in the study was generated by executing the tests against the projects (Table 3.1) in the packages. The data is available in the replication package [26]. This process is further detailed in Section 3.3.3.

3.3 Taxonomy Development Process

This section describes the process used to empirically construct the taxonomy for coverage failures of SBST tools with the purpose of answering RQ1. The methodology is largely based on the principles introduced by Usman et al. for creating taxonomies in the field of software engineering [28]. Taking into consideration these principles and considering the goal of identifying solutions for relevant taxonomy categories (RQ5), a hierarchical taxonomy is considered for this analysis. The taxonomy creation process consisted of a few steps, described in the following subsection.

Table 3.1: Projects used in the Taxonomy

ID	Project_Name	Language	Tool
1	typesystem	Python	Pynguin
2	spotify-metrics-munin-reporter	Java	EvoSuite
3	dataclasses-json	Python	Pynguin
4	javascript-algorithms	JavaScript	Syntest
5	Twitch-Python_441	Python	Pynguin
6	pytutils	Python	Pynguin
7	supertunnel_6727	Python	Pynguin
8	vcver-python_7255	Python	Pynguin
9	isort	Python	Pynguin
10	pypara	Python	Pynguin
11	oscpy_4284	Python	Pynguin
12	aegisql-ftry	Java	EvoSuite
13	Claudenw-classpath-utils	Java	EvoSuite
14	zerolog_7561	Python	Pynguin
15	pyMonet	Python	Pynguin
16	python-string-utils	Python	Pynguin
17	adyliu-idcenter	Java	EvoSuite
18	iexfinance_3031	Python	Pynguin
19	thonny	Python	Pynguin
20	lodash	JavaScript	Syntest
21	fmi_2509	Python	Pynguin
22	docstring_parser	Python	Pynguin
23	lxchinesszz-tomato	Java	EvoSuite
24	apimd	Python	Pynguin
25	moment	JavaScript	Syntest
26	commanderjs	JavaScript	Syntest
27	pdir2	Python	Pynguin
28	mimesis	Python	Pynguin
29	sanic	Python	Pynguin
30	py-backwards	Python	Pynguin
31	python-ubersmith_5776	Python	Pynguin
32	scrapyrt_6252	Python	Pynguin
33	3redronin-mu-server	Java	EvoSuite
34	opengeospatial-ets-georss10	Java	EvoSuite
35	DataONEorg-d1-cn-common	Java	EvoSuite
36	opengeospatial-ets-indoorgml10	Java	EvoSuite
37	AussieGuy0-SDgen	Java	EvoSuite
38	binfalse-XMLUtils	Java	EvoSuite
39	matthewhorridge-mdock	Java	EvoSuite
40	JSON-java-20201115	Java	EvoSuite
41	fastjson-1.2.68	Java	EvoSuite
42	gson	Java	EvoSuite

First, the specific unit of analysis is described in Section 3.3.1. Further, the procedure for conducting the manual analysis of the collected data is depicted in Section 3.3.2. Finally, the iterative process of refining the taxonomy categories and structure is explained in Section 3.4.3.

3.3.1 Unit of Analysis

For the purpose of developing the taxonomy, the basic unit of analysis is defined as the coverage target element at which the SBST tool led to failure in generating coverage for a specific section of code.

Based on the coverage criteria relevant to the study, the primary target element types are:

- **Branch-True** - A failure where the tool failed to generate coverage because the conditional statement did not evaluate to **True**
- **Branch-False** - A failure where the tool failed to generate coverage because the conditional statement did not evaluate to **False**
- **Runtime Error Origin** - A failure where the tool throws a runtime error. This can occur either within a branch statement or a regular line of code.

This granularity was determined because it directly relates to the way SBST tools operate and is in line with the main goal of the taxonomy—to create a classification for coverage failures in such tools.The SBST tools' fitness function is typically guided by branch coverage, making it a clear choice for a unit of analysis.**Runtime Error Origin** is the other cause of lack of coverage in a tool, as it can prevent subsequent code from being executed. This definition of the unit of analysis paves the way for further analysis, while also creating a clear outline for what exactly should be classified during the manual analysis.

3.3.2 Manual Analysis Procedure

Following the definition of the coverage failures and their identification in Section 3.3.1, manual analysis was conducted to identify relevant information about the root cause, context and impact of the failure. The analysis relied on the coverage reports generated based on the unit tests and source code available in the replication packages and the source code of the generated unit tests.

3.3.2.1 Analysis Steps per Failure Instance

Manual analysis of the coverage data was performed following a series of steps:

1. **Failure Point Identification** - Identifying the failure point is generally straightforward. By analysing coverage reports, they can be found by identifying the first line prior to the beginning of an uncovered block.

2. **Code Inspection** - This stage relates to analysing the code snippet affected by the failure point and understanding what conditions are required for the failure to be resolved. In some cases, this can be rather difficult as some projects contain nested calls to other classes or modules. For the purpose of this thesis, the flow of the program will not be explored beyond a dependent module, classifying all such failures into a separate category in the taxonomy.

3. **Test File Inspection** - In some cases, the cause of failure is not readily apparent from the source code. In such cases, additionally, the test cases are analyzed for additional context. This step can be considered optional, and it is context-dependent.

4. **Hypothesis Formulation** - In this step, a probable cause is formulated for the failure. This can be considered a tentative category in the taxonomy that will be refined later on. There are, however, a few considerations to take into account when formulating such categories. Previously, it was mentioned that a hierarchical taxonomy would be used, mimicking a tree structure. This implies that a single failure gap must belong only to a single category. However, there are cases where a failure might belong to multiple categories.

   For example, we might have the following function in Listing 3.1:

   Listing 3.1: Function dependant on multiple parameters

```
public static boolean checkBothConditions(int inputA, String
    inputB) {
    boolean conditionA_met = false;
    if (inputA > 100) {
        conditionA_met = true;
    }
    boolean conditionB_met = false;
    if ("valid".equals(inputB)) {
        conditionB_met = true;
    }
    if (conditionA_met && conditionB_met) {
        return true;
    } else {
        return false;
    }
}
```

   To deal with such examples, it is important to create well-defined rules that determine how they should be classified. But first, it is important to define some concepts.

   An operation A is dependent on another operation B only if it reads, uses, or is affected by data values or object states that were written, created, and modified by B. When there are multiple operations in a row dependent on one another,

such as A dependent on B and B dependent on C, they form a 'chain'. Given these definitions, the following rules are defined:

a) **First Unmet Dependency** - In cases where a target leads to a failure, the dependency chain should be examined. If there is a single chain ending in the target of interest, the failure should be considered the earliest point in the dependency chain where the SBST tool fails to generate the appropriate values. If there are multiple chains with different failures leading to the target in question, it is not possible to assign a single category, so a separate category is saved for this case. The taxonomy categories that impact the target are saved as metadata items for further analysis. This choice was made to preserve the hierarchical structure of the taxonomy.

b) **Short-Circuit Evaluation Logic** - In control structures where there are multiple conditions linked through conjunction, only the first condition that did not evaluate to True or False (depending on the context) is considered to be the root cause of the failing statement. The other conditions are not considered since solving the problem leading to the first blocking point changes the conditions under which the subsequent conditions can manifest.

c) **Ambiguous classification** - In some cases ,it is difficult to identify a clear category that fits well for a certain target. This can be due to a lack of log data or a lack of other contextual information around some failure. In such cases, the failure should be classified in a special category for unclear classifications. This decision was made to ensure the validity of the taxonomy and reduce the number of ambiguous cases.

5. **Impact Metric Calculation** - For each target that is classified, the number of lines of code that are not covered due to a specific failure is tracked. In this case, not only the immediate block is considered, but also the number of lines of code of nested functions called within that block. However, this is only done for the file under test, not for other files which might contain dependencies. Moreover, the total number of statements in a single file is saved for further analysis.

3.3.3 Category Identification and Refinement

In this category, the iterative process of creating and refining the final categories and hierarchical structure of the taxonomy is described. The main goal of this step is to create a meaningful and consistent classification structure for the observed SBST failures.

Initially, in the data analysis step for each failure, a hypothesis regarding the possible cause is created. In many cases, these tentative labels are inconsistent, as they might have similar meanings but different names. The analysed data is reviewed again, and such inconsistencies are resolved. This process is performed iteratively

until no more inconsistencies can be found. The resulting categories represent the leaf nodes in the hierarchical taxonomy.

Furthermore, these leaf nodes would be further clustered together in categories that logically belonged together. However, this was done more with RQ5 in mind, such that these groupings could be done in a way that would serve a purpose from a solution identification standpoint. This process would be iteratively repeated until only one taxonomy category is left, thus defining the tree-like structure of the taxonomy.

Some special categories are used to define the special cases mentioned in the previous section. For cases that cannot be classified definitively, the 'Ambiguous' label is used. For cases that seemingly belong to multiple categories according to the **First Unmet Dependency** rule, the 'Converging Dependency Failure' category is used.

In some cases during the refinement process, some categories in the leaf nodes were too precise, while others were too broad. While there is no definitive rule on the specificity level of a category, in general, it should be considered from the perspective that the category is a problem to be solved. It should be broad enough to have general applicability but specific enough to represent a single problem, not a class of problems. This process led iteratively to multiple reviews of the hierarchy and recalibration in order to adjust the scope of the categories. In the end, the refinement process led to the creation of stable categories based on a well-defined process, which led to the finalisation of the taxonomy.

3.4 Data Analysis Methods

This section describes the subsequent analysis performed on the analysis. The main goal of the analysis is to gather insights about the behaviour of SBST tools and the reasons for their respective failure modes. This methodology section aims to emphasise the taxonomy's utility and practical applicability.

3.4.1 Impact Analysis

Measuring the impact of a class of failure types is not entirely trivial. There are different metrics that can be used; however, it's important to select one that does not paint an incorrect picture. In genera,l for a good metric, we need to take into account the following parameters: the number of projects in which it occurs, the number of files in which it occurs and the number of lines of code that it prevents from being tested. Additionally, we only want to use coverage data for calculating the metric, so the number of covered, uncovered and total lines.

3.4.1.1 Total uncovered lines

The simplest metric is just the total lines of code uncovered by a failure type. While easy to compute, this fails to capture the distribution of failures. For example, One

hundred untested lines in one file vs. 10 untested lines in each of 10 files both sum to 100 lines, but the latter case impacts far more of the codebase. Considering only the total uncovered lines of code would miss scattering across gaps and could be misleading. Global coverage percentages have a similar shortcoming: a single untested A 100-line file in a 100k LOC project might only drop coverage by 0.1%, yet that entire file could have high importance to the functionality of the system. A metric that focuses on such scattered coverage gaps is required.

3.4.1.2 Project/File Count

Another approach is to score each failure by the number of files or projects it affects. At one extreme, we could count each affected file equally, regardless of gap size, essentially using file_cnt or project_cnt as the score. This would strongly favour distribution, for example, 10 files each missing a few lines would score higher than one file missing 100 lines. However, completely ignoring the size of gaps isn't ideal. A failure leaving dozens of files 1-2 lines untested would score high, even though each gap is small. Conversely, a single large gap in one file would score low, yet it represents a substantial untested piece of that file. Such scenarios could occur in certain projects that repeat a similar coding pattern in multiple files, skewing the result and making it difficult to understand the reality of the impact by simply looking at a number. So, using only file count doesn't really capture the essence of impact, overemphasising breadth at the expense of depth.

3.4.1.3 Weighted sums of factors

Another approach would be to use a linear combination of the three dimensions we are interested in: lines of code, project count and file count. This allows tuning the weight of size vs. distribution. In practice, though, choosing the weight for the dimensions becomes arbitrary and hard to justify objectively. A poor choice might reintroduce bias toward one factor. Moreover, adding these different units is not straightforward. A numeric weight on file count doesn't have a value that is in any way relative to the count of lines of code. While a weighted sum is simple, it would not create a metric capturing the reality of the impact, as the impact value would highly depend on the chosen weight distribution in all cases.

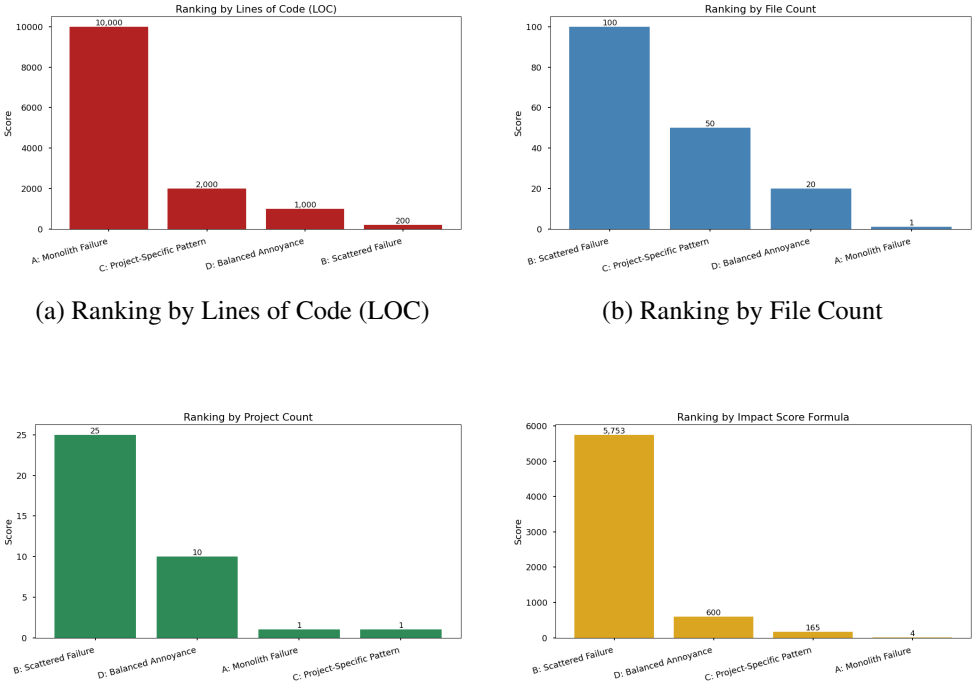
3.4.1.4 Impact Score Formula

So, it is necessary to find a formula to capture the nature of the impact correctly. One important factor to consider is that in terms of impact, we can consider $LOC <$ file_count $<$ project_count in terms of importance. That is because lines of code are often misleading, since a condition can prevent hundreds of lines from being evaluated. This condition can be circumstantial and can severely affect the impact of a classification type. Instead, the logarithm of the number of lines can be considered. Using the logarithm would ensure that the metric would not be severely affected by differences in the number of lines of code.

