KotSuite: Unit Test Generation for Kotlin Programs in Android Applications

Feng Yang, Qi Xin
School of Computer Science
Wuhan University
Wuhan, China
{yangfeng, qxin}@whu.edu.cn

Zhilei Ren
School of Software
Dalian University of Technology
Dalian, China
zren@dlut.edu.cn

Jifeng Xuan *
School of Computer Science
Wuhan University
Wuhan, China
jxuan@whu.edu.cn

***Abstract*—Unit testing plays a pivotal role in safeguarding functional requirements and supporting the maintainence during the development of Android applications. The Kotlin programming language emerges in developing Android applications since Kotlin is considered due to its simplicity, safety, and interoperability with Java. It is time-consuming to manually write unit test cases for Kotlin programs. To mitigate labor costs, automated unit test generation techniques are developed. However, existing tools of unit test generation, such as EvoSuite and Randoop, are primarily optimized for traditional Java projects. This make these tools incapable of generating test cases for Kotlin projects in Android. In this paper, we introduce KotSuite, an automated tool of unit test generation for Kotlin applications in Android. KotSuite employs static analysis techniques to extract the syntactic structure of the target methods and transforms the syntactic structure into control flow representations. Then, KotSuite automatically generates a suite of test cases using a genetic algorithm and test reuse. We evaluate KotSuite on eight modules from four widely-used and open-source Kotlin projects in Android. Experimental results show that KotSuite can effectively generate high-coverage test cases with average line coverage of 66.0% and branch coverage of 60.4%. Additionally, test reuse in KotSuite can improve the efficiency of test case generation by 22%.**

***Index Terms*—Kotlin programs, test case generation, Android applications, test reuse, test tools**

I. Introduction

Android applications are widely used in mobile devices. Developers are required to meet high standards to avoid performance issues and security vulnerabilities [1], [2]. Unit testing can help developers verify the behavior of individual components and catch bugs in the early stage. The unit testing is crucial for maintaining the high availability of applications and for reinforcing the security.

However, it is time-consuming to manually write comprehensive unit tests. Automated techniques of test case generation utilize optimization algorithms to generate unit tests based on predefined criteria [3]. These approaches can accelerate the process of test development and save the time of developers. Previous studies [3]–[5] have shown that automated test generation can improve testing efficiency and enhance the reliability by increasing the consistency and accuracy of test cases [6].

*Jifeng Xuan is the corresponding author. This work was partially supported by the National Natural Science Foundation of China under the grant number 62202344 and the OPPO Research Fund.

The shift from Java to the Kotlin programming language in Android application development is reshaping the mobile programming landscape. Since 2017, Kotlin has been the recommended language for Android applications by Google [7]. This benefits from the concise syntax, enhanced safety features, and interoperability with Java. The support of Kotlin for modern programming paradigms, such as extension functions and null safety, improves code readability and reduces common programming errors, especially those related to null references, which are among the leading causes of application crashes. Furthermore, the integration of coroutines in Kotlin offers developers an efficient and readable approach to asynchronous programming. This is a crucial aspect of Android development due to the single-threaded UI framework. All these features in Android applications make existing tools of test generation not work for Android Kotlin programs.

Existing automated tools of test case generation, such as EvoSuite [8] and Randoop [9], are primarily designed for Java applications. Since the unique language features in Kotlin and Android are different from Java programs, these existing tools cannot automatically generate test cases for Kotlin programs. First, the design of existing tools of test generation heavily rely on the features of Java programs. For instance, handling null safety in Kotlin requires a different approach from that in Java. Java-based tools are generally not equipped to recognize the null-safe types of Kotlin and may miss potential corner cases in their test coverage. Additionally, features like extension functions, which add methods to existing classes without altering their source code, can lead to unexpected behaviors. Second, the complexity of Android applications cannot be directly handled in existing tools like EvoSuite [8] and Randoop [9]. The architecture of Android introduces unique runtime considerations, such as activity life cycle management and asynchronous tasks. Java-based testing tools typically lack built-in mechanisms to account for these nuances, making it difficult to capture issues arising from life cycle events or concurrent operations. Without adaptation for language features in KotSuite and the architecture of Android, these tools may leave critical issues for compiling and coverage. This adds risks to the reliability and stability of the final result. Given these limitations, there is a demand for Kotlin-based tools of test case generation for Android applications.

