Test Program-Based Generative Fuzzing for Differential Testing of the Kotlin Compiler

Călin-Andrei Georgescu

Test Program-Based Generative Fuzzing for Differential Testing of the Kotlin Compiler

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Călin-Andrei Georgescu
born in Bucharest, Romania

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

Applied Program Analysis Lab
Intelligent Collaboration Tools Lab
JetBrains N.V.
Amsterdam, The Netherlands
https://www.jetbrains.com/

Test Program-Based Generative Fuzzing for Differential Testing of the Kotlin Compiler

Author: Călin-Andrei Georgescu
Student id: 4794672
Email: caling@protonmail.com

Abstract

Kotlin is a programming language best known for its interoperability with Java, as well as the measurable improvements it offers over it. Since it became Android's go-to language in 2019, the popularity and impact of Kotlin have risen greatly. Amidst this surge in popularity, the Kotlin developer team is working on a new version of the compiler that introduces sweeping changes to the ecosystem.

Traditional compiler testing is a manual and laborious task that requires extensive developer effort and expertise. In an attempt to mitigate this, researchers have invested great resources in developing and perfecting automated compiler testing tools over the last decades. These approaches generate new pieces of code to test the behavior of compilers, which is assessed through differential testing. However, the usage of heuristics as guidance for the generative process is not well understood, and no approach that generates Kotlin code from scratch currently exists.

In this thesis, we propose a novel method of enriching standard grammar specifications with language-targeted semantic context that is integrated in the sampling process. We structure generated code hierarchically and use it as the base of an evolutionary computation framework. Within this framework, we introduce two classes of algorithms that are novel to the field of compiler fuzzing, based on syntactic diversity and semantic proximity, respectively.

We carry out an empirical analysis spanning 200K generated Kotlin files, which we analyzed through different Kotlin compiler versions. Our results uncovered five previously unreported categories of bugs, which we reported to the Kotlin compiler developer team. The developers verified and replicated our instances on the current release of the Kotlin compiler, and have assigned target release dates for fixes within the current major version of the compiler. The study also provides new insight into the effects of heuristic-specific hyperparameters such as expression simplicity, dissimilarity measurements, and target selection.

Thesis Committee:

Committee Chair:	Dr. A. Panichella, Faculty EEMCS, TU Delft
Committee Member:	Dr. S. E. Verwer, Faculty EEMCS, TU Delft
Company Supervisor:	Ir. M. Akhin, Applied Program Analysis Lab, JetBrains
Company Supervisor:	Dr. P. Derakhshanfar, Intelligent Collaboration Tools Lab, JetBrains
Daily Co-Supervisor:	Ir. M. Olsthoorn, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

Compilers are ubiquitous and pivotal components at the core of innumerable software and programming language ecosystems. They translate the high-level, human-readable source code of computer programs into target-specific instructions that machines can parse and execute. As a consequence, the ramifications that arise from flawed compiler implementations can be both far-reaching and severe. The outstanding complexity and feature richness of modern compilers has created several formidable challenges, the scope of which is perhaps best exemplified by Hoare [41] declaring the task of building a provably correct compiler to be a *Grand Challenge* in computing research. In recent decades, researchers and practitioners have invested tremendous resources into compiler research, attempting to improve their performance, quality, and reliability. Such efforts are especially beneficial for newer software systems, that require robust compilation pipelines to establish themselves as worthy alternatives to mature standards.

1.1 Compiler Testing

Because of their cornerstone role in many critical applications, compilers are often subject to thorough quality assurance processes. In practice, compiler developers often rely on testing as a means of assessing and improving the correctness of compilers. Empirical evidence supports the benefits of such approaches, with Sun et al. [82] and Holler et al. [43] highlighting the existence of bugs in widely used C compilers and JavaScript engines, respectively. These findings stand out because of the maturity of the software ecosystems that pivot around these technologies, showcasing the necessity of effective testing methods.

One approach to testing compilers is to assemble manually-written test suites. This strategy has several advantages, including the possibility to individually target specific components of the compiler and leveraging developers' capacity to reason about the tests' semantics and expected outputs. However, the labor-intensive nature of this process and the imposing complexity of compiler code bases have led to automated test case generation methods superseding their manual counterparts [15, 93]. Though compelling and time-effective, automated test case generation strategies concurrently give rise to an arduous set of challenges. The infinite number of correct possible programs, the syntactic dependen-

cies between code snippets, and the many semantic nuances programming languages all constitute obstacles that effective test generation tools must address.

Fuzzing, or random testing, lends itself naturally to this task thanks to its ability to generate code snippets that trigger diverse behavior within the compiler. Because of this, researchers have proposed a myriad of language-specific approaches, fixated around two overarching paradigms. Several studies focus on algorithms that generate standalone, independent code [89, 43, 88, 39]. Such approaches have the ability to produce entirely novel programs, which in turn may contain seldom encountered constructs and relations. By contrast, mutation-centric approaches [49, 50, 81, 79] rely on an input (or seed) program, that serves as a basis for subsequent transformations. While the former approach may be able to explore more of the space of feasible input programs, the latter effectively leverages large corpora that become available as programming language ecosystems mature.

1.2 The Kotlin Programming Language

Kotlin [1] is a relatively new programming language whose development initiative is led by JetBrains. Initially designed as a Java alternative, the Kotlin ecosystem has steadily grown and currently envelops a broad range of environments, including server-side development, web frontend interfaces, and mobile applications. Kotlin showcases compelling features, including expressiveness, null safety, and Java interoperability, which played a key role in Google embracing a "Kotlin-first" approach in its Android mobile operating system [21].

The number of research initiatives investigating Kotlin's impact in the software development process has risen in tandem with its adoption. Several studies have analyzed both Kotlin's influence on the quality of code bases, as well as developers' perception of Kotlin in comparison to their previous experience. Góis Mateus and Martinez [33] and Flauzino et al. [26] independently found that code bases which at least partially employ Kotlin tend to exhibit fewer code smells than their Java counterparts. Ardito et al. [8] suggest that transitioning projects from Java to Kotlin can meaningfully increase code conciseness. Chauhan et al. [14] provide empirical evidence indicating that Kotlin's coroutine implementation vastly outperforms Java's thread-centric concurrency framework in terms of speed. Oliveira et al. [67] study Android practitioners' opinions regarding Kotlin adoption and reveal that developers find Kotlin easy to adopt and understand. Their study also showcases that developers value Kotlin's null-safety guarantees, as well as its interoperability with Java.

Despite the sharp rise in Kotlin's popularity and an increasing number of systems relying on its ecosystem, the study by Stepanov et al. [79] is the only one to propose a tailored algorithm for automatically testing the Kotlin compiler. Their work heavily relies on manually written test suites to provide seeds for an enumeration-based fuzzer aimed at detecting compilation errors between different compiler implementations. Currently, no tool that is capable of generating entirely novel, semantically meaningful, and diverse test programs for the Kotlin compiler exists. Creating such a tool would not only aid the quality assurance process of the Kotlin compiler, but also provide valuable insight into the relation between language features and compiler defects.

[1] https://kotlinlang.org/

1.3 Research Aim and Scope

This thesis seeks to advance the current understanding surrounding test program-based fuzzing as a means of automated compiler testing. To narrow the scope of this goal, we identify and acknowledge several key limitations. First, this thesis only concerns itself with empirical analyses regarding the Kotlin compiler. Though the prototype implementation may well be extended to target other systems, we restrain the evaluation to Kotlin and its current compiler versions ensure feasibility.

Second, this study investigates the ability of the proposed fuzzing techniques to find bugs in the targeted compiler, and does not emphasize the practical or commercial implications of the uncovered bugs. We provide a qualitative analysis of the uncovered defects, but do not further investigate the ramifications of their existence. Moreover, research concerning other widely studied defect-centric practices such as bug localization, program minimization, and bug deduplication are outside the scope of this study. Within these constraints, we seek to achieve the following research aim:

The aim of this research is to gain insight into how effective heuristic-driven test program-based fuzzing is at uncovering bugs in the Kotlin compiler.

To address this aim, we implement a grammar-aided fuzzing tool that generates varied and valid pieces of Kotlin code. The generated code serves as input to the compiler, whose behavior is in turn assessed through differential testing. We divide the overarching research aim into several, more granular research questions.

A vital point to consider when designing a fuzzing algorithm is its potential to adapt to the goals of its users. Amalfitano et al. [1] show that random testing approaches eventually reach a saturation point, where novel input is exceedingly unlikely to trigger new faults. To address this, practitioners can either design new tools, adapt existing ones, or intertwine approaches. While flexible design can help mitigate this shortcoming, studying the impact of cornerstone hyperparameters and heuristic choices is crucial for understanding the scope of a fuzzer and its behavior. To this end, we formulate the following research question:

RQ1: *How do meta-heuristic guidance criteria influence the properties of test programs generated by fuzzing?*

Next, we focus the study on comparing the performance of several established and novel search formulations. The second research question aims to investigate the relative bug-uncovering potential of such heuristics within an evolutionary framework:

RQ2: *How do the proposed generative heuristics compare in terms of uncovering bugs in the Kotlin compiler?*

1.4 Contributions and Significance

This thesis makes both theoretical and practical advances, relevant to both the field of compiler testing and to the Kotlin community. The study makes the four following contributions:

1. Three novel formulations of grammar-aided test program-based compiler fuzzing, including a random sampling approach and two evolutionary algorithms revolving around diversity- and proximity-driven criteria.

2. A modular, configurable, and extendable framework for end-to-end automated fuzzing of the Kotlin compiler, implementing the three proposed approaches.

3. Empirical analyses comparing the bug finding capabilities and heuristic-driven behavior of the three novel fuzzing algorithms.

4. A set of open-source artifacts, and analytics tools for validating, verifying, and extending the current study. Artefacts include the code implementing the proposed approach [12] and the data that we base the empirical study upon [11].

From a theoretical perspective, we investigate the applicability and adaptation of evolutionary fuzzing techniques to the task of compiler testing. The significance of this knowledge is tied to the continued interest in establishing reliable quality assurance measures for compilers. On a practical level, we provide prototype implementation that supports the proposed algorithms and includes additional infrastructure surrounding the core fuzzing process. We aim for this tool to become a useful instrument for improving the reliability and quality of the Kotlin compiler. Compiler engineers can leverage such a tool as an additional step in a continuous integration pipeline, or choose to employ it when developing novel features in future versions of the language.

1.5 Overview of the Study

Chapter 2 introduces the theoretical background that constitutes the foundation of this study. Chapter 3 discusses related work and positions this thesis within the current research landscape. Chapter 4 describes the conceptual outline of the fuzzing procedure and the rationale behind its underpinning design choices. Chapter 5 delineates the implementation of our tool and highlights its practical applications. Chapter 6 outlines the design of the empirical study carried out to evaluate the performance of our tool. Chapter 7 analyzes the results of the empirical study and ties them to the main research questions. Chapter 8 discusses the main findings of this research, its limitations, and makes additional recommendations for future research. Finally, Chapter 9 concludes the thesis.

Chapter 2

Background

This chapter establishes the theoretical basis required for investigating the aim introduced in Section 1.3. This includes the individual nuances of the Kotlin compiler, the encompassing field of software testing, and the theoretical background of fuzzing. In addition, this chapter lays a contextual and historical foundation required for the analysis of related work examined in Chapter 3.

2.1 The Kotlin Compiler

The Kotlin project began in 2010, with version `1.0` officially releasing six years later [79]. Following Google's decision to support it as an official Android programming language [22], Kotlin's popularity increased drastically. The Stack Overflow annual developer survey exemplifies this exceptional growth well. The survey aggregates data from tens of thousands of active developers from varied backgrounds and clearly showcases Kotlin's sharp ascent: the percentage of developers who actively use Kotlin increased from 0.12 per cent in 2016 to 9.16 per cent in 2022 [68, 69]. Such sustained growth brought both opportunities and challenges to the expanding community, which resulted in the development of increasingly complex and feature-rich tools.

Such changes have not gone unnoticed by Kotlin developer community, who for the past seven years has introduced numerous changes to all parts of the language ecosystem to accommodate changes in the scope of the language. Inevitably, the compiler too must adapt to the direction that the ecosystem is taking. Perhaps one of the biggest changes to Kotlin as a whole is the upcoming release of version 2.0 of the language, complete with a sweeping re-write of the compiler frontend [1]. This component is responsible for parsing, analyzing, and translating source code into an intermediate representation.

The new version, code-named `K2`, of the compiler seeks to increase performance and maintain backwards compatibility, all while entirely replacing the architecture of the current compiler frontend. Naturally, any defects in the new compiler implementation could have drastic repercussions for Kotlin users, which coupled with the sheer size of the rewrite, warrants extensive testing. The remainder of this study targets the testing of the new (exper-

[1] https://blog.jetbrains.com/kotlin/2023/02/k2-kotlin-2-0/

imental) implementation of the `K2` compiler against its previous counterpart, with the goal of uncovering defects that the introduction of `K2` brings with it.

The Kotlin ecosystem supports several platforms, but large parts of it pivot around the Java Virtual Machine (JVM) system and its adjacent Java bytecode instruction set. Kotlin is one of several successful languages to leverage the JVM, together with Java, Scala, Clojure, and Groovy. One can think of the JVM as a compatibility layer between a Java bytecode input and a target language. The advantage of this system lies in the flexibility it affords language engineers. Leveraging the JVM allows language designers to focus on building a single compilation tool that targets Java bytecode. This approach delegates the task of machine code generation to the JVM and frees compiler engineers from platform- and hardware-specific idiosyncrasies. This study focuses on testing the end-to-end compilation pipeline on the JVM platform of Kotlin through the open-source `kotlinc` utility.

2.2 Software Testing

Testing is an integral component of the Software Development Lifecycle (SDLC) [44]. At its core, software testing is the process of validating the behavior of a target System Under Test (SUT). Over the years, researchers expanded vast resources into developing, and understanding the nuances of software testing, both manual and automated. To effectively navigate the numerous alternatives, we use the taxonomy established by Umar [85]. We introduce software testing in a general setting and highlight the intricacies that are most relevant to the automated testing of the Kotlin compiler.

Black-, White-, and Grey-Box Testing

A key characteristic of a testing strategy consists of the requirements it imposes on the SUT. White-Box (WB) approaches require a full understanding of the target system, which often translates to unrestricted access to its source code. The information extracted through access to the system's structure and implementation often provides valuable guidance criteria. By contrast, Black-Box (BB) approaches function without access to the underlying source code, which diminishes the amount of information they can exploit. This sacrifice, however, extends the applicability of BB approaches to a broader range of systems. Grey-Box (GB) approaches constitute a trade-off between the two extremes, and assume access to a subset system components. This requires some level of understanding of the SUT's behavior, but imposes lesser constraints than WB standards. In the scenario of compiler testing, we focus our attention on BB approaches because of their versatility.

Unit-, Integration-, and System-Level Testing

Testing methods also differ in the degree to which they exercise the SUT. Three main paradigms emerge from this distinction: Unit-Level Testing (UT), Integration-Level Testing (IT), and System-Level Test (ST). Daka and Fraser [19] demarcate UT as generally small and automatically executable tests that aim to validate the correctness of small components (units) of the SUT. Leung and White [52] establish IT at a higher-level scope, targeting the

integration of several smaller modules within a system. Finally, Umar [85] identifies ST as tests that validate the entire system's compliance against its underlying requirements. In this thesis, we concern ourselves with the end-to-end behavior of the Kotlin compiler. For this reason, we shift our attention away from the more granular levels and towards ST.

The Oracle Problem

Irrespective of access to the SUT and its level of granularity, test cases must be capable of distinguishing between correct and faulty behavior. This challenge constitutes the *test oracle problem*, and its automation is among the greatest obstacles in automating test case generation [10]. Approaches like specification inference and regression testing offer approximate oracles that can partially automate this process. First introduced in 1998 by McKeeman [60], Differential Testing (DT) offers a compelling and versatile alternative, that uses different versions of the same SUT to heuristically discover bugs. DT identifies incorrect behavior when at least one version of the SUT returns a result that differs from the others. While inexact in its nature, DT has become popular in the field of compiler testing thanks to its lightweight constraints, which increase its applicability to especially complex pieces of software, like compiler ecosystems.

2.3 Automated Test Case Generation

Traditional (manual) software testing poses several practical challenges, not least because of its imposing complexity. Candea and Godefroid [13] regard testing as the most demanding and laborious task in the SDLC. Lehman [51] established that to maintain relevance, software systems must adapt, while Zaidman et al. [90] showed that test suites often evolve in tandem with their code bases. These findings suggest that to adequately test an evolving system, practitioners not only allocate significant resources to writing test cases, but also do so throughout the software's lifetime. In an effort to alleviate the human resources that traditional compiler testing requires, researchers developed a plethora of automated test case generation methods. The remainder of this section briefly covers the most prevalent techniques in this field.

2.3.1 Fuzzing

Miller et al. [63] coined the term *fuzz* in 1979 referring to a program capable of generating a stream of random characters. This program served as a means of assessing the reliability of several tools that were part of the Unix Operating System (OS). At its core, fuzzing is perhaps the simplest possible implementation of input generation: the *fuzzer* simply samples data driven by a pseudo-random number generator and executes the target program with the generated input. This original formulation of fuzzing bears a close resemblance to well established test case generation algorithms such as Random Testing (RT) [24] and its descendant, Adaptive Random Testing (ART) [16]. Nonetheless, fuzzing is a straightforward and deceptively effective tool for defect discovery, which led it to become one of the most prevalent techniques in the field of test case generation.

Since its inception, the meaning of fuzzing has adapted to better accommodate the extensive domain of applications that benefited from its application. Such applications include security testing [66, 80], data structure signature generation [23], database system management testing [94], and REST API testing [9, 32]. To encapsulate the various complexities that fuzzing has grown to exhibit, Manès et al. [59] define fuzzing as the execution of the SUT using input sampled from an input space protruding that of the SUT's. Saavedra et al. [75] adopt a similar definition and additionally establish a taxonomy, that distinguishes between three fundamental fuzzing paradigms:

- **Mutation-based fuzzers** rely on predefined input (seeds), which they manipulate in heuristic ways to better cover that SUT's input space. The performance of these approaches is often heavily correlated with the quality of the corpus of seed inputs as the main source of exploiting program knowledge.
- **Generation-based fuzzers** sample the input space guided by an a-priori configuration or specification of the SUT. Such approaches are generally more sophisticated than their mutation-based counterparts, which they tend to outperform [75]. However, these benefits come at the cost of complexity and adherence to the limitations provided through the configuration.
- **Evolutionary fuzzers** employ evolutionary computation techniques, including fitness evaluation, mutation, crossover, and selection, to iteratively improve the quality of a set of solutions. These algorithms benefit from the rich literature surrounding evolutionary computation and may be better suited at exploring specific volumes of the search space. Evolutionary fuzzers impose an additional requirement on the SUT, namely that it must be capable of meaningfully evaluating generated input.

In this study, we propose approaches that intertwine generation-based and evolutionary fuzzers. We use techniques resembling the former to establish the baseline tools for code generation, and further modify novel code through evolutionary techniques. The latter allows us to enhance the core generative component with high-level heuristics that past work only scarcely covered.

2.3.2 Search-Based Software Testing

Software engineering entails numerous problems that present nuanced, multivariate, and often conflicting objectives. Metaheuristic search algorithms lend themselves naturally as frameworks for effectively exploring the space of possible solutions with the aim of finding (near-)optimal candidates [37]. The study and application of metaheuristics to software-centric tasks has brought forth many state-of-the-art approaches, and is jointly known as Search-Based Software Engineering (SBSE). Approaches fixating on the testing phase of the SDLC belong to the Search-Based Software Testing (SBST) branch of SBSE.

Search-based methods generally involve more sophisticated mechanisms than fuzzing. Irrespective of the problem, a notion of *fitness* allows the comparison of any two (valid) solutions on the basis of at least one *objective*. In the realm of SBST, solutions may consist

of test cases, and the corresponding fitness of a solution may be determined on the structural coverage a solution (test case) achieves, with the underlying objective of uncovering bugs. Search algorithms often change solutions by means of *mutation*. Mutation operators introduce small, possibly stochastic perturbations to alter the representation of a candidate, implicitly equipping the search space with a *topology*. Over the years, researchers developed successful variations of this high-level framework, which in many cases are able to satisfactorily solve combinatorial problems in reasonable time [62]. The remainder of this subsection coarsely examines some of the most widespread metaheuristic search approaches and grounds them through an SBST lens.

Local Search Metaheuristics

Local Search (LS) is among the most straight-forward metaheuristic frameworks. Hill Climbing (HC) is a simple implementation of LS, that greedily exploits the neighborhood structure of a solution. It consists of an iterated procedure that repeatedly attempts to improve a solution by applying mutations [76]. If the mutated candidate exceeds the fitness of the previous solution, the former replaces the latter, concluding the iteration. This basic strategy has been an effective tool for solving combinatorial problems for over half a century [55]. Both greedy and stochastic variations of LS exist, providing compromises between the quality of the final solution and the time to convergence. Arcuri [4] shows from a theoretical perspective that LS asymptotically outperforms RT, but falls short of more sophisticated global search techniques. Harman and McMinn [38] reinforce this hierarchy.

Miller and Spooner [64] have demonstrated the first application of HC in automated test data generation in 1976. Since then, researchers developed several variations of the LS framework, such as Tabu Search (TS) [31] and Simulated Annealing (SA). Introduced by Kirkpatrick et al. [47] in 1983, SA finds its inspiration in statistical mechanics. SA behaves similarly to LS, but additionally incorporates an annealing mechanic that enables the stochastic exploration of candidates that exhibit a lower fitness than previous solutions. A cooling schedule governs these mechanisms, which generally encourages a broader exploration of the domain early in the search procedure.

Evolutionary Algorithms

While LS and its variants provide useful optimization frameworks, Global Search (GS) techniques often enable more effective and robust domain sampling mechanisms [61]. Of the many GS search algorithms developed over the years, Evolutionary Algorithms (EA) have gained the most notoriety within the SBSE community [36].

EAs are widely applicable and robust, population-based algorithms that practitioners have employed in a broad range of engineering tasks [36]. Though its conceptual origins trace back to Alan Turing's "*Computing Machinery and Intelligence*" [84], it was John Holland [46] who in 1975 formalized and cemented the modern EA paradigm. A plethora of specialized EAs have emerged from a vast spectrum of research domains, but arguably the most widespread variation is the class of Genetic Algorithms (GA). Algorithm 1 provides the pseudocode for the simple GA proposed by Holland. This constitutes the foundation

of specialized automated test case generation algorithms for several tasks, including unit testing [83, 70], data flow testing [30], and REST API testing [5]. Inspired by the theory of natural evolution, standard GA formulations share several core ingredients:

- A *representation* of *individuals*, often referred to as *chromosomes*. Each individual represents a solution to the underlying problem. Together, the collection of individuals present at a given iteration forms a *population*.
- A *fitness* function that determines the quality of individuals. The fitness function is generally linked to one or more underlying *objectives*.
- A *variation* mechanism that alters the individuals in a population to create *offspring*. Variation often consists of a *mutation* operator that changes an individual's chromosome, and a *crossover* operator that combines individuals to create new offspring.
- A *selection* mechanism that is responsible for pressuring the population toward better solutions. Selection is generally driven by the fitness of individuals and strongly influences the rate of convergence of the algorithm.