Furthermore, file_count and project_count metrics can be multiplied with each other and with the log of LOC, resulting in the following final formula:

$$\begin{aligned} I(\text{ClassificationType}) = \log(&\text{LOC}(\text{ClassificationType})) \\ &\times \text{FILE\_CNT}(\text{ClassificationType}) \\ &\times \text{PROJECT\_CNT}(\text{ClassificationType}) \end{aligned} \tag{3.1}$$

This metric prioritises dispersion. So if a metric impacts a similar number of LOC in a similar number of files but only in a single project, it would yield significantly lower impact compared to one that impacts multiple projects. This metric ensures that more general classification types would be more easily uncovered.



(a) Ranking by Lines of Code (LOC)

(b) Ranking by File Count

(c) Ranking by Project Count

(d) Ranking by Impact Score Formula

Figure 3.1: A comparison of different impact metrics across four hypothetical failure scenarios, highlighting the biases of simpler metrics and the balanced view provided by the Impact Score formula.

In Figure 3.1 we can see how the different approaches of measuring impact compare with one another. They have been performed on four different scenarios:

1.

Chapter 4

Taxonomy of SBST Limitations

In this chapter, a systematic taxonomy that categorises the limitations of search-based test generation tools, based on data generated by the studied SBST tools: SynTest, Pynguin and EvoSuite. As mentioned in the related work chapter, prior research has addressed individual shortcomings or focused on specific tools; however, a structured and comprehensive classification of these limitations has been lacking. This taxonomy aims to bridge that gap by offering a detailed framework for understanding where and why these tools fall short. In later chapters, the impact of these shortcomings is also analyzed to provide additional context and understanding of the state-of-the-art tools.

Figure 4.1 presents the hierarchical structure of the proposed taxonomy for SBST limitations. The root ”SBST Tool Failure Modes Taxonomy” branches into several main categories, which are further refined into more specific challenges. Subsequently, each final sub-category, with no children, is detailed with a description, criteria for its application, an illustrative code example demonstrating the SBST challenge (where applicable), and the reasoning behind its inclusion as a distinct classification criterion. Intermediate subcategories have a description and motivation, but no example, since they are provided in their respective subcategories. A flowchart for classification is presented in the next subsection 4.1 to explain the relationship between the identified coverage gaps and highlight its practical application. The taxonomy serves as a foundation for the subsequent exploration of individual problems and ultimately informs potential improvements to SBST practices.

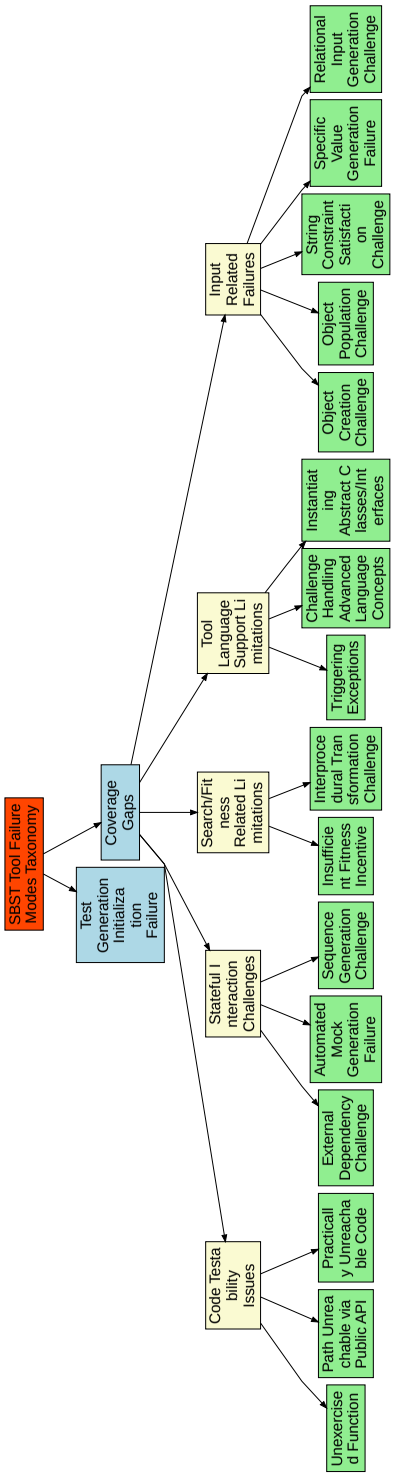


Figure 4.1: Taxonomy structure

4.1 Classification Flowchart

To ensure an unbiased and reproducible approach to the design of the taxonomy, a flowchart 4.2 has been created, presenting a clear, step-by-step approach for assigning a category to an identified coverage gap. While the taxonomy itself presents a grouping of failure nodes based on problem areas, it does not explain how these failures relate to one another. The flowchart can be used as a diagnostic tool by narrowing down the possible classification categories based on identified symptoms in the coverage data and generated tests.

The referenced flowchart contains a tree-like structure, where leaf nodes indicate classification categories, while intermediate nodes represent questions that help in finding the right one. A person analysing coverage data would start at the top of the flowchart, answering the questions in the nodes. According to the answer (YES/NO), the respective path would be taken, and either a new question would be answered, or a classification category would be assigned.

As an example, the code snippet 4.1 and the respective test 4.2 can be considered. Let's assume a coverage gap is identified,d, consisting of the entire body of the multiply method. The first question is considered: "Does the function under analysis have any lines covered?". Since no lines are covered, the next question corresponding to the "NO" branch is considered. Tests exist for the methods "add" and "get_status"; therefore, the "YES" branch is considered. The function is clearly reachable, so again, the "YES" branch is the correct choice. Thus, the final node is "Unexercised Function". Using a similar approach,h we can classify any encountered coverage gap.

Listing 4.1: Code Under Test: ‘MyCalculator‘ class with an unexercised ‘multiply‘ method.

```
class MyCalculator:
    def add(self, x, y):
        """Adds two numbers."""
        result = x + y
        if result > 100:
            print("Large sum!") # Illustrative branch
        return result

    def multiply(self, x, y):
        """Multiplies two numbers.
        This public method is intentionally not called by the example tests.
        """
        return x * y

    def get_status(self):
        """A utility function to check status."""
        return "Calculator is active."
```

Listing 4.2: Minimal conceptual test cases for ‘MyCalculator‘, exercising ‘add‘ and ‘get_status‘ but not ‘multiply‘.

```
# file: test_my_calculator.py (Conceptual)
```

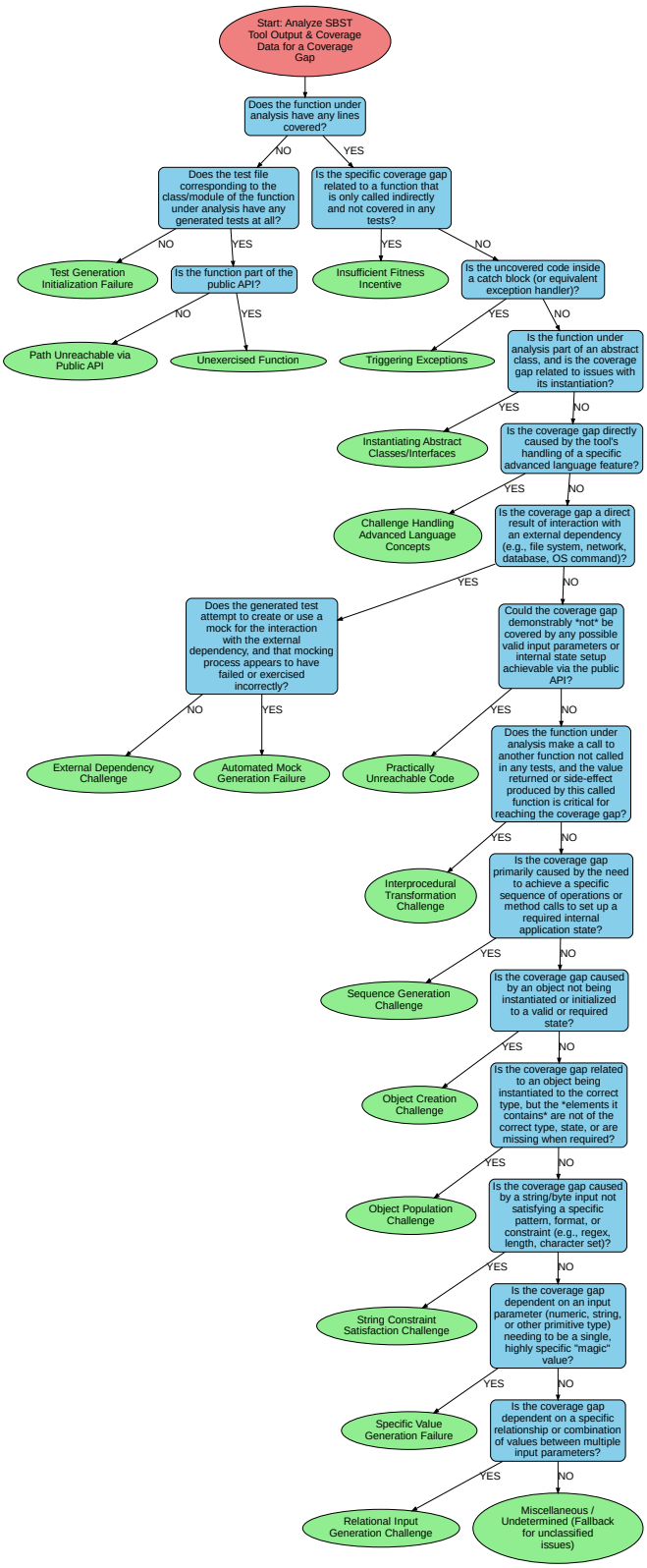


Figure 4.2: Decision flowchart for classifying SBST tool failure modes.

```
import unittest
import my_calculator # Assuming my_calculator.py is accessible

class TestMyCalculatorMinimal(unittest.TestCase):
    def setUp(self):
        """Set up for each test method."""
        self.calculator = my_calculator.MyCalculator()

    def test_add_simple(self):
        """Tests the add method with basic input."""
        result = self.calculator.add(7, 8)
        self.assertEqual(result, 15)

    def test_get_status_is_called(self):
        """Tests the get_status method."""
        status = self.calculator.get_status()
        self.assertEqual(status, "Calculator is active.")

if __name__ == '__main__':
    unittest.main()
```

It is worth mentioning that the structure of the flowchart 4.2 does not mimic exactly the structure of the taxonomy 4.1. The main purpose of the taxonomy is to create categories that can be interpreted as potential problems to be solved or researched. This kind of structure is not useful in identifying the coverage gaps when analysing code and test data. The decision flowchart can be used as a starting point in understanding the taxonomy. The categories are explained in more detail with examples in the following subsection 4.2.

4.2 Taxonomy Details

This section describes the taxonomy that explains the causes of coverage gaps in search-based test generation tools, with a focus on SBST limitations. The main goal of this section is to explain each type of failure and to provide illustrative examples, offering a comprehensive overview of each classification category.

4.2.1 Test Generation Initialisation Failure

This primary category encompasses scenarios where the SBST tool fails at the very initial stages of test generation before it can even begin its main search process or generate any test cases. This results in an empty test file with no tests, and no coverage for the file that is being tested. Although this fault has occurred across different tools, it is difficult to infer the exact reason for the failure based solely on coverage data. Potential issues can range from errors during test generation to missing dependencies, malformed code or misconfigured project environments. Analysing the logs related to test generation,

which were not available in the replication packages, could provide useful insights into this specific fault. Such failures are critical as they result in no coverage for the entire tested file.

Criteria for Classification

- Test for which the generated test file is empty, containing no tests.

Motivation This category has been created to focus on the scenarios where the tool fails to produce any coverage at all, to understand the frequency of such occurrences, but also to outline the existence of the problem and to document future potential research directions for understanding the cause more accurately. The category is distinct from other kinds of failures, since the failure here occurs before any test cases are generated. In the 'Coverage Gap' category, tests are generated, and the identified issues are related to the way in which they are generated.

4.2.2 Coverage Gaps

This category addresses coverage gaps identified in a target file for which a corresponding successfully generated test file exists. Unlike "Test Generation Initialization Failure", where failures arise during the test generation process, this category focuses on failures and limitations encountered during the tools' search, input generation and test setup.

Motivation This category has been created to contrast "Test Generation Initialization Failure" and focus on failures that happen when the tools successfully complete their test generation phase but still leave portions of the code unexecuted. The main goal of this category is to serve as an entry point into more detailed subcategories, which break down the identified coverage gaps into more manageable components from a research perspective, such as search, input generation, state handling or dealing with custom language features.