In this paper, we propose KotSuite, an automated tool of unit test generation for Kotlin programs in Android based on a genetic algorithm and test reuse. KotSuite mainly consists of two phases, the analysis phase and the test generation phase. First, in the analysis phase, KotSuite incorporates customized handling for the language syntax in Kotlin, such as extension functions and null safety, and Android-specific runtime behaviors like activity life cycle management and coroutine handling. To support compatibility and test coverage, KotSuite can model the application functions in both Kotlin and Java by building a detailed Control Flow Graph (CFG). Second, in the test generation phase, based on CFGs, KotSuite leverages a genetic algorithm and test reuse to automatically generate unit test cases. In the genetic algorithm, KotSuite employs test mocking to generate test mocks for internal Android objects. KotSuite employs the test reuse to improve the efficiency of test generation to avoid the time cost of similar methods under test. The algorithm of test reuse allows KotSuite to repurpose previously successfully-generated test cases. The test reuse algorithm can reduce the computational overhead associated with generating all test cases.

We evaluate KotSuite on eight modules from four widely-used and open-source Android projects. Experimental results show that KotSuite can effectively generate high-coverage test cases with the average line coverage of 66.0% and the branch coverage of 60.4%. Additionally, test reuse in KotSuite can improve the efficiency of test case generation by 22%, and the average time of generating unit test cases for a single method is 4.2 seconds. The evaluation on eight modules also shows that KotSuite can support both Java and Kotlin code for Android applications.

This paper makes the following contributions:

- We propose an automated approach, KotSuite, which employs a genetic algorithm, test mocking, and test reuse to generate test cases for syntax features of Kotlin programs in Android. To the best of our knowledge, this is the first work of unit test generation for Kotlin projects in Android.
- We design a method of test case reuse to improve the efficiency of test generation.
- We evaluate our approach on eight modules from four open-source Kotlin projects.
- We implement an Android Studio plugin and a command-line tool for KotSuite. The plugin enables developers to quickly get started while the command-line tool facilitates the integration of KotSuite into the CI/CD. The tools and the experimental data are publicly available.[1]

II. Background and Motivation

A. Background

Android applications, along with their corresponding tests, are primarily written in either Java or Kotlin, the two most widely used programming languages for Android development. These tests can be executed in different environments depending on the type of testing being performed, either on the Java Virtual Machine (JVM) or directly on an Android device [10]. JVM tests are run on a simulated environment or a local machine, offering developers a faster feedback loop for code correctness and behavior at the unit or component level. Device tests are run on actual Android devices or emulators and can encompass a broader range of test types, including unit, integration, system, and GUI (graphical user interface) tests.

In an Android Kotlin project, the identification of relevant classes and methods for unit testing is crucial to ensure effective test coverage. Focal classes are those that contain the core functional logic of the application, which directly contributes to the business behavior. These classes are the primary targets for testing. On the other hand, certain classes are excluded from testing, such as anonymous classes, abstract classes, data classes, and non-public classes, as they either serve as auxiliary structures or do not influence the core application behavior. Similarly, focal methods are the methods within these focal classes that encapsulate the application's essential logic. Methods such as constructors, anonymous methods, getter and setter methods, and life cycle methods are excluded from testing because they either do not perform significant business logic or are automatically managed by the Android framework. Therefore, focal methods are selected by filtering out those that fall into these excluded categories, focusing on methods that require testing for their functional contributions to the application.

B. Motivation

Automated unit test generation has become increasingly important in modern software development, especially with the rapid adoption of continuous integration and delivery practices. Android applications, particularly those written in Kotlin, present unique challenges in this area due to their intricate interactions with the Android framework and the distinct language features of Kotlin, such as null safety, extension functions, and coroutines. These features can complicate the process of generating effective test cases, as they often require careful handling to accurately represent application behavior.

Existing tools such as EvoSuite [8] and Randoop [9] primarily focus on Java and do not fully address the particularities of Kotlin and Android. Furthermore, recent advances in deep-learning-based test generation, as seen in tools like A3Test [11] and ChatUniTest [12], have shown promise but still face limitations in handling the specific requirements of Android applications and Kotlin syntax. Moreover, Android unit tests must run on the JVM, which necessitates using mock objects to isolate Android runtime dependencies. Without efficient mocking and test reuse mechanisms, the testing process can be both time-consuming and resource-intensive.

The code example of Fig. 1 comes from Android architecture components at Google.[2] The `fetchUserProfile`

[1]Tools and data are publicly available at https://github.com/Maple-pro/kotsuite-opendata.

[2]https://github.com/android/architecture-components-samples.

```kotlin
class UserRepository(private val apiService: ApiService) {

    // Extension function to check for null and empty strings
    private fun String?.isValid(): Boolean {
        return this != null && this.isNotEmpty()
    }

    // Suspending function to fetch user profile
    suspend fun fetchUserProfile(userId: String?): Result<User> {
        if (!userId.isValid()) {
            return Result.failure(
                IllegalArgumentException("Invalid user ID"))
        }
        return try {
            // This is a network call
            val user = apiService.getUserProfile(userId)
            if (user != null) {
                Result.success(user)
            } else {
                Result.failure(
                    NullPointerException("User data is null"))
            }
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}
```