Algorithm 1: Simple Genetic Algorithm

Input : Population size n
Output: N solutions

```
1 t ← 1
2 P_1 ← InitializeAndEvaluatePopulation(n)
3 while ¬ ShouldTermiante(P_t) do
4     O_t ← CreateAndEvaluateOffspring(P_t, n)
5     P_{t+1} ← SelectIndividuals(P_t, O_t, n)
6     t ← t + 1
7 return P_t
```

2.4 Context-Free Grammars

Grammars are powerful tools used to study, specify, and understand languages. A Context-Free Grammar (CFG) is a recursive structure that specifies a language's syntactic properties, without accounting for semantics. CFGs have become central to many fuzzers thanks to their ability to effectively constrain generative processes to input that is syntactically meaningful for the target SUT. For the remainder of this study, we use the terms *grammar* and CFG interchangeably, and we adhere to the definition put forward by Sipser [77].

Formally, a CFG is a 4-tuple $G = (V, \Sigma, R, S)$, with (i) V a finite set of *variables* or *symbols*, (ii) Σ a set of *terminals* or *terminal symbols* such that $\Sigma \cap V = \emptyset$, (iii) $R \subseteq V \times \Sigma \cup R$ a set of *rules* or *productions* that define the ways in which a variable can be *expanded*, and (iv) $S \in V$ a *start symbol* or *start variable*.

Chapter 3

Related Work

This chapter surveys relevant research work within the fields of compiler testing and fuzzing. Section 3.1 examines existing automated compiler testing tools and their underlying rationale. Section 3.2 surveys algorithms for specification fuzzing, a problem whose elemental challenges share many commonalities with compiler testing.

3.1 Automated Compiler Testing

Over the years, specialized testing approaches have emerged as tools for improving compiler reliability. To navigate the ample space of approaches, we adopt a simplified version of the taxonomy proposed by Chen et al. [15]. Most notably, we distinguish between *test program generation* and *program mutation* approaches, discussed in subsections 3.1.1 and 3.1.2, respectively. The former generates programs from scratch, without any external seed, while the latter mutates existing programs to generate new variations. Test program generation approaches also differ in the way in which they utilize the grammar of the target language. Based on this property, Chen et al. [15] distinguish three categories. Grammar-directed approaches solely rely on the language grammar to generate novel test cases. Grammar-aided approaches heuristically exploit a given grammar, which is often enriched with semantically significant context. Fuzzers may also not directly rely on a grammar, and instead define sound heuristic that respect a given linguistic syntax. Finally, we distinguish a third, separate category of fuzzers that fundamentally rely on machine learning techniques to generate input. Since these fuzzers do not fit directly in with the standard taxonomy, we address them separately in Section 3.1.3.

3.1.1 Test Program Generation

Purdom [72] devised one of the first algorithms aimed at validating the correctness of CFG parsers. This grammar-directed approach applies iterative rewriting rules starting from a starting symbol, eventually reaching all grammar productions. The algorithm heuristically favors short sentences that aid in the debugging of uncovered faults. Faults are detected when the production sequence used to generate a particular sentence differs from the representation returned by the parser under test. Purdom's [72] empirical analysis demonstrates

that this algorithm is effective at exploring the available states and transitions of CFGs. However, this simplistic approach does not generalize well to contemporary programming language compiler testing, as it contains no mechanism to account for the rich semantics that accompany CFG specifications.

Yang et al. [89] propose CSMITH, a grammar-aided test program generation tool aimed at finding bugs in commercial C compilers using DT. CSMITH first creates a set of `struct` type declarations, with a stochastically-driven number of members, which it stores as additional information next to the C programming language grammar. CSMITH then creates programs in a top-down fashion, starting from a single program entry point, the `main` function. At each iteration, it interweaves two steps: (i) instantiating a randomly chosen production from its grammar, accompanied by target completion, and (ii) validating the intermediate program using a series of safety checks with the goal of avoiding undefined behavior. The generation process follows an "on-demand" paradigm, where `struct` members and functions are generated when previously generated expressions access them. This approach fundamentally differs to Purdom's [72] paradigm, in that it strongly emphasizes the semantic constraints of C over structural CFG coverage. CSMITH's effectiveness has inspired several variations such as Lidbury et al.'s CLSMITH [54] and Morisset et al.'s [65] tool for uncovering concurrency bugs.

Livinskii et al. [56] introduce YARPGEN, a C/C++ program generation tool that aims to increase the expressiveness and diversity of test programs. YARPGEN proceeds in top-down fashion and does not explicitly utilize a formal language grammar. Instead, it tracks a *type environment* that stores all visible composite data types, which implicitly guide the generative process. The environment is iteratively enriched with newly generated types, each having access to all previous entries. Once established, the generated types drive the creation of global variables that can be used in future statements. YARPGEN incrementally builds functions that utilize the previously generated types and symbols. Each expression in a statement is recursively constructed such that it avoids Undefined Behavior (UB). To further increase the soundness of its output, YARPGEN performs local fixes that eliminate several common UB instances. Finally, *generation policies* provide a mechanism for adapting the probability distribution dictating YARPGEN's choices. This feature serves as a way of incorporating prior knowledge into the program and enables users to target specific features of the language or compiler.

Holler et al. [43] developed LANGFUZZ, a grammar-directed fuzzing approach that combines generative and mutative processes. The mutation process follows a two-phase approach, first *learning* code fragments from existing test suites, and then constructing executable mutants from its input. A grammar enables the extraction of code fragments, which LANGFUZZ obtains by breaking down hand-written test suites. In practice, a *fragment pool* emerges from the non-terminal instantiations of grammar rules within the input test suites. To complement this mutative process, LANGFUZZ leverages a generative procedure that stochastically expands the input grammar in a breadth-first manner. The two approaches work in tandem, comprising a trade-off between the flexibility of the input grammar and the rich semantics of test suites.

Han et al. [35] introduce the notion of *semantics-aware assembly*, which they implement in the CODEALCHEMIST tool, aimed at JavaScript engine testing. The key concept

in semantics-aware assembly consists of building blocks, also referred to as *code bricks*, a notion similar to LANGFUZZ's *code fragments* [43]. A code brick consists of a valid JavaScript Abstract Syntax Tree (AST) that is additionally annotated with an *assembly constraint*. Assembly constraints are sets of pre- and post-conditions that must be satisfied for two code bricks to be compatible. Such constraints specify what variables must be defined in preceding code bricks, and what additional objects are visible in subsequent bricks, respectively. CODEALCHEMIST produces programs by splitting seeds into code bricks, which it then stochastically interconnects in ways that adhere to the bricks' assembly constraints.

Though effective for their own target ecosystems, none of the approaches discussed in this subsection target Kotlin, and adaptation of already mature tools is likely a costly and challenging endeavour. In addition, the core mechanism that underlies these approaches is fundamentally random. Unlike this study, no approaches discussed in this subsection attempt to explore whether heuristic guidance criteria can make a significant contribution to the quality of the fuzzers. In this study, we aim to conjoin notions of syntactical structure and semantic constraints already present in current literature while also investigating whether heuristics can improve fuzzer performance when compared to sheer randomness.

3.1.2 Program Mutation

Le et al. [49] introduce the notion of Equivalence Modulo Inputs (EMI), a framework for establishing semantic equivalence between programs. Given a set of valid program inputs I, two programs P_1 and P_2 are equivalent modulo I if and only if their outputs are identical for all inputs $i \in I$. This notion lends itself naturally to the DT framework: since P_1 and P_2 are expected to produce identical results for all input in I, any discrepancies that emerge during execution can be attributed to compiler faults. ORION is a mutation-centric realization of this framework for the C programming language. To derive semantically equivalent programs, ORION profiles the execution of a program P over an input set I and determines which code snippets are not executed. ORION then exploits this information to stochastically prune the unexecuted code, maintaining the semantics of the original program P.

Le et al. [50] extend ORION in a tool named ATHENA, equipped with a heuristic guidance mechanism. Still reliant on the EMI framework, ATHENA improves the random mutation-driven search of ORION with a heuristic diversity-based probability distribution. The goal of this strategy is to exercise an ample range of compiler optimization mechanics, which is more likely as when the equivalent programs themselves are more diverse. To achieve this, ATHENA leverages a Markov Chain Monte Carlo (MCMC) sampling technique. A CFG-based similarity metric guides the MCMC search, favoring programs that significantly differ in their CFG representation. Novel programs emerge from either the deletion of unused code, or the insertion of code in unused sections.

Zhang et al. [91] introduce the notion of Skeletal Program Enumeration (SPE), a framework for exhaustively representing variable usage patterns in small programs. SPE breaks a program down into three components: a syntactic skeleton, a collection of variable placeholders, and a set of variables. Variable usages within a program are conceptually transformed into *holes*, that can be *filled* to *realize* a concrete program. This framework enables the enumeration of all possible arrangements of variables in such a way that filling the holes

results in novel programs. The authors further define the concept of α-*equivalence* between programs. Two programs are α-equivalent if they exhibit the same control- and data-dependence properties. Notably, naive enumeration can result in numerous α-equivalent programs under different variable permutations. To avoid the generation of such equivalent programs due to α-*renaming*, the authors show the SPE can be reduced to the *set partition problem*, which displays the same equivalence structure. The proposed algorithm handles variable scopes by promoting certain *local holes* to global ones and computing the Cartesian product between the two disjoint sets.

Stepanov et al. [79] propose Type-Centric Enumeration (TCE), an extension of SPE that strongly emphasizes types as a powerful medium for generating semantically diverse programs. Similarly to SPE, TCE deconstructs a program into a skeleton, a collection of placeholders, and a set of variables. In addition to this breakdown, however, TCE equips each placeholder with an adherent type, and further considers a set of *callables* that it uses alongside variables. The algorithm proceeds in two phases. The *generation* phase is responsible for the creation of a set of typed expressions, which it extracts from the callable set of its seed program. The *mutation* phase operates on a different seed program, which it first merges with that of the previous stage. Subsequently, the expressions obtained in the generation phase fill type-compatible placeholders placed in the latter seed program. Additional expressions obtained either from the standard library or primitive types are additionally employed to populate remaining unfilled placeholders.

Program mutation approaches generally face less stringent constraints than their generative counterparts, thanks to their reliance on existing seeds. This makes the application of complex guidance heuristics a more feasible endeavour for mutative approaches rather than generative ones. The work of Le et al. [50] is a powerful example of the effectiveness of sophisticated heuristics in a compiler testing scenario. However, whether the value of heuristics also applies to generative approaches remains an open research question that we aim to begin addressing in this study.

3.1.3 Machine Learning-Based Fuzzers

Amodio et al. [2] develop an approach called Neural Attribute Machines (NAM), that utilizes a Long Short-Term Memory (LSTM) [42] neural network to embed multifaceted constraints and structural context into neural program generation. Their method extends the *attribute grammar* formalism by leveraging an LSTM that is capable of generating AST nodes of test programs. The NAM architecture exploits a tri-level loss function that seeks to minimize (i) the model's divergence from the training set distribution, (ii) the model's likelihood of violating semantic constraints, and (iii) the model's likelihood of generating structurally valid, but nuanced incorrect code. Training occurs on the level of AST nodes, additionally equipped with surrounding context and encoded by the NAM's *logical unit*. Experiments show that structure emerges from the training procedure, but the likelihood of breaking common language ordering and type-safety constraints remains meaningful.

Cummins et al. [18] build a test program-based generative fuzzer that relies on a neural network for code generation. The system depends on a corpus of code repositories that serve as training data for the neural network. At the core of the system resides a LSTM that

models the distribution of vocabulary tokens within the input corpus. To generate programs, the LSTM is sampled to produce novel vocabulary words in a token-by-token fashion, which are first decoded before being packaged together into viable test programs. The authors' empirical analysis shows that despite having no direct model for the language structure, the tool, called DEEPSMITH, is able to generate highly structured code and uncover dozens of bugs in the OPENCL system.

We also note the existence of many statistics- and machine learning- driven approaches for tangential tasks, such as in the fields of program synthesis [34] and program completion [74]. These tasks share similarities with the challenges of this study, namely the generation of valid and contextually-aware code. Though extendable to the field of compiler testing, these methods additionally account for *user intent* (i.e., the goal of the program a user is attempting to synthesize). Compiler fuzzers broadly disregard this aspect, as their goals differs from the former categories.

The works of Amodio et al. [2] and Cummins et al. [18] both show that machine learning-based techniques can be powerful mediums for encoding and generating code. In recent years, researchers have pushed the boundaries of task-specific code generation even farther. However, whether such techniques can integrate well into the more typical fuzzers addressed in previous subsections remains an open research question. In our approach, we aim to leverage the power of current state-of-the-art machine learning code models within a classical fuzzer architecture to enable the development of novel heuristics.

3.2 Specification Fuzzing

Steinhöfel and Zeller [78] propose the Input Specification Language (ISLA), a declarative language for integrating context-dependent semantic constraints in otherwise coarser CFG definitions. Programming languages often impose complex semantic relations that CFGs cannot encapsulate. This includes even commonplace constraints such as *"a variable must be declared before it is assigned"*. ISLA builds upon constraint solvers to incorporate complex annotations that can model such relations. The ISLA solver iteratively expands a tree representing potential admissible expansions of grammar nodes under annotated grammar rules. Expansions that do not match the annotations are excluded from the cost function-guided solving procedure, as valid candidates emerge. Eventually, if a the solver discovers a valid terminal expansion, it first (approximately) translates it to a regular expression representation before querying a specialized constraint solver. ISLA then converts this solution back to a derivation tree representation and substitutes it in the original grammar graph.

Havrikov and Zeller [39] study coverage criteria of CFGs and accompanying coverage-centric generative algorithms. *Symbol coverage* is a straight-forward metric for evaluating how much of a specification grammar a derivation tree covers. Symbolic coverage simply measures the fraction of symbolic nodes in the grammar that the derivation tree covers. *k-Path* coverage is a more nuanced criterion that additionally takes into account the *context* surrounding a symbolic node's position within a tree. Formally, a *k-path* is a directional sequence of nodes in a derivation that contains exactly k symbolic nodes. The concept of *k-path* coverage naturally follows from this definition: a collection of inputs achieves *k-path*

coverage if each *k-path* in the grammar graph is present in the underlying set of derivation trees. The authors derive an enumeration-based *k-path* algorithm stochastically iterates through all *k-paths* up to a user-defined depth. The terminal derivation trees uncovered through this guided traversal instantiate specification-abiding sentences.

Kreutzer et al. [48] introduce the language specification language (LALA), a framework for semantically enriching the syntactic specification of languages. At its core, LALA offers a declarative implementation of an attribute grammar that operates over programs represented as ASTs. Each production in the underlying syntactic CFG takes the form of a class that embeds additional mechanisms to provide semantics to the overarching AST. ASTs model information flow both bottom-up (through *synthesized* attribute values) and top-down (through *inherited* attribute values), and their semantic validity is determined through boolean-valued *guard* attributes embedded in some nodes. The *SMITH fuzzer leverages this representation to generate valid program representations through ASTs with conforming guard attributes. The algorithm generates programs by instantiating nodes visited through the random walking over available productions, additionally caching *fail patterns* that occur when nodes with false guard attribute values emerge. Additional heuristics for fail pattern recognition, as well as subtree replacement and deconstruction additionally improve the effectiveness of *SMITH.

Specification fuzzing approaches demonstrate that targeted fuzzers can benefit from an increased level of abstraction. While language-specific approaches can be highly effective for the particular software systems they target, the specification fuzzing class of algorithms brings other advantages. Most notably, the more general representations language fuzzers adopt and the weaker semantic assumptions they rely on make them more versatile tools. Though in this thesis we only consider the Kotlin compiler, we aim to design a tool whose potential is not bounded to this ecosystem. Instead we seek to strike a balance between the highly constrained and targeted nature of older compiler fuzzers and the highly customizable features of specification fuzzers. Such a balance allows the integration of rich domain-specific knowledge with a generalizable interface without necessitating the vast amount of effort associated with building language-agnostic tools.

Chapter 4

Test Program-Based Fuzzing for the Kotlin Compiler

4.1 Approach Overview

This chapter details the foundational techniques, abstractions, and trade-offs that our novel approach employs to generate syntactically valid and semantically meaningful Kotlin code. We design our approach along three main pillars, broadly depicted in Figure 4.1. The first two pillars tackle the fuzzer's internal connection to Kotlin through semantics and syntax. The third coalesces the interfaces provided by the first two into a powerful representation, which allows the integration of complex heuristic approaches into the fuzzing process.

Section 4.2 addresses the first core component of the fuzzer, which handles the semantics of Kotlin. This module contains representations for a wide range of Kotlin code constructs, a detailed type hierarchy, and adaptable constraints that aid with code generation at runtime. In addition, this *context* component is equipped with an extraction procedure that allows users to customize the semantic environment in which the fuzzer operates.

Section 4.3 discusses the syntax handling module of the fuzzer. This component operates on a specification of the Kotlin CFG, which it transforms into a truncated version. The transformation additionally equips the grammar with operations that integrate with its context-handling counterpart, in turn enhancing versatility and configurability of the fuzzer.

Sections 4.4 and 4.5 outline the final pillar of our approach. The former establishes a hierarchical structure that aligns code within generated files with its local dependencies, enabling powerful mutation operators. The latter develops this structure into three classes of search algorithms. Finally, a fourth auxiliary configuration component augments the three main pillars and allows users to select from the many techniques established in this chapter.

The three main components of the fuzzer come together in a framework that enables the guided generation of code. From a high-level perspective, the semantic and syntactic modules cooperate to produce novel pieces of code. The extended CFG ensures that generated code structurally obeys the rules of Kotlin, while the context counterpart helps assure that the code is semantically sound. Our approach uses this sampling procedure as a basis for more sophisticated GA-based heuristics that *guide* generation towards different objectives.

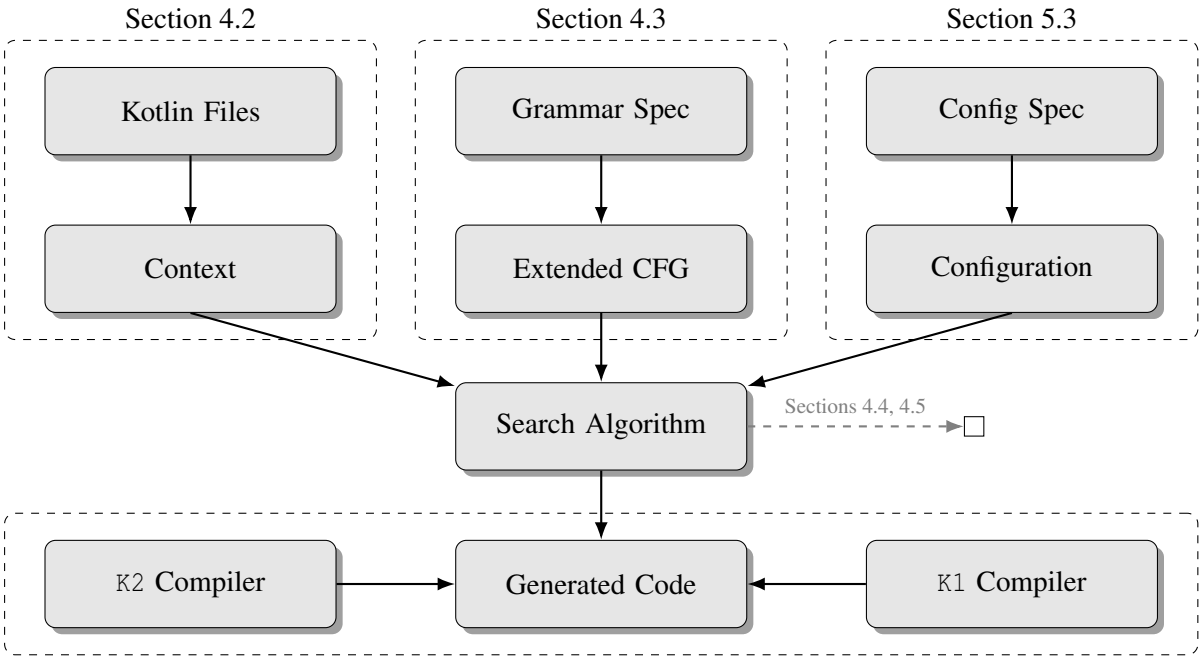


Figure 4.1: Conceptual layout of the approach.

4.2 Semantic Interface

Kotlin is a rich programming language, that conjoins various paradigms, each building upon numerous layers of theoretical abstractions. These abstractions collectively govern the semantics of the language, and implicitly determine the validity of programs. We jointly represent the core concepts that compose the Kotlin semantics, as well as the constraints that underlie their behavior, under an construct concept called the fuzzer's *context*.

The context contains both adaptive, program-specific details (such as scopes and visible identifiers), and universally applicable statutes (such as type hierarchies). The information encapsulated in the context drives the generative process, similarly to YARPGEN [59]. However, unlike YARPGEN, we endow our representation of the context with an extraction feature that allows the parsing of arbitrary Kotlin code to serve as the basis of further code generation. The remainder of this section describes three key features of the context. Section 4.2.1 first addresses the type hierarchy representation. Section 4.2.2 describes the internal representation of language structures. Finally, Section 4.2.3 lays out the automated context extraction process.

4.2.1 Type Hierarchy Representation

Kotlin is a strongly-typed language, with well-formed semantics that are crucial for generating valid code snippets. We represent the Kotlin type system using a labeled Directed Acyclic Graph (DAG) data structure. Formally, a DAG G is a tuple $\langle V, E \rangle$ composed of a set of vertices V and a function $E : (V \times V) \to L^+$ that links pairs of vertices in V through a

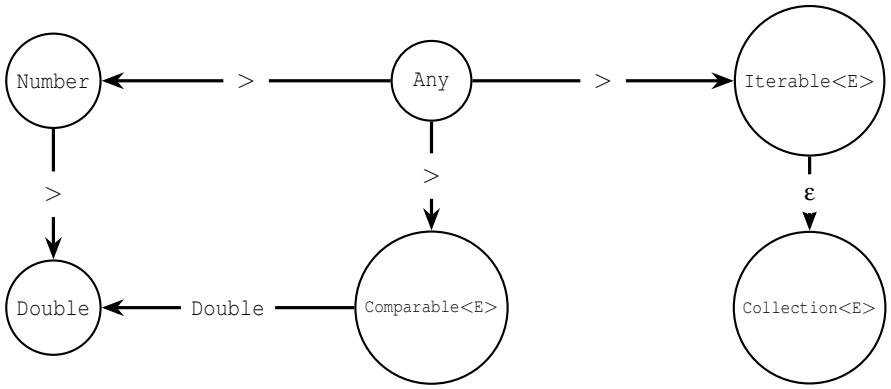


Figure 4.2: A sample of the type hierarchy representation using standard Kotlin types.

composite transition from a domain L.

The set V comprises all the accessible *classifier* (classes and interfaces) types (i.e., (abstract) classes and interfaces) in the environment. The function E maps each pair of vertices $\langle u, v \rangle$ with $u, v \in V$ to a label $l \in L = \{\emptyset, >, \varepsilon\} \cup V$, where the label signifies the inheritance relation between the two types. A $\emptyset$ transition signals that there is no direct relation between types, while a $>$ transition symbolizes that v is a direct and unparameterized subtype of u. To accommodate the semantics of parameterized types, we enrich the label set L with all types belonging to V, as well as an additional symbol ε. A label belonging to V signifies that one of the parameterized types of u explicitly belongs to a type hierarchy, while the ε transition represents the inheritance of a symbolic type.