4.2.2.1 Code Testability Issues

The category focuses on coverage gaps that are primarily characterised by the test tool not testing or not being able to test the part related to the coverage gap at all. These problems are more basic compared to other problems, like input generations or those related to search capabilities. Essentially, these represent scenarios where the tools don't consider some function or class a viable test target or are not able to reach it by design.

Motivation Compared to other problems that deal with input generation or setting up some kind of test setup, this failure does not relate to the internal logic of the function at all. These failures require different research approaches and directions compared to other categories. For example, if the tool was unable to test a function it was able to test, it is important to understand why this happens. In the case of unreachable parts of the code, the research would focus on finding ways of making these parts reachable or for identifying different coverage metrics to exclude such code from coverage. More precisely categories have been defined: **Unexercised Function**, related to functions that are not called, even if the tool is capable of calling them, **Path Unreachable via Public API** for functions that the tool is unable to reach and **Practically Unreachable Code** for code snippets that cannot be consistently covered through SBST capabilities.

Unexercised function

Description In this category, all functions that were not called and used as targets, that are valid targets, are included. In this category, it's difficult to pinpoint a more precise cause due to a lack of logging information from running the algorithm. Potential causes include reduced time budget of the algorithm or functions that require complicated inputs, making it hard for the SBST tool to exercise the function.

Criteria for Classification

- The function lacks any coverage, is public and is a valid target for the SBST tool, and coverage exists for other functions in the targeted file. Private functions should not be included in this category.

Motivation Although it is unlikely to find the root cause or a general solution for this problem, the category serves as a place to gather general insights about potential improvements for the tools in this area while also serving as a statistic to understand how much of the lack of coverage is caused by the tool's inability to explore the function effectively. This category is distinct from *Insufficient Fitness Incentive*, where functions are called but their internal branches lack coverage and from *Path Unreachable via Public API*, where functions cannot be targeted directly by the tool.

Path Unreachable via Public API

Description Refers to private methods that can never be called and are never called by other functions. Usually, these are caused by certain programming patterns, like private constructors patterns in Java or represent dead code, but generally, they should not be a common occurrence.

Criteria for Classification

- Coverage gap caused by private methods that can never be accessed by the SBST tools and are never called by other functions.

Illustrative Code Example

Listing 4.3: Conceptual Example for Path Unreachable API

```
class ReportManager {
    public String generateStandardReport(String reportData) { //
        Path GSR.1
        if (reportData == null || reportData.isEmpty()) { // Path
            GSR.2
            return "STANDARD_EMPTY_REPORT"; // Line L1
        }
        return "STANDARD_REPORT_FOR:" + reportData.toUpperCase();
            // Path GSR.3, Line L2
    }
    public boolean HealthCheck() { // Path HC.1
        return true; // Line L3
    }
    private String generateSpecialAuditReport(String auditId) {
        // Path GSAR.1 (Target for 100% coverage gap)
        if (auditId == null) { // Path GSAR.2
            return "AUDIT_ERROR_NO_ID"; // Line L4
        }
        // Complex logic for a special audit, e.g., connecting to
            a secure log
        String result = "AUDIT_REPORT_FOR_ID:" + auditId + "
            _PROCESSED_SECURELY"; // Line L5
        return result; // Path GSAR.3
    }
}
```

Explanation of example: The generateSpecialAuditReport is never called and is private. It is likely dead code, yet it decreases the overall coverage when the function cannot be used in practice.

Motivation It is very similar to Practically Unreachable Code, but the difference is the possible solution to the problem. It is much easier to exclude uncalled private functions from coverage than it is to detect what is dead code

and what isn't in the context of researching new metrics for computing coverage. If we focus on the perspective of including them in coverage, the solution is also different (perhaps using a wrapper), thus warranting a separate category.

Practically Unreachable Code

Description This category refers primarily to code that is impossible to reach. This can happen when future-proofing some code by creating an exception that is impossible to trigger or can be just dead code that is impossible to trigger. In this category, flaky code that causes some branches to be impossible to trigger consistently is also included.

Criteria for Classification

- Coverage gap caused by being unable to consistently cover the code.
- Coverage gap caused by code that is not reachable

Illustrative Code Example

Listing 4.4: Conceptual Example for Practically Unreachable

```
import random

def trigger_rare_bonus_feature(user_id: str) -> str:
    if random.randint(1, 1_000_000) == 42: # Path RBF.A (Target
        for coverage gap)
        return f"User {user_id}: Congratulations! You've hit the
            ultra-rare bonus!" # Line L1
    else: # Path RBF.B
        return f"User {user_id}: No bonus this time. Standard
            processing." # Line L2
```

Explanation of example: This function emulates a lottery event. As it can be seen, it is highly unlikely to trigger the function and is also impossible to do so reliably.

Reasoning for Classification Criterion Although not a fixable coverage gap, it is still classified as there might be research efforts that focus on changing how coverage is calculated to exclude such occurrences, thus indirectly increasing the overall coverage.

4.2.2.2 Stateful Interaction Challenge

This category has been created to focus on failure modes relating to a dependency on an external setup, outside of the function. This problem is different from others since it relates a lot more to the environment in which the function is run and the interaction between different functions and components in the project, rather than focusing directly on the function under test itself.

Motivation Compared to other categories, this category focuses on how to create the external state such that the function can be successfully tested. It is a unique kind of problem, since it involves mostly understanding the type of dependency the requirements the tested function has from the environment and how to effectively mock or emulate the scenario that the function requires. In this category a few subcategories are identified: **External Dependency Challange**, relating to issues that interact with external environments, such as the OS, or the file system, **Automated Mock Generation Failure**, which relates to cases where such external dependencies are successfully mocked, but where the mocks fail to emulate a behaviour and **Sequence Generation Challange**, which relates to creating the right state (perhaps by calling methods like setters, for example) prior to testing the function.

Environmental Interaction / External Dependency Challenge

Description This category refers to all cases where a coverage gap is caused by some external dependency that cannot be influenced trivially through a simple test case. This could refer to accessing the File System, interacting with the OS, dependency on a programming language or state, an external library or even another application running in the background. Typically such errors create vast gaps in code coverage. The lack of support for such features often leads to runtime errors. There is also an inherent danger to running such tests, considering they could interact in a non-modular way with external systems.

Criteria for Classification

- The Unit Under Test interacts directly with resources or stats that are outside the immediate scope of the tested unit.
- The SBST algorithm lacks direct control over the dependency.
- There is a direct link between the coverage gap and the interaction with the external system. A direct link refers to either causing a runtime error or influencing values or external states that are further used in the same function in explicit or implicit conditional statements.

Illustrative Code Example

Listing 4.5: Function behavior depends on an environment variable

```
import os

def get_feature_flag_status(feature_name: str) -> bool:
    """Checks if a feature flag is enabled via an environment
        variable."""
    env_var_name = f"FEATURE_{feature_name.upper()}"
    status_str = os.environ.get(env_var_name)

    if status_str is None:
        return False # Path A: Environment variable not set

    if status_str.lower() == "true" or status_str == "1":
        return True # Path B: Environment variable is true
    else:
        return False # Path C: Environment variable is set but
            not 'true'
```

Explanation of example: An SBST tool testing ‘get_feature_flag_status‘ would likely only cover Path A (and perhaps Path C if ‘os.environ.get‘ returns an empty string or another default for unset variables in the test environment). To cover Path B, the specific environment variable would need to be set to ”true” in the test execution environment, which the SBST tool typically does not control as part of its input generation for the ‘feature_name‘ parameter.

Reasoning for Classification Criterion This type of failure has been grouped together because they have similar solutions. Mocking implies creating emulated environments that mimic the behaviour of the real environments on which the code depends. Although this category has only one subcategory, failures for which mocking was not used are still classified either due to it not existing in some tool under test or due to the algorithm not instantiating it properly. Failures which occur due to mocking not functioning properly belong to the next subcategory.

Automated Mock Generation Failure

Description This failure relates to cases where the SBST tools successfully attempt to mock an external environment dependency but fail. In a sense, this failure mode is similar to the Object Creation/Initialisation Challenge. The main difference is that instead of creating a direct input parameter using a constructor of an existing type, a mock has the goal of mocking an external dependency.

Criteria for Classification

- Some evidence must exist that the tool attempted to generate a mock object (typically in EvoSuite, this can be detected by inspecting variable names in the test file).
- The mocked object or environment leads directly to a coverage gap, either through a runtime error, or by influencing a state or value that is later used in either an explicit or implicit conditional branch.

Reasoning for Classification Criterion This category is significant because it emphasises the existing limitations in mocking implementations. Since mocking is the most widely known solution also for the coverage gaps classified in the Environmental Interaction / External Dependency Challenge category, it makes sense to have this category to understand how it relates to other gaps that could potentially be solved by it. It also helps in comparing tools and understanding the differences in their features and solutions.

Sequence Generation Challenge

Description This case relates to coverage gaps caused by a lack of setup in the test. This can be either having to call functions in a certain sequence in order to trigger a state and generally relates to emulating an implicit manner in which the function that is tested is supposed to be called in a broader context.

Criteria for Classification

- Setting the value of an object parameter under test to a function value that is not called/tested after the function returns.
- A branch is not executed due to some global variable or object parameter that has its value updated through a different function goal or directly.
- A function or a class is returned, which remains untested.
- A function is not called enough times to trigger a state.
- Functions related to the environment of the tested function are not tested in a logical order in order to trigger

Illustrative Code Example

Listing 4.6: Conceptual Example for Stateful Logic

```
class UserSession:
    def __init__(self):
        self._is_logged_in = False # Internal state: user is
            initially not logged in
```

```
def login(self, username: str, password_hash: str) -> bool: #
     Method 1: Enables a state
    if username and password_hash:
        self._is_logged_in = True # State is changed
        return True
    return False

def access_premium_feature(self) -> str: # Method 2: UUT,
    needs prior state
    if self._is_logged_in: # Path US.APF.A (Target for
        coverage gap)
        return "Accessing Premium Feature..." # Line L_APF1
    else: # Path US.APF.B (Default path if not logged in)
        return "Access Denied: Please log in first." # Line
            L_APF2

def logout(self): # Another state-changing method
    self._is_logged_in = False
    return "Logged out"
```

Explanation of example: This example represents a simple login system. When testing access_premium_feature, the login method needs to be called first. Without calling it initially, it is impossible to cover the code following the first branch in the access_premium_feature method.

Reasoning for Classification Criterion A known and common issue with the SBST code. Although it might seem closer to integration testing, it is still part of unit testing as long as the additional setup is supposed to be done within the unit that is being tested. This issue is also related more to how the test organises the calls to the functions rather than specific inputs.

4.2.2.3 Search/Fitness Related Limitations

This category focuses on cases where the capabilities of the search algorithm are diluted, meaning the guidance provided by the fitness function becomes weak and insufficient to guide the algorithm towards specific uncovered targets. Judging by the title of the category, it might seem like any issue related to input can fall into this category. The distinction is that the focus in this case is on issues related to target identification for computing the fitness and guiding the algorithm, rather than the computation of fitness based on the input in relation to those targets.

Motivation This is a different category from others since it focuses on challenges in how fitness targets are selected and prioritised. In this context, two

categories have been identified that represent the same problem but are considered from different perspectives: **Insufficient Fitness Incentive** and **Interprocedural Transformation Challenge**. The first focuses on functions that are called by a tested function but the fitness mechanism fails to adequately guide the search to focus on the internal branches of the indirectly reached function. The second one focuses on functions that have coverage gaps directly caused by calls to such functions that hinder coverage.

Insufficient Fitness Incentive / Goal Limitation

Description This category refers to cases where functions receive no guidance for their objectives and therefore no coverage. Essentially, they are only used as a black box and do not contribute to the overall fitness goal of the function.

Criteria for Classification

- Functions are called within functions, and are tested primarily by the SBST tools. These functions don't contribute to the fitness incentive as they do not provide any targets or branches to be covered directly due to being inaccessible for instrumentation.

Illustrative Code Example

Listing 4.7: Conceptual Example for Insufficient Fitness

```
class UserProfileProcessor {
    private String determineUserSegment(int activityScore, int
        tenureDays) {
        if (activityScore > 950 && tenureDays > 730 && (
            activityScore % tenureDays < 5)) { // Path DUS.A
            return "SEGMENT_HEV"; // Highly Engaged Veteran - Line
                 L_H1
        } else if (activityScore > 800 || tenureDays > 365) { //
            Path DUS.B
            return "SEGMENT_ENGAGED";
        } else { // Path DUS.C
            return "SEGMENT_STANDARD";
        }
    }

    // Public UUT method
    public String getMarketingCampaign(int userActivity, int
        accountAgeDays) {
        String segment = determineUserSegment(userActivity,
            accountAgeDays);
        if ("SEGMENT_HEV".equals(segment)) {
```

```
            return "Campaign: Exclusive Veteran Offer"; // Line
                L_UUT2
        } else if ("SEGMENT_ENGAGED".equals(segment)) { // Path
            GMC.B
            return "Campaign: Engagement Bonus";
        } else { // Path GMC.C (Covers "SEGMENT_STANDARD")
            return "Campaign: Standard Newsletter";
        }
    }
}
```

Explanation of example: The lack of coverage in the determineUserSegment, by not having all the branches triggered, is due to the fact that this function's branches are not used as an objective in the fitness function for testing getMarketingCampaign.