Fig. 1. Motivating example of test case generation for Kotlin programs. This function fetches a user profile data asynchronously using coroutines. The function is designed to work within the Android architecture, including null safety checks and extension functions.

function fetches a user's profile data asynchronously using coroutines. The function includes null safety checks, extension functions, and is designed to work within Android's architecture, specifically dealing with nullable data types and coroutine contexts. Traditional tools like EvoSuite and Randoop faces difficulties as follows:

- **Kotlin-specific null safety**. Traditional tools lack built-in support for null safety and extension functions in Kotlin.
- **Coroutines**. Traditional tools cannot handle `suspend` functions or coroutine context, which are essential in Kotlin for asynchronous operations.
- **Android-specific Behavior**. Testing on JVM with Android components often requires mock objects, especially for networking, which needs to be handled by mock library.

Motivated by the fact that existing tools of test generation cannot handle Android context, we design a new tool of test case generation for Kotlin programs, called KotSuite.

III. The KotSuite Approach

A. Overview

Fig. 2 illustrates the overall architecture of the proposed approach KotSuite. The KotSuite framework is designed to automatically generate unit test cases for Android Kotlin projects by integrating both static analysis and evolutionary algorithms within a customizable tool chain. As shown in Fig. 2, KotSuite consists of two major phases: the analysis phase and the test generation phase.

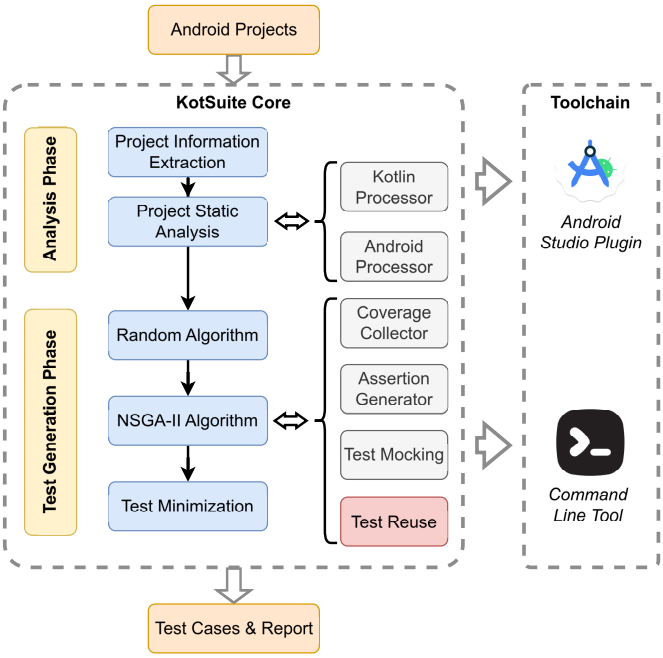

Fig. 2. Overview of the KotSuite approach.

In the analysis phase, KotSuite begins by extracting project-specific information, including the structure and dependencies required for effective test case generation. This phase incorporates a customized project static analysis stage, which leverages the Kotlin processor and Android processor to accommodate Kotlin-specific syntax and Android framework constructs. This allows KotSuite to handle Kotlin language features and Android-specific elements, ensuring compatibility and efficiency when analyzing Kotlin-based Android applications.

The test generation phase initiates with two primary methods for test case generation. The first method uses a Random Algorithm to produce an initial pool of test cases, serving as a baseline for further refinement. The second, more sophisticated method utilizes an NSGA-II Algorithm (a multi-objective genetic algorithm) [13], aiming to optimize test case generation for higher coverage and efficiency. The NSGA-II Algorithm iteratively refines the initial test cases based on fitness functions such as line and branch coverage, thereby improving test quality and coverage metrics.

Throughout the test generation process, KotSuite incorporates several specialized modules to enhance the effectiveness and quality of generated tests. These include a coverage collector to monitor coverage metrics, an assertion generator to generate assertions within the tests, and a test mocking module to simulate dependencies and isolate the focal methods under test. Additionally, a test reuse component is integrated to leverage previously generated high-quality test cases, reducing redundant computations and increasing test generation efficiency.

Upon completion of test generation, a test minimization step is employed to filter out redundant or ineffective test