Figure 4.2 depicts the DAG representation a small subset of built-in and standard Kotlin types, with $\emptyset$ transitions left out. Within this hierarchy, the `Number`, `Comparable<E>`, and `Iterable<E>` types all directly inherit from the `Any` node, which is the root of the type hierarchy. The `Number` and `Comparable<E>` have a common immediate descendant in the `Double` type, though their transitions differ. The transition between `Comparable<E>` and `Double` itself contains the annotation `Double`, an exact type referring to a node in the graph that is required to complete the inheritance. The ε label on the edge between `Iterable<E>` and `Collection<E>` denotes that `E` is a symbolic type, and that a link between the two parameterized types must be consistent. The meaning of consistency is intricate, as parameterized types in Kotlin may contain bounds that further limit the domain of valid parameter types. To integrate this constraint, bound information is additionally stored in edge labels. When attempting to infer the position of a node in a graph, we additionally include information about that node in the inference procedure. This allows the graph to account for recursive or self-referential type definitions (i.e., `Double : Comparable<Double>`).

The DAG type hierarchy offers several advantages. First, a graph representation is flexible in that new types can easily be added, removed, or changed. This flexibility enables the automated construction of a type hierarchy from pre-written Kotlin files, which allows for near-endless customization of the fuzzing space. Second, a DAG topology enables the encoding of strict semantic constraints into graph problems. This applies to simple inheritance semantics (i.e., subclasses may be used wherever a superclass is expected), but can

```
1  class Foo <E>(bar: String, baz: E) {
2      val fooProp1: String = bar
3      var fooProp2: E = baz
4      fun <P> fooFunc(parameter: P) : E {
5          return fooProp2
6      }
7  }
```

Figure 4.3: Simple Kotlin snippet.

also generalize to Kotlin-specific traits, such as requiring subtypes of `Iterable<E>` whenever a `for` loop occurs. Finally, the DAG greatly prunes the search space of semantically valid solutions, as simple traversal algorithms suffice to solve problems related to sound parameterized type inference, a common problem when fuzzing the Kotlin standard library.

4.2.2 Callable Representation

In addition to a type hierarchy, the fuzzer uses knowledge to which *callables* it can act upon. We use the term callable in a similar fashion to Stepanov et al. [79]. The collection of callables in the context encompasses the set of functions, methods, constructors, variables, constants, and predefined primitive values available to the fuzzer. We model each callable as a 5-tuple $\langle N, O, P, I, R \rangle$ that builds on top of the type hierarchy. Let N the name of the callable, O the type of the callable's *owner* class, P a list of (symbolic) parameterized types specific to the callable, I a list of input types that the callable requires, and R the return (or output type) of the callable. This representation draws inspiration from lambda calculus and aims to provide a uniform representation for all callables that Kotlin can represent.

To help build intuition on the applicability of this representation, consider the snippet depicted in Figure 4.3. Though simple, the `Foo` class contains several representationally challenging constructs that help illustrate the capability of the callable construct. First, the `Foo<E>` type itself belongs to the type hierarchy of the underlying context. The `fooProp1` class property becomes the $\langle$`fooProp1`, `Foo<E>` $, \emptyset, \emptyset,$ `String`$\rangle$ quintuple, which is akin to a lambda-calculus function that takes no input and returns a `String` value. The `fooProp2` property is analogous, with the exception of the return type, which is now a symbolic type `E`, which is linked to the owner class' parameter type. During runtime, the fuzzer leverages this link to first determine other contextual constraints on `E`, and then sample an appropriate concrete type to fulfill the link.

Finally, the `fooFunc` method becomes the $\langle$`fooFunc`, `Foo<E>` $, \{$`P`$\}, \{$`E`$\},$ `String`$\rangle$ 5-tuple, which links both parameter's input type to the method's own parameterized type `P` and the return type to the class' parameterized type `E`. A similar translation occurs for the primary constructor of `Foo`, which becomes the $\langle$`Foo`$, \perp, \{$`E`$\}, \{$`String`$,$`E`$\},$ `Foo<E>` $\rangle$ quintuple that takes two inputs and returns an object of type `Foo<E>`. In the latter case, the constructor's owner class is left out, as a constructor call does not require access to an object. As in previous cases, the link exists between the `E` type of the callable representation, and the (possibly constrained) `E` type in the definition of `Foo`.

4.2.3 Context Extraction

To achieve the desired utility, the context representation must be capable of fulfilling two core tasks. First, it should contain sufficient information to enable a fuzzer to generate valid Kotlin code through successive queries. Second, it should be flexible enough to support the *extraction* of relevant types and callables from pre-existing pieces of code, without requiring the user to manually intervene in the process. The latter requirement drastically increases the applicability of such a tool in the real world, where practitioners might want to investigate particular language features, specific implementations of standard interfaces, or proprietary code bases. To accomplish this, we developed a three-phase context extraction process that parses a collection of input Kotlin files and produced a sound representation using the type hierarchy and callable abstractions. Figure 4.4 depicts this method.

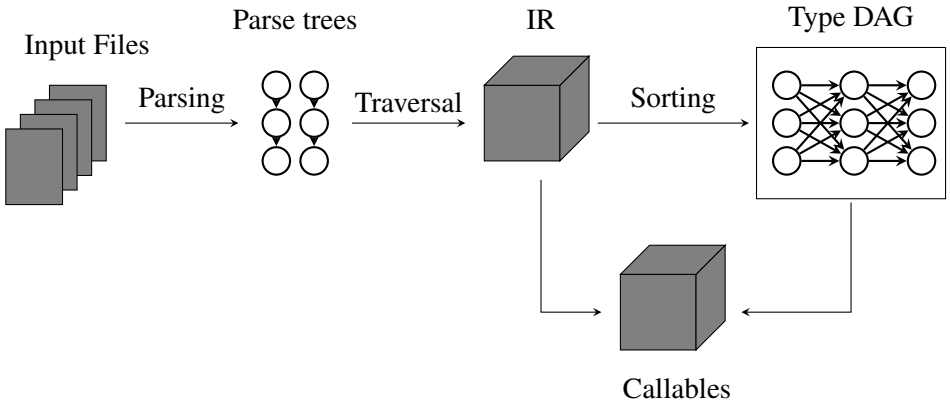


Figure 4.4: Simplified visualization of the context extraction process.

The extraction process operates on a parsed representation of the input files, which for the scope of this research consists of built-in and standard library Kotlin files. The first phase traverses the parse trees of the Kotlin files and maps relevant information into an intermediate representation. The retained information includes the necessary details to identify language constructs, available types, and relations between parameterized types. The second step then creates the links between parameterized types in class declarations and their properties, while simultaneously traversing the intermediate representation and inserting metadata with regarding visibility and inheritance modifiers.

The third step of the procedure involves building the type hierarchy DAG. At this point, extracted class definitions contain links that allow for the differentiation between symbolic and concrete types, which makes resolving inheritance relations possible. Because of this, a simple topological sorting algorithm suffices to construct the DAG. An initial `Any` node serves as the root of the graph and represents the parent for all classifiers that do not explicitly inherit from other constructs. This design choice simplifies the application of further graph traversal algorithms.

The topological sorting then proceeds by filtering available parsed classifiers on three conditions: (i) that the direct parents of the classifier have been added to the hierarchy, (ii) that the concrete parameterized types of the classifier have been added to the hierarchy, and

(iii) that the owner type of the classifier has been added to the hierarchy (if nested types are encountered). Classifiers that meet these requirements can be added in an arbitrary order. The algorithm then proceeds until all visible classifier types have been processed. This stopping condition places the burden of selecting a sound subset of input files on the user. However, we provide a subset of Kotlin standard library files as a sensible default option.

Once a type hierarchy has been established, the final step creates links between the processed types and the relevant callables. Each intermediate representation of a callable receives a copy of its owner's type before being converted into its final representation, which enables conversion to valid Kotlin code. The context tracks each callable by its owner's type, with a special empty owner for built-in functions. This method of storage not only models the intuitive structure of Object-Oriented (OO) code (with container classes and their properties), but decreases the complexity of common queries during runtime.

4.3 Syntactic Interface

Effectively leveraging the syntax of a programming language is a prerequesite for ensuring the correctness of automatically generated code. In this work, we seek to exploit the usage of the Kotlin CFG as both a means of increasing the flexibility of the fuzzer and a guide that enforces the validity of generated code. This places our fuzzing approach in the *grammar-aided* category defined by Chen et al. [15].

Though highly informative, a CFG-based model of linguistic syntax suffers from several limitations which curtail the grammar's standalone potential within the fuzzer. The remainder of this section describes the implementation of the grammatical representation within the fuzzer, its integration with the context, and the reasons behind the enhancements we enriched the grammar with.

4.3.1 Extending the Kotlin CFG

In attempting to enhance the contextual integration of the Kotlin CFG, it is important to consider its practical setbacks. The Kotlin developer team provides a standard grammar specification [45], implemented using the ANTLR4 framework. ANTLR4 [71] is a parser generator that offers a broad range of language-targeted functionality, including the validation and structured parsing of input with respect to a given grammar.

The available grammar implementation encapsulates a highly complex and finely detailed set of rules. In its current form, the CFG consists of two components - a Lexer grammar and a Parser grammar. The former pertains to the lower-level components of the Kotlin language, including keywords, symbols, and numerical representations, split between 315 symbols and 378 productions. The latter models the higher-level constructs of the language, ranging from simple statements to class definitions, and totals 170 variables and 353 rules.

Altogether, the two compartments create a substantial collection of non-trivial syntactic constraints, the complexity of which increases with each nested relation. In practice, this complexity causes straight-forward specification fuzzing algorithms (such as, for instance, traversing the grammar) to generate semantically invalid Kotlin code with overwhelming likelihood. This in turn imposes challenges on any fuzzer attempting to utilize these rules in

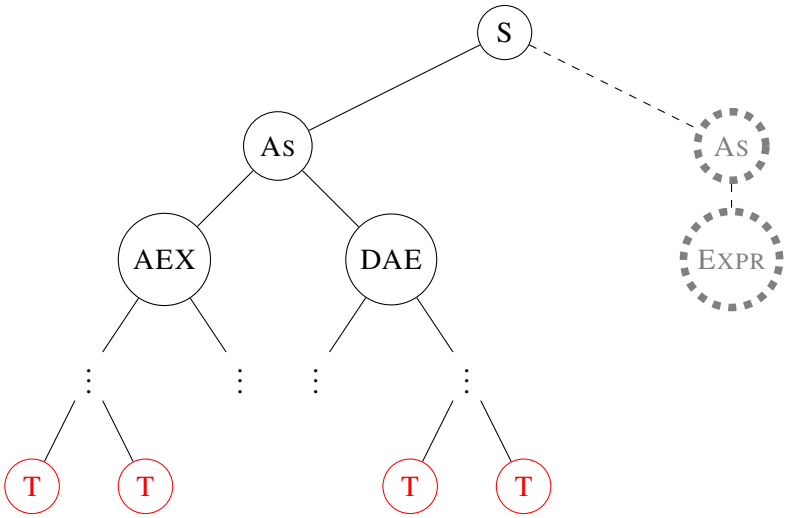


Figure 4.5: Sample CFG transformation.

any substantial sense. An implementation of grammar-aided techniques requires a solution that reduces the likelihood of generating invalid code by sampling grammar productions.

To fulfill this requirement, we propose an approach that *extends* the core Kotlin CFG with an enhanced representation that circumvents several key shortcomings. The transformation from the barebones CFG to its extended version takes place at the node (symbol) level in the graph representation, as to improve comprehension. To obtain this transformation, we use a traversal algorithm which takes as input the ANTLR4 core-grammar node and outputs its enhanced version. This algorithm performs a single key operation for each symbol: it endows each node in the core CFG with a grammar-abiding `sample` property, which aims at reducing the complexity of generating valid samples. The atomic node transformation can either leave a node unchanged or *truncate* its productions. Simple nodes such as ones solely containing unicode characters are preserved, while nodes which introduce substantial contextual semantics are subject to change. This reduces the problem of obtaining valid samples from the grammar to implementing appropriate methods of leveraging the `sample` property of each affected node.

Figure 4.5 depicts an example of a transformation performed on a section of the Kotlin grammar. In this scenario, the root node S represents a *statement* in the Kotlin language and directly maps to a symbol in the core CFG. The symbol S could follow a production to sample an *assignment* node AS as depicted in the left half of the figure. This rule introduces additional semantic constraints, which the grammar omits. The AS node may be extended into either an assignable expression (AEX) or a directly assignable expression (DAE), the details of which are not important for the sake of this example. Either production can in turn unravel into a chain of derivations that eventually sample terminal nodes T. Following this series of derivations, the semantic constraints that the production $S \rightarrow AS$ entails (such as type consistency in the assignment) do not propagate to further links in the chain, and thus the likelihood of generating valid code by trivially sampling terminal nodes in the derivation is vanishingly small.

The transformed grammar shown on the right branch of the S symbol circumvents this problem by truncating the grammar at the node that introduces such constraints. In this case, the algorithm transforms AS node into its extended version, which encodes relevant constraints within its `sample` property. Transformed nodes may additionally transfer such constraints between them when sampled: for instance, the assignment node may propagate its type-consistent constraint to a generic EXPR node that generates constraint-aligned and valid Kotlin expressions.

This extended CFG technique provides several advantages. First, it forms a framework that solves the problem of semantically constrained grammar sampling. By implementing the appropriate `sample` properties, one can effectively evade the problems that the context-invariant grammar raises. Second, it is an effective trade-off between implementation difficulty and core grammar usage. This technique preserves the key properties and relations in the Kotlin grammar, while not necessitating that every production is individually catered to. For instance, transformed nodes may simply inherit part of their structure or their entire specification from the ANTLR4 definition.

Finally, the extended CFG framework serves as a versatile foundation for fuzzing algorithms, by allowing higher-level routines to target specific symbols in the grammar. This property favors configurability and allows for extensive adaptations of the core grammar. Practitioners may additionally choose to sample only select nodes in the grammar, thus effectively targeting desired subsets of the Kotlin language.

This approach stands out from the current body of literature through the balance it strikes between the level of user ”intervention” in re-writing grammar rules and the utilization of pre-written grammar specifications. In isolation, neither side of this trade-off is novel. CSMITH [89] has demonstrated the effectiveness of manually constructing a sampling structure without direct usage of a grammar over a decade ago, while approaches like ISLA [78] show that constraint-annotated grammars can effectively encode common and meaningful semantic constraints. However, our integration of contextual sampling within an extended grammar representation enables algorithms built on top of it to individually tune the semantic constraints of each symbol, while always retaining to option to ”fall back” on the pure grammar representation where appropriate, preserving the *shape* of the grammar.

4.3.2 Connecting Context and Grammar

To fully leverage their `sample` property, nodes in the extended grammar representation require an extensive connection to their local context. To facilitate this arrangement, we connect the initial (or *root*) context to the node from which sampling starts. As sampling algorithms recursively visit nodes in the extended grammar, additional information embedded in each node helps the traversal decide whether the current context should be transmitted as-is, cloned, or mutated.

Simple nodes like expressions utilize an unchanged context, while more complex constructs, like variable declarations may insert additional identifiers. Encompassing constructs like class and function definitions are scope sensitive, which requires careful handling of local and nested operations. To address this, each such node creates a local copy of the context, to which it might add isolated information, before persisting globally accessible

data (i.e., function names, or new types) to the root context. In essence, this integration allows for autonomous context-sensitive random sampling from the extended grammar: the extended nodes each query the context in such a way that consistent constructs emerge. The context, in turn, provides an interface that provides several guarantees with respect to future parameterized queries.

4.4 Code Representation

To facilitate the implementation of measurable optimization criteria, we propose a versatile GA-based framework that supports common evolutionary search operators. Such a framework requires a central representation of Kotlin code that is granular enough to leverage the rich semantic information accessible through the context, and simple enough to support large numbers of operator calls during search. To achieve this, we designed a hierarchical chromosomal representation that distinguishes between three levels of separation.

The hierarchy roots itself in a tri-level complexity abstraction that discerns between pieces of code based on scope and isolated validity. Figure 4.6 illustrates this hierarchy. *Fragments* lie at the base of the pyramid and compose low-level language constructs, such as expressions and statements. These pieces of code do not capture any logic structure and rely heavily on surrounding semantic context. A step above are the code *snippets*: individually encapsulated pieces of code comprised of fragments. Snippets emerge when changes in scope are significant enough to merit a logical separation. Such scenarios generally occur along the lines of OO design, and include functions, methods, and classes.

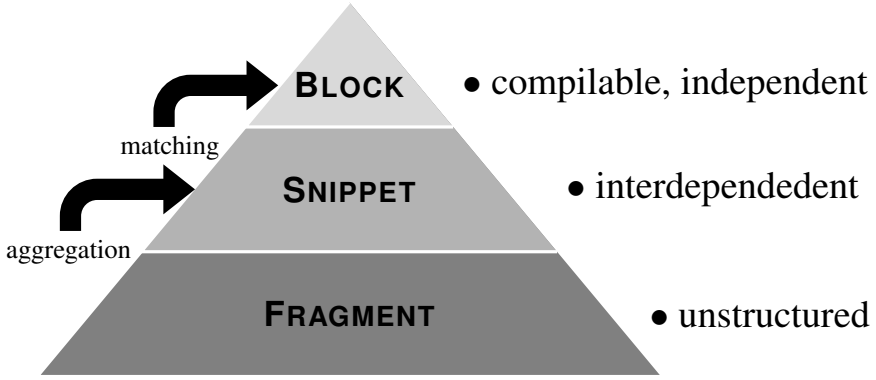


Figure 4.6: Representational hierarchy of code complexity abstractions.

Snippets serve as an intermediate layer that captures sufficient information to be individually actionable (i.e., a function that can be called), but not enough to be context-independent. The interdependency of snippets materializes as a consequence of the sequential nature of node sampling: as nodes recursively unfold their `sample` property, prior changes to the context trigger subsequent consequences (i.e., previously generated functions may appear in later expressions). To overcome this obstacle, several snippets can assemble into *blocks*, the highest construct of the pyramid. This snippet *aggregation* procedure works

```
1  fun x() : Double {
2          return 0.0
3  }
4
5  fun y() : Double {
6          return x() + 1.0
7  }
8
9  fun z() {
10         return y()
11 }
```

Figure 4.7: Illustration of fragments, snippets, and blocks.

on the basis of a dependency matching algorithm that recursively traverses snippets within a Kotlin file and retrieves external snippets that their fragments depend on.

The pyramidal abstraction construct shares conceptual similarities with both the works of Holler et al. [43] and Han et al. [35]. However, there are fundamental differences between our method and past work. Unlike Holler et al. [43], more sophisticated, language-targeted generation procedures serve as the basis of novel code blocks, which do not rely on input seed programs. The aggregation of snippets into blocks also differs from the semantics-aware assembly proposed by Han et al. [35], which infers language constraints from a corpus of over 200,000 files. Our approach requires no such input, making the generation process entirely self-sustaining.

Figure 4.7 exemplifies the differences between the three levels of the hierarchy. The statements in lines 2 and 6, as well as the expression in line 10 are all instances of *fragments*: pieces of code that are too small to track in the model. The functions `x`, `y`, and `z` are all *snippets*: independently separable pieces of code that are not necessarily self-sufficient. Finally, there are three *blocks* in this example: $\langle \texttt{x} \rangle$, $\langle \texttt{x}, \texttt{y} \rangle$, and $\langle \texttt{x}, \texttt{y}, \texttt{z} \rangle$. All three of these blocks share the property that they are context-independent: the dependencies for all snippets in the block are also part of the block. Preserving this property during search is critical not only because it is the most effective way of employing the context's semantic guarantees, but also because it ensures that compiler resources do not go to waste in analyzing incomplete files. The *block* abstraction is the cornerstone of the GA framework, shaping all higher-level operators and constructs. In the remainder of this section, we analyze the foundational elements of the evolutionary fuzzing toolkit.

Individuals. The population of the GA consists of individual *blocks* encoded in a chromosomal representation as follows. A chromosome consists of a list $\mathcal{L}$ of triples $\mathcal{S} := \langle \mathcal{R}, \mathcal{D}, \mathcal{T} \rangle$ representing snippets, with $\mathcal{R}$ a representation of the signature of the snippet, $\mathcal{D}$ a list of signatures of snippets that $\mathcal{S}$ depends on, and $\mathcal{T}$ the text that the snippet contains. Considering again the code in Figure 4.7, the chromosome encoding the block $\langle \texttt{x}, \texttt{y} \rangle$ is $\mathcal{L}_{\langle \texttt{x},\texttt{y} \rangle} = [\langle \texttt{x} : \lambda \rightarrow \texttt{Double}, \emptyset, \mathcal{T}_\texttt{x} \rangle, \langle \texttt{y} : \lambda \rightarrow \texttt{Double}, \{\texttt{x} : \lambda \rightarrow \texttt{Double}\}, \mathcal{T}_\texttt{y} \rangle]$ The first entry in the list models the `x` function, with a λ-calculus inspired signature, an empty dependency set, and the first lines 3 of the code. The second entry is analogous, except for the de-

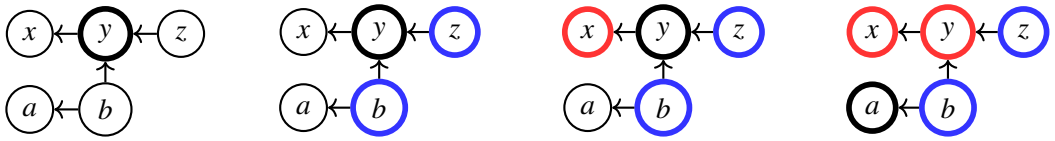
pendency set that contains the signature of x, signaling that lines of code 5-7 require the presence of x for completeness.

Context-Aware Partitioning. A downside of the established chromosome model has to do with the ordering of snippets inside $\mathcal{L}$. Though representationally irrelevant, the order of snippets within the list becomes crucial when the chromosome instantiates a concrete piece of code. Snippets in Kotlin can be order-sensitive depending on their scope. This means that to instantiate valid code, the list of snippets within the chromosome should be arranged in a dependency-aware topological order. Using again the example in Figure 4.7, this means that within the chromosome representing the entire piece of code, the snippet x must always precede y and z. Similarly, y must always appear before z.

There are several ways to achieve this ordering. Since the generative process is iterative, the first instantiation of a block always obeys this constraint. However, preserving this property during the application of variation operators proves more challenging. To effectively mutate and recombine individuals, blocks require the capability of identifying entire dependency structures centered around particular snippets. In other words, given a snippet s present in a block b, b should be able to identify a *partition* of itself that includes (i) s, (ii) all snippets that depend on s, and (iii) all downstream dependencies of (i) and (ii). We refer to a partition that such a structure as a *self-sufficient* partition of a block.

To build such a partition starting from an arbitrary snippet requires careful analysis of the dependency structure within a block. To implement this, we propose a technique that models the structure of a block as a *dependency graph*, a common data structure used in static analysis of programs. Each node n represents a snippet within the block, and each directed edge $n_1 \rightarrow n_2$ denotes that snippet n_1 depends on n_2. To construct a self-sufficient partition starting from a node (snippet) n, we employ two procedures that traverses the dependency graph to gather information regarding its topology. An *upstream* traversal starting at n walks along every possible path *ending* in n, in effect gathering all blocks that either directly or indirectly depend on n. Reflexively, a *downstream* traversal beginning at n walks on every path originating in n and gathers all of n's dependencies.

Figure 4.8 depicts several sample traversals of a snippet dependency structure similar to that introduced in Figure 4.7. Nodes with thick borders are affected by the traversal, while thinly bordered ones are not. Black depicts that the node is the starting point of the traversal, while blue and red denote nodes selected during up- and downstream traversals, respectively. Subfigure (a) gives the standalone topology of the block, and selects snippet y as the root of the traversal. An upstream traversal rooted in y would select b and z, which directly depend on y, as depicted in panel (b). A downstream traversal rooted in y would additionally yield snippet x, which is y's only direct dependency, as shown in subfigure (c). Finally, it is important to note that the repeated application of up- and downstream traversal on increasingly larger sets of nodes (i.e., each traversal considers *all* nodes returned by the previous) yields the largest connected subgraph containing the original starting node. Subfigure (d) illustrates three such traversals: starting at a, b is selected during an upstream traversal. Next, y and x from the downstream traversal rooted at b. Lastly, an upstream traversal rooted at either x or y yields z. The additional benefit of traversals is that they preserve the ordering of nodes in graph, which graph walking algorithms may fail to do.