Reasoning for Classification Criterion This is a special case, considering these functions are not directly targeted. It might seem similar to the Interprocedural Transformation Challenge; however, there is a difference of perspective. In this case, we consider the gaps in the function that was the callee, while in the other category, we consider the gaps in the function that was the caller due to this limitation.

Interprocedural Transformation Challenge (Guidance obscured by callee)

Description This is a similar fault to Insufficient Fitness Incentive. In that case, the coverage gaps within the private function were classified, where in this case, the coverage gaps in the caller function are classified.

Criteria for Classification

- The reasons for a failure are directly connected to a state not being returned (or set globally) from a called function for which no branching information can be extracted from within the function that is tested. It does not apply when the failure is in a dependent function that only has one direct execution path

Illustrative Code Example

Listing 4.8: Conceptual Example for Interprocedural

```
class UserProfileProcessor {
    private String determineUserSegment(int activityScore, int
        tenureDays) {
        if (activityScore > 950 && tenureDays > 730 && (
            activityScore % tenureDays < 5)) { // Path DUS.A
```

```
            return "SEGMENT_HEV"; // Highly Engaged Veteran - Line
                L_H1
        } else if (activityScore > 800 || tenureDays > 365) { //
            Path DUS.B
            return "SEGMENT_ENGAGED";
        } else { // Path DUS.C
            return "SEGMENT_STANDARD";
        }
    }

    // Public UUT method
    public String getMarketingCampaign(int userActivity, int
        accountAgeDays) {
        String segment = determineUserSegment(userActivity,
            accountAgeDays);
        if ("SEGMENT_HEV".equals(segment)) {
            return "Campaign: Exclusive Veteran Offer"; // Line
                L_UUT2
        } else if ("SEGMENT_ENGAGED".equals(segment)) { // Path
            GMC.B
            return "Campaign: Engagement Bonus";
        } else { // Path GMC.C (Covers "SEGMENT_STANDARD")
            return "Campaign: Standard Newsletter";
        }
    }
}
```

Explanation of example: Here, we reuse the same example as in Insufficient Fitness Incentive. This time, we consider the branches that won't be covered in getMarketingCampaign as the coverage gaps classified in this example.

Reasoning for Classification Criterion This is a commonly known research problem. Additionally, it relates directly to how a lack of visibility in fitness affects the exploration of branches. Even though they are closely related to Insufficient Fitness Incentive, they are still quite different, as they focus on different perspectives.

4.2.2.4 Tool Language Support Limitation

This category focuses on coverage gaps that result in a language feature not being supported properly in a specific tool. These types of challenges are more related to engineering problems rather than research ones. Typically, generated tests for functions that use such features result in weird, unexpected behaviours that can lead to runtime errors in the test function, even before the tested function is called.

Motivation This category focuses more on areas of improvement for the tools' implementation. They do not relate directly to the search process, but rather to general support for some features. Some features repeat more often, while others repeat more rarely. The **Triggering Exceptions** and **Instantiating Abstract Classes** were identified as recurring issues across multiple languages and projects. The rest of the specific features were classified as **Challenge Handling Advanced Language Concepts**.

4.2.2.5 Challenge Handling Advanced Language Features / Protocols

Description This category refers mainly to certain programming patterns or language features which don't generalise well beyond their programming language. Implementing solutions for this category is a very targeted and likely low-impact effort. In this failure mode, annotation handling or generator functions in Python can be included. In JavaScript, features like frozen objects or argument objects could fall within this category.

Criteria for Classification

- The coverage gap must relate to a feature that is specific to a certain programming language and has no feature in the other languages used in the classification (Python, Java), that could be solved in a similar way.
- The coverage gap could potentially only be overcome by addressing the identified language-specific problem.
- Generators in Python
- Frozen state, arguments object in JavaScript

Illustrative Code Example

Listing 4.9: Function branches based on Object.isFrozen()

```
// Function that behaves differently based on whether the object
    is frozen
function checkMutability(obj) {
  if (Object.isFrozen(obj)) {
    // PATH A: Logic for frozen objects
    return 'object_is_frozen';
  } else {
    // PATH B: Logic for mutable objects
    // (Could potentially attempt modification here if needed for
        further logic)
    // obj.status = 'mutable_and_processed';
    return 'object_is_mutable';
  }
}
```

Explanation of example: This JavaScript function checks if the input ‘obj‘ is frozen using ‘Object.isFrozen()‘. An SBST tool needs to generate both mutable and frozen object instances for the ‘obj‘ parameter to cover both Path A and Path B. If the tool’s strategies for object creation do not include applying ‘Object.freeze()‘, or if it rarely explores this possibility, it may fail to generate a frozen object, leaving Path A uncovered.

Reasoning for Classification Criterion Although specific language features do not typically have solutions that generalise well, they are still important as a classification criterion. It allows us to understand the impact of such features, how often they occur and whether it is worth investing time in finding custom solutions for these features for individual tools.

Triggering Exceptions

Description This category addresses the common challenge where SBST tools fail to generate test cases that intentionally trigger specific, handled exceptions. They primarily refer to implicit branches that are not triggered and are not detected by the testing tool. As a result, the code within catch or except blocks, which is designed to handle these exceptional circumstances, often remains untested, leaving the application’s error-handling logic unverified.

Criteria for Classification

- The uncovered lines of code reside entirely within a catch (or except, rescue, etc.) block.
- The tool fails to generate a test case that causes the specific exception type handled by that block to be thrown from the corresponding try block.

Illustrative Code Example

Listing 4.10: Conceptual Example for Triggering Exceptions

```
public class UserValidator {
public String processUserAge(int age) {
       try {
           if (age < 0) {
               throw new IllegalArgumentException("Age cannot be
                   negative.");
           }

           return "Processing user with age: " + age; // Line L1
       } catch (IllegalArgumentException e) {
           // This entire block is the coverage gap
```

```
            System.err.println("Invalid age provided."); // Line
                L2
            return "ERROR: Invalid Age"; // Line L3
        }
    }
}
```

Explanation of example: An SBST tool testing processUserAge will likely generate various positive integers for the age parameter, successfully covering Line L1. However, the search algorithm often lacks a specific fitness incentive to generate a negative integer, which is required to throw the IllegalArgumentException. Consequently, the catch block (Lines L2 and L3) remains unexecuted, as the condition to trigger it is never met.

Reasoning for Classification Criterion Exception handling is a fundamental aspect of robust software design, and this failure pattern is common across most modern programming languages. It requires a separate category because it relates to a very specific problem: triggering an implicit branch or violating a contract within a code block.

Instantiating Abstract Classes

Description Instantiation of Abstract Classes appears to be a recurring problem with SBST tools. This programming pattern often leads to untested code and runtime errors, since most languages have prevention mechanisms for preventing direct testing. This fault is different from Challenge Handling Advanced Language Features precisely because it is a pattern popular across many languages.

Criteria for Classification

- Untested method that follows the abstract class implementation pattern in that language
- Abstract class implementation pattern is successfully used in testing, but uses an implementation object that leads to a lack of coverage

Illustrative Code Example

Listing 4.11: Conceptual Example for Abstract Class

```
from abc import ABC, abstractmethod

# ---- Abstract Class Definition ----
class DataHandler(ABC): # Inherits from ABC
```

```
def process_data(self, data: str) -> str: # Path DH.1
    if not data: # Path DH.2
        return self.get_empty_message() # Line L1
    transformed_data = self.transform_data(data) # Line L2
    return f"Processed: {transformed_data}" # Line L3

@abstractmethod
def transform_data(self, data: str) -> str: # Line L4
    pass # Must be implemented by concrete subclasses

@abstractmethod
def get_empty_message(self) -> str: # Line L5
    pass # Must be implemented by concrete subclasses
```

Explanation of example: When such an example is encountered, Pynguin would typically attempt to directly test the abstract class by instantiating a DataHandler() object. This would lead to concrete methods like process_data left untested, even if they are defined.

Reasoning for Classification Criterion The usage of Abstract Classes is a common programming pattern in most programming languages that support object-oriented programming. Although the implementation can differ across different languages, they still share similar features and, more importantly, solutions. In most cases, coverage can be achieved by identifying a class that is a concrete implementation of the abstract class, which can be tested to produce coverage for the abstract class.

4.2.2.6 Input Related Failures

This category relates to the most common and impactful problem in SBST tools. Generating appropriate input is of utmost importance for achieving appropriate coverage. In dynamically typed languages, the additional challenge of identifying the type correctly makes the problem even more difficult. Other challenges include finding appropriate gradient functions to guide the algorithm for certain types of inputs or simply generating objects with the correct structure.

Motivation This category is different from others since it primarily deals only with the analysed function. The main problem is understanding the structure of the input and the values that should be yielded in order for the relevant coverage gaps to be triggered. This category is the most populous as it has five subcategories: **Object Creation Challenge**, focusing on problems with generating objects with the right structure, **Object Population Challenge**, focusing on cases where lists and dictionaries are successfully created but are populated

with incorrect objects, **String Constraint Satisfaction Challenge**, focusing on problems with generated strings not conforming to the pattern expected by the function, **Specific Value Generation Failure** focusing on cases where a condition requires a certain value to be triggered and **Relational Input Generation Challange**, focusing on cases where multiple input parameters that interact between each other and need specific values in order to resolve a coverage gap.

Object Population Challenge

Description This category describes a specific failure mode where an SBST tool correctly identifies and instantiates a collection-type object (such as a list, array, or map) but fails to populate it with elements of the correct type or structure. Essentially, the object is correctly built, but the objects related to it are not.

Criteria for Classification

- A collection-type object (e.g., list, dictionary, array) is correctly instantiated for a function parameter and a coverage gap is caused because the elements populated by the tool do not match the expected type or structure, leading to a runtime error that halts execution before subsequent lines are covered.

Illustrative Code Example

Listing 4.12: Conceptual Example for Object Population Challenge

```
def calculate_total_bill(orders: list):
    total_price = 0.0
    # The tool might create a list, but populate it with
    # integers or strings instead of the expected dictionaries.
    try:
        for order_item in orders:
            # A TypeError occurs here if order_item is not a
                dictionary.
            # A KeyError occurs if the dictionary lacks the 'price
                ' key.
            item_price = order_item['price'] * order_item['
                quantity']
            total_price += item_price # This line is the coverage
                gap
    except (TypeError, KeyError):
        return -1.0 # Error code

    return total_price
```

Explanation of example: An SBST tool testing calculate_total_bill may correctly infer that the orders parameter should be a list. However, it might populate it with simple data like [1, ”apple”, 3]. When the code enters the for loop and attempts to perform order_item['price'] on the integer 1, a TypeError is immediately raised.

Reasoning for Classification Criterion This failure mode deserves its own category because it is distinct from the broader ”Object Creation Challenge.” In this case, the object is not created incorrectly, but is typically a dynamic structure that has objects. So, in a way, it is a more complex failure mode as it encapsulates object creation.

Object Creation Challenge

Description This category refers to a rather broad category. It encapsulates essentially any object that is not instantiated with the correct type or the correct structure. This includes cases in dynamically typed languages where an incorrect type is passed (a string instead of a number), but also cases where some property is missing from an object or an object is incorrectly identified as either being callable or iterable. This might seem like a problem specific only to dynamically typed languages, but it is also rather frequent in statically typed languages. A common pattern is using the general Object type in Java and having conditional branches using instances. Moreover, in either type of language, it can happen that a certain state is required from the object that represents some kind of complex logic (a graph object that has a loop).

Criteria for Classification

- Incorrect type provided to trigger a conditional statement. This also applies to generic types or subtypes in statically typed languages.
- A property on an input object is missing, which leads to a runtime error or to the failure of triggering a conditional statement
- An object is constructed with values that fail to trigger a certain execution path when a function is called on that object that uses its state
- An object is required to represent complex logic in order to trigger conditional statements, which it fails to do
- Failure to provide an object type that meets some property based on code execution (an iterable or callable object)

Illustrative Code Example

Listing 4.13: Conceptual Example for Object Creation

```
def get_user_access_details(user_account):
    if not user_account.is_active(): # Raises AttributeError or
        TypeError
        return "Status: Inactive Account" # Path A

    profile = user_account.profile # Raises AttributeError if
        missing

    if not isinstance(profile, dict):
        return "Error: Profile is not a valid dictionary" # Path
            B

    subscription_level = profile.get("subscription_level", "
        default")

    if not isinstance(subscription_level, str):
        return "Error: Subscription level has invalid type" #
            Path C

    if subscription_level == "premium":
        return "Access: Granted to Premium Features" # Path D
    elif subscription_level == "standard":
        return "Access: Granted to Standard Features" # Path E
    else:
        return "Access: Basic/Default Features" # Path F
```

Explanation of example: The function presented represents multiple ways in which an incorrectly generated object can fail. In order for Path A to be triggered, the user_account parameter should have an is_active parameter that is callable. Access to the profile attribute on the next line will raise an AttributeError if the specific attribute does not exist on the provided object. Path B can only happen if the generated profile attribute is of an exact dictionary type. Next, the subscription_level variable requires that the key with the same name exists in the dictionary, which imposes another tighter restriction on the input object generated by the tool. Path C requires that the value corresponding to the key in the dictionary be a string.

Reasoning for Classification Criterion These classification criteria have been created to encompass the most likely prevalent problem that affects SBST tools. Although this category could be split into multiple subcategories, it has been decided to keep it as one, considering that many solutions for one of the problems presented in the Criteria of Classification affect one another. Additionally, many of these challenges have common solutions. Increasing granularity would increase the focus on very specific problems instead of the more general ones.