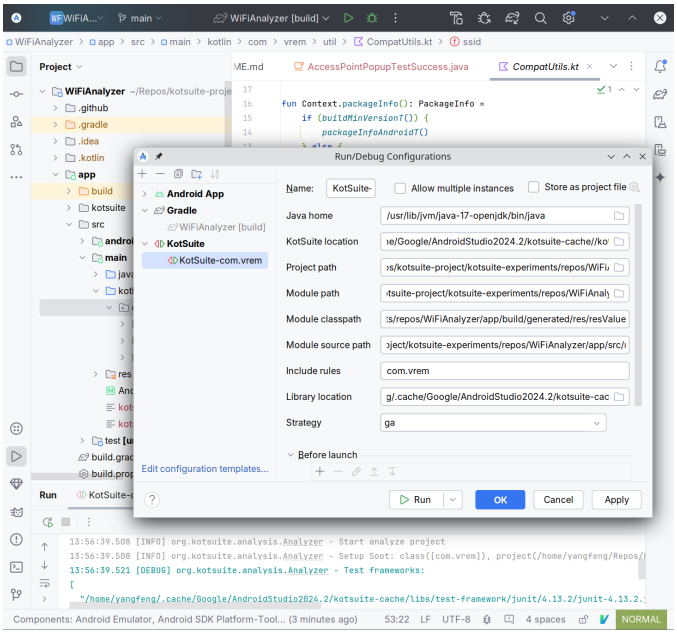

Fig. 3. Screenshot of the Android Studio Plugin of KotSuite.

cases, resulting in a final set of optimized test cases. KotSuite outputs these tests along with a comprehensive test cases and reports, summarizing the generated test coverage and efficiency metrics.

KotSuite offers a dual-interface approach to testing, with both an Android Studio plugin [14] and a command-line tool. Fig. 3 shows the user interface of the plugin. The Android Studio plugin provides an intuitive interface that integrates seamlessly with the Android development environment, allowing developers to quickly adopt KotSuite and start generating tests with minimal setup. It allows developers to interactively configure and execute test cases, view coverage results, and refine test parameters in real-time, facilitating a smoother workflow [15]. For teams that require flexibility in how they incorporate testing tools into their development pipeline, the command-line tool of KotSuite offers robust options for automation and integration into CI environments. The command-line tool enables KotSuite to be incorporated into scripted workflows, allowing for automatic test generation, execution, and reporting as part of the CI/CD process [16]. This capability is essential for development teams that aim to maintain high-quality code through automated testing practices and need testing tools that can adapt to various environments and workflows [17]. By providing both an IDE plugin and a command-line tool, KotSuite enhances accessibility and adaptability, catering to individual developers, as well as teams with complex integration needs, making it a versatile solution in automated testing for Android.

B. Analysis Phase

The Analysis Module in KotSuite is specifically designed to conduct static analysis on Android applications, providing foundational insights for effective automated testing. This module processes the bytecode of the target program and its dependencies, leveraging the capabilities of the Soot framework [18] for Java bytecode analysis. Soot, a widely used open-source toolkit, facilitates extensive analysis and transformation of Java applications by converting Java class files into an intermediate representation. Through Soot, the Analysis Module extracts critical structures such as the abstract syntax tree (AST) and the control flow graph (CFG) of target functions, which are essential for understanding code logic and flow at a granular level.

The control flow graph provides a map of all potential execution paths within each function, which is crucial for identifying test cases that cover a variety of scenarios. By outputting the results in Jimple format [19], a simplified three-address code representation used by Soot, the Analysis Module allows the genetic algorithm module in KotSuite to process and manipulate these structures effectively. Jimple is particularly advantageous for optimization and static analysis, as it abstracts complex Java bytecode operations into a simpler, more manageable format, making it easier for KotSuite to generate high-coverage test cases efficiently.

With the robust capabilities of Soot in static analysis, including inter-procedural analysis and data flow analysis, the analysis phase can detect and account for unique Kotlin and Android-specific features in the bytecode, laying the groundwork for comprehensive test generation. This systematic approach allows KotSuite to generate tests cases that cover diverse paths in the code and consider the specific behaviors and constructs present in Kotlin-based Android applications.

C. Test Generation Phase

The test generation phase in KotSuite is responsible for generating high-quality unit test cases that achieve substantial code coverage for Android Kotlin projects. This phase is structured into several key steps and leverages multiple supporting modules to ensure that generated test cases are efficient, effective, and capable of maximizing coverage across the codebase.

Four key modules support this process: coverage collector, assertion generator, test mocking, and test reuse. The coverage collector module, implemented using tools like JaCoCo and Java Instrumentation, collects detailed coverage information on each test case, which is essential for evaluating fitness in the NSGA-II Algorithm. The assertion generator module automatically creates assertions based on expected program outputs, allowing generated test cases to verify program behavior and detect potential errors. Test mocking enables the simulation of dependencies within test cases, making it possible to test methods that interact with external components or services, which is crucial for thorough coverage of Android applications. Finally, the test reuse module facilitates the reuse of previously generated high-quality test cases, reducing redundant iterations and improving the efficiency of test generation.

The unit-level test code is generated heuristically. Based on the abstract syntax structure of the bytecode, a preliminary test code skeleton is constructed, which essentially consists of a

sequence of API calls for the function under test. To create an initial test code, the module encodes the bytecode and applies a heuristic optimization algorithm to form a program sequence, or test code segment. The optimized fitness function comprises line coverage and branch coverage.