(a) Initial choice of snippet.

(b) Upstream traversal from node `y`.

(c) Successive downstream traversal from node `y`.

(d) Successive up-, down-, and upstream traversals starting at `a`.

Figure 4.8: Sample upstream and downstream traversals of snippet dependencies.

Mutation. To fully take advantage of the properties of blocks, mutation operators must ensure the preservation self-sufficiency. Since perturbing the dependency hierarchy within blocks would render any validity guarantees pointless, we designed mutation procedures to operate on partitions of code blocks selected by means of the previously established traversals. This includes three mutation varieties:

- **Removal** of a selected partition. This operator first chooses a random snippet within the block and performs an upstream traversal from that snippet. The operator then removes all collected snippets from the chromosome.
- **Context-Free Addition** performs a sample operation on the node that the given block originates from, and appends the result to the end of the chromosome.
- **Context-Aware Addition** first brings the context in a state that is compatible with the block subject to mutation, and then performs a sample operation. The result extends the chromosome as in the context-free case.

Recombination. We propose a simple recombination operator that takes in a pair of chromosomes $\mathcal{L}_{p_1}$ and $\mathcal{L}_{p_2}$ and creates two offspring $\mathcal{L}_{o_1}$ and $\mathcal{L}_{o_2}$. The offspring start out as identical copies of each parent, before crossover swaps two collections of snippets between them. The collection of snippets emerges using repeated up- and downstream traversals initially rooted a randomly chosen snippet in the block. The resulting selections preserve both the self-sufficiency and ordering constraints of both the extracted and remaining partitions. The selected partitions are swapped and appended to the opposite chromosome.

Selection. We provide standard selection operators for both single- (tournament and proportionate selection) and multi-/many-objective (domination rank and domination count selection) fitness functions.

To exploit the power of this framework to its full extent, we allow for a large degree of external configuration. The remainder of this section covers three meta-heuristic strategies implemented on top of this general framework and analyzes their merits and potential.

4.5 Generative Heuristics

The semantic, syntactic, and representational interfaces described in Sections 4.2, 4.3, and 4.4 lay the foundation for more complex overarching heuristics to guide a generative process underlying fuzzing. This section explores three meta-heuristic guiding oracles shaped into several concrete algorithm variations and their application in the context of compiler fuzzing. Subsection 4.5.1 establishes a standard baseline algorithm, while Section 4.5.2, and 4.5.3 elaborate on two more complex heuristic approaches, centering on syntax and semantics, respectively.

4.5.1 Random Sampling

The simplest method to leverage the context and grammar constructs is to repeatedly sample a given node in the extended grammar by traversing context-informed random paths. We refer to this generative procedure as Random Sampling (RS). Though trivial in its scope, this approach has proven deceptively powerful and is the dominant search strategy employed in some shape by all state-of-the-art fuzzers discussed in Section 3.1. Despite its simplicity, RS establishes a powerful interface, which all algorithms involved in this study follow. Algorithm 2 describes the RS procedure.

Algorithm 2: Random Sampling

Input : Node to sample $\mathcal{N}$, Context $\mathcal{C}$, Time Budget s
Output: Collection of generated Kotlin files

```
1 A ← ∅
2 while ¬ TimeElapsed(s) do
3 |   c ← Clone(C)
4 |   b ← Sample(N, c)
5 |   A ← A ∪ {b}
6 return A
```

The sampling procedure takes as input a node $\mathcal{N}$ of the extended Kotlin CFG , a context $\mathcal{C}$, and a time budget s that determines how long to run for. The loop simply invokes the $\mathcal{N}$'s `sample` property repeatedly, generating code that is syntactically rooted in the target. Since this procedure mutates the provided context, the main loop creates a fresh clone of original context $\mathcal{C}$ to prevent cross-referencing between different files. An archive A tracks and collects each generated file, and is returned once the allotted time has exhausted.

As a result of the prevelance of RS in conjunction with a BB treatment of the compiler under test, many of today's compiler fuzzing tools utilize heuristics to shape the direction of stochastic sampling procedures rather than directly optimize solutions under concrete objectives. The remainder of this chapter explores auxiliary optimization settings, that seek to substitute the standard RS procedure with directly measurable objective functions, as an alternative paradigm to unguided sampling.

4.5.2 Syntactic Diversity-Driven Search

Generating code that covers a broad range of the Kotlin input space is a key challenge for thoroughly exercising the many features of the compiler. This subsection delves into two ways a GA can model, measure, and optimize for the diversity of its generated files, with the goal of triggering varied internal compiler behavior.

Modeling Syntactic Diversity

Diversity is intrinsically linked to the concept of (dis)similarity. To effectively build such a notion in the domain of semantically complex Kotlin code, the fuzzer requires a transformation capable of mapping Kotlin blocks to a space embedded with a notion of distance. For this purpose, we opted to measure the number of distinct language constructs that each block contains. To this end, the grammar sampling process actively tracks its stochastic traversal of the CFG graph such that each new occurrence of a node causes a dedicated counter to increment.

Once generation has concluded, the newly established block internally maintains a vectorized numerical representation of all its counters. This simplified representation implicitly describes a coordinate space that contains every possible valid piece of Kotlin code (though, clearly, this mapping is surjective as many pieces of code may be mapped to identical points). Furthermore, this syntactically-minded space largely overlooks the semantic nuances of the blocks that inhabit it, driving the genetic process almost entirely through the CFG. This is key for promoting structurally diverse blocks in the population.

Direct Single-Objective Diversity Optimization

We integrate the diversity model into the GA framework by means of a custom Single-Objective (SO) fitness function. Let b_1 and b_2 be two code blocks, generated using the procedure described in the previous paragraphs. Let $m : \mathcal{B} \to \mathbb{N}^k$ be the mapping that operates on the input space of all possible blocks $\mathcal{B}$ and outputs a k-dimensional vector of natural numbers representing, for each position, the number of language structures of a certain type present within the block. Using this mapping, any common measure of distance $d : \mathbb{N}^k \times \mathbb{N}^k \to \mathbb{R}$ (i.e., Euclidean) may be used to determine the distance between two blocks. Within this setting, we equate the notion of similarity the distance between two blocks. From these conventions, the notion of population-wide dissimilarity follows:

$$dis(b,P) = min_{b^{(i)} \in P - \{b\}} \{d(m(b), m(b^{(i)}))\} \tag{4.1}$$

Intuitively, the dissimilarity of a block b with regard to a population P is the minimum distance from b to a different block $b^{(i)}$ belonging to the population. This notion gives rise to a population-wide diversity fitness function:

$$\min_{b \in \mathcal{B}} f_{\texttt{DIV}}^{(SO)}(b,P) = \frac{1}{1 + dis(b,P)} \tag{4.2}$$

This formulation inverts and normalizes the similarity metric, meaning that $f_{\texttt{DIV}}^{(SO)}$ has reached an optimum when $dis(b,P)$ has reached its maximum. In other words, the higher

Algorithm 3: Single-Objective Diversity Genetic Algorithm

Input : Population size n, Node to sample $\mathcal{N}$, Context $\mathcal{C}$, Time Budget s
Output: Collection of generated Kotlin files

```
1  t ← 1
2  P_1 ← InitializeAndEvaluatePopulation(n, N, C)
3  P* ← P_1
4  while ¬ TimeElapsed(s) do
5      O_t ← CreateAndEvaluateOffspring(P_t)
6      P_{t+1} ← SelectIndividuals(P_t, O_t, f_DIV^(SO))
7      if Σ_{b∈P_{t+1}} f_DIV^(SO)(b, P_{t+1}) > Σ_{b∈P*} f_DIV^(SO)(b, P*) then
8          P* ← P_{t+1}
9      t ← t + 1
10 return P*
```

the minimum distance from a block to its population peers is, the better its fitness. While the notions of (dis)similarity and diversity allow for different computations, the one put forward in Equation (4.1) fits the purposes of compiler fuzzing especially well. Because the fitness of the individual is determined on the basis of its *most similar* counterpart, this evaluation method decreases the likelihood of preserving blocks that are very similar to other individuals. Other methods of determining similarity (such as, for example, the *mean* distance to the rest of the population) do not maintain this property as well, since blocks that map farther away in the mapping space might significantly improve the fitness of several blocks that are very similar to each other, but dissimilar from the rest of the population.

Algorithm 3 shows the GA incorporating this fitness criterion. The algorithm first initializes a population by sampling a provided node $\mathcal{N}$ (line 2). The variable P^* (line 3) tracks the most diverse population generated at any point during the algorithm's runtime, which in the beginning is the randomly sampled collection. After it has sampled the initial population, the algorithm proceeds by iterating through the standard GA loop. New offspring emerge as a result of the recombination and mutation operators (line 5), which then participate in the selection step (line 6). Once the new population has been established, the algorithm compares its diversity to that of the most diverse population encountered, and updates the P^* variable if appropriate (lines 7, 8). The algorithm finally returns the most diverse population encountered at the end of its runtime (line 10).

Indirect Many-Objective Diversity Optimization

Though it models the concept of inter-block syntactic similarity well, the SO optimization criterion suffers from one key limitation. $f_{\texttt{DIV}}^{(SO)}$ fails to faithfully depict the entire similarity landscape, implicitly requiring a trade-off in importance between similarity to close (or similar) and far away (or dissimilar) blocks. The higher the weight on the local topology, the more likely it is for the SO formulation to "miss" the global landscape. The converse is also

true, requiring the algorithm design to carefully strike a balance between the two conflicting objectives. Within the scope of compiler fuzzing, such a balance is very challenging to reason about, not least because the key to unlocking a balanced fitness criterion rests on many layers of abstractions and tools, all centered around the compiler.

To overcome this limitation, we propose a second method of obtaining a diverse set blocks, centered around a Many-Objective (MO) interpretation of diversity. Let $f_l(b) : \mathcal{B} \rightarrow \mathbb{N}, \forall l \in \mathbb{L}$ be a function that maps arbitrary blocks to a natural number equal to the number of occurrences of language construct l in the input block, with $\mathbb{L}$ the set of all valid language constructs of Kotlin. In addition, let $f_{sz}(b) : \mathcal{B} \rightarrow \mathbb{N}$ be a function that maps input blocks to their size. There are multiple sensible choices for the measurement of block size, but in this study we limit our measurements to a single metric that counts the number of characters in the text representation of the individual. Altogether, these functions give rise to a novel MO objective function:

$$\max_{b \in \mathcal{B}} f^{(MO)}_{\texttt{DIV}}(b) = \{-f_{sz}(b),\ f_{l_1}(b),\ f_{l_2}(b),\ \ldots,\ f_{l_n}(b) |\ l_i \in \mathbb{L}\} \tag{4.3}$$

Within this formulation, $f^{(MO)}_{\texttt{DIV}}$ attempts to simultaneously *maximize* the number of each language feature and *minimize* the size of each block. In essence, this gives rise to a Pareto frontier of blocks that have a high number of different combinations sets of language features within a limited size. The size minimization is essential to this interpretation of diversity. Omitting the size objective causes $f^{(MO)}_{\texttt{DIV}}$ to simply maximize the number of each language feature independently. In practice, this is a poor decision for two key reasons.

First, the nature of language feature objectives is strictly linear and additive. Since larger blocks contain more code, they also contain, by extension, more language features on average. This causes larger blocks to be (on average) much more likely to Pareto dominate smaller blocks and take over the population very quickly. This setting would heavily favor larger blocks, entirely missing the enveloping goal of optimization. Second, exploring a search space composed of smaller blocks may uncover more valuable compiler defects. Smaller blocks not only make it easier to pinpoint the cause of the uncovered faults, but are also more likely to lead to more impactful faults: the smaller the code pertaining to a compiler bug, the more generalizable and likely it is to appear in real-world projects.

Algorithm 4 contains the pseudocode for the MO optimization algorithm based on $f^{(MO)}_{\texttt{DIV}}$. It is a straight-forward extensions of SO GA that utilizes an archive (lines 1, 5) to track the set of non-dominated solutions that together comprise the approximation set. With each iteration, the archive could both grow (by inserting new non-dominated blocks) and shrink (by discarding older individuals dominated by newer additions). The algorithm returns this approximation set as its final output (line 9).

Differences in Diversity Nuance

Grasping implications of the different diversity formulations may prove tricky. To help build intuition regarding how the selection criteria and their respective formulations impact the landscape of the search process, we provide an example in Figure 4.9. Subfigure (a) shows a population of 30 points, the features of which were each drawn from categorical uniform

Algorithm 4: Many-Objective Diversity Genetic Algorithm

Input : Population size n, Node to sample $\mathcal{N}$, Context $\mathcal{C}$, Time Budget s
Output: Collection of generated Kotlin files

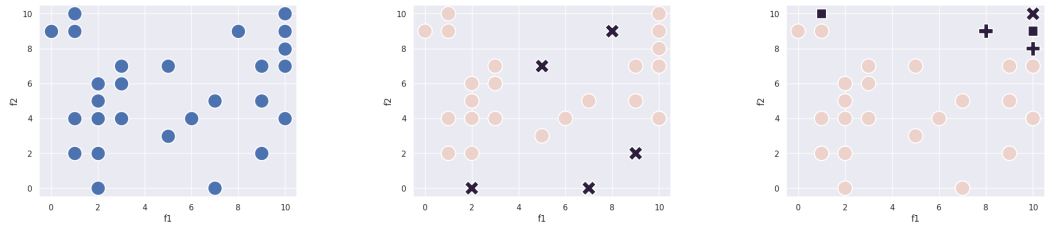
$A \leftarrow$ `InitializeElitistArchive(`$f_{DIV}^{(MO)}$`)`
$t \leftarrow 1$
$P_1 \leftarrow$ `InitializeAndEvaluatePopulation(`$n, \mathcal{N}, \mathcal{C}$`)`
while $\neg$ `TimeElapsed(`s_t`)` **do**
    $A \leftarrow$ `ProcessNewArchiveEntries(`$P_t, f_{DIV}^{(MO)}$`)`
    $O_t \leftarrow$ `CreateAndEvaluateOffspring(`P_t`)`
    $P_{t+1} \leftarrow$ `DominationSelection(`$P_t, O_t, f_{DIV}^{(MO)}$`)`
    $t \leftarrow t+1$
return A

distributions. In practice, the axes represent the number of language features (statements, functions, etc.) each individual (block) exhibits. Within this space, the choice of modeling heuristic drastically affects which blocks get selected to the next round of the algorithm.

Consider a scenario in which $f_{\texttt{DIV}}^{(SO)}$ is the fitness criterion, and the algorithm uses a simple truncation selection method to determine the survivors to the next generation. In this case, the selection operator favors blocks that are the farthest away from any other blocks in the population, as subfigure (b) depicts. The points depicted by black crosses are the ones who survive the selection round. While this mechanism does well to identify portions of the search space which the population sparsely inhabits, it is prone to "forgetting" areas of high density that do not contribute to the selection. This issue resembles the problem of *front degradation* in traditional MO evolutionary computation. The populaiton-wide retention mechanism of the SO diversity GA seeks to combat this.

Alternatively, consider the MO setting, in which the algorithm carries out selection by means of truncation based on the domination rank of the individual. This combination of operators results in the selection depicted in subfigure (c). Points depicted in black are selected, and the shape of the point depicts its domination rank. Within this pure maximization framework, the problem makes itself apparent: points depicting blocks that are larger drastically diminish the probability of smaller blocks (mapped to the bottom-left quartile of the figure) propagating to the next round of the algorithm. It is for this reason that $f_{\texttt{DIV}}^{(MO)}$ contains a size component that aims to circumvent this obstacle and allow for a more even distribution of individuals.

Notably, only one selection overlaps between the two selection mechanisms proposed here. Within the scope of compiler testing, it is extremely difficult to analytically reason about which formulation is better suited for uncovering defects. For this reason, we set out to empirically analyze both formulations within this study.



(a) Initial distribution of code blocks in the syntactic similarity space.

(b) Truncated selection of individuals based on $f_{\mathrm{DIV}}^{(SO)}$.

(c) Selection of individuals using domination rank and $f_{\mathrm{DIV}}^{(MO)}$ without the size component.

Figure 4.9: Illustration of different optimization criteria behavior in the diversity space.

4.5.3 Semantic Proximity-Driven Search

Syntactic analysis alone cannot paint a sufficiently detailed picture for comparing blocks. While effective as a means of measuring the structural composition of blocks, the syntactic diversity criteria introduced in the previous section fail capture the semantic nuances of generated code. In this subsection, we introduce a slew of techniques culminating in three complementary GA formulations that seek to better account for the indispensable semantic dimension of Kotlin code.

Modeling Semantic Proximity

The mathematical requirements to effectively measure the semantic similarity between two blocks are the same as those that syntactic diversity requires: a sound mapping from the domain of code, and a measure of distance in the mapped space. However, the multifaceted nature of Kotlin semantics present difficulties that syntactic models do not face. The ANTLR4 CFG implementation that the Kotlin developer team supplies as part of the language specification condenses the rules of the language in a structured and processable manner. This reduces the complexity of interpreting the syntactic similarity of two blocks down to assessing their relation with respect to the grammar. Unfortunately, no such construct exists for the semantic counterpart of Kotlin.

To address this problem, we turn to techniques used in the field of Machine Learning for Software Engineering (ML4SE). Ever since Hindle et al. [40] shared their findings on the high regularity of code in 2016, the field of ML4SE has experienced a boom in popularity and development. Crucially, the study suggested that software regularity emerging from a property the authors call *naturalness*, as opposed to syntax. Since then, researchers adapted, tuned, and extended techniques at the intersection of Machine Learning (ML) and Natural Language Processing (NLP) and successfully applied them at scale for common software engineering tasks. Non-trivial chores previously exclusively reserved for developers could now be at least in part automated at scale through new ML4SE techniques.

Of particular interest to our research is a class of models that use an internal numeric representation of code to simultaneously model both the syntax and the semantics of the input. These models largely rely on architectures revolving around a Neural Network (NN)

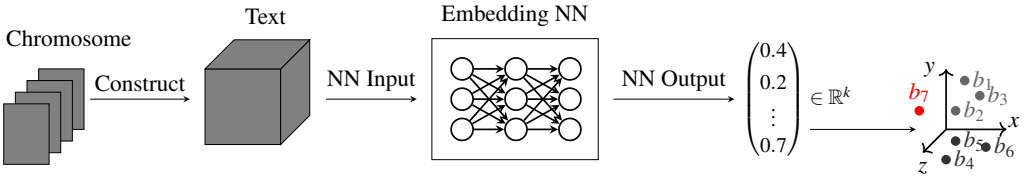


Figure 4.10: Visualization of the code embedding process.

trained on one or more tasks, generally involving both code and natural language. In NLP research, the (lower-dimensional) numeric representation of token-represented input is referred to as an *embedding*.

Embeddings exhibit two key properties that make them suitable candidates for the goals of our optimization framework. Though impossible to interpret in isolation, embeddings retain an interpretation of the textual representation fed into them. We specifically select embedding mechanisms that have been trained in an end-to-end fashion using multimodal training criteria, in the hope of taking advantage of the generalizability of such approaches. Second, by populating a space with the embeddings of Kotlin code, the notion of *proximity* can simply be expressed by means of a distance metric in the projected domain.

Section 4.5.3 contains an illustration of the embedding pipeline. To compute the proximity of two chromosomes during search, we first convert the ordered snippet list into the Kotlin code, without retaining any additional information. We then feed the code as input to the embedding NN of choice, and retrieve the real-valued vector it computes as output. By caching previously computed blocks, the topology of a projected space emerges, and the similarity of two blocks is measured in terms of their proximity (distance). The points depicted in the coordinate system on the right of Section 4.5.3 are color-coded to highlight their proximity within the space. Blocks corresponding to points b_1, b_2, and b_3 all are in close proximity to one another, as are the blocks pertaining to b_4, b_5, and b_6. b_7 is farther apart, denoting its dissimilarity to either cluster.

Selecting Proximity Targets

To completely integrate the notion of semantic proximity in our GA, we require a notion of individual fitness. The notion of diversity proposed in the previous section is a candidate for this purpose, however, it suffers from two notable drawbacks. First, embedding NNs operate on the entire textual representation of the code, which means that (unlike our syntactic formulation) information such as randomly generated identifier names influences the embedding of the chromosome. In practice, this means that input is likely to be noisy, artificially affecting diversity estimates. Second, embeddings of current code models tend to have a relatively high number of dimensions, and relatively small numerical values at each position. This does not bode well with our population-adaptive diversity heuristic, which

attempts to simultaneously consider all dimensions in a scalarization of the distance metric.

As an alternative, we propose a heuristic which seeks to *maximize the proximity* between the population of a GA and a *static* collection of *target* pieces of Kotlin code. Intuitively, this approach seeks to generate code that is *as close as possible* to a pre-defined set of targets. Naturally, we must ensure that a suitable collection of code serve as targets during optimization. For this purpose, we turn our attention to a collection of test inputs that contributors to the Kotlin put together with the goal of verifying a broad range of compiler and language features. These inputs are publicly available as part of the Kotlin repository [1].

The proximity optimization concept rests upon two core assumptions. The first assumption is that the set of targets provided to the algorithm consists of "*interesting*" test cases for the compiler. By interesting, we refer both to the relevancy of the test cases to the broader scope of the language, as well as the behavior they cause during compilation. Since the data set we use by default has been curated by compiler developers over a long period of time, we have no reason to doubt the validity of this assumption. Second, we assume that within the embedding space, projections that are close to *interesting* test cases are likely to be, themselves, *interesting*. On the surface, this is a reasonable proposition: pieces of code that are similar to hand-crafted compiler test cases may well trigger compiler edge case behavior that the static test misses. However, the practical implications of this assumption heavily depend on the choice of similarity metric.

To ensure that the mapping to the embedding space appropriately shapes the output domain, we must carefully choose a model that has shown good performance on multiple code-centric tasks. To our knowledge, there are currently no available open-source models that have either been pre-trained or fine-tuned for Kotlin-specific tasks and thoroughly evaluated in an empirical study. With this in mind, we select CODEBERT [25] as the default option for the remainder of this study. CODEBERT is a Transformer-based [87] model that has been pre-trained on a large corpus of open-source repositories containing several languages. Though Kotlin is not part of CODEBERT training set, its closest relative, Java, is. Furthermore, the architects of the CODEXGLUE [57] , a popular ML for code-related tasks dataset, have selected CODEBERT as the default benchmark to compare against in several categories, attesting to its generalizability.

Finally, to better assess the performance of the algorithm, some degree of interpretability is required. The data set of tests provided in the Kotlin repository is vast, encompassing over 19,000 files organized in almost 2,000 directories. To parse the result of our fuzzers with respect to this extensive dataset, we reduced the number of selected targets by means of clustering. We empirically assessed several clustering algorithms and provide sensible default options based on how well the clusters reflect the original directory structure.