String Constraint Satisfaction Challenge

Description This category refers essentially to any fault caused by string input that does not follow a specific pattern intended to trigger an implicit or explicit branch. The difficulty of this problem lies in the fact that, in many cases, it is challenging to generate fitness for various string operators, regular expressions, and the numerous string-related functions in the standard library, which must be handled separately.

Criteria for Classification

- A string input (direct or as a parameter on an object) fails to trigger a branch as a result of being used directly or indirectly in multiple functions related to string processing. A string processing function can be something like: find(), startsWith(), match(), etc.

Illustrative Code Example

Listing 4.14: Conceptual Example for String Constraint

```
import re

def get_session_id_from_log(log_line: str) -> str | None:
    # "SID:[A-F0-9]{8}-USER:[a-z]+"
    # Example valid string: "SID:A1B2C3D4-USER:john"

    # Line L1: Regular expression search for the pattern
    pattern = r"SID:[A-F0-9]{8}-USER:[a-z]+"
    match_obj = re.search(pattern, log_line if log_line else "")

    if match_obj: # Path SUT.A (Target for potential coverage gap
        )
        # This path is taken only if the complex string
            constraint (regex) is met.
        return match_obj.group(0) # Line L2 (Returns the matched
            session ID string)
    else: # Path SUT.B
        # This path is taken if the string does not match the
            pattern.
        return None # Line L3
```

Explanation of example: In the example, in order for the path SUT.To be taken, the input parameter log_line needs to have the pattern for the described session_id in order to trigger that branch.

Reasoning for Classification Criterion Although this seems like a broad category, it has been grouped since all the solutions are related to string-related fitness calculations. Even though it might seem that there are many different standard library functions in different languages for processing strings, in many languages, they are the same, and handling fitness for 10-15 functions would cover most of the coverage gaps.

Relational Input Constraint Challenge (between multiple direct UUT parameters)

Description This category relates to a specific case where triggering a branch requires some complex interaction between multiple parameters of the function. Handling fitness calculation for such cases is often more complex, considering the search space is significantly larger. The interdependency between multiple parameters might obfuscate the fitness and make guidance less effective.

Criteria for Classification

- Two or more input parameters contribute to the computation of a value used in a branch. Cases for boolean parameters or those with a limited number of possible values are excluded. The emphasis is placed on relational constraints where the involved parameters have continuous, large, discrete, or complex structural domains.

Illustrative Code Example

Listing 4.15: Conceptual Example for Relational Input

```
def evaluate_sensor_synergy(reading_a: float, reading_b: float,
    optimal_midpoint: float) -> str:
    closeness_penalty = abs(reading_a - reading_b) # Lower is
        better

    current_midpoint = (reading_a + reading_b) / 2.0
    midpoint_deviation_penalty = abs(current_midpoint -
        optimal_midpoint) # Lower is better

    synergy_metric = (0.7 * closeness_penalty) + (0.3 *
        midpoint_deviation_penalty) # Line L1

    if synergy_metric < 0.5: # Path RSS.A (Target for coverage
        gap)
        return "High Synergy Detected" # Line L2
    elif synergy_metric < 2.0: # Path RSS.B
        return "Moderate Synergy"
```

```
    else: # Path RSS.C
        return "Low Synergy or Dissonance"
```

Explanation of example: Analysing the provided example, it can be seen that in order to generate input to trigger the branches, there needs to be synergy between the three input variables. Standard fitness guidance is unlikely to trigger the branch as the search space is immense.

Reasoning for Classification Criterion This special case should not be combined with others because it operates under more complex rules. It operates in a larger search space, so the solutions for other problems might not appear as effective in this specific case.

Specific Value Generation Challenge (e.g., Magic Numbers, Sentinels)

Description This category refers to the challenge when branches trigger when compared directly to some value, or some specific single value is required for them to trigger a branch.

Criteria for Classification

- A specific constant that is defined in the codebase is not compared against in the branch.
- Some input is expected to meet a specific special value, like isNan, or isInfinite.

Illustrative Code Example

Listing 4.16: Conceptual Example for Specific Value

```
import math

def handle_computation_result(result: float) -> str:
    if math.isinf(result): # Path SVal.A (Checks for infinity)
        if result > 0: # Path SVal.A1 (Positive Infinity)
            return "Computation resulted in Positive Infinity" #
                Line L1
        else: # Path SVal.A2 (Negative Infinity)
            return "Computation resulted in Negative Infinity" #
                Line L2
    elif math.isnan(result): # Path SVal.B (Handles another
        specific value: NaN)
        return "Computation result is Not a Number (NaN)" # Line
            L3
    return f"Result: {result}" # Line L5
```

Explanation of example: In the example, it is shown simply how an input is expected to either hold an infinity (positive or negative), or a NaN value in order to trigger branches.

Reasoning for Classification Criterion These classification criteria are separate mainly because, in such cases, there is a single possible case where the branches are triggered. Consequently, a different approach should be taken rather than trying to guess the value through standard fitness approaches. In many cases, the input required to trigger a branch could be inferred statically.

Chapter 5

Analysis of Taxonomy

Despite the increase in research and the maturity of search-based software testing, this thesis demonstrates that significant performance limitations still exist, even in the state-of-the-art tools. The analysis quantifies these gaps, revealing the failure patterns that are responsible for the majority of untested code. By examining these patterns through various metrics, this section establishes a foundational understanding of where and why these tools fail. This data-driven, metric-based approach allows for the identification of opportunities for improvement, with specific solutions proposed for the more critical failure categories.

5.1 How much code is still untested?

To start creating an understanding of what each coverage gap represents and how it affects the test generation process, a few basic metrics can be used to create an initial impression and understanding: Total lines of code, the number of projects in which a fault is found (Project count) and the number of files in which a fault is found (File count).

Looking at Figure 5.1, it is clear that the **Unexercised Function** is the largest category by lines of code, affecting 33% of the total failure code. This category represents uncalled functions, so a fully unexecuted function will contribute to a large number of uncovered lines. The **Object Creation Challenge** is also very common across all languages. It is a broad category, leading to a large amount of uncovered code. It can also be seen that the top two categories are far more prevalent than the rest and that the first six categories make up 80% of the total uncovered LOC across all categories. While LOC highlights the quantity of contested code, it can easily be skewed by some large functions. For example, it is unclear whether the 'Unexercised Function' category dominates because it is concentrated in a few large problematic files or if it is a truly widespread issue. Figure 5.2 addresses this uncertainty.

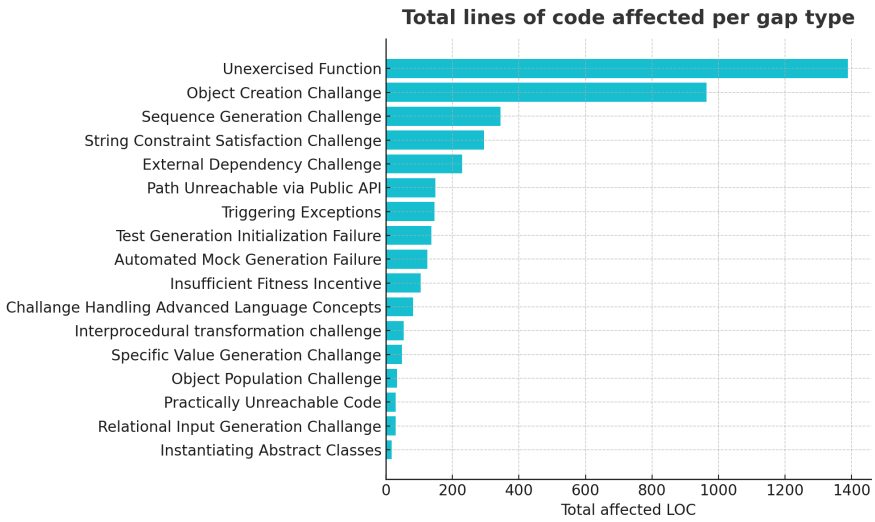


Figure 5.1: Total LOC by Final Classification Category

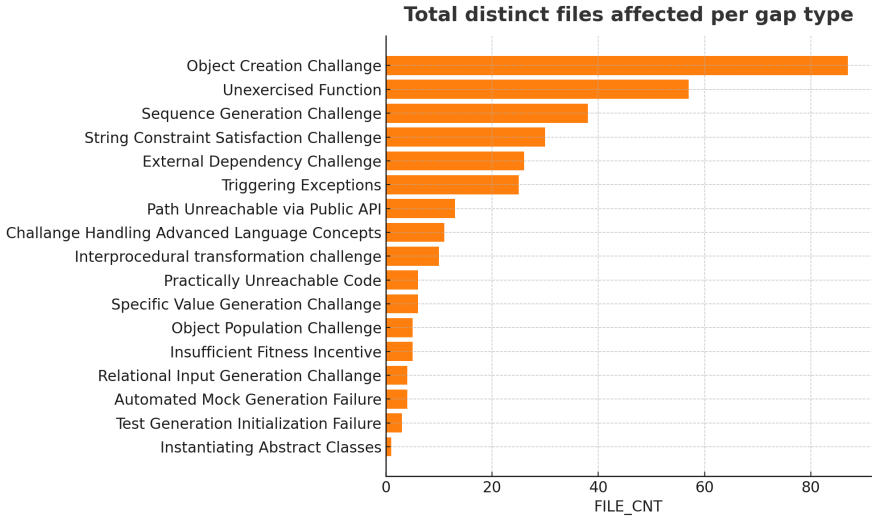


Figure 5.2: Total Files by Final Classification Category

It can be observed that the distribution has a similar shape, but is more evenly distributed in this case. It can be seen that some categories change their order, most importantly, Object Creation Challenge and Unexercised Function. As mentioned in the previous paragraph, this is due to Unexercised Functions contributing a disproportionately large amount of uncovered code, given the nature of the category. However, files alone also do not tell the whole story. There can be a project that uses a pattern that repeats itself many times over, leading to the same failure. It would be useful to also look at Figure 5.3 to understand the distribution by projects.

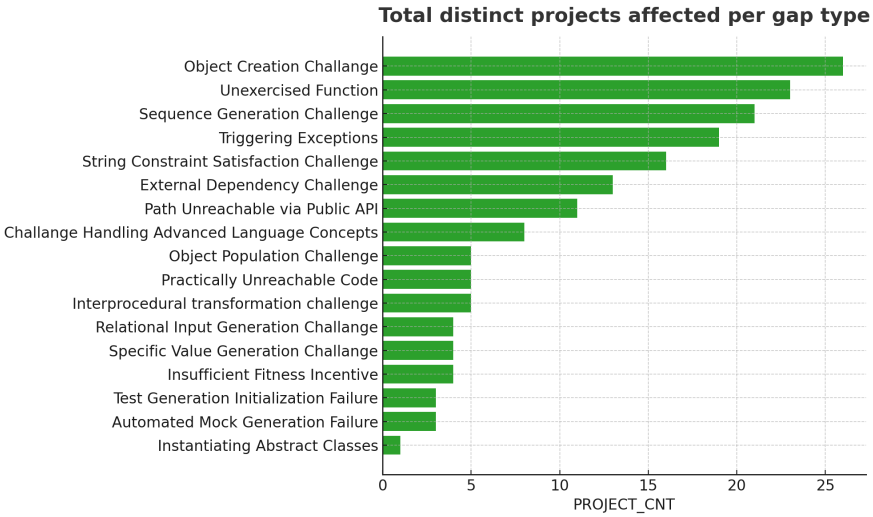


Figure 5.3: Total Projects by Final Classification Category

In Figure 5.3, a different pattern can be noticed. The distribution is almost linear, indicating less domination of the top categories. This makes sense, since covering a larger number of projects is not difficult, even with a lower number of faults. This type of spread is an indicator that the taxonomy is likely to generalise well, as the categories are present across multiple projects, rather than being concentrated in a small cluster of categories. Again, while being present in many projects indicates that a category generalises well, it does not indicate that it is truly problematic. A specific failure may only occur once or twice in each project, affecting a small number of lines of code, making it a low-priority issue. The top categories appear to be consistent across all three charts; however, they do change positions in some cases, making these metrics somewhat ambiguous. To facilitate a better understanding of what metric is truly impactful, we will use the impact metric from Section 3 to continue the analysis.

5.2 High-Leverage gaps

The previous analysis demonstrated that some categories seem more prevalent than others; it is difficult to determine a clear order between them. Even the top two functions across all three charts, 'Object Creation Challenge' and 'Unexercised Functions', are not consistent in their ranking. In this section, we will focus on the previously defined impact metric to create a clear ranking in terms of impact.

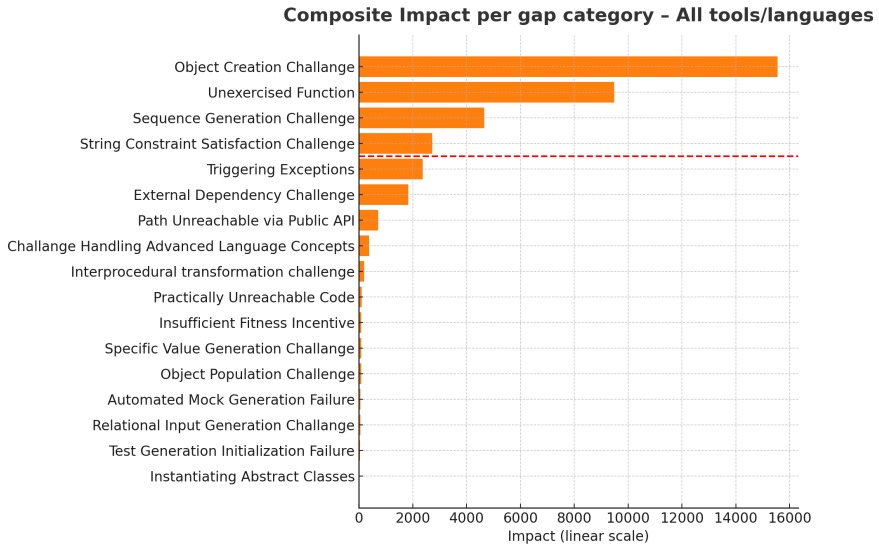


Figure 5.4: Impact by Final Classification Category

The Figure 5.4 reveals the calculated impact. The graph is similar in shape to the previous ones, with a larger tail that highlights clearer differences between categories. Interestingly, the first four categories —'Object Creation Challenge', 'Unexercised Function', 'Sequence Generation Challenge', and 'String Constraint Satisfaction Challenge' — represent 84.3% of the total impact across all categories. So 23.5% of the categories cause 84.3% of the total impact, highlighting an alignment with the Pareto principle. The primary insight in this case is that focusing on and solving these four problems is sufficient to improve the performance of SBST tools significantly. However, it is unclear whether this impact is universal to the entire SBST paradigm or whether it is language-specific, and this particular aggregation is not representative of the general problems that occur.