A random heuristic algorithm [20]–[22] generates the initial test case set, where each test case represents a sequence of function calls on the function under test. To increase coverage, the test code undergoes evolutionary generation. Once the preliminary test code is created, further diversity is introduced to improve coverage. An initial population of test cases is formed based on the bytecode encoding. The evolutionary computation model iteratively optimizes the test cases according to the fitness function, yielding higher coverage. During evolution, hard-to-cover nodes are identified, and the process iterates until no new test code segments can be generated.

In test case generation, each individual in the evolutionary algorithm represents a test case set, with the optimization objective of maximizing code coverage. The optimization targets within the algorithm include line coverage and branch coverage, using the NSGA-II algorithm [17] for evolution.

The test minimization stage in KotSuite is an essential post-processing step that follows the NSGA-II Algorithm in the test generation Phase. Positioned as the final operation in the test generation pipeline, this stage aims to refine the generated test suite by identifying and removing redundant test cases, thereby improving the efficiency of regression testing [23] and reducing execution time without compromising coverage. The stage works closely with the coverage collector, which records the line and branch coverage achieved by each test case. By analyzing the coverage data collected, the test minimization stage detects duplicate test cases that cover identical execution paths or reach the same set of lines in the code. This comparison allows it to systematically prune test cases that do not contribute new coverage information, resulting in a more concise and meaningful test suite. This method leverages techniques similar to those described in [8] for reducing test redundancy through coverage-based filtering.

1) Coverage Collector: The coverage collector module in KotSuite plays a pivotal role in gathering coverage data for generated test cases. Positioned within both the NSGA-II algorithm and test minimization stages, this module provides essential feedback that guides the evolutionary search process and assists in pruning redundant test cases. Specifically, the coverage collector captures information on line and branch coverage, which is then utilized in three key ways: calculating the fitness of individuals in the NSGA-II Algorithm, supplying coverage details to the test minimization stage to identify duplicate tests, and generating a comprehensive coverage report at the end of test case generation.

To implement coverage collection, this module employs JaCoCo [24] and Java Instrumentation. JaCoCo is an open-source library for Java code coverage analysis, which provides detailed metrics on line and branch coverage during execution. Java Instrumentation, on the other hand, enables dynamic modification of bytecode at runtime, allowing for a more flexible and customizable approach to tracking execution paths. By integrating these tools, KotSuite effectively captures the necessary coverage data to optimize the generated test suite both in terms of coverage completeness and efficiency.

2) Assertion Generator: The assertion generation module is an integral component of the KotSuite framework, responsible for automatically generating assertions within test cases. Assertions serve as key elements in validating the correctness of the system under test, ensuring that the actual behavior of the application conforms to its expected behavior. The implementation of the assertion generation module is based on a combination of dynamic execution [25] and Java instrumentation API. It identifies key conditions and properties that must hold true during test execution. The module analyzes the dynamic execution of the code, particularly monitoring method invocations, variable states, and data flow throughout the program. Using Java instrumentation, the module can inject hooks into the bytecode, enabling it to observe and manipulate the program's execution in real-time. Based on this execution data, the module generates assertions that assert the correctness of specific values, control flow outcomes, and system behaviors.

3) Test Mocking Module: The test mocking module in KotSuite is essential for creating mock objects that simulate dependencies within generated test cases. Positioned within the test generation phase, this module enables KotSuite to handle Android-specific dependencies by generating mock instances, thereby isolating the target methods from the Android runtime environment. This isolation is crucial because Android unit tests are executed on the JVM rather than on an actual Android device or emulator, where certain Android APIs are not available. Consequently, using mock objects allows KotSuite to test Android application logic in isolation from platform-specific implementations, ensuring that generated test cases achieve comprehensive coverage while remaining compatible with the JVM-based test environment.

The test mocking module is implemented using MockK [26], a widely used library for creating mock objects in Kotlin. MockK offers powerful features, including mocking of classes, functions, and coroutines, which are frequently used in Android applications. By leveraging MockK, KotSuite can simulate complex dependencies and interactions, such as database operations, network requests, and lifecycle-based events, without requiring access to actual Android runtime components. To enable the creation of mock objects at the bytecode level using the MockK framework, KotSuite creates a customized library named JmockK [27]. The JmockK library facilitates seamless integration between bytecode-level analysis and mock object generation, allowing KotSuite to generate mock objects that can interact directly with the bytecode representations of Android application classes. By leveraging JmockK, KotSuite ensures that mock objects are compatible with the bytecode manipulation required for thorough and accurate test generation.