Single-Target Proximity Optimization

To tie our proximity formulation together, we first propose an algorithm that iteratively attempts to minimize the distance to each of the targets in the data set. This follows the paradigm that Tonella [83] established in one of the earliest successful applications of GAs

[1] https://github.com/JetBrains/kotlin/tree/master/compiler/testData

Algorithm 5: Single-Target Proximity Genetic Algorithm

Input : Target Set $\mathcal{T}$, Population size n, Node to sample $\mathcal{N}$, Context $\mathcal{C}$, Time Budget s, Target Time Budget s_t
Output: Collection of generated Kotlin files

```
1  Shuffle(T)
2  i ← 1
3  while ¬ TimeElapsed(s) do
4      T ← T_i
5      t ← 1
6      P_1 ← InitializeAndEvaluatePopulation(n, N, C)
7      while ¬ TimeElapsed(s_t) do
8          ChooseBestForEachTarget(P_t, T)
9          O_t ← CreateAndEvaluateOffspring(P_t)
10         P_{t+1} ← SelectIndividuals(P_t, O_t, f_PRO^(SO))
11         t ← t + 1
12     i ← i + 1
13 return BestBlockForEachTarget(T)
```

to the task of automated unit test generation. The function to optimize is given by a particular selected target, and measures the distance from the individual to that target. Equation (4.4) gives the definition of this SO objective, with b the block to a evaluate, t_i a target that the algorithm is attempting to reach, d a measure of distance, and e an embedding function as described in the previous subsection. Algorithm 5 illustrates the pseudocode of the entire procedure.

$$\min_{b \in \mathcal{B}} f_{\text{PRO}}^{(SO)}(b, t_i) = d(e(b), t_i) \tag{4.4}$$

The algorithm is structured in two nested loops. Before iterations start, the algorithm first shuffles the collection of targets. The outer loop then selects a target (line 4) from the randomized ordering, before attempting to optimize a random population towards that target, in a nested loop that applies a GA (lines 5-11). In addition to running the nested GA, each inner iteration also considers whether the current population has contains better improvements with regard to *any* target in the data set, not just the one being optimized. This information does not affect selection - the scope of this additional step (line 8) is to avoid discarding individuals that would have been closest to *some* target. The algorithm returns a set of blocks that are closest to at least one target.

Many-Objective Proximity Optimization

To complement the iterative SO formulation, we also propose an MO alternative that attempts to simultaneously minimize the distance to each target in the data set. Compared to the previous alternative, the MO option could provide a more comprehensive overview of

the search landscape by also selecting blocks which are not necessarily *closest* to any one target, but relatively close to multiple targets at the same time. The MO fitness function underlying the optimizaiton method is given by Equation (4.5).

$$\min_{b \in \mathcal{B}} f^{(MO)}_{\texttt{PRO}}(b, \mathcal{T}) = \{d(e(b), t_i) |\, t_i \in \mathcal{T}\} \tag{4.5}$$

The pseudocode of the algorithm is analogous to that of Algorithm 4, except selection and archival are carried out based on $f^{(MO)}_{\texttt{PRO}}$ rather than its diversity counterpart. Of note is that the number of individual objectives is generally much higher for the proximity formulation. This is because the number of individual language that the fuzzer supports is relatively limited, whereas the number of available test cases, even when reduced by clustering, is much higher. This warrants the use of standard techniques for bounding the elitist archive, such as discretization or pruning. Alternatively, a user could also configure the GA to use a lower population size.

Scalarization-Based Proximity Optimization

The final proximity-centered formulation we propose finds its roots in the Whole-Suite (WS) approach put forward by Fraser and Arcuri [28]. This approach follows the common GA framework explored in this thesis, with one key exception. Individuals in the WS approach are *test suites*, which constitute unordered collections of lower-level test individuals. In our case, each suite S is a group of blocks. To evaluate a suite, WS performs a technique called *scalarization* [20], which translates the MO problem down to an SO counterpart. Equation (4.6) encapsulates the fitness this approach is based on, with each suite S having a fitness based on the sum of smallest distances between the blocks in S and each target t.

$$\min_{b \in \mathcal{B}} f^{(SO)}_{\texttt{WS}}(S, \mathcal{T}) = \sum_{t \in \mathcal{T}} \min_{b^{(i)} \in S} d(e(b^{(i)}), t) \tag{4.6}$$

We incorporate $f^{(SO)}_{\texttt{WS}}(S, \mathcal{T})$ in a GA nearly identical to Algorithm 1. However, unlike all previous formulations, WS performs variation and selection at the suite-level, rather than on individual blocks. Mutation operators include addition, removal, and swapping of individual blocks, while recombination simply swaps blocks between two separate test suites. Finally, we note that WS has several known limitations, including that it can fail to retain individually valuable blocks. This shortcoming emerges as a consequence of the scope of the variation and selection operators, neither of which attempts to retain individual components of the suite. We note that newer approaches like DYNAMOSA have demonstrated increased performance relative to WS. However, the simplicity of the WS approach makes it a better candidate for studying and interpreting the effects of operating at a larger level than blocks.

Chapter 5

KOTFUZZ

This chapter details the implementation of the conceptual methods laid out in Chapter 4 in a system we named KOTFUZZ. We first describe the scope of the system and its goals, before analyzing its architecture and components. Finally, we examine the many ways in which practitioners can configure the system based on their individual intended use cases.

5.1 Implementation Overview

We designed KOTFUZZ to be a self-contained system providing tools that (i) enable the generation of Kotlin code, (ii) allow the differential testing of specific compiler versions, and (iii) perform analysis on the performance and behavior of its algorithms. We make the source code of the tool available through a public GitHub repository [1]. The implementation spans over 10,000 lines of code across five programming languages and more than a dozen libraries. In addition to providing a versatile toolkit, we aim for KOTFUZZ to fulfill two other key qualities: customizability and reproducibility.

First, KOTFUZZ should be modular enough to allow users to swap different components according to their needs. Second, it should allow for a large degree of reproducibility and experimentation, as to enable further empirical research into automated testing of Kotlin compiler, while at the same time allowing users to reproduce results that might be relevant to the future of the language. Fortunately, containerization is a technique that excels at both distributing software in isolation and building a controlled environment in which strict version requirements can be met. These properties significantly lessen the burden of modularity and reproducibility. As a result, the entire KOTFUZZ system makes heavy use of containerization, both to deliver software and to empirically assess it.

5.2 Architecture of the System

The ambitious and varied tasks that KOTFUZZ sets out to accomplish require a versatile architecture that encourages modularity and extendability. For this reason, we design KOTFUZZ to follow a Service-Oriented Architecture (SOA). The service-centric model allows

[1] https://github.com/ciselab/kotlin-compiler-fuzzer

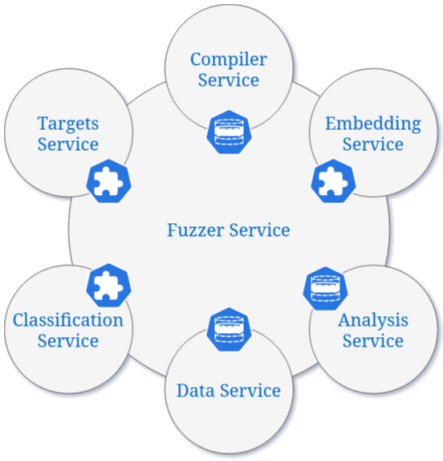


Figure 5.1: Service-Oriented Architecture of KOTFUZZ.

for the separation of individual tasks into self-contained components, called *services*, that can interface with each other through APIs. This is especially desirable for our system, as separating between fuzzing, done in Java and Kotlin, embedding, performed by a Python wrapper over CODEBERT, and analytics software, written in Bash, Python, and R would present a sizable challenge without this separation.

The organization of services within KOTFUZZ is not symmetric, instead revolving around the main fuzzer service, which implements the theoretical methods proposed in the previous chapter. In support of this main component are six additional services, which the remainder of this section describes more closely. Figure 5.1 depicts the overall architecture, with the six complementary services orbiting around the fuzzer. Services communicate either through REST APIs (denoted by puzzle pieces), or by shared volumes, depending on the use case. All services belong to otherwise isolated containers.

Fuzzer Service

The fuzzer service is tasked with running the generative fuzzer, and contains the implementation of both the syntactic and semantic interfaces, as well as the generative heuristics proposed in Chapter 4. This implementations spans circa 8,800 lines of code split between Java and Kotlin. Java is the main language of the fuzzer, which we chose due to previous experience, as well as interoperability with Kotlin, which enables the usage of several libraries developed as part of the Kotlin specification. One such library is `grammar-tools`[2], a collection of tools that can parse and interface between ANTLR4 and an intermediate parse tree representation. This is crucial for the context extraction step described in Section 4.2.

The fuzzer service is internally structured in alignment with the sections of Chapter 4. Figure 5.2 provides a UML visualization of the component-level composition of this service. The center runner block is the user-facing interface of the service, which processes command line arguments provided by users and configures the appropriate internal state.

[2]https://github.com/Kotlin/grammar-tools

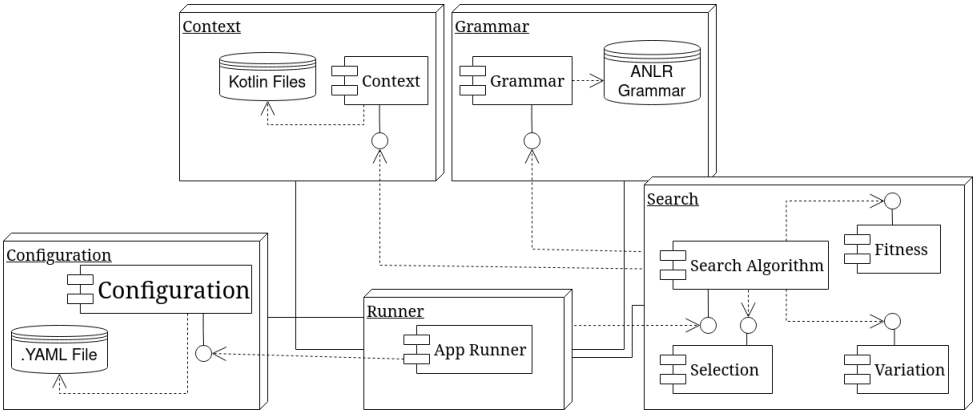


Figure 5.2: Component diagram of the KOTFUZZ fuzzer service.

Three additional components implement the context, grammar, and generative heuristic modules of the fuzzer. A final configuration component is responsible for allowing users to fine-tune the behavior of the fuzzer according to their goals. The remainder of this subsection briefly details the most important aspects of each module.

The context component is tasked with extracting relevant structural information from a provided collection of Kotlin files and providing a semantic interface to the fuzzer at runtime. The default context uses a subset of Kotlin built-in and standard library types, including numerical and string types, exceptions and throwables, and container type interfaces. Though limited, this subset suffices to grasp the effectiveness of the fuzzer prototype and establish the properties of the proposed heuristics. The fuzzer is internally able to parse a much broader and generalizable subset of the Kotlin language, including (nested) classifier types and their members, functions, annotations, and visibility and inheritance modifiers. However, due to the relatively costly nature of implementing sound semantic constraints for all constructs of the Kotlin language, sampling within this research prototype is limited to functions, expressions, and statements.

By default, we use the standard specification of ANTLR4 lexer and parser grammars[3]. The grammar component of the fuzzer traverses the graph representation of the CFG and transforms each node into its extended counterpart as established in Section 4.3. We envision that users can provide a target node for sampling from this transformed CFG, however, for the scope of this research prototype, we restrict the samplable nodes to functions in the top-level scope of a file. We selected this option because functions are powerful constructs in Kotlin, stressing many components of language such as name and scope resolution, type constraints, and parameter matching. Functions also behave as dictated by the grammar, and can contain an arbitrary number of expressions and statements in their bodies.

The search component of the fuzzer service implements both the genetic representation of code snippets, as well as the generative heuristics built on top, as per Section 4.4 and Section 4.5. For the latter, standard interfaces model common elements shared throughout algorithms, including SO and MO fitness functions, selection and variation operators, and

[3]https://kotlinlang.org/docs/reference/grammar.html

block generation. This enables a large degree of configurability and extendability, as common procedures likely to be used in most algorithms are already implemented in basic GA classes. Implementing novel algorithms that conform to a somewhat standard GA structure is straightforward within this framework: in most cases a developer simply needs to implement a novel `FitnessFunction` class, choose one of the five provided selection operators, and optionally implement a novel iteration loop.

One notable extension of this framework is the inclusion of a `RemoteEvaluation` fitness function interface. This interface extends its base (local) counterpart, but allows the fuzzer to determine the fitness of individuals through HTTP requests to remote services. Such methods greatly increase the flexibility of the tool, allowing remote applications to implement tailored evaluation procedures that would otherwise prove challenging to integrate in the main service. The embedding-based proximity optimization algorithm discussed in Section 4.5.3 is an example of such a use case.

Finally, a configuration component parses and interprets a user-provided file that governs the behavior of nearly every aspect of the fuzzer. We delve into the details of configuring KOTFUZZ in Section 5.3.

Data Service

The data service is an auxiliary component of KOTFUZZ that hosts a large amount of data pertaining to adjacent services. This data includes the compiler versions utilized in the compiler and analytics services and the pre-processed Kolin files that serve as targets for proximity optimization. This component is part of the Data Packed Volume Container (DPVC) design pattern that several services adhere to, thus increasing the amount of data cached and shared between services, without requiring multiple processing pipelines.

Embedding Service

The embedding service is at the core of the proximity heuristics formulated in Section 4.5.3. This service contains a custom object that wraps the CODEBERT model and allows the embedding of arbitrary pieces of text. The text is first split and tokenized, before the model performs a forward pass on the tokens, retaining their only embeddings. To further allow the user to customize KOTFUZZ, we do not limit the model selection to just CODEBERT. Instead, we allow the user to inject models of their choice by mounting different volumes in the embedding container. By default, the container running this service shares volumes with the data service, where the default model resides.

The main fuzzer service makes use of embeddings by utilizing a REST API implemented through `FastAPI` [4], a popular Python library for building lightweight and performance-oriented APIs. The API provides two endpoints: one for embedding a single file, and one for returning an annotated list of files. The latter is used more frequently within proximity GAs, as it requires less traffic between services and reduces overhead. To avoid costly re-evaluations of blocks, all algorithms that rely on remote evaluations maintain a local cache of previously considered individuals, further reducing connectivity traffic.

[4] https://fastapi.tiangolo.com/

Targets Service

Like its embedding counterpart, the target service supplies critical information to the proximity-based GAs. This service performs two key tasks. First, it uses the embedding service to compute numerical representations of all test programs available in the Kotlin repository. Second, it uses a custom object to wrap standard clustering algorithms that help reduce the shear number of targets. The latter step operates in the embedding space, and uses standard clustering algorithms implemented in the `skelarn` [5] Python library to construct a cluster structure around the projected points.

Once established, the cluster structure provides either an implicit or explicit list of *centroids*. Some algorithms, like *k-means* do so directly, while others, like `BIRCH` [92] and `OPTICS` [3] require an additional step to infer the underlying centroids based on the data. In the latter case, we first run all projected points through the inferred cluster structure and assign each point to a cluster (unless the algorithm rejects it). Then, we compute the centroids of each cluster as the mean of all points assigned to it. Whether implicit or explicit, the emerging centroids serve as targets for the proximity optimization GA.

Also similar to the embedding service is the targets services's configurability. By default, we provide six clustering objects that users of KOTFUZZ can select from. We selected these objects based on (i) the scalability of the underlying `sklearn` algorithm and its ability to handle the vast amount of test data, (ii) the centroid's retention of the original directory structure within the Kotlin repository, and (iii) the number of centroids. Most notable is the second property, which serves as a heuristic evaluation criterion. We score each model based on (i) the percentage of files that belong to the same directory in the Kotlin repository and also belong to the same cluster in the model and (ii) the percentage of files in different directories, also assigned to different clusters.

Through this evaluation, we opt for several k-means models, which offer a good trade-off between mirroring the original directory structure and allowing the selection of an intended number of targets (centroids). However, we also provide several additional models as default options within this service. Unlike the embedding service, there is no additional implementation requirement to extending the targets functionality. Users can simply inject pre-trained clustering objects in the appropriate directory, and the runtime `FastAPI` application handles the communication with the fuzzer service. Finally, we employ the Data Version Control (DVC) [6] system to allow practitioners to orderly modify, update, and maintain different models.

Compiler and Analysis Services

We address the compiler and analysis services together because of their similar roles within KOTFUZZ. Both services add auxiliary functionality to the fuzzer system and aid in its empirical assessment. The compiler service is the lighter of the two, and its role is to perform DT on the output of the fuzzer. To speed up analysis, we allow for this service to run asynchronously to the main service, by sharing a volume between the two containers.

[5] https://scikit-learn.org/stable/modules/clustering.html
[6] https://dvc.org/doc

In practice, this means that the compiler service passes newly generated files through the DT pipeline as soon as they emerge from the fuzzer. This feature is desirable for extending the fuzzer and continuously assessing its bug-finding capabilities.

The analysis service provides a more complete set of analytics tools than its light-weight counterpart, and allows for insightful analysis of KOTFUZZ's performance. This service also performs DT, however, it simultaneously monitors the compiler process through the GNU `time` command line utility [7], which captures additional details regarding speed and memory usage. In addition, each file analyzed through this service is additionally inspected through `detekt` [8], a static analysis tool for Kotlin code. Both of these features aid in detecting patterns in the generated code, and can help link compiler defects (or performance limitations) to properties of source code.

Finally, the analysis service also contains utilities that extract statistics regarding the performance of the fuzzer. These utilities include the classification of defects uncovered in a particular fuzzer run, as well as statistical tests that determine the relative performance of more than one runs. Because of the more performance-intensive nature of this service, we advise against utilizing it at the same time as the fuzzer, as it can limit compute resources. For this use case, the compiler service is a better fit.

Classification Service

KOTFUZZ is intended to uncover defects in the Kotlin compiler. However, our empirical analyses and fuzzing campaigns revealed that the distribution of defects KOTFUZZ unocovers in a typical run is significantly skewed. In particular, we found that there exist certain kinds of files that trigger recurring problems in the compiler. Once KOTFUZZ has uncovered a sufficient number of bugs of a certain type, it is preferable to avoid generating files that trigger the same defects in the future, because (i) these files reveal no new information about defects in the compiler, and (ii) they consume computational resources during both search and analysis, that could be better used in the generation of new files.

It is the function of the classification service to detect such occurrences during search. The simplest way to do so is to compile intermediate populations during search, cache the compiler output in a temporary file, and analyze the file to detect patterns linked to known defects. However, this method defeats the purpose of the classification service altogether, as it requires *more* computational resources to be expended by compiling even more code. To circumvent this limitation, we implemented a ML classification algorithm for the most common error that KOTFUZZ generated code triggers: out of memory errors. Chapter 7 provides deeper analysis into these errors.

Similar its counterparts, we designed the classification service with modularity and versatility in mind. A REST API provides access to an endpoint that consumes the text pretaining to an individual during search and returns a boolean value representing the model's prediction (whether the text would trigger an out of memory error in the compiler). Custom objects can be mounted to a dedicated directory in the running container to allow for further user customization. By default, we provide two classifiers based on Decision Trees and

[7] https://www.gnu.org/software/time/

[8] https://detekt.dev/

Argument	Type	Effect	Required
`configFile`	Path	Path to the configuraition file.	✔
`lexerFile`	Path	Path to the ANTLR4 *lexer* grammar specification.	✔
`grammarFile`	Path	Path to the ANTLR4 *parser* grammar specificaiton.	✔
`compilerPath`	Path	Path to the compiler to be used during fuzzing.	✘
`classPath`	Path	Path to the directory containing context files.	✔
`ctxSeed`	Int	Seed to use for stochastic-driven processes.	✔
`searchSeedSeed`			✔
`selectionSeed`			✔
`recombinationSeed`			✔
`ctxSeed`			✔
`time`	Int^+	Time budget (ms) to run the fuzzer for.	✔
`takeSnapshots`	Bool	Whether the fuzzer takes intermediate snapshots.	✔
`snapshotInterval`	Int^+	Time (ms) between snapshot intervals.	✘
`output`	Path	Time budget (ms) to run the fuzzer for.	✔

Table 5.1: Overview of command line argument options.

`Adaboost` [29], respectively. Both models operate on the embedding numerical representation of files and were trained on circa 8,000 data points of generated code. Empirical k-fold cross validation revealed an accuracy of over 0.99 for both models.

5.3 Configuring the System

To navigate the full breadth of features proposed in Chapter 4, while simultaneously masking the complexity of their implementation, we implemented a comprehensive configuration system that allows users of KOTFUZZ to seamlessly compose the desired behavior of the tool from many individual components.

The configuration system incorporates two modules. The first and simple of the two consists of command line arguments that users provide when executing the main fuzzer service. Table 5.1 contains a description of all available command line arguments, which are divided between three broad categories. The first (lines 2-6 of the table) specify the resources required to configure the heuristic, context, and extended grammar components. The second category (lines 7-12) specify the seeds that each random-driven process in the fuzzer should use. The final group of command line arguments dictate the time budget and output directory of the fuzzer, further allowing for the GAs to take intermediate snapshots of their populations during runtime.

This simple system suffices for specifying the runtime behavior of the system but its capabilities fall short of those required to fully customize the system. Providing fine-grained information pertaining to the many heuristic options available in KOTFUZZ through the command line is both cumbersome and error prone. To mitigate this, we enable users to configure the syntactic, semantic, and heuristic attributes of KOTFUZZ through a static con-

figuration file, which they can link to main fuzzer service through a command line option. Figure 5.3 provides a truncated version of a configuraiton specification. The specification follows the popular `yaml` [9] serializaiton language, which offers a clear and readable layout.

The configuration itself is split between two large sections. The first (lines 1-26) relates to the algorithmic engine of the fuzzer service. In sections, users can define the class of search algorithm they prefer to use (line 2), as well as finer details regarding its variation and selection operators. Lines 3, 4, and 6 control the size of the population for GAs, while lines 8-17 dictate the behavior of genetic operators. Here, users can select the selection operator they favor for both the SO (tournament, truncation, proportionate) and MO (domination rank, domination count). Further SO selection mechanisms can be configured for the `WS` scalarization approach (lines 15-17). Moreover, this section covers the integration of remote services to the main fuzzer component. The `remote` section (lines 18-26) is where URLs to the endpoints for the targets, embedding, and classification services are specified.

The second section of the configuration (lines 27-51) contains information regarding the probability distribution that governs the CFG interpretation of the fuzzer. This is highly valuable for scenarios in which practitioners are interested in stressing a particular subcomponent of Kotlin. The first and most impactful parameter of this section is the *simplicity bias* (line 28). The simplicity bias of the fuzzer is the probability with which a *simple* expression is sampled over any alternative in the CFG. Simple expressions are instances of callables (Section 4.2), and include function calls, constructor calls, and directly assignable values. The simplicity bias of a fuzzer run directly determines how complex and large generated blocks are, by allowing tuning the intricacy of generated expressions.

Further options in the latter part of the configuration center on tuning the distributions of both syntactic (lines 29-32) and semantic (lines 33-41) constructs. Distributions can be either uniform or geometric, and allow the user to favor or limit the extent with which language features manifest. The level of detail is granular enough to allow users to independently control the number of occurrences of language features (like `catch` blocks in lines 36-38) as well as the number of statemts within these instances (lines 39-41). The same system applies to larger-scope language features, like entire expression categories (lines 42-51). Users can provide weights to expressions and statements, which determine their relative frequency within generated code, and even disable whole categories altogether.

Parsing, and tuning the extensive options of the configuration system may prove challenging to users of KOTFUZZ. To alleviate the issue of understanding the 100+ field configuration, we provide exhaustive documentation explaining all possible values for each category of settings, their effects on the system. Sensible defaults are also available in the source repository. Finally, the configuration component of the fuzzer service alerts the user at runtime whether fields of the configuration are missing or otherwise misconfigured. Expected values are also printed out in common cases of erronous specifications.