Figure **??** represents the normalised impact metric for each metric. Normalising impact makes the comparison between distinct categories clearer. It reveals where the most impact comes from in a specific language.

In the chart, JavaScript is ahead in the 'Object Creation Challenge' compared to the other languages. This can be explained by the more structured ap-

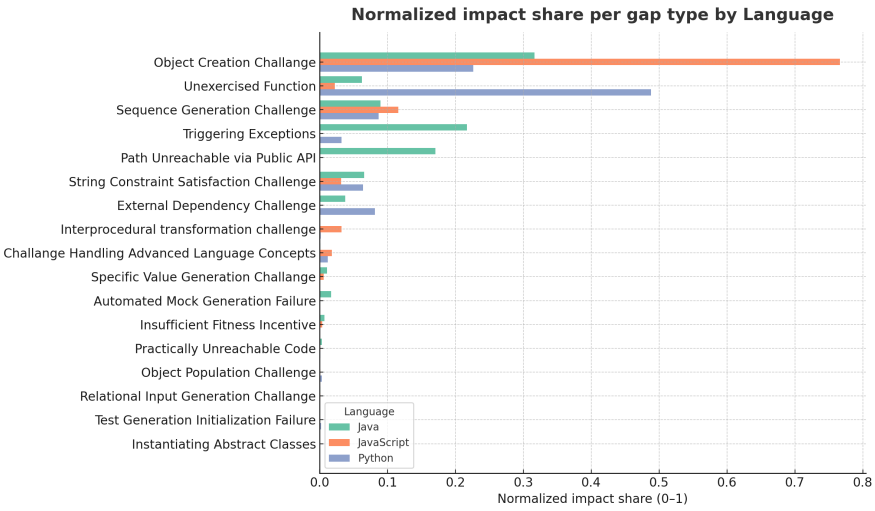


Figure 5.5: Impact by Language by Final Classification Category

proaches in Java, being a statically typed language, and in Python, where constructors with defined parameters are extensively used. It is much more common in JavaScript to create objects on the fly, compared to Python.

Python emerges as a notable outlier in the 'Unexercised Functions' category. This suggests a potential inefficiency within the Pynguin algorithm's function selection process, a point that will be revisited later in the analysis.

Two of the bigger categories in which Java is an outlier are 'Triggering Exceptions' and 'Path Unreachable via Public API'. In the analysed code, Java used exceptions far more frequently than the other two languages, which explains the spike in this direction. For the 'Path Unreachable via Public API' category, Java uses this pattern extensively, while Python does not have private functions at all. This fault was not observed in the smaller JavaScript dataset.

There are other discrepancies across smaller features; however, they will not be discussed due to their lower respective impact. Popular coding patterns or language features have an impact on which category is more impactful, simply because more code is written in these patterns. This analysis also reveals that, although the four identified problems are prevalent across all languages, their impact is not consistent across them. While these language-specific differences are insightful, they lead to a higher-level question: how does the typing system of a language, static or dynamic, influence the impact of the failure modes?

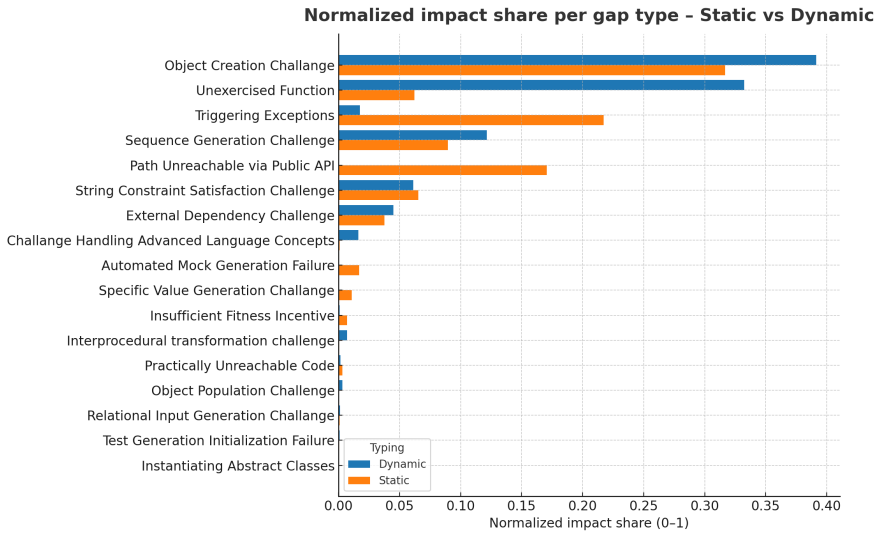


Figure 5.6: Impact by Language Type by Final Classification Category

5.3 Divergent Impact Signatures: A Static vs. Dynamic Comparison

To answer this question, Figure **??** directly compares the normalised impact between statically and dynamically typed languages. The top category for either type of language is 'Object Creation Challenges'. While it might seem counterintuitive, due to Java being statically typed, EvoSuite also struggles with initialising objects to the correct states and, in many cases, even to the correct type. Usage of 'instance of' and 'Object' type is common in Java.

The 'Unexercised Function' spike is explained by Pynguin inflating the impact. Conversely, Java inflates the 'Triggering Exceptions' and 'Path Unreachable via Public API' categories as described in the previous paragraph. The other larger categories are quite equal. So, in general, we can see that the main differences lie in the 'Object Creation Challenges', 'Triggering Exceptions', 'Path Unreachable via Public API', and 'Unexercised Functions'. The only category seemingly related to the typing system in the language is 'Unexercised Functions'. A plausible hypothesis is that EvoSuite's awareness of types allows it to be more efficient in creating the correct input when using functions as a target.

While some differences have been identified, the analysis reveals that there are also similarities between the two types of languages. It is, however, unclear whether the similar categories arise due to some common issues in the tools themselves or if there is a hidden correlation or relationship between certain categories. For example, does a failure in 'Object Creation' frequently co-occur with 'Sequence Generation Challenges'? To uncover these relation-

ships, the next section will analyse the correlation between failure categories to identify underlying 'failure motifs'.

5.4 Co-occurance Patterns

To answer the question of whether failures are isolated or part of a larger pattern, this section examines the co-occurrence of failures. The primary goal is to identify 'failure motifs' - groups of distinct failure categories that frequently appear together, or if any exist at all. By analysing a correlation heatmap of failure categories, the systemic relationships between classification categories can be identified.

To categorise the statistical relationship, the Pearson's Φ Coefficient was used. For every individual file, "baskets" were counted, which represent a set of coverage gaps that occurred in that file. For every pair of gap types, we measure how often they appear together in the same file using Pearson's Φ (a –1 to +1 correlation for yes/no data). Statistical significance: with 194 analysed files, any —Φ above 0.14 passes the conventional p ¡ 0.05 test. Practical importance: following Cohen's widely-used guidelines [9], we call a link meaningful when —Φ— is at least 0.30. So, we will consider a 'failure motif' a meaningful link.

The results of the correlation analysis reveal that strong failure motifs are present within each tool. As illustrated in the correlation matrices for Syntest (Figure 5.7) and Pynguin (Figure 5.8), each dynamic tool individually exhibits strong correlations between failure modes, similar in complexity to that of the Java tool (Table 5.1). However, a critical insight emerges when the data from the dynamic tools is aggregated. As shown in Table 5.2, the combined dynamic dataset shows only a single remaining meaningful correlation.

This result shows that failure motifs are not only paradigm-specific but are highly tool-specific. The individual failure mode correlations of Syntest and Pynguin are distinct, so that they effectively cancel each other out when their data is combined. While a detailed comparative analysis of each tool's specific motifs is beyond the scope of this thesis, this finding has a critical implication: aggregating results from different tools, even those within the same paradigm, can hide important, tool-specific weaknesses. It indicates that understanding and improving SBST requires a focus on the behaviour of individual tools, not only broad categories.

5.5 Parameter Count

In the previous section, it was concluded that each testing tool has a unique failure 'signature' visible in distinct failure motif patterns. To further analyse the cause of these differences in signatures, the analysis can also be expanded

Table 5.1: Meaningful gap pairs (—φ—$\geq$0.30) – Java

Gap A	Gap B	φ
Insufficient Fitness Incentive	Specific Value Generation Challenge	0.552
Specific Value Generation Challange	Specific Value Generation Challange	0.484
Sequence Generation Challenge	Specific Value Generation Challange	0.418
Interprocedural transformation challenge	Sequence Generation Challenge	0.415
Interprocedural transformation challenge	String Constraint Satisfaction Challenge	0.392
Specific Value Generation Challange	String Constraint Satisfaction Challenge	0.391
Insufficient Fitness Incentive	Interprocedural transformation challenge	0.383
Interprocedural transformation challenge	Practically Unreachable Code	0.383
Path Unreachable via Public API	Specific Value Generation Challange	0.367
Insufficient Fitness Incentive	Unexercised Function	0.36
Path Unreachable via Public API	Sequence Generation Challenge	0.335
Interprocedural transformation challenge	Specific Value Generation Challange	0.323
Interprocedural transformation challenge	Triggering Exceptions	0.322
Insufficient Fitness Incentive	Object Creation Challange	0.313
Specific Value Generation Challange	Triggering Exceptions	0.309

Table 5.2: Meaningful gap pairs (—φ—$\geq$0.30) – Dynamic tools (Pynguin + Syntest)

Gap A	Gap B	φ
Interprocedural transformation challenge	Specific Value Generation Challange	0.489

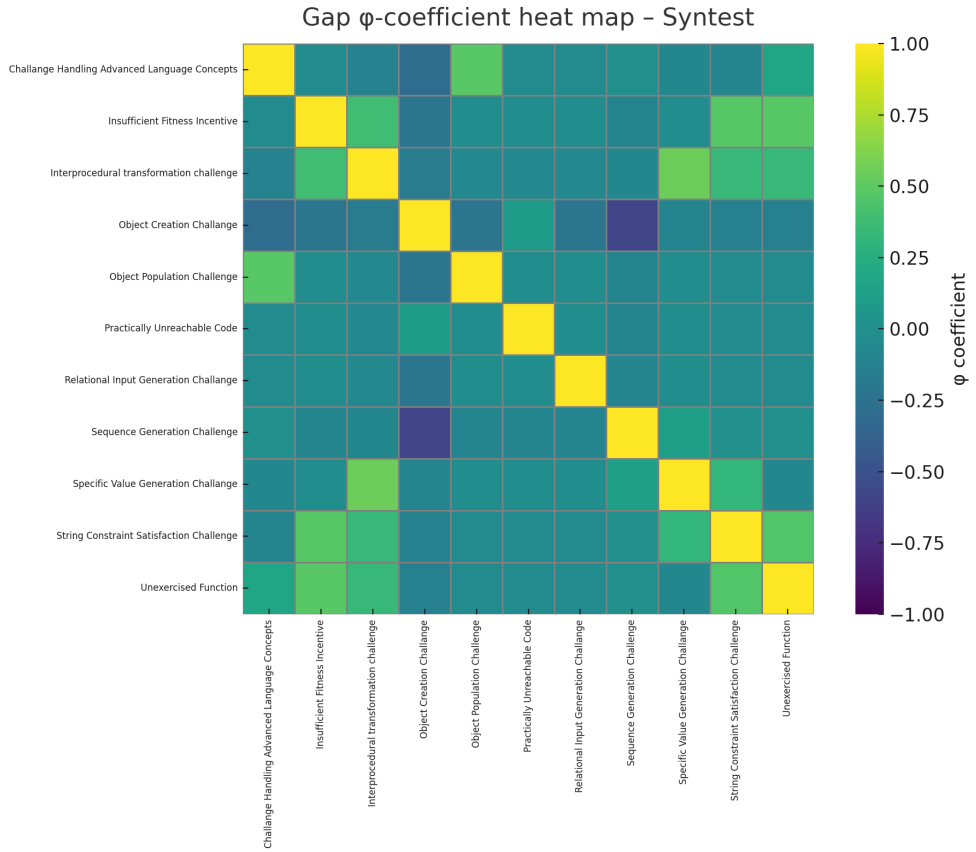


Figure 5.7: Syntest Correlation Matrix

Table 5.3: Meaningful gap pairs (—φ—$\geq$0.30) – All tools combined

Gap A	Gap B	φ
Interprocedural transformation challenge	Specific Value Generation Challange	0.362
Insufficient Fitness Incentive	Specific Value Generation Challange	0.347
Path Unreachable via Public API	Specific Value Generation Challange	0.309