4) Test Reuse Module: During the process of generating test code segments, KotSuite encodes and indexes high-quality test

Algorithm 1 Algorithm of test reuse in KotSuite

Input TargetFunction, ExternalVaribales, TestCaseLibrary
Output Updated TestCaseLibrary and generated test cases

```
function GENERATETESTCASES:
    Initialize: InitialPopulation ← []
    for each testCase in TestCaseLibrary do
        if Match(TargetFunction.Features,
                                testCase.FunctionFeatures)
            parameters ← ExtractParameters(testCase)
            apiCalls ← ExtractAPICalls(testCase)
            newTestCase ← AssembleTestCase(
                                parameters, apiCalls)
            InitialPopulation.append(newTestCase)
    end for
    if InitialPopulation.Size < POPULATION_SIZE then
        InitialPopulation ← GenerateRandomPopulation()
    end if
    for iteration = 1 to MAX_ITERATIONS do
        ApplyGeneticAlgorithm(InitialPopulation)
        for each testCase in InitialPopulation do
            if IsHighQuality(testCase) then
                TestCaseLibrary.append(testCase)
            end if
        end for
        if CheckStopCondition() then
            Break
        end if
    end for
end function
```

```kotlin
class UserRepositoryTest {

    private val apiService = mockk<ApiService>()
    private val userRepository = UserRepository(apiService)

    // test fetchUserProfile with valid userId
    // returns user data
    @Test
    fun test_fetchUserProfile_1() = runBlocking {
        // Mock data
        val validUserId = "12345"
        val mockUser = User("xxx", "yyy", validUserId)

        // Mock apiService to return a user when called
        coEvery {
            apiService.getUserProfile(validUserId)
        } returns mockUser

        // Execute function
        val result = userRepository
            .fetchUserProfile(validUserId)

        // Validate the result
        assertTrue(result.isSuccess)
        assertEquals(mockUser, result.getOrNull())
    }

    // test fetchUserProfile with null userId
    // throws IllegalArgumentException
    @Test
    fun test_fetchUserProfile_2() = runBlocking {
        // Execute function with a null userId
        val result = userRepository.fetchUserProfile(null)

        // Validate the result
        assertTrue(result.isFailure)
        assertTrue(result.exceptionOrNull()
            is IllegalArgumentException)
    }
}
```

Fig. 4. Two test cases generated by KotSuite for the example in Fig. 1.

cases that have already been produced, establishing a reusable test dataset that can streamline future test generation tasks. This approach allows KotSuite to leverage previously validated test cases, saving resources and reducing the redundancy associated with generating new tests from scratch. By building a test reuse dataset, KotSuite enhances both the efficiency of test case generation and the overall test coverage across diverse application modules.

The primary objective of test case reuse is twofold: to accelerate the generation process by reusing pre-validated test patterns and to broaden the scope of testing by reapplying effective test cases to similar functions or components. Reusing test cases is particularly valuable in complex applications, where high-quality tests often reveal intricate control flows and edge cases that may be relevant across multiple modules. Studies have shown that test reuse can improve coverage while reducing the time required for test generation, making it a critical strategy in automated testing for large projects.

By incorporating this reuse strategy, KotSuite optimizes the test creation pipeline, allowing developers to focus resources on generating unique test cases only when necessary. This approach enhances efficiency and contributes to the robustness and reliability of the final software product.

Algorithm 1 shows how the test reuse algorithm is applied in the genetic algorithm. The input of the algorithm contain three parameters, including `TargetFunction` (the focal method to be tested), `ExternalVariabales` (external variables on which the function depends), and `TestCaseLibrary` (library of high-quality, reusable test cases).

The algorithm iterates through `TestCaseLibrary` to identify previous generated test cases with function features matching `TargetFunction`. The `FunctionFeature` component encapsulates key characteristics of the target function to be tested, such as the external variables used within the function and its parameters. These features are extracted during the static analysis phase and serve as input for the test generation process, helping the genetic algorithm focus on relevant aspects of the function. When a match is found, it extracts the input parameters and API call sequence from the matching test cases to create new test cases in the `InitialPopulation`. If the size of the `InitialPopulation` is less than `POPULATION_SIZE`, the algorithm generates new test cases by random strategy util the size reaches the target.

The algorithm then iterates through a genetic algorithm iteration, refining the `InitialPopulation` by applying selection, crossover, and mutation operations. The `IsHighQuality()` function evaluates whether the generated test cases meet a threshold of 60% coverage. This threshold applies to both line and branch coverage and ensures that only tests that contribute meaningfully to coverage are retained in the final test suite. Following each iteration, high-quality test cases are appended to `TestCaseLibrary`, expanding the library with validated, reusable test cases.

The algorithm terminates when a predefined stopping condition is met. The output is an updated `TestCaseLibrary`, now enriched with a broader set of optimized test cases

that can support future test generation tasks. This approach improves efficiency by reusing existing, validated test cases where applicable, while ensuring high coverage and robustness through targeted genetic optimization. Fig. 4 shows two test cases that are automated generated by KotSuite for the method from the example in Fig. 1.

D. Implementation

The KotSuite framework primarily utilizes the following open-source libraries.