[9] https://yaml.org/

```yaml
heuristic:
  type: proximity-ga
  population-size: 10
  new-blocks-generated: 2
  distance-metric: euclidean
  blocks-per-suite: 20
  num-iters-per-target: 10
  suite-mutation-probability: 0.1
  selection:
    individual: # Selection for GAs where individuals are blocks
      mo-selection: domination-rank # For MO algorithms
      so-selection : tournament     # For SO algorithms
      tournament-size: 4
      maximum-length: 500           # Additional selection filters
    suite:      # Selection for GAs where individuals are suites
      so-selection: tournament
      tournament-size: 4
  remote:
    embedding-single: http://localhost:9090/embedding-single/
    embedding-multi: http://localhost:9090/embedding-multiple/
    targets: http://localhost:9091/centers/kmeans100/
    num-targets: 100
    oom:    # Out-of-memory detection model
      enable: true
      oom-single: http://localhost:9092/predict-single/
      oom-multi: http://localhost:9092/predict-multiple/
grammar:
  simplicity-bias: 0.6 # Likelihood of sampling a simple expression
  plus-node-dist:
    type: uniform # Unifrom categorical distribution between 1 and 5
    lower-bound: 1
    upper-bound: 5
  try-block-dist:
    type: geometric # Unbounded geometric distribution
    lower-bound: 0
  catch-block-dist: # The number of catch blocks to generate
    type: geometric
    lower-bound: 0
  catch-block-stmt-dist: # Number of statements within each block
    type: geometric
    lower-bound: 0
  language-features:
  functions:
    enable: true
  expressions:
    simple-expr:
      enable: true
      weight: 1.0 # Relative weight to all other expressions
    if-expr:
      enable: true
      weight: 19.0 # Relative weight to all other expressions
```

Figure 5.3: Sample configuration subset.

```
docker run --name kotfuzz-runner -it --rm\
-v $(pwd)/output:/output\
-v $(pwd)/resources:/resources:ro\
-v $(pwd)/configs:/configs:ro kotfuzz-fuzzer\
-DlexerFile="/resources/antlr/KotlinLexer.g4"\
-DgrammarFile="/resources/antlr/KotlinParser.g4"\
-DcompilerPath="/resources/kotlinc"\
-DclassPath="/resources/kotlin/"\
-DctxSeed=0 -DsearchSeed=0 -DselectionSeed=0 -DmutationSeed=0\
-DrecombinationSeed=0 -Dtime=60000\
-DtakeSnapshots="false" -DsnapshotInterval=20000\
-Doutput="/output" -DconfigFile="/configs/rs/cfg-0.6.yaml"\
-jar fuzzer.jar
```

Figure 5.4: Sample command to run the fuzzer.

5.4 Running the System

The highly modular nature of KotFuzz is an important factor when running the tool. A dedicated directory within the repository structure contains all image specifications in the `Dockerfile` format. A single setup script is available, that prepares all images locally in a dependency-senstive manner. Once built, users can first run the auxiliary services, before running the main fuzzer container with an appropriate configuration.

However, this option might not suffice for users interested in extending or tracing the execution of the system. For this purpose, the fuzzer service can also be run locally in two ways. First, the `Gradle` build tool [10] manages the dependencies of the system. We provide the local `Gradle` configuration with an additional *task* called `uberJar`, which compiles the entire fuzzer service, together with all of its dependencies, down to a single `jar` file. Users can experiment with this implementation of the system in isolation from the other containerized services. Finally, an additional entry point exists within the class structure, that allows users to run the application through their Java Integrated Development Environment (IDE) of choice, without providing additional command line arguments. Default options are instead tailored to the structure of the repository.

Whichever method a user chooses (container or `jar`), the final running command is a simple listing of all command line arguments described in Table 5.1 through the `java` command line utility. When running in a containerized environment, additional volume mounts specifying the output, resources, and configuration directories should be made available. Figure 5.4 contains an example of running the container version of the fuzzer service, the entry point of which is `java`. To run the service locally, a user need only replace the first four lines of the command with `java`, and adjust the resource paths accordingly.

[10] https://gradle.org/

Chapter 6

Empirical Study

This chapter outlines the empirical evaluation we carried out to evaluate the prototype implementation of KOTFUZZ and its six constituent search algorithms. Section 6.1 begins by establishing the algorithms we set out to assess within this study, and Section 6.2 fixates the research questions we aim to answer. Section 6.3 establishes the metrics we use to determine the performance and behavior of each algorithm. Lastly, Section 6.4 details the experimental protocol we follow to obtain concrete results.

6.1 Evaluation Subjects

To gather a comprehensive overview of the performance, behavior, and implications of the approaches proposed in Chapter 4, we evaluate six prototype implementations, split between three distinct categories. The first category contains a single algorithm, RS, introduced in Section 4.5.1. The second class contains two instances of syntactic-oriented algorithms: Single-Objective Diversity Genetic Algorithm (SODGA) and Many-Objective Diversity Genetic Algorithm (MODGA), based on the formulations proposed in Section 4.5.2. The final group of algorithms focuses on the semantic proximity formulations detailed in Section 4.5.3, and contains three instances. Single-Target Proximity Genetic Algorithm (STPGA), Many-Objective Proximity Genetic Algorithm (MOPGA), and Whole-Suite Proximity Genetic Algorithm (WSPGA) implement the three proposed approaches.

Table 6.1 contains an overview of the six implemented algorithms. SODGA, STPGA, WSPGA behave like SO algorithms (though STPGA repeatedly switches objectives), while MODGA and MOPGA implement MO formulations. We opt for the use of tournament selection for SO scenarios, and domination rank counterparts for MO instances. For GA-based algorithms, we turn to established literature values to determine the size of the population and further selection details. However, applications of evolutionary fuzzing to compiler testing are scarce, and established evolutionary fuzzing tools like VUZZER [73] and V-FUZZ [53] make no recommendation in these regards.

As such, we turn to standard values used in software testing literature, particularly of EVOSUITE, which uses a population size of 50 and a tournament size of 10 [27, 70]. Past research by Arcuri and Fraser [7] has shown that default values can provide solid perfor-

Heuristic	Fitness	# Objectives	Selection
RS	-	-	-
SODGA	$f^{(SO)}_{\texttt{DIV}}(b,P) = \frac{1}{1+dis(b,P)}$	1	Tournament
MODGA	$f^{(MO)}_{\texttt{DIV}}(b) = \{-f_{sz}(b),\ f_{l_i}(b) \mid l_i \in \mathbb{L}\}$	$\mid \mathbb{L} \mid +1 = 6$	Dom. Rank
STPGA	$f^{(SO)}_{\texttt{PRO}}(b,t_i) = d(e(b),t_i)$	1, adaptive	Tournament
MOPGA	$f^{(MO)}_{\texttt{PRO}}(b,\mathcal{T}) = \{d(e(b),t_i) \mid t_i \in \mathcal{T}\}$	$\mid \mathcal{T} \mid$	Dom. Rank
WSPGA	$f^{(SO)}_{\texttt{WS}}(S,\mathcal{T}) = \sum_{t\in\mathcal{T}} \min_{b^{(i)}\in S} d(e(b^{(i)}),t)$	1	Suite Tournament

Table 6.1: Overview of evaluated algorithms.

mance in a broad set of scenarios, in the context of software testing. Though these findings provide no guarantees for the task of compiler testing in particular, we believe these are sensible starting points that circumvent the expensive requirements of parameter tuning over all proposed algorithms.

6.2 Research Questions

The empirical analysis of the proposed approach aims to answer the following research questions, which we reiterate from Section 1.3:

RQ1: *How do meta-heuristic guidance criteria influence the properties of test programs generated by fuzzing?*

To better differentiate between the six proposed approaches and their individual most significant guiding parameters, we split the *RQ1* into three subquestions:

RQ1.1: *How does the simplicity bias impact the properties of files generated by RS?*

RQ1.2: *How does the projection space topology influence the properties of files generated by syntactic diversity-driven heuristics?*

RQ1.3: *How does target selection influence the properties of files generated by semantic proximity-driven heuristics?*

The three subquestions address each of three generative heuristic categories by tuning their core elements. For RS, by far the most influential hyperaparameter is the simplicity bias, which governs both the structure and complexity of generate expressions. For diversity heuristics, the manner in which diversity is computed drives the selection procedure. Finally, the number of targets that proximity-heuristics receive is crucial for their search

process. Once established, the most sensible settings revealed by the previous subquestions can be used to empirically assess the performance of each category of algorithms in comparison to the others, as stated in *RQ2*:

> ***RQ2:*** *How do the random search, syntactic diversity, and semantic proximity generative heuristics perform in terms of uncovering bugs in the Kotlin compiler?*

To answer the second research question, we seek to measure the performance of the best algorithm from each class and compare both their bug-finding capabilities and efficiency.

6.3 Evaluation Metrics

To measure the performance of our algorithms, we focus on the two dimensions of *effectiveness* and *efficiency*. Before addressing those measures, however, we first establish the meaning of a compiler defect within the framework of DT. For the purposes of this study, a generated piece of Kotlin code *b* passes through two versions of the compiler, `K1` and `K2`. *b* is said to have uncovered a deffect if either of following cases occurs:

1. `K1` successfully compiles *b* and `K2` does not.
2. `K1` compiles *b* and `K2` crashes.
3. `K2` successfully compiles *b* and `K1` does not.
4. `K2` compiles *b* and `K1` crashes.
5. `K1` and `K2` both crash.

In the above listing, we define a *successful* compilation as one that results in an output object (for our purposes, a `jar` file). An unsuccessful compilation is one that *completes*, but does not result in an output object (i.e., the compiler has detected that the code is erroneous). A *crash* occurs when compilation fails to complete altogether. Table 6.2 contains a visualization of the types of defects that can emerge during a differential testing round.

Only two types of behaviors are considered correct within this system: either both systems successfully compile an input file to a `jar` object, or both fail to do so, without crashing. In any other case, we consider the behavior erroneous and attach it a label that categorizes the defect according to the list above. Since the tangible goal of this study is to help improve the quality of the `K2` compiler, we only focus on error types that break backwards compatibility (I and III), or that cause `K2` to crash (IV and V). We emphasize errors that only cause defects in `K1` (type II) less than their counterparts, as this version is to be replaced in the future.

We use this notion of defects to measure the performance of the six proposed algorithms. The effectiveness of a fuzzer run is the number of files generated within that run that cause any defects (types I-V) after differential testing. This metric captures the performance of the fuzzer after the exhaustion, however, an additional measure of efficiency is required to

		K1 outcome		
		Success	Failure	Crash
K2 outcome	Success	✔	Type I	Type II
	Failure	Type III	✔	Type II
	Crash	Type IV	Type IV	Type V

Table 6.2: Types of compiler defects.

underline an algorithm's behavior over time. To this end, we express the efficiency of a run through the Area-Under-Curve (AUC) metric.

Informally, when running a fuzzer algorithm, snapshots of its performance can be expressed as 2-dimensional points, where the x axis represents the time elapsed, and the y axis, the number of uncovered defects. After several snapshots have been taken at approximately comparable times for multiple algorithms, the curve that emerges from connecting the points corresponding to snapshots splits the space in two. The area under this curve provides a means of comparing how quickly algorithms uncover bugs. Given a set P of two dimensional points p_i, with $p_i^{(1)}$ the time at which the snapshot was taken and $p_i^{(2)}$ the number of defects uncovered by generated files until that point, we compute the AUC according to the trapezoidal rule approximation given by Equation (6.1).

$$\frac{\sum_{i=1}^{|P|-1}(p_i^{(2)}+p_{i+1}^{(2)})\cdot(p_{i+1}^{(1)}-p_i^{(1)})}{p_{|P|}^{(1)}-p_1^{(1)}} \tag{6.1}$$

6.4 Experimental Protocol

To ensure the robustness of the empirical evaluation, we perform repeated runs to determine the relative performance of the six proposed methods. Due to the large number of possible hyperparameter and heuristic combinations, as well as the large quantity of computational resources required to perform DT on generated files, we focus these techniques on performance evaluations. To determine appropriate parameter ranges for variables not explored in previous work, we rely on fewer, longer runs, instead.

Statistical Analysis

Where appropriate, we additionally perform statistical tests in accordance with standard practices related to empirical analysis of randomized software engineering algorithms [6]. We employ the paired-data version of the Wilcoxon [17] signed-rank test to verify whether the performance (either efficiency or effectiveness) of one algorithm differs from that of another. This test operates on a null hypothesis that the distribution of the difference between two sample populations is symmetric around a mean of 0. Informally, the alternative hypothesis states that there is a significant difference between the measurements emerging from the two algorithms subject to comparison. The p-value computed by this test de-

termines whether the evidence supports the null or the alternative hypothesis. *Significant* p-values (< 0.05) imply a high probability of the latter being true.

To measure how much the performance of two algorithms differs in favor of either one, we use the method established by Vargha and Delaney [86] A_{12} for computing the *effect size* of a population of samples, if the associate p-value is significant. The A_{12} captures the stochastic difference between the distributions approximated by the two populations and, with respect to our study, shows which distribution skews higher based on evidence from several runs. An A_{12} measurement of 0.5 in a comparison between two algorithms a_1 and a_2 denotes the two approaches are equivalent. Higher values favor a_2, while, symmetrically, lower values favor a_1.

Experiment Length and Heuristic

To understand how the simplicity bias influences the nature of generated files (*RQ1.1*), we test several values of this parameter under the setting of RS. Random Sampling is a prime candidate for uncovering the effects of this option, as there are very few external factors that could otherwise alter the shape of block generation. We experiment with simplicity bias values of between 0.4 and 0.6, as we empirically found that values outside this range either lead to blocks too large to scale effectively, or too small to build a sufficiently complex context. We run KOTFUZZ using RS for 90 minutes for each setting of simplicity bias.

To answer *RQ1.2*, we perform two 90-minute runs for both SODGA and MODGA: one with a Euclidean norm distance measure ($d(e(b_1), e(b_2)) = \sum_{1 \leq i \leq |e(b_{\{1,2\}})|} \sqrt{(b_1^{(i)} - b_2^{(i)})}$), and one with an l^∞ norm equivalent ($d(e(b_1), e(b_2)) = \max_{1 \leq i \leq |e(b_{\{1,2\}})|} \mid b_1^{(i)} - b_2^{(i)} \mid$). The results would help uncover how a more stringent measures of diversity influences the quality of the generated files. Next, we attempt to answer *RQ1.3* by varying the number of targets provided to STPGA, MOPGA, and WSPGA between 50 and 200, which helps understand how the number of objectives affects the generative process. Finally, we answer *RQ2* by considering the effectiveness and efficiency of RS, and the two most sensible approaches that emerge from the analysis of the previous subquestions.

Experimental Environment

We carry out all DT procedures through the publicly released Kotlin version `1.8.20-RC-release-288` [1]. We ran all generation, DT, and analysis experiments on a AMD Ryzen 7 5800H machine, running on the Manjaro Talos 22.1.2 OS, with 16 gigabytes of RAM. We executed all experiments in isolated containers using Docker.

We ran five instances of RS to answer *RQ1.1*. Three runs SODGA and MODGA provide the data for *RQ1.2*, while four runs for each of the three proximity algorithms constitute the experimental analysis for *RQ1.3*. We additionally executed ten runs each for each of the most promising algorithm from each class to answer *RQ2*, for a total of $5+3+12+3\times 10 =$ 50 total fuzzing sessions. We additionally perform one other long-term (82 hour) fuzzing campaign to gain insight into the scale-related behavior and limitations of the fuzzer.

[1] https://github.com/JetBrains/kotlin/releases/tag/v1.8.20

Chapter 7

Results

The aim of this thesis is to gain insight into the behavior and applicability of heuristics to the task of differential testing of the Kotlin compiler. To achieve this goal, this chapter seeks to answer the research questions posed in Section 6.2 through empirical evidence and statistical analysis. We structure the analysis of the results as to mirror the hierarchy of research questions. Section 7.1 analyzes the effect of the most impactful hyperparameters on the generative heuristics. Section 7.2 details the performance of the best representative of each heuristic category. Section 7.3 concludes this section by highlighting individual bugs KOTFUZZ automatically uncovered an analyzing their origins.

7.1 Influence of Guidance Parameters

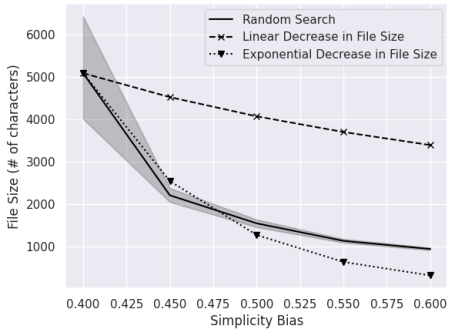
We begin our analysis of hyperparameter influence by considering the effects of the simplicity bias on the structure, size, and number of files that KOTFUZZ generates in Section 7.1.1, followed by the effects of distance metrics on diversity heuristics in Section 7.1.2, and finally, the impact of target selection in Section 7.1.3.

7.1.1 Effects of Simplicity Bias on Random Sampling

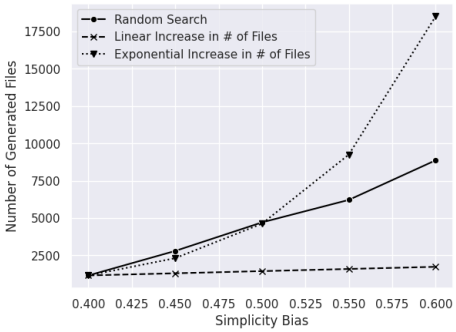
To study the effects of the simplicity bias, we focus on the size and number of files generated by RS. We limit the analysis of simplicity bias values to a range between 0.4 and 0.6, which we empirically evaluated to strike a balance between complexity and size. Values of this parameter exceeding 0.6 tend to severely limit the semantic context in which code is generated, while values below 0.4 increase the probability of producing output whose size and complexity diminish its real-world application.

Figure 7.1 depicts the relation between (a) the average size of the blocks generated through random sampling and (b) the number of files the algorithm outputs in a 90-minute interval. To help contextualize the rate of change, we introduce additional data points representing linear (dashed lines) and geometric (dotted lines) changes in the data, where each previous value is either halved (for subfigure (a)) or doubled (for subfigure (b)).

For values of the simplicity bias between 0.40 and 0.50, both rates of change are comparable to their geometric counterparts. The average size of a file decreases from 5,084



(a) Mean size of block generated by RS as a function of simplicity bias.



(b) Number of blocks generated by RS per 90 minutes as a function of simplicity bias.

Figure 7.1: Comparison of the number of files generated by RS and their size.

characters for a bias of 0.40 to 2,205 for a bias of 0.45 and 1,545 for a bias of 0.50. The rate of change diminishes for bias values 0.55 and 0.6, with average sizes of 1,132 and 941 characters, respectively. Conversely, the number of generated files increases from 1,157 for the lowest value of bias (0.40) to 8,869 for the highest (0.60). Values attached to biases between the two extremes follow the same trend, with 2,799, 4,712, and 6,218 files generated for simplicity biases of 0.45, 0.50, and 0.55, respectively. Both rates of change significantly outpace their linear equivalents for the entire testes interval. In the first first two intervals for both subfigures, the rate of change also surpasses its geometric counterpart.

To further understand the variability of random sampling, we consider the distribution of file sizes for each value of the simplicity bias. Figure 7.2 and Table 7.1 give the visual and numerical interpretation of this distribution. While the mean size of files varies by over 3000 characters between the extremes of the simplicity bias, the medians only vary by 201. Further, the Inter-Quartile Range (IQR) more than halves from 2013 for the lowest simplicity bias, to 834 for the highest. This suggests that while all tested biases are likely to produce many files of less than 700 characters, lower simplicity biases result in many more large outliers than their higher counterparts. This is in accordance with the depiction of Figure 7.2, which highlights that as the simplicity bias increases, its upper quartile significantly shrinks while the lower quartile remains unchanged.

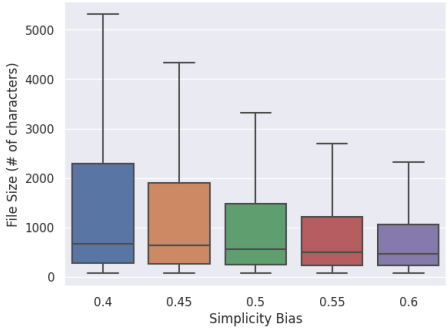


Figure 7.2: Block size distribution of RS.

Bias	Mean	Median	IQR
0.40	5084	669	2013
0.45	2205	644	1631
0.50	1545	555	1227
0.55	1132	499	984
0.60	941	468	834

Table 7.1: Statistics of block size distributions of RS.

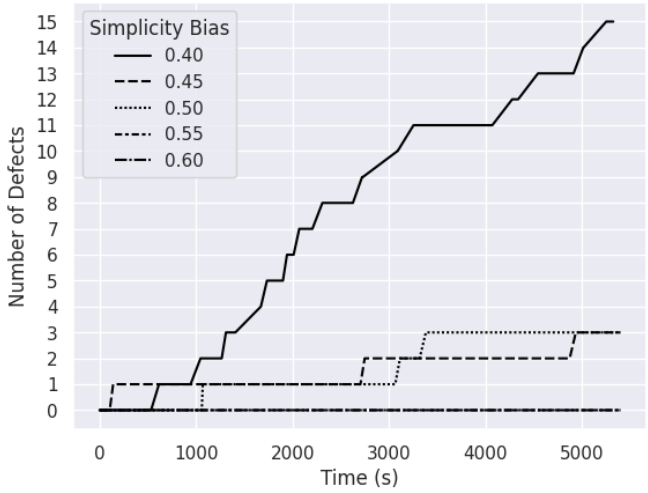


Figure 7.3: The number of defects uncovered by RS over different simplicity bias values.

Finally, we analyze the relation between the simplicity bias and the effectiveness of random sampling as a mechanism for uncovering bugs in the Kotlin compiler. Figure 7.3 provides a visualization of the number of defects uncovered through differential testing of the generated files, for each simplicity bias value tested. The most important aspect of these results is that *all* defects uncovered by all 5 versions of RS consist of Type I errors (as described in Table 6.2). These errors all trigger Out-Of-Memory (OOM) errors in the `K1` compiler, but not in its newer `K2` counterpart.

All OOM errors encountered in the five experimental runs exceed 10,000 characters, and their distribution among the simplicity bias values aligns with the distribution of file sizes that each value entails. RS with a simplicity bias of 0.40 generates 15 OOM-inducing files, while higher values of 0.45 and 0.50 both produce 3 such instances. Simplicity bias values exceeding 0.50 do not generate such files, due to their narrower file size distribution.

Summary RQ1.1: Lower simplicity bias values cause RS to generate significantly larger and significantly fewer files, given the same computational budget. Simplicity biases of 0.50 and lower are likely to occasionally generate files of over 10,000 characters, which often trigger OOM errors in `K1`, but not in `K2`.

7.1.2 Effects of Distance Metric on Syntactic Diversity-Driven Search

We consider the implications of using the l^2 and l^∞ norms as distance measures for SODGA. We additionally take the behavior of MODGA into account as a point of comparison, which does not utilize distance metircs in its optimization.

We begin by analyzing the properties of the files captured in each snapshot of the algorithms. Figure 7.4 provides the visualization of the evolution of the average generated file size over time. For both variations of SODGA, the mean file size fluctuates between 1000 and 3500 characters, which are significantly under the threshold of 10,000 characters that

Figure 7.4: Mean size of blocks generated by diversity algorithms over time.

may trigger common OOM errors. The fluctuations are a consequence of the population-sensitive fitness function: as the population tends towards larger files, the selection operator pressures the population towards smaller blocks that inherently belong to a farther away region of the search space. As smaller files take over the population, the trend reverses and larger files again introduce more diversity. This trend is apparent for both the l^2 and l^∞ norms. However, the latter introduces larger shifts in the size of blocks because of its more stringent measure of (dis)similarity. As a result, the l^∞ variant generates both the largest (3500) and smallest (900) average file size population out of the considered snapshots.