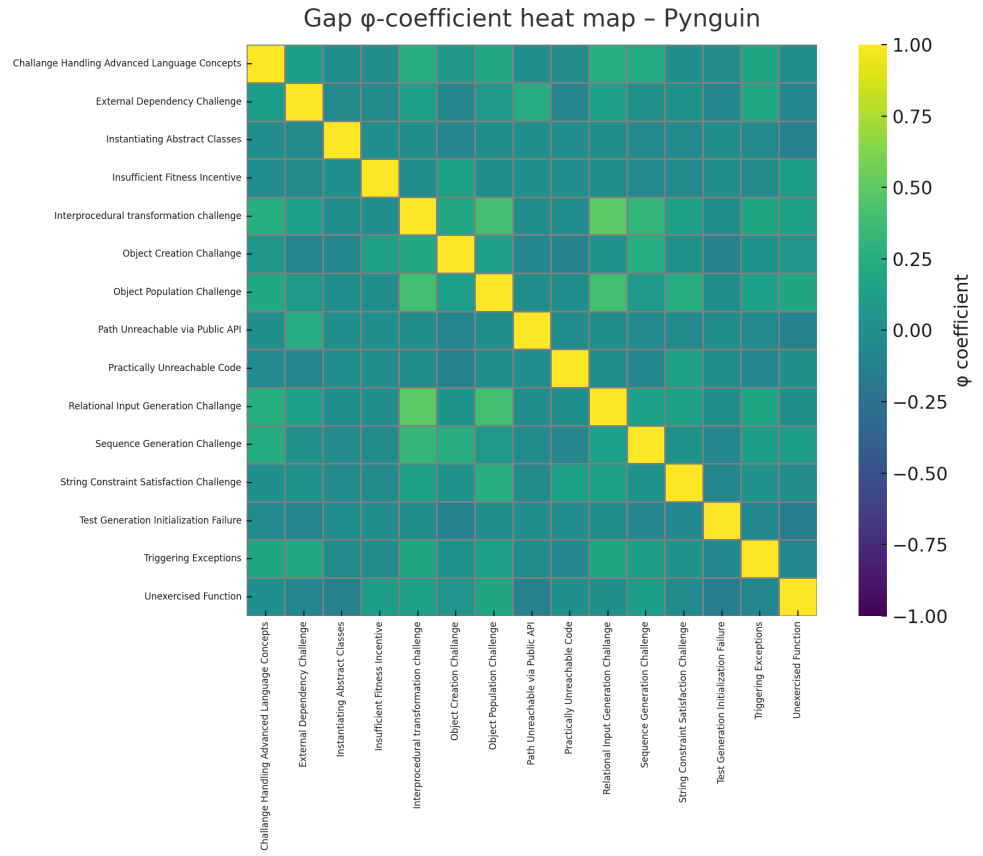


Figure 5.8: Pynguin Correlation Matrix

to include the analysis of function signatures, focusing on the parameter count. Investigating how each tool's performance relates to the number of parameters can show whether function signatures are a key component of these distinct tool-specific failure profiles.

Analysing the distribution of failures by parameter count (Figure 5.9) provides the first layer of evidence for the tools' failure signatures. The profile for Python's Pynguin is the most distinct, since it has the largest share of failures on functions with four or more parameters. In contrast, EvoSuite (Java) struggles significantly more with parameterless functions (22% of its incidents) than either of the dynamic tools. SynTest (JavaScript) has another distinct profile. Less than 10% of the incidents come from functions with zero parameters, yet 59% come from functions with a single parameter. These clear differences in where failures most frequently occur are a core component of each tool's specific operational profile.

While understanding the parameter distribution involved in incidents is important, examining severity can also provide valuable insights. In Figure 5.10, the

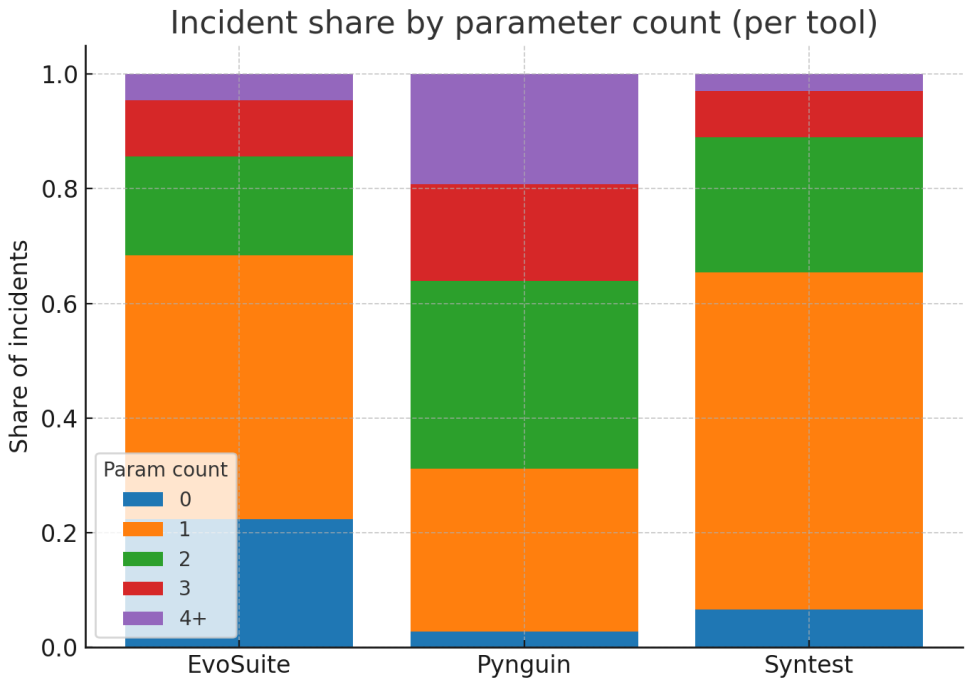


Figure 5.9: Incident share by parameter count per tool

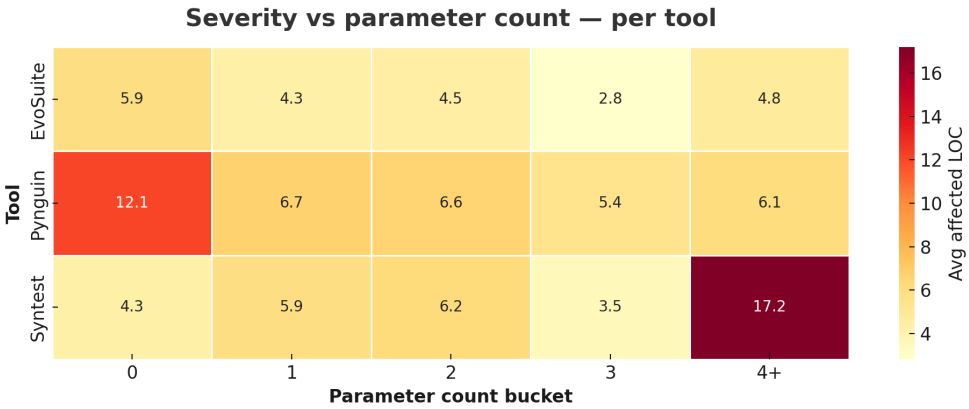


Figure 5.10: Severity vs parameter count

severity is displayed per tool per parameter. In the case of Pynguin, in the case of zero parameters, there are 12.1 LOC affected on average. This is likely connected to the large amount of 'Unexercised Functions' in this language and the low share of functions of zero parameters in the total incident distribution. On average, EvoSuite has the lowest value across all parameters. This is expected considering it is a statically typed language and has less difficulty generating input for test case generation. Syntest has the highest severity by far for functions with four parameters or more. Similarly to how Pynguin has a very small share of zero-parameter functions, Syntest has a very low share of functions with four parameters or more. The most likely explanation is that these small categories contain outliers. An interesting detail is that all tools have the lowest severity for three parameters; however, it is not clear why.

5.6 Solutions and Research Directions

The previous analysis provides a comprehensive, data-driven analysis. By quantifying coverage gaps, ranking them by impact, and identifying distinct, tool-specific failure signatures, a clear set of high-impact challenges has now been established. This section explores and outlines solutions and research directions aimed at mitigating the most critical identified coverage gaps. By analysing existing research for tools like EvoSuite, Pynguin, and Syntest, the proposed strategies are categorised into two types: practical engineering solutions, which often involve adapting successful features from one tool to another, and forward-looking research directions for problems requiring more novel approaches.

5.6.1 External Dependency Challenge

The External Dependency Challenge revolves around the interaction of the tested function with elements outside its control. In the analysed data, the most common types of interactions were File System Operations, Network Communication, OS Interactions, usage of External Libraries, and GUI interactions. Although the challenge manifested with similar frequency in the impact analysis chart, when comparing Syntest and Pynguin with EvoSuite, EvoSuite handles most cases well, except GUI-related dependencies due to its internal virtualisation mechanism. SynTest and Pynguin lack such a system, leading to an opportunity to extend the tools' capabilities by adapting the same principles implemented in EvoSuite. Moreover, there are promising research directions that could also be used to extend the capability of all studied SBST tools even further.

5.6.1.1 Engineering Solutions

Currently, the internal virtualisation mechanism of EvoSuite is quite robust. It supports Console Inputs, a Virtual File System, and a Virtual Network[5]. EvoSuite uses all these virtual solutions within a security manager, which protects the system on which the tests are executed from potential side effects [4]. Considering that neither SynTest nor Pynguin implement any kind of system of this type and since the principles used by EvoSuite are not inherent to statically typed languages, similar mechanisms could be developed in dynamically-typed languages.

EvoSuite approaches the emulation of console input by replacing System.in with a custom stream, and if read attempts are detected, the evolutionary algorithm involved in generating the test cases injects string inputs using a helper utility function to simulate user interaction and guide the test suite towards higher coverage.[4]

To manage File System dependencies, EvoSuite's approach consists of sending the standard I/O file operations to a controlled, in-memory virtual file system where all file operations are redirected. This allows the search algorithm to look up, create and modify virtual file content as required, similarly to other inputs, to increase the coverage of the test file. [4]

Although this was not a widespread problem in the analysed data, mocking networking calls can also be a valuable addition to Pynguin and SynTest. To manage network interactions in these environments, the idea is to virtualise network dependencies by replacing the standard library calls with mocks that interact in a simulated network environment. The search algorithm can change the virtual network's state to guide test execution and focus on network-dependent code. [5]

While several possible solutions to some of the challenges encountered in the analysed data have been proposed here, the more granular implementation details can be found in the referenced research papers. While this section highlights only a few solutions adaptable to SynTest and Pynguin, other capabilities can be added to these tools according to the referenced literature [4]. While less relevant to the analysed data, they offer potential directions for future improvements.

5.6.2 String Constraint Satisfaction Challenge

The String Constraint Satisfaction Challenge relates to the inherent difficulty of generating appropriate string inputs for triggering branches, especially when inputs must satisfy chained string method invocations or conform to regular expressions. This problem is common in all studied SBST tools, due to shared techniques for string manipulation and fitness guidance. Typically, SBST tools

implement a constant pool and edit distance for equality operators to guide the search. Given the shared challenges, solutions for this particular challenge are likely portable across all three tools. While explicit documentation is lacking regarding how strings are handled in each specific tool, during the data analysis stage, it was clear that they encountered similar challenges across all tools in dealing with string-related methods and regular expression matching.

5.6.2.1 Engineering Solutions

One of the more interesting potential solutions treats complex string API combinations as regular expressions and then generates inputs based on that regular expression [30]. The idea is first to create a mapping from the string-related methods to regular expressions and then use the inference algorithm described in the paper to combine the string method calls into a larger regular expression. Generating input for this expression would help in exploring certain branches. Although it was not integrated with EvoSuite, it was used in conjunction with it, yielding promising results, including a 17

Therefore, implementing dedicated fitness functions for common string operators within each target language provides a more straightforward, SBST-native approach to improving search guidance. While the string method collection differs in each language, many operations are shared across multiple languages. For some of them, such fitness functions have been identified and tested in some versions of EvoSuite [3]. Although designing effective fitness functions for every complex string operation is challenging, implementing them for a core set of frequently used and researched operators (e.g. startsWith, contains, endsWith) could significantly enhance tool performance.

5.6.3 Sequence Generation Challenge

The Sequence Generation Challenge involves generating the appropriate function calls related to the module under test in a way that ensures a correct state is reached, thereby triggering a conditional statement within a branch. Understanding how to create this call ordering is a notoriously difficult problem because a good contextual understanding of the class/module and the project in which the method is implemented is required. Capturing this relationship between different methods is far from trivial; often, the logic and requirements to trigger a certain branch are convoluted.

5.6.3.1 Research Directions

One interesting implementation of an LLM-based approach is called TELPA. TELPA is a Pynguin extension that uses LLMs in combination with program

analysis to generate test cases when Pynguin fails. It is promising, showing an average increase of 33.39% in terms of coverage. The tool extracts sequences of method calls that lead to the invocation of the target method under test. These sequences, representing valid construction and setup paths, are then provided to the LLM to guide it in generating the necessary prerequisite steps for a test. Although it seems that the solution should have a positive impact on the sequence generation challenge, it would be useful to understand its impact on this specific type of problem in isolation.