- **Soot**.[3] This library enables program static analysis, providing foundational support for static analysis capabilities in KotSuite.
- **JaCoCo**.[4] This library powers the coverage collector module in KotSuite, enabling the collection and calculation of test case coverage metrics.
- **MockK**.[5] KotSuite builds upon this library by developing the JMockK library[6] to mock code dependencies, allowing accurate unit testing in the Java Virtual Machine environment.
- **Fernflower**.[7] This library facilitates bytecode-to-Java conversion within KotSuite, producing final Java test case outputs from generated bytecode.

IV. Evaluation Setup

A. Research Questions

We design three Research Questions (RQs) to evaluate the effectiveness and efficiency of KotSuite.

RQ1. How effective is KotSuite in generating unit test cases for real-word Android Kotlin projects? KotSuite aims to provide a framework of test generation specifically tailored for Android Kotlin projects. This RQ seeks to determine whether KotSuite can fulfill its intended role as a practical and effective test generation tool.

RQ2. Can the genetic algorithm and test mocking components improve the coverage of generated test cases? KotSuite generates high coverage test cases by incorporating a genetic algorithm and test mocking components specifically designed to optimize test generation coverage. This RQ is designed to check the effectiveness of these components in real-world scenarios.

RQ3. Does the test reuse algorithm in KotSuite enhance the efficiency of test case generation? In automated test generation, efficiency is a key factor, especially when large-scale applications require extensive and complex test cases. This research question addresses the need to evaluate whether this approach effectively improves the efficiency of test generation without sacrificing the quality of coverage.

[3]https://soot-oss.github.io/soot/
[4]https://www.eclemma.org/jacoco/
[5]https://mockk.io/
[6]https://mvnrepository.com/artifact/io.github.Maple-pro/JMockK
[7]https://github.com/fesh0r/fernflower

B. Experiment Setup

To assess the effectiveness and efficiency of KotSuite, we conducted an experiment in which we investigated the code coverage of test cases generated by KotSuite as well as the time cost of test case generation. Specifically, we compared the impact of different modules within KotSuite on both test case coverage and generation efficiency. To perform the study, we selected the four Android applications with the highest number of functions to be tested from different categories on F-Droid [28], all of which are also available on GitHub [29], each from different category. Table I summarizes the properties of these case study subjects. The CineLog project is purely written in Java and does not contain any Kotlin code. This is to verify that KotSuite can be applied to both Kotlin and Java programs. The experiments were conducted on a system with 64GB of memory.

For each Android project, we randomly selected two modules for the experiment based on the number of focal methods. The number of focal classes and focal methods in each module is shown in Table II.

The genetic algorithm is influenced by a great number of parameters [30]. In this experiment, we employed a genetic algorithm to generate unit test cases, with the goal of optimizing code coverage. The key parameters for the genetic algorithm were set as follows: the mutation rate was set to 0.5, meaning that half of the selected individuals underwent mutation during each generation, promoting diversity in the population and enabling the algorithm to explore a wider range of solutions. The crossover rate was set to 0.5, ensuring that half of the pairs of selected individuals underwent crossover to combine their genetic information and potentially generate better offspring. This balance between mutation and crossover is intended to maintain diversity while also fostering the generation of higher-quality solutions. The maximum number of iterations was set to 10, meaning that the algorithm would terminate after 10 generations, ensuring a sufficient amount of exploration while preventing excessive computational cost.

To assess the effectiveness of the genetic algorithm, the fitness function was designed to optimize test case coverage, specifically focusing on line and branch coverage as the primary objectives. The population size was set to 100, ensuring a large enough pool of candidates to effectively explore the solution space while maintaining manageable computational complexity. The algorithm was initialized with a randomly generated population of test cases, and each generation was evaluated based on its ability to cover untested paths in the code. The test cases evolved over successive generations through selection, crossover, and mutation, with the aim of improving coverage and ensuring the robustness of the generated tests.

V. Evaluation Results

A. RQ1. How effective is KotSuite in generating unit test cases for real-word Android Kotlin projects?

To evaluate the effectiveness of KotSuite in generating unit test cases for real-world Android Kotlin projects, we con-

TABLE I
OVERVIEW OF THE FOUR ANDROID APPLICATIONS IN THE EVALUATION

Application	Category	#Stars	#Focal classes	#Focal methods	Percentage of Kotlin code
Ad-Free	Internet	273	241	1380	99.3%
GPSTest	Navigation	1775	380	1925	68.7%
CineLog	Multimedia	47	288	1758	0%
WiFiAnalyzer	Connectivity	3483	276	1708	97.3%

TABLE II
LINE COVERAGE AND BRANCH COVERAGE OF TEST CASES GENERATED UNDER DIFFERENT APPROACHES (LINE COVERAGE / BRANCH COVERAGE)