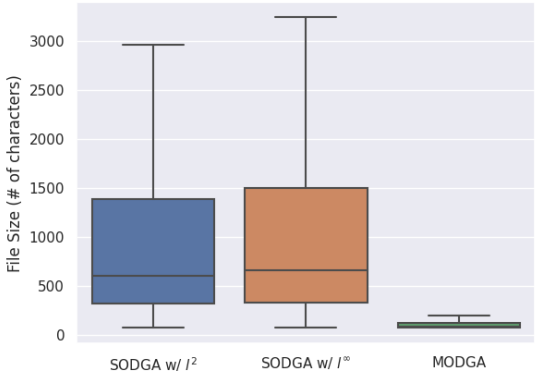


Figure 7.5: Block size distribution of diversity heuristic variations.

Variant	Mean	Median	IQR
SODGA-l^2	1598	599	1067
SODGA-l^∞	1574	658	1177
MODGA	322	83	49

Table 7.2: Statistics of block size distributions of diversity heuristics.

By contrast, the elitist archive of MODGA retains files that are far smaller than the population of its single-objective counterparts. After initial fluctuations in the first 6 snapshots, the average size of files in the MODGA elitist archive converges to around 300 characters. This coincides with the members of the archive itself, which also converge to a static set

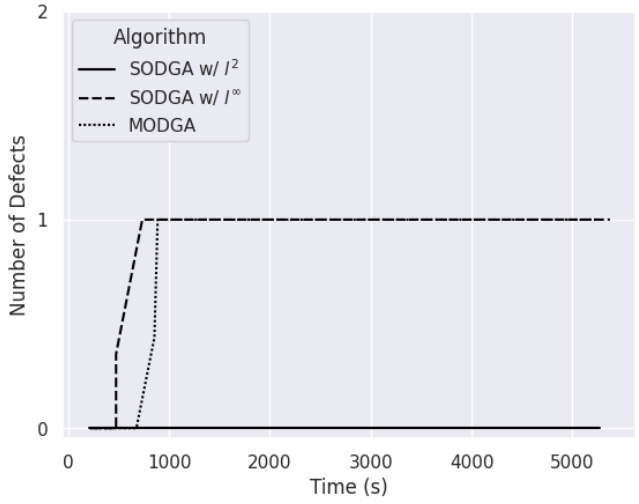


Figure 7.6: The number of defects uncovered by SODGA and MODGA.

of 70 solutions at the sixth snapshot. Further changes for the remainder of the runtime are minimal by comparison, as the archive grows by only 3 additional entries.

Figure 7.5 and Table 7.2 provide additional details regarding the file size distribution of syntactic diversity heuristics. The means of the distributions of the two SODGA variations are comparable: the l^∞ variant produces file that are on average only 1.50% (1574 characters) smaller than the Euclidean norm equivalent (1598 characters). However, the shape of the distribution differs between the two, with the l^∞ alternative generating more outliers. This property reveals itself in the median and IQR measures, which are 9.8% and 10.3% (658 and 1177) larger for the l^∞ optimizer than in the Euclidean revision (599 and 1067). MODGA produces files that are both much smaller and less divergent than either of the single objective options. This proclivity is attributed to (i) the size component significantly diminishes the size of the archive and (ii) the tendency of the archive to retain few (< 10) very large files, which alone dominate large regions of the search space.

Finally, Figure 7.6 depicts the number of defects uncovered by each algorithm variation. Only two defects emerge from differential testing of the generated files of the three algorithms. SODGA equipped with the l^∞ norm and MODGA each independently uncover one Type III defect that causes `K1` to raise an error, while `K2` successfully compiles the generated code. Both instances of the defect emerge as a consequence of the genetic recombination operator, and could not have been otherwise generated by RS. We discuss further implications of this defect and the operators involved in its creation in Chapter 8.

Summary RQ1.2: SODGA displays comparable performance and behavior under both l^2 and l^∞ norms, with the latter displaying a broader distribution. Both variations fluctuate between selecting larger and smaller files, based on the algorithm's recent history. MODGA maintains an archive that stagnates after a limited number of iterations, but effectively retains smaller files. All three algorithms are capable of uncovering defects that RS could not.

7.1.3 Effects of Target Set on Semantic Proximity-Driven Search

We analyze the behavior of STPGA, MOPGA, and WSPGA under four different target sets, of 50, 100, 200, and 400 targets, respectively. For each algorithm, we analyze snapshots that consist of either the closest solution to each target (for STPGA), the set of non-dominated solutions (for MOPGA), or the best suite generated (for WSPGA).

We again begin our analysis by considering the size of the files retained by each algorithm. The distribution over each snapshot for STPGA is given in Figure 7.7 and Figure 7.8. Means vary by only 14 between the smallest (341 for 400 targets) and the largest (355 for 200 targets) values. The file size distribution contains few outliers, as captured by the IQR measurements that do not exceed 260. While all measurements exceed the size achieved by MODGA (mean 322, IQR 49), they are far lower than those generated by RS with an equal simplicity bias (mean 1545, IQR 1227). Since STPGA does not contain an explicit size objective like MODGA, the only mechanisms to which the retention of smaller files can be attributed to are either the archival or the embeddings.

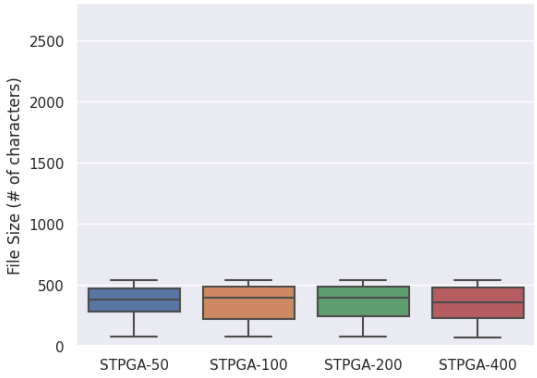


Figure 7.7: Block size distribution of STPGA variations.

Variant	Mean	Median	IQR
STPGA-50	353	378	196
STPGA-100	350	390	260
STPGA-200	355	391	218
STPGA-400	341	357	250

Table 7.3: Statistics of block size distributions of STPGA variations.

To further understand this phenomenon, we analyze the corresponding behavior of MOPGA and WSPGA. Figure 7.8 and Table 7.4 capture the behavior of the former. While not as drastic, all three measurements of the distribution display significantly lower values (with means between 986 and 1124, and medians between 344 and 390) than the RS counterpart. The larger difference between means and medians, together with the higher IQRs highlight the appearance of more outliers in MOPGA than in STPGA. This can, in large part, be attributed to the differences in the implementation of the archive, which holds far more values for the MO variant, and is thus less stringent on the blocks it retains.

WSPGA employs the least constraining retaining mechanism, simply tracking the test suite with the best scalarized fitness. The distribution captured by Figure 7.9 and Table 7.5 reflects this. WSPGA contains, on average, the largest files of any proximity algorithm, with means more than triple those of STPGA. WSPGA also displays the highest variance between its four versions, with an average difference of 446 characters between the files pertaining to the WSPGA-100 and WSPGA-200. The whole-suite approach also produces the most spread out distribution, with the highest IQR values of all proximity-class methods.

Despite this, all variants of this algorithm retain files that are, on average, smaller than the RS counterpart with the same simplicity bias.

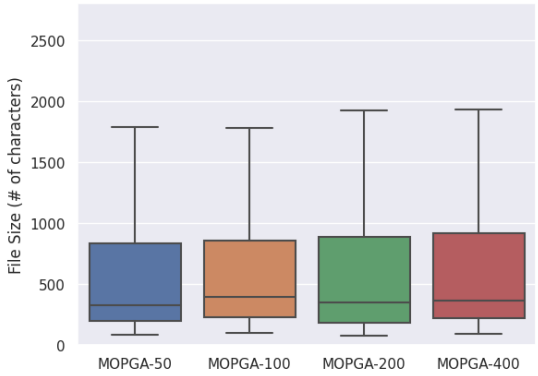


Figure 7.8: Block size distribution of MOPGA variations.

Variant	Mean	Median	IQR
MOPGA-50	991	378	638
MOPGA-100	986	390	626
MOPGA-200	1124	344	702
MOPGA-400	988	359	696

Table 7.4: Statistics of block size distributions of MOPGA variations.

This phenomenon occurs in all twelve variations, thus suggesting that the embedding mechanism is the root cause of this bias. Reasoning about why bias is implanted in the embedding procedure is difficult, as the CODEBERT NN at the core of this process is hardly interpretable. However, most files that we include in the target set (or aggregations emerging as results of operations on the embedding of those files) are short and contain isolated test cases of specific language features. This stands out as the most intuitive explanation of the implicit bias of the algorithm: short generated files embedded by CODEBERT are more likely to be similar to selected targets, which are also short. Though this occurrence warrants further investigation, it is outside the scope of this thesis.

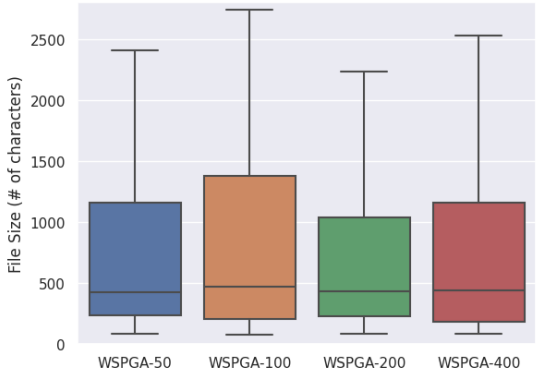


Figure 7.9: Block size distribution of WSPGA variations.

Variant	Mean	Median	IQR
WSPGA-50	1053	423	828
WSPGA-100	1455	468	1178
WSPGA-200	999	432	816
WSPGA-400	1109	436	880

Table 7.5: Statistics of block size distributions of WSPGA variations.

We additionally consider the behavior over time of the two archive mechanisms of STPGA and MOPGA. This aspect is relevant for understanding the scalability of the approaches. Non-dominated archives such as that of MOPGA, tend to scale with the number of solutions in the optimal Pareto set, which, in general, scales with the number of objectives of the problem [58]. Figure 7.10 and Figure 7.11 show the behavior of the archives

of STPGA and MOPGA, respectively. The former, which only retains the closest block for each target, behaves as intuitively expected, with variations that have more targets retaining more files than counterparts with fewer targets. The number of files retained by the archive of STPGA at the end of the run scales sublinearly with the number of targets, with STPGA-400 only having 5 times more blocks in its archive than STPGA-50, despite having 8 times more targets. This result implies that scenarios in which one file is the closest approximation of many targets are common.

Figure 7.11 displays significantly different behavior for MOPGA. Unlike its iterative counterpart, the archive size of MOPGA never reaches a stage where its size converges and fluctuates around a certain value. Instead, all variations of MOPGA grow for the entire duration of the run, suggesting convergence had not occurred at the end of the experiment. Additionally, the archive sizes of MOPGA variations only scale with the number of targets in the beginning of the run. After 6 snapshots (cca. 18 minutes of runtime), the archives MOPGA-50 and MOPGA-100 reach the same size as their higher-target counterparts. In further snapshots, the variants with the fewest targets retain the highest number of files. This occurs as a consequence of the way in which we capture snapshots: the time-based method we used (intervals of three minutes) does not take into account the number of generations each algorithm has completed. MOPGA variations with higher numbers of targets incur significant overhead as an effect of requiring additional computation with each iteration, as each archive computation scales with the number of objectives. In practice, this means the algorithm progresses at a slower pace the more objectives it optimizes for, and thus variants with fewer targets retain more files when runtime is limited.

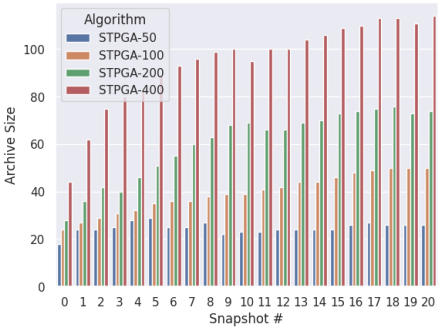


Figure 7.10: Evolution of STPGA archive size over time.

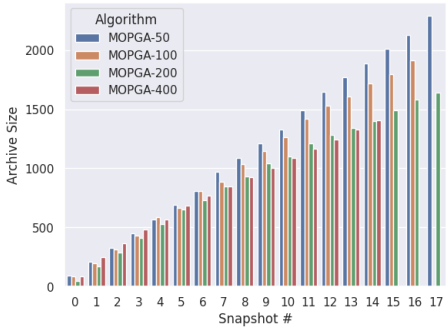


Figure 7.11: Evolution of MOPGA archive size over time.

Finally, we consider the bug finding capabilities of STPGA and MOPGA, which are the only two algorithms to uncover defects in our experiments. Figure 7.12 and Figure 7.13 depict these results, respectively. While STPGA algorithms only uncover one instances of a Type III defect (`K1` rejects, `K2` accepts), MOPGA algorithms uncover between 5 and 44 defects each. These defects include Type II and V OOM errors, as well as instances of the same Type III defect fond by STPGA and diversity-class algorithms. In addition, MOPGA with 400 targets finds the fewest defects (5), while MOPGA-50 uncovers the most (44). This coincides with the size of the archives of each algorithm, with the former having the smallest, and the latter, the largest. This trend is not entirely consistent with the MOPGA-

100 and MOPGA-200, with the former finding fewer bugs, despite having the larger archive.

All four algorithms uncover the same *categories* of errors, both among themselves, as well as in comparison to diversity algorithms. This suggests that the number of bugs uncovered by each heuristic is a consequence of the number of blocks retained in the archive, rather than a result of the target choice. Once introduced in the archive, files that cause crashes tend to persist for the duration of our experiments, thus explaining the consistently increasing "step size" in the convergence graphs. The high degree of randomness involved in the generative process, together with the fact that none of the algorithms had converged at the end of the experiment, could explain the fact that MOPGA-100 and MOPGA-200 do not clearly follow this trend. These results are not conclusive, as we only performed one run for each variation within this study. Further analysis based on more empirical data, as well as longer runs that reach convergence, would improve the understanding of this behavior.

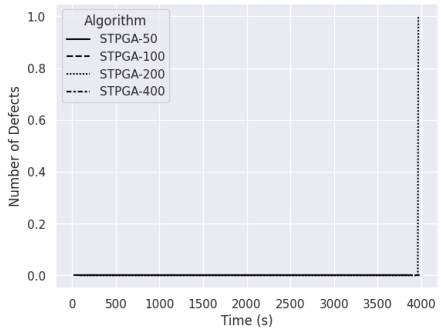


Figure 7.12: Evolution of STPGA archive size over time.

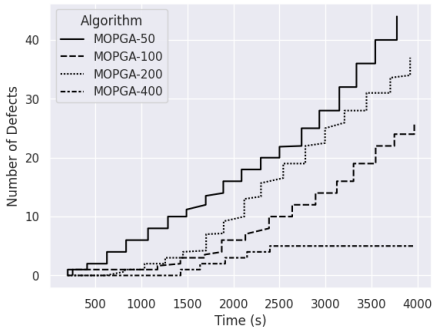


Figure 7.13: Evolution of MOPGA archive size over time.

Summary RQ1.3: Irrespective of the number of targets, all proximity-driven algorithms retain files that are, on average, smaller than RS with an equivalent simplicity bias, likely as a result of biases implanted in the embedding process. STPGA produces the smallest files, followed by MOPGA, and WSPGA. MOPGA finds the most defects of any algorithm, which is linked to the non-dominated archive retaining files that cause defects for long periods of time.

7.2 Comparative Heuristic Performance

To gain additional insight into the comparative performance of the three classes of heuristics, we compare the defect uncovering capabilities of one algorithm from each class. To this end, we performed 10 fuzzing sessions with RS, SODGA-l^2, and STPGA-200, respectively. We selected SODGA-l^2 on the basis that it generates fewer outlier files than its l^∞ counterpart, and does not suffer from possibly premature convergence, like MODGA. We also selected STPGA-200 rather than the MO-based counterparts because the latter's extreme archive growth would make for an unfair comparison to the other two algorithms, that do not retain such a mechanism. All algorithms used a simplicity bias of 0.5.

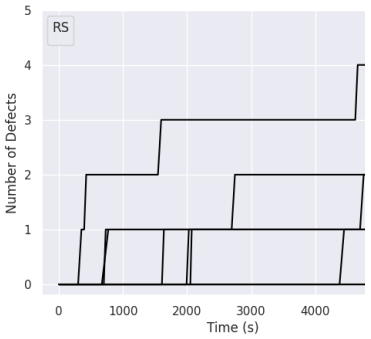


Figure 7.14: Convergence plot of RS defects.

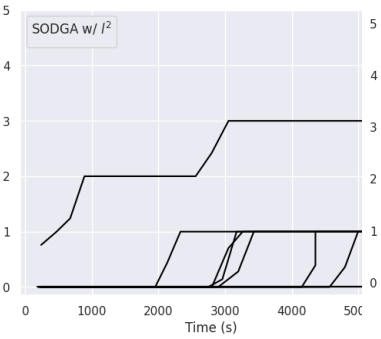


Figure 7.15: Convergence plot of SODGA defects.

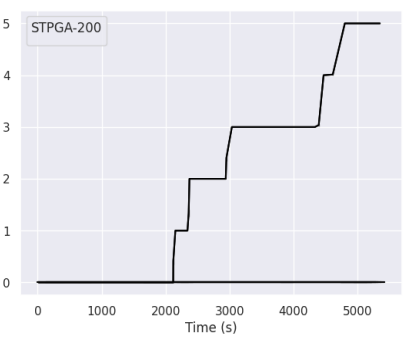


Figure 7.16: Convergence plot of STPGA defects.

Figures Figure 7.14, Figure 7.15, and Figure 7.16 show the convergence plots emerging from the three experiments. Table 7.6 summarizes the collected metrics and statistical analyses. In total, RS uncovers more defects (12) than both SODGA (9) and STPGA (10). The statistical analysis of the distribution of the number of bugs reveals that RS is not significantly better at uncovering bugs than either SODGA ($p = 0.459$) or STPGA ($p = 0.570$). There is also no significant difference in the effectiveness of SODGA and STPGA ($p = 0.666$). The RS runs reveal one novel Type III defect (`K2` does not compile, `K1` does) not found by any previous experiment, in addition to several Type II and V errors (crashes in either just `K1`, or both `K1` and `K2`). Both the STPGA and SODGA runs additionally reveal previously uncovered Type I and Type II defects, and do not generate any novel crashes. We defer the analysis of individual bugs to Section 7.3.

Since all algorithms perform similarly in terms of effectiveness, we turn to the analysis of efficiency using the AUC metric. The 10 RS runs result in the highest mean AUC of 0.719, which is greater than both SODGA (0.446) and STPGA (0.385). Pairwise statistical analysis shows that no algorithm class is statistically better than any other, as per Table 7.6.

			Effectiveness p			Efficiency p		
	# Defects	Mean AUC	RS	SODGA	STPGA	RS	SODGA	STPGA
RS	12	0.719	-	0.549	0.570	-	0.322	0.400
SODGA	9	0.446	0.549	-	0.666	0.322	-	0.483
STPGA	10	0.385	0.570	0.666	-	0.400	0.483	-

Table 7.6: Summary of effectiveness and efficiency analysis.

To better understand whether the similar behavior in terms of both effectiveness and efficiency is inherent to our implementation or a consequence of the relatively limited runtime, we further analyze the empirical behavior of a longer fuzzer campaign. We limit this experiment to the same configuration of SODGA analyzed in this section, for three main reasons. First, the behavior of SODGA is transparent, in the sense that it contains no BB components, such as the embeddings of the proximity algorithms, that could hinder the interpretability of the results. Second, unlike the proximity-class algorithms and MODGA,

SODGA evaluates individuals relative to an evolving population. This volatile fitness function means the algorithm will not converge upon a stable set of solutions, which is desirable for longer fuzzing campaigns. Finally, unlike RS, SODGA leverages both random sampling and genetic operators.

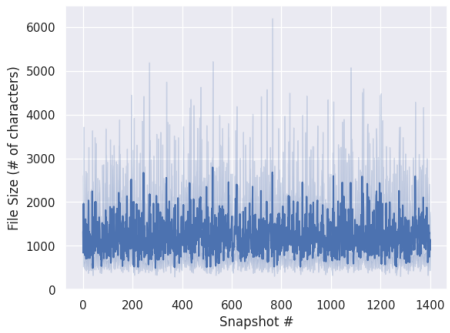


Figure 7.17: File size distribution of SODGA over 82 hours.

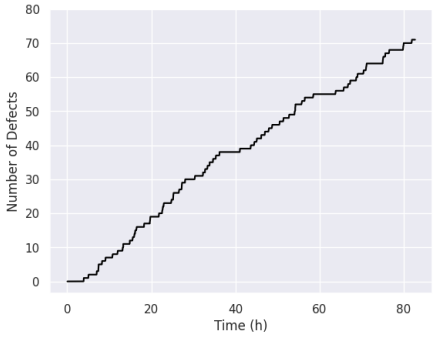


Figure 7.18: Convergence plot of SODGA defects over 82 hours.

Figure 7.17 and Figure 7.18 show the results of the long-term experiment. The fluctuations in size discussed in Section 7.1.2, that emerge as a consequence of the population-relative fitness function, persist for the entire duration of the experiment, as depicted in Figure 7.17. In total, the 82 hour fuzzing campaign produced 1401 snapshots and a total of 70,050 files. Of those files, 70 (0.099%) failed to compile for both `K1` and `K2`, as a result of an implementation error in KOTFUZZ. The experiment generated 71 files that cause defects, however, no new defects were uncovered. The 71 files that cause DT errors include instances of all previously uncovered bugs, with the exception of the novel Type III error uncovered by RS and highlighted previously in this section. We discuss the potential causes underlying this phenomenon in Chapter 8.

Summary RQ2: Neither effectiveness, nor efficiency are significantly different between RS, SODGA-l^2, and STPGA-200. However, the three algorithms find distinct inidividual categories of defects.

7.3 Defect Analysis

Throughout the experiments carried out in this study, we encountered several hundred instances of erroneous behavior in the Kotlin compiler. After individual analysis and consultation with the Kotlin compiler developer team, we divided these instances into five distinct categories. This section briefly analyzes these categories, their impact on the Kotlin ecosystem, and the components of the fuzzer that were responsible for their emergence.

`K1` OOM Errors

As our analysis from Section 7.1.1 revealed, OOM errors occurring only in the `K1` compiler are common among files that exceed 10,000 characters in length. The fact that such files do

```
fun main() {
    fun p () : Char {
        return 'c'
    }

    fun p () : Float {
        return 13.0f
    }
}
```

Figure 7.19: K2 false negative conflicting overloads.

not cause equivalent errors in K2 makes them less important for the scope of this study, as they showcase a measurable improvement in performance, rather than an issue that requires developer attention. As a result, we did not report any of these issues to the developer team, instead focusing on bugs that affect K2.

K2 OOM Errors

We occasionally encountered code that either (i) triggered OOM errors in both K1 and K2, or (ii) only triggered OOM errors in K2. Though also large, these files are not as clearly correlated with size as their K1 OOM counterparts. We reported two instances, one for both cases, to the Kotlin developer team. Compiler developers confirmed both instances for the current release of Kotlin, and traced the root cause to the compiler's backend JVM component. Both instances were categorized as *perfromance issues*, rather than bugs, and one of them [1] was deemed acceptable. Compiler engineers tracked the root cause of this defect to the quadratic space complexity of a subsystem that performs analysis. Upon receiving this feedback, we focused our analysis on shorter, more actionable cases.