Another highly relevant piece of research is TestFul. TestFul is an evolutionary testing framework designed for object-oriented systems, which automatically generates sequences of method calls to reach certain object states. Its evolutionary algorithm evaluates how well each generated sequence performs by measuring the code coverage that results from it. It applies standard evolutionary operators on sequences. While SBST tools primarily focus on optimising the fitness function in relation to the input that is provided to the function, using evolutionary algorithms for generating sequences of method calls is a novel, promising approach for SBST tools. Some interesting research directions would be understanding how to integrate such an algorithm in existing frameworks efficiently, but also finding ways to detect when to trigger it. This problem occurs in a minority of cases, so a more intelligent approach to the usage of such an evolutionary algorithm would have an important effect on the overall performance of the algorithm.

5.6.4 LLMs as a solution

With the rise in popularity of LLMs, it is natural to wonder whether they are a magical solution to all testing problems. However, it is essential to consider their limitations. One such limitation is the context window, which directly relates to the cost of an LLM request. Different problems have different solutions and require different amounts of context when considered from an LLM-centric perspective.

The main challenge remains understanding which coverage gaps are more expensive and require more context, and which LLM would require minimal context. Based on the information gained during the taxonomy-building process, it becomes apparent that some categories would be greatly reduced even without extensive context. Understanding this dimension would allow us to analyse better and reason about the potential utility of the tool as a solution in this direction. During the analysis process, some interesting findings emerged regarding the most popular categories.

First of all, significant coverage would likely be gained by simply providing input using an LLM for the unexercised functions, since they are not executed at all. However, the extent to which they can be covered is unpredictable. This

solution is not inherent to LLMs and could likely also be solved by SBST tools, provided a larger time budget is allocated.

The Object Generation Challenge could also have a very variable context window. In cases where the object structure is clear and easily understandable from the function at hand, LLMs should be able to generate an object with the correct structure. However, if the object is passed around many functions, the context window increases accordingly. There are cases where complicated logical relationships also require a lot of context.

One case where LLMs could shine is for the String Constraint Satisfaction Challenge. In most cases, this challenge relates to cases where a string needs to correspond to a pattern, blocking an if condition, simply. The pattern itself is often obvious from the context of the condition. This is indeed similar to the Triggering Exceptions coverage gap, where the name of the exception is likely to provide enough context to trigger it.

In the case of the Sequence Generation Challenge, it is unlikely that the context window will be small in most cases. Understanding how to reach a complex state through method calls requires a thorough understanding of the entire codebase, the way different methods interact with one another, and the global states they affect.

Some other faults, like External Dependency Challenge, cannot be fixed by an LLM if the existing infrastructure for mocking such dependencies is lacking. Even though the LLM tool might be able to generate the correct commands to trigger the paths, the test would be flaky, inconsistent and likely dangerous to run, as it could have unexpected effects on external environments.

Chapter 6

Conclusions and Future Work

6.1 Contributions

This thesis makes several contributions to the field of Search-Based Software Testing (SBST). The primary contribution is the development of a comprehensive, empirically grounded taxonomy of coverage failures in SBST tools. This taxonomy provides a common vocabulary and a structured framework for analysing and understanding why these tools fail to achieve complete code coverage.

Building on this taxonomy, a second contribution is the data-driven analysis of these failures. By quantifying the impact of each failure category and performing a systematic comparative analysis across different tools and language paradigms (statically typed vs. dynamically typed), this work provides new, objective insights into the most significant challenges in automated test generation.

Finally, for the most impactful failure categories identified, this thesis documents potential engineering solutions and promising future research directions, providing a roadmap for improving the effectiveness of SBST tools.

6.2 Conclusions

The research conducted in this thesis leads to several key conclusions. The primary conclusion is that coverage gaps in SBST tools are not random but can be systematically categorised into a hierarchical taxonomy with well-defined failure modes, such as Code Testability Issues, Stateful Interaction Challenges, and Input Related Failures.

The impact analysis performed using the proposed formula reveals that the most significant challenges are the `Object Creation Challenge` and the

presence of `Unexercised Functions`. Other highly impactful categories include the `Sequence Generation Challenge`, `String Constraint Satisfaction Challenge`, and the `External Dependency Challenge`.

The comparative analysis concludes that while many failure types are common across language paradigms, their impact differs. Statically-typed languages like Java suffer more from unreachable paths due to patterns like private constructors, whereas dynamically-typed languages show a higher frequency of `Test Generation Initialisation Failures`. Furthermore, the comparison between Python and JavaScript highlights that the specific use cases of a language influence failure modes; for instance, Python projects in the benchmark were more affected by the `External Dependency Challenge`.

6.3 Threats to Validity

6.3.1 Internal

A risk related to the internal validity of the analysis is associated with the methodology itself. Due to the limited number of replication packages and the demand for data to construct a robust taxonomy, some compromises had to be made. More precisely, some of the coverage data used in tests was run only once, while in other cases, it was run multiple times, and the median was chosen as a compromise. In some cases, different versions of the same tool were used. In some cases, data from different replication packages was used for the same tool. However, this was a necessary compromise. The alternative would have meant significantly less data and created a much higher risk to the internal validity of the analysis. Moreover, it would have risked making the results less generalizable as they would be biased towards a single predefined set of examples.

6.3.2 External

One risk to consider is the size of the analysed data. While 42 projects were examined and over 700 coverage gaps were identified, it is still possible that some coverage gaps were over- or underrepresented. Replication packages and their availability also induce the bias. In this case, a solution would be to increase the analysed data and to include more people in the classification process. However, this study aims to mitigate this issue by focusing on multiple languages, projects, and tools, utilising replication packages from multiple studies to increase the generalizability of the results.

6.3.3 Construct

Building a taxonomy is based on a subjective vision and carries implicit bias. One threat is that the rules defined do not align with the initial vision intended for the categories. The classification of the categories carries an inherent risk. To mitigate both of these risks, a decision tree was created, and tiebreaker rules were defined for cases where two categories might overlap with one another.

Another threat is the Impact Score Formula used throughout the analysis. The choice of the formula was made as a proxy for the general applicability of a particular issue. To ensure that the metric represented this intent, it was shown in the methodology chapter how it solves the problematic biases. Of course, another risk is that a definition of impact can be quite subjective, leaving room for disagreement.

6.3.4 Conclusion

One risk could be the correctness of the classification on which the conclusions and the analysis rely. While ideally, more people would be involved in the development process, the defined tiebreaker rules and the flowchart attempt to mitigate the risk and create a reproducible taxonomy.

Moreover, the analysis relies on coverage generated by non-deterministic tools. Tests generated especially for large files might skew the data due to ”lucky” or ”unlucky” runs. To mitigate the risk, in replication packages where multiple runs were available, the median coverage was selected to represent an expected run.

6.4 Future work

The findings and contributions of this thesis open up several avenues for future work. The taxonomy could be expanded and compared against other tools, not necessarily SBST-based. Given the rise in popularity and practical application of LLM-based (hybrid) solutions, it would be interesting to also include such solutions in the comparison. More projects and more extensive benchmarks can be identified to achieve a more comprehensive analysis.

Another significant advancement would be the automation of the classification process. While the current application of the taxonomy relies on manual analysis, future research could focus on developing a tool that utilises machine learning or static analysis techniques to automatically classify coverage gaps. This would increase the practical utility of the taxonomy. It would allow for the development of tools that track the improvement of testing tools over time, and it could also be used to develop a general testing algorithm. Based on the identified coverage gap, the algorithm could decide which specialized solution

for the coverage gap to choose to increase coverage, making automated testing more robust as a whole and increasing the capability of interaction between different algorithms and solutions.

Bibliography

[1] Code Coverage Best Practices, . URL https://testing.googleblog.com/2020/08/code-coverage-best-practices.html.

[2] Releases · se2p/pynguin, . URL https://github.com/se2p/pynguin/releases.

[3] Andrea Arcuri and Juan P. Galeotti. Enhancing Search-based Testing with Testability Transformations for Existing APIs. *ACM Transactions on Software Engineering and Methodology*, 31(1):1–34, January 2022. ISSN 1049-331X, 1557-7392. doi: 10.1145/3477271. URL https://dl.acm.org/doi/10.1145/3477271.

[4] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 79–90, Vasteras Sweden, September 2014. ACM. ISBN 9781450330138. doi: 10.1145/2642937.2642986. URL https://dl.acm.org/doi/10.1145/2642937.2642986.

[5] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Generating TCP/UDP network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 155–165, Bergamo Italy, August 2015. ACM. ISBN 9781450336758. doi: 10.1145/2786805.2786828. URL https://dl.acm.org/doi/10.1145/2786805.2786828.

[6] Atlassian. What is Code Coverage? URL https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018. doi: 10.1145/3182657.

[8] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785.

[9] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 2nd edition, 1988.

[10] Federico Formica, Tony Fan, and Claudio Menghi. Search-Based Software Testing Driven by Automatically Generated and Manually Defined Fitness Functions. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–37, February 2024. ISSN 1049-331X, 1557-7392. doi: 10.1145/3624745. URL https://dl.acm.org/doi/10.1145/3624745.

[11] Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, Szeged Hungary, September 2011. ACM. ISBN 9781450304436. doi: 10.1145/2025113.2025179. URL https://dl.acm.org/doi/10.1145/2025113.2025179.

[12] Gordon Fraser and Andrea Arcuri. EvoSuite: On the challenges of test case generation in the real world. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 362–369, 2013. doi: 10.1109/ICST.2013.51.

[13] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. *Empirical Software Engineering*, 20(3):611–639, 2015. doi: 10.1007/s10664-013-9288-2.

[14] Gordon Fraser, Annibale Panichella, and Andrea Arcuri. A retrospective on whole test suite generation: On the role of sbst in the age of llms. *IEEE Transactions on Software Engineering*, 51(3):1648–1669, 2025. doi: 10.1109/TSE.2025.3539458. This corresponds to sources [13], [21], etc., in your document.

[15] Martin Gruber, Muhammad Firhard Roslan, Owain Parry, Fabian Scharnböck, Phil McMinn, and Gordon Fraser. Do Automatic Test Generation Tools Generate Flaky Tests? In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, Lisbon Portugal, February 2024. ACM. ISBN 9798400702174. doi: 10.1145/3597503.3608138. URL https://dl.acm.org/doi/10.1145/3597503.3608138.

[16] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12, 2015.

[17] Kiran Lakhotia, Myra Adham, and Mark Harman. Austin: An open source tool for search based testing of c programs. In *2010 Third International Conference on Software Testing, Verification and Validation Workshops*, pages 433–436, 2010. doi: 10.1109/ICSTW.2010.23. Corresponds to the work on AUSTIN mentioned for C/C++ challenges, related to source [291].

[18] Stephan Lukasczyk and Gordon Fraser. Pynguin: automated unit test generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, Pittsburgh Pennsylvania, May 2022. ACM. ISBN 9781450392235. doi: 10.1145/3510454.3516829. URL https://dl.acm.org/doi/10.1145/3510454.3516829.

[19] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of automated unit test generation for Python. *Empirical Software Engineering*, 28(2):36, March 2023. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-022-10248-w. URL https://link.springer.com/10.1007/s10664-022-10248-w.

[20] Valentin J. M. Manès, Hyungon Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019. doi: 10.1109/TSE.2019.2946300.

[21] Chao Mao, Ling Yu, and Zude Chen. A search-based test data generation method for concurrent java programs. *International Journal of Computational Intelligence Systems*, 9(1):15–31, 2016. doi: 10.1080/18756891.2016.1142517. Corresponds to source [403] in your document.

[22] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004. doi: 10.1002/stvr.294.

[23] Atif M. Memon, Bao Nguyen, and Adithya Masri. Guitar: an innovative tool for automated testing of gui-driven software. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '08 Companion, pages 837–838. ACM, 2008. doi: 10.1145/1449806.1449866. Corresponds to source [356] in your document.

[24] Mitchell Olsthoorn, Arie Van Deursen, and Annibale Panichella. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1224–1228, Virtual Event Australia, December 2020. ACM. ISBN 9781450367684. doi: 10.1145/3324884.3418930. URL https://dl.acm.org/doi/10.1145/3324884.3418930.

[25] Mitchell Olsthoorn, Dimitri Stallenberg, and Annibale Panichella. Syntest-JavaScript: Automated Unit-Level Test Case Generation for JavaScript. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, pages 21–24, Lisbon Portugal, April 2024. ACM. ISBN 9798400705625. doi: 10.1145/3643659.3643928. URL https://dl.acm.org/doi/10.1145/3643659.3643928.

[26] Nicolae Rusnac. Replication Package for: "A Comprehensive Taxonomy of SBST-Tool-Limitations", June 2025. URL https://doi.org/10.5281/zenodo.15757570.

[27] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Guess what: Test case generation for javascript with unsupervised probabilistic type inference. In *Search-Based Software Engineering - 14th International Symposium, SSBSE 2022, Online*, volume 13749 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2022. doi: 10.1007/978-3-031-27429-7_5.

[28] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method. *Information and Software Technology*, 85:43–59, May 2017. ISSN 09505849. doi: 10.1016/j.infsof.2017.01.006. URL https://linkinghub.elsevier.com/retrieve/pii/S0950584917300472.

[29] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007. ISBN 978-0123725576.

[30] Miaomiao Wang, Baoquan Cui, Jiwei Yan, Jun Yan, and Jian Zhang. String Test Data Generation for Java Programs. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 251–262, Charlotte, NC, USA, October 2022. IEEE. ISBN 9781665451321. doi: 10.1109/ISSRE55969.2022.00033. URL https://ieeexplore.ieee.org/document/9978963/.

[31] Zhinia Wang, Jie M. Chen, and Ahmed E. Hassan. Software testing with large language models: Survey, landscape, and vision. *arXiv*

preprint arXiv:2307.07221, 2023. URL https://arxiv.org/abs/2307.07221.