K2 Nested Functions

Figure 7.19 contains a piece of code that triggers a Type I failure. K2 compiles the code without warnings, while K1 fails to do so, and throws a *conflicting overloads* error. The latter is the intended behavior. The two p functions cause a resolution conflict that the compiler is meant to warn of. After experimentation with this instance, we observed that K2 always resolves a p() function call to the first definition of the function, irrespective of the return type or the number of re-declarations. Notably, this problem only occurs when the definitions of p() are both *nested* inside a higher-scope function. The Kotlin compiler developers confirmed the existence of this bug in the current release version, and assigned it medium priority. The developers additionally fixed a target release date for a fix in a future minor release of the compiler.

We uncovered several dozen instances of this bug, all generated through GA-based algorithms. The sampling process that RS depends on contains a check that prevents the

[1] https://youtrack.jetbrains.com/issue/KT-59190/OutOfMemoryError-Java-heap-space-with-fuzzer-generated-code

generation of functions with the same name, which is tracked through the share context. However, this constraint is relaxed during recombination, allowing for such scenarios to occasionally transpire during crossover. We further discuss the implications of generative constraints and their relaxation in Chapter 8.

K2 RuntimeException

Figure 7.20 highlights a Type III error that affects the `RuntimeException` class of the Kotlin standard library. The `K2` compiler throws an overload resolution ambiguity exception stemming from the `RuntimeException(R)` expression in line 3. `K1` compiles the code without error, which is the intended behavior. The Kotlin developers verified the occurrence of this bug in several compiler versions that were released in 2023, including the most recent one. The compiler team traced the bug to the resolution and inference components of the compiler frontend, and assigned a future minor release of Kotlin 1.9 as the target release date for a fix.

Though any permutation of algorithm and configuration described in this study is in principle capable of generating the code in Figure 7.20, it was a version RS not covered in our empirical analysis that first uncovered it. The 82 hour campaign of SODGA subsequently revealed the same defect. From the perspective of the fuzzer, the piece of code that triggers the error is a simple fragment generated from an expression node. Finding such bugs entirely relies on effectively covering the input of the fuzzer's context, which RS-based algorithms might be more effective at, since they do not incur additional overhead from other GA operators, that operate on higher-levels language structures.

```
1  var R: UninitializedPropertyAccessException = UninitializedPropertyAccessException("fooBar")
2  R = R ?: UninitializedPropertyAccessException(
3      "fooBar".plus(ArithmeticException()), RuntimeException(R))
```

Figure 7.20: `K2 RuntimeException` overload resolution ambiguity.

K2 ConcurrentModificationException

Akin to the `RuntimeException` scenario, the code in Figure 7.21 highlights a `K2` defect originating from the constructor of the `ConcurrentModificationException` class from the Kotlin standard library. Similar to the previous case, the code surfaced from a fuzzing campaign of RS, in this case from the experiments carried out in Section 7.2. Also similar to its peer, this defect was confirmed for the current release of the Kotlin compiler, and its errors trace back to the same frontend components, called Checkers. Generating this instance is also feasible for every proposed approach.

```
1  var J_: ConcurrentModificationException = ConcurrentModificationException()
2  J_ = J_ ?: ConcurrentModificationException(ConcurrentModificationException(J_))
```

Figure 7.21: `K2 ConcurrentModificationException` overload resolution ambiguity.

Chapter 8

Discussion

This section discusses the totality of the study and contextualizes potential implications. We begin by analyzing the limitations of our approach in Section 8.1, before addressing potential threats to the validity of the research in Section 8.2. Section 8.3 provides recommendations regarding potential avenues of future expansion of this study.

8.1 Limitations

Throughout this study, we generated, compiled, and analyzed roughly 200,000 Kotlin files, and uncovered five categories of defects affecting three components of the Kotlin compiler. The Kotlin compiler team verified all of these errors and is working on targeted fixes for several of them. In this process, several non-trivial underlying biases and restraints that KOTFUZZ carries came to light. In this section, we analyze both the practical and theoretical limitations of the fuzzer, critically assess them, and propose solutions.

Language Features The set of language features that the fuzzer currently supports is limited to functions, four types of expressions, and statements. This limitation, which we imposed to ensure the feasibility of the study, drastically limits the domain of code that KOTFUZZ can generate. In practice, this restriction means that any defects that are related to core language features, such as classes, are outside the reach of the current research prototype. Extending the set of language features is paramount for enhancing the practicality of the tool and for effectively exploring the Kotlin syntax. In its current state, extending novel language features consists of simply implementing either one or two additional Java classes and making them accessible in parent CFG nodes.

Context Throughout the study, the root context used for sampling consisted of a limited collection of standard library files mostly containing exceptions, throwables, as well as numeric, logical, and string types. Besides severely limiting the number of objects and classes that KOTFUZZ can generate, this factor also affects the set of language features that our tool can leverage. For instance, `for` statements in Kotlin can only be applied to

constructs that implement specific methods, also known as *container* classes. In the current context, no such type is available, further limiting the syntax KOTFUZZ can leverage.

Generative Constraints To ensure the validity of generated code, we embed numerous constraints and biases in the sampling process. The implementation of these constraints has proven effective, as all sampled code examined in this study was valid (i.e., either `K1` or `K2` compiled the code, unless a `OOM` error occurred). However, these constraints may drastically reduce the domain of code that KOTFUZZ can generate, thus possibly also missing out on additional defects. For instance, constraints such as *the return type of a function must be samplable in the current context* may render much of the context useless. While intuitively respecting such a constraint is sensible (if there is no way to sample a type in the current context, it is impossible to write a function that returns such a type), it could be that breaking this constraint might lead to false negative defects. The genetic recombination proves this by breaking one such constraint, which lead to the nested function defect discussed in Section 7.3. Exploring how generative constraints affect the quality of the generated code and whether they lead to additional defects requires careful and incremental changes, that we do not cover in this study.

Representation and Genetic Operators The hierarchical *fragment, snippet, block* representation is sufficient to meaningfully separate code based on complexity and scope, but lacks the precision needed to explore certain corner cases. For instance, neither snippets, nor blocks are recursive, which means variation operators in GA algorithms are only able to operate on the highest scope of the file. This limitation leaves out numerous cases in which code can be exchanged on smaller levels, in turn exploring a new dimension of the language. A representation capable of capturing this level of detail requires a recursive implementation of blocks and fragments, the foundation of which is already implemented in the current implementation of KOTFUZZ.

GA Convergence The application of GA-based formulations to fuzzing raises the problem of convergence. Especially visible in the cases of SODGA and STPGA, the problem stems from the fact that relatively stable sets of solutions may emerge quickly in those GAs. Since longer fuzzing campaigns become wasteful and redundant in those scenarios, a mechanism that breaks this stability would enable longer fuzzing runs. Two options to alleviate this problem include restart mechanisms to effectively reset the search process, and altering the targets of proximity-based algorithms.

8.2 Threats to Validity and Reproducibility

Threats to *construct validity* stem from the connection between the practical measures employed to quantify theoretical aspects of the study. In this regard, we use standard DT procedures to quantify compiler defects, and link uncovered defects to static, measurable properties of the generated code. We also employ common metrics of effectiveness and efficiency that are standard practice throughout empirical software engineering research.

Threats to *internal validity* regard factors that could lead to confounding the causes of observed phenomena. In our study, the main obfuscating factor in regards to causality is the heavy reliance of KOTFUZZ on randomness. We take several measures to address the large degree of randomness inherent to fuzzing approaches. First, we individually assess each uncovered defect and analyze the components of the fuzzer involved in its generation. Through this procedure, we identify the root causes of generating compiler-crashing code and isolate the merits and disadvantages of different heuristics. Second, we ensure the fairness of the comparisons by sharing parameters between different configurations with default values from literature. We also perform the comparison using a single implementation of the tool, that relies on the same sampling process and genetic operators. Lastly, to assess the effectiveness and efficiency of our algorithms, we repeat independent runs 10 times, and report average values that are subject to standard statistical analyses.

Threats to *conclusion validity* affect the relation between the available data and the credibility of the conclusions we derive based on it. To this end, we base our analysis on over 200,000 generated files, and vary the essential hyperparameters of our algorithms to several sensible values. We additionally perform statistical tests to compare algorithms representative of their respective class. In particular, we base our performance conclusions on the application of the Wilcoxon signed-rank test, which is a statistical procedure that does not make any assumptions about the distribution of the underlying data.

Threats to *reproducibility* concern factors that might cause the application of the same research methods to result in significantly divergent or conflicting observations. To mitigate this, we supply the entire code base that implements KOTFUZZ [12], in addition to the entire set of generated files and adjacent preprocessed data [11]. We also provide extensive documentation to detail our tool, its configuration, and its applicability. To ensure that no environmental factors interfere with the fuzzer, we use containerization to isolate the dependency management and runtime of KOTFUZZ.

8.3 Future Work

On top of addressing the limitations underlined in Section 8.1 and exploring additional heuristics in the context of compiler fuzzing, there are several directions future work could follow to extend this study.

Compiler Integration The heuristic fitness criteria explored in this study work as methods to guide the generative sampling process towards promising areas of the Kotlin code space. In parallel to those methods, one could leverage compiler information directly into the search algorithm. The compiler service described in Chapter 5 provides an interface for this by allowing the fuzzer to query for files that trigger faults according to DT. However, the fuzzer does not currently integrate this information into the search process. Integration could follow different blueprints, such as maintaining code that uncovers faults as a permanent part of the population, or avoiding generating novel blocks that resemble known faulty code. This promotes generating more instances of the same bug, while the latter prevents it.

Defect Clustering at Runtime In its current implementation, KOTFUZZ only analyzes the generated files after the fuzzing process has terminated. We perform defect analysis automatically by parsing cached compiler messages for common patterns, and manually analyze files that do not fit known patterns. Heuristically performing this task at runtime could provide additional information to the fuzzer in such a way that KOTFUZZ could orient the search process away from proeminent clusters, thus better exploring the search space. This process could take several forms. For instance, the aforementioned compiler integration could be extended to incorporate additional clustering capabilities. Additionally, the embedding process could act as a mapping tool that allows the application of standard clustering algorithms without the need for compilation. Both options would incur significant computational overhead in relation to RS.

Integration with Mutation Fuzzing Stepanov et al. [79] introduced a mutation-based fuzzer for Kotlin, that alters input code in a sound and type-aware manner. Since the variation operators currently available in KOTFUZZ are both vastly different and less powerful than those in the mutation fuzzer, exploring the integration of the two could give rise to new, otherwise unattainable pieces of code. This integration could follow a two-step approach, where files generated through KOTFUZZ serve as input to the mutation fuzzer. Alternatively, a closer integration of the two methods could see KOTFUZZ use the mutation fuzzer as a powerful variation operator implemented in one of its GAs.

Specialized Embeddings The semantic proximity GAs rely on embeddings to guide the search process. Throughout this study, we rely on CODEBERT Feng et al. [25] as a tool for providing this critical piece of the algorithm. However, this choice might not be optimal. First, newer models have superseded CODEBERT on several ML4SE tasks, and leveraging these newer solutions might increase the quality of the embeddings. Second, CODEBERT is not specifically pre-trained for Kotlin-related tasks. Given the recent rise in popularity of Kotlin, there is likely sufficient available open-source data to either pre-train or fine-tune a large code model specifically for Kotlin. In addition to increasing the quality of the embeddings, such a model could prove useful for many other tasks centered around Kotlin.

Chapter 9

Conclusion

We proposed a generalizable three-pillar approach that intertwines grammar specification with semantic context and meta-heuristic search. Our method prunes semantically rich grammar productions and replaces them with context-aware samplable counterparts to drastically decrease the likelihood of generating invalid code. We structure the code that emerges from sampling the enriched grammar structure into a hierarchical representation based on scope and complexity. This representation forms the basis of an evolutionary framework that provides guidance to the sampling process.

Within this evolutionary framework, we introduced two classes of algorithms that are novel to the field of compiler fuzzing. Syntactic diversity optimization seeks to drive the population toward a diverse collection of files that exercise different combinations of language features. Semantic proximity optimization aims at nudging the population towards a pre-defined set of targets using an imperfect approximation of similarity. We used machine learning techniques to obtain a numerical interpretation of the semantics of code, and used this representation to derive notions of similarity. We proposed and implemented both single- and many-objective formulations for both heuristics, in addition to the literature-standard random sampling.

We analyzed the behavior and performance of the proposed approaches in an empirical analysis spanning 200,000 Kotlin files, which we analyzed through differential testing between the current `K1` compiler and the upcoming `K2` standard. Our results uncovered six previously unreported categories of bugs, five of which we reported to the Kotlin compiler developer team. The developers verified and replicated our instances on the current release of the Kotlin compiler. Compiler developers are working on fixing the reported issues, and have assigned target release dates for fixes within the current major version of the compiler. Our empirical analysis also provided new insight into the relations between expression simplicity and file complexity, distance metric and code distribution, and target selection and archive growth for each of the proposed heuristics. Finally, the performance analysis of our methods have shown that comparable algorithms do not significantly differ in the number of bugs they find or the rate at which they do so. Despite this, the driving mechanisms of our algorithms favor different code patterns, which in turn materialized in distinct bugs uncovered by each heuristic implementation.

Bibliography

[1] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M Memon. Exploiting the saturation effect in automatic random testing of android applications. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 33–43. IEEE, 2015.

[2] Matthew Amodio, Swarat Chaudhuri, and Thomas W Reps. Neural attribute machines for program generation. *arXiv preprint arXiv:1705.09231*, 2017.

[3] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *ACM Sigmod record*, 28(2):49–60, 1999.

[4] Andrea Arcuri. Theoretical analysis of local search in software testing. In *International Symposium on Stochastic Algorithms*, pages 156–168. Springer, 2009.

[5] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1):1–37, 2019.

[6] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

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

[8] Luca Ardito, Riccardo Coppola, Giovanni Malnati, and Marco Torchiano. Effectiveness of kotlin vs. java in android app development tasks. *Information and Software Technology*, 127:106374, 2020.

[9] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[10] 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, 2014.

[11] Georgescu Calin. Empirical Study Data for Test Program-Based Generative Fuzzing for Differential Testing of the Kotlin Compiler, August 2023. URL `https://doi.org/10.5281/zenodo.8221889`.

[12] Georgescu Calin. Fuzzer Implementation for Test Program-Based Generative Fuzzing for Differential Testing of the Kotlin Compiler, August 2023. URL `https://doi.org/10.5281/zenodo.8222995`.

[13] George Candea and Patrice Godefroid. Automated software test generation: some challenges, solutions, and recent advances. *Computing and Software Science*, pages 505–531, 2019.

[14] Kshitij Chauhan, Shivam Kumar, Divyashikha Sethia, and Mohammad Nadeem Alam. Performance analysis of kotlin coroutines on android in a model-view-intent architecture pattern. In *2021 2nd International Conference for Emerging Technology (INCET)*, pages 1–6. IEEE, 2021.

[15] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, 53 (1):1–36, 2020.

[16] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

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

[18] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 95–105, 2018.

[19] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.

[20] Kalyanmoy Deb and Kalyanmoy Deb. Multi-objective optimization. In *Search methodologies: Introductory tutorials in optimization and decision support techniques*, pages 403–449. Springer, 2013.

[21] Android Developers. Android's kotlin-first approach, 2019. URL `https://developer.android.com/kotlin/first`. Visited on 2022-12-29.

[22] Android Developers. Celebrating 5 years of kotlin on android, 2022. URL https://android-developers.googleblog.com/2022/08/celebrating-5-years-of-kotlin-on-android.html. Visited on 2022-12-29.

[23] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577, 2009.

[24] Joe W Duran and Simeon Ntafos. A report on random testing. In *ICSE*, volume 81, pages 179–183. Citeseer, 1981.

[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[26] Matheus Flauzino, Júlio Veríssimo, Ricardo Terra, Elder Cirilo, Vinicius HS Durelli, and Rafael S Durelli. Are you still smelling it? a comparative study between java and kotlin language. In *Proceedings of the VII Brazilian symposium on software components, architectures, and reuse*, pages 23–32, 2018.

[27] 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, 2011.

[28] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[29] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55 (1):119–139, 1997.

[30] Moheb R Girgis. Automatic test data generation for data flow testing using a genetic algorithm. *J. Univers. Comput. Sci.*, 11(6):898–915, 2005.

[31] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[32] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. Intelligent rest api data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 725–736, 2020.

[33] Bruno Góis Mateus and Matias Martinez. An empirical study on quality of android applications written in kotlin language. *Empirical Software Engineering*, 24(6):3356–3393, 2019.

[34] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[35] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[36] Mark Harman. Software engineering meets evolutionary computation. *Computer*, 44 (10):31–39, 2011.

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

[38] Mark Harman and Phil McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 73–83, 2007.

[39] Nikolas Havrikov and Andreas Zeller. Systematically covering input structure. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 189–199. IEEE, 2019.

[40] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[41] Tony Hoare. The verifying compiler: A grand challenge for computing research. In *International Conference on Compiler Construction*, pages 262–272. Springer, 2003.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

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

[44] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software testing techniques: A literature review. In *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*, pages 177–182. IEEE, 2016.

[45] JetBrains. Kotlin grammar specification, 2023. URL `https://kotlinlang.org/docs/reference/grammar.html`. Visited on 2023-06-17.

[46] Holland John. Adaptation in natural and artificial systems. *Ann Arbor*, 1975.

[47] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[48] Patrick Kreutzer, Stefan Kraus, and Michael Philippsen. Language-agnostic generation of compilable test programs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 39–50. IEEE, 2020.

[49] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[50] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.

[51] Meir M Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1979.

[52] Hareton KN Leung and Lee White. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301. IEEE, 1990.

[53] Yuwei Li, Shouling Ji, Chenyang Lv, Yuan Chen, Jianhai Chen, Qinchen Gu, and Chunming Wu. V-fuzz: Vulnerability-oriented evolutionary fuzzing. *arXiv preprint arXiv:1901.01142*, 2019.

[54] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 50(6):65–76, 2015.

[55] Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965.

[56] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proceedings of the ACM on Programming Languages*, 4 (OOPSLA):1–25, 2020.

[57] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[58] Hoang N Luong and Peter AN Bosman. Elitist archiving for multi-objective evolutionary algorithms: To adapt or not to adapt. In *Parallel Problem Solving from Nature-PPSN XII: 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part II 12*, pages 72–81. Springer, 2012.

[59] Valentin JM Manès, HyungSeok 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.

[60] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

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

[62] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[63] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[64] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, (3):223–226, 1976.

[65] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the c11/c++ 11 memory model. *ACM SIGPLAN Notices*, 48(6):187–196, 2013.

[66] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2): 58–62, 2005.

[67] Victor Oliveira, Leopoldo Teixeira, and Felipe Ebert. On the adoption of kotlin on android development: A triangulation study. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 206–216. IEEE, 2020.

[68] Stack Overflow. Stack overflow 2016 developer survey, 2016. URL `https://insights.stackoverflow.com/survey/2016`. Visited on 2022-12-29.

[69] Stack Overflow. Stack overflow 2022 developer survey, 2022. URL `https://survey.stackoverflow.co/2022/#most-popular-technologies-language`. Visited on 2022-12-29.

[70] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[71] Terence Parr. The definitive antlr 4 reference. *The Definitive ANTLR 4 Reference*, pages 1–326, 2013.

[72] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[73] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[74] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pages 419–428, 2014.

[75] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133*, 2019.

[76] Bart Selman and Carla P Gomes. Hill-climbing search. *Encyclopedia of cognitive science*, 81:82, 2006.

[77] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1): 27–29, 1996.

[78] Dominic Steinhöfel and Andreas Zeller. Input invariants. In *ESEC/FSE 2022: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 583—-594, 2022.

[79] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. Type-centric kotlin compiler fuzzing: Preserving test program correctness by preserving types. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 318–328. IEEE, 2021.

[80] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[81] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 849–863, 2016.

[82] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 294–305, 2016.

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

[84] Alan M Turing and J Haugeland. Computing machinery and intelligence. *The Turing Test: Verbal Behavior as the Hallmark of Intelligence*, pages 29–56, 1950.

[85] Mubarak Albarka Umar. Comprehensive study of software testing: Categories, levels, techniques, and types. 2020.

[86] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[87] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[88] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[89] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[90] Andy Zaidman, Bart Van Rompaey, Arie Van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3): 325–364, 2011.

[91] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–361, 2017.

[92] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. *ACM sigmod record*, 25(2):103–114, 1996.

[93] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated test program generation for an industrial optimizing compiler. In *2009 ICSE Workshop on Automation of Software Test*, pages 36–43. IEEE, 2009.

[94] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–970, 2020.

Appendix A

Acronyms

ART Adaptive Random Testing. 7

AST Abstract Syntax Tree. 13, 14, 16

AUC Area-Under-Curve. 52, 63

BB Black-Box. 6, 29, 63

CFG Context-Free Grammar. v, 10–13, 15–17, 22–24, 29, 30, 34, 41, 46, 67

DAG Directed Acyclic Graph. 18–21

DPVC Data Packed Volume Container. 42

DT Differential Testing. 7, 12, 13, 43, 44, 51–53, 64, 68, 69

DVC Data Version Control. 43

EA Evolutionary Algorithms. 9

EMI Equivalence Modulo Inputs. 13

GA Genetic Algorithms. 9, 10, 17, 25, 26, 30–38, 42, 43, 45, 46, 49, 65, 66, 68, 70

GB Grey-Box. 6

GS Global Search. 9

HC Hill Climbing. 9

IDE Integrated Development Environment. 48

IQR Inter-Quartile Range. 55, 57–60

IT Integration-Level Testing. 6

JVM Java Virtual Machine. 6, 65

LS Local Search. 9

LSTM Long Short-Term Memory. 14, 15

MCMC Markov Chain Monte Carlo. 13

ML Machine Learning. 34, 36, 44

ML4SE Machine Learning for Software Engineering. 34, 70

MO Many-Objective. 32, 33, 37, 38, 41, 46, 49, 59, 62

MODGA Many-Objective Diversity Genetic Algorithm. v, 49, 50, 53, 56–59, 62, 63

MOPGA Many-Objective Proximity Genetic Algorithm. v, vi, 49, 50, 53, 59–62

NAM Neural Attribute Machines. 14

NLP Natural Language Processing. 34, 35

NN Neural Network. 34, 35, 60

OO Object-Oriented. 22, 25

OOM Out-Of-Memory. 56, 57, 61, 64, 65

OS Operating System. 7, 53

RS Random Sampling. v, vi, 29, 49, 50, 53–56, 58–60, 62–66, 70

RT Random Testing. 7, 9

SA Simulated Annealing. 9

SBSE Search-Based Software Engineering. 8, 9

SBST Search-Based Software Testing. 8, 9

SDLC Software Development Lifecycle. 6–8

SO Single-Objective. 30–33, 37, 38, 41, 46, 49

SOA Service-Oriented Architecture. 39

SODGA Single-Objective Diversity Genetic Algorithm. v, vi, 49, 50, 53, 56–58, 62–64, 66, 68

SPE Skeletal Program Enumeration. 13, 14

ST System-Level Test. 6, 7

STPGA Single-Target Proximity Genetic Algorithm. v, vi, 49, 50, 53, 59–64, 68

SUT System Under Test. 6–8, 10

TCE Type-Centric Enumeration. 14

TS Tabu Search. 9

UB Undefined Behavior. 12

UT Unit-Level Testing. 6

WB White-Box. 6

WS Whole-Suite. 38

WSPGA Whole-Suite Proximity Genetic Algorithm. v, 49, 50, 53, 59, 60, 62