Application	Module	#Focal classes	#Focal methods	KotSuite *	KotSuite variant		
					Without GA	Without reuse	Without mocking
Ad-Free	ch.abertschi.adfree.detector	46	255	53.4% / 54.9%	37.5% / 22.2%	53.4% / 54.9%	9.2% / 12.4%
	ch.abertschi.adfree.model	25	201	44.4% / 64.3%	28.7% / 19.0%	44.4% / 64.3%	10.0% / 23.1%
GPSTest	com.android.gpstest.util	7	39	83.2% / 78.4%	61.9% / 66.6%	83.2% / 78.4%	6.9% / 6.4%
	com.android.gpstest.io	8	51	61.1% / 52.9%	52.6% / 44.7%	61.1% / 52.9%	9.4% / 8.1%
CineLog	com.ulicae.cinelog.data.dao	69	689	68.5% / 56.6%	63.4% / 50.9%	68.5% / 56.6%	6.8% / 5.2%
	com.ulicae.cinelog.utils	18	57	76.1% / 62.4%	59.3% / 36.9%	76.1% / 62.4%	13.3% / 10.5%
WiFiAnalyzer	com.vrem.wifianalyzer.wifi.graphutils	20	133	75.5% / 73.2%	52.8% / 36.8%	75.7% / 73.2%	11.2% / 19.8%
	com.vrem.wifianalyzer.wifi.model	65	261	79.5% / 64.5%	59.8% / 44.6%	79.5% / 64.5%	9.2% / 7.6%
Average	-	-	-	66.0% / 60.4%	53.5% / 40.4%	66.0% / 60.4%	8.6% / 10.2%

*The percentage on the left indicates the line coverage while the percentage on the right indicates the branch coverage.

ducted experiments on eight Android Kotlin modules. These modules were selected from a variety of Android applications to ensure diversity in the test cases generated. For each module, we utilized KotSuite to automatically generate test cases and then measured both the line coverage and branch coverage achieved by the generated test cases.

The results of the experiment are summarized in Table II, where we present the average line and branch coverage for each module. As shown, KotSuite was able to achieve an average line coverage of 66.0% and branch coverage of 60.4% across the 8 modules. The highest coverage achieved was 83.2% for line coverage and 78.4% for branch coverage in the `com.android.gpstest.util` module, which had a relatively simple code structure. In contrast, more complex modules such as `ch.abertschi.adfree.detector` and `ch.abertschi.adfree.model` exhibited slightly lower coverage, with average line and branch coverage of around 55%.

KotSuite was specifically designed to support both Kotlin and Java languages within Android projects. Our evaluation results confirmed that KotSuite can seamlessly handle both Kotlin and Java code. In projects written mostly in Kotlin, such as Ad-Free and WiFiAnalyzer projects, KotSuite was able to effectively generate test cases and achieve comparable coverage metrics to those obtained in Java-based modules. For example, in the WiFiAnalyzer project, KotSuite achieved 78.1% line coverage and 67.4% branch coverage, demonstrating its effectiveness even with Kotlin-specific features such as coroutines and null safety.

Overall, the experimental results indicate that KotSuite is effective in generating unit test cases for real-world Android Kotlin projects. The coverage makes the tool a valuable addition to the Android development workflow.

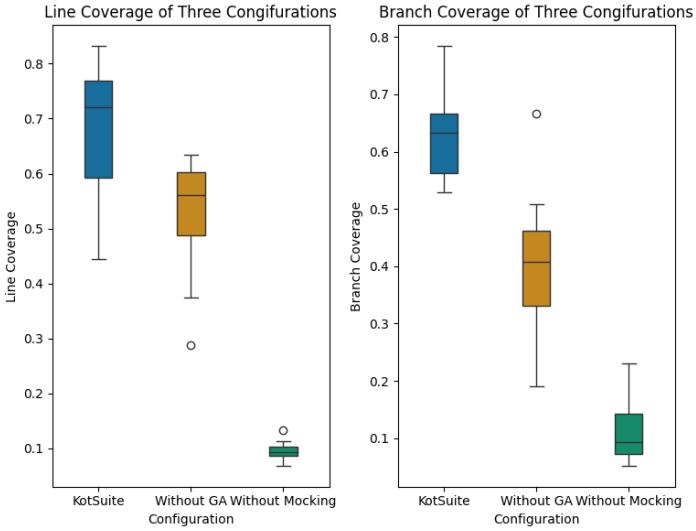

Fig. 5. Box-plots for line coverage and branch coverage of three versions of KotSuite.

B. RQ2. Can the genetic algorithm and test mocking components improve the coverage of generated test cases?

In this section, we evaluate the impact of the genetic algorithm and test mocking components in KotSuite on the effectiveness and efficiency of test case generation. To address this research question, we conducted experiments using eight Android Kotlin modules. For each module, we measured both line coverage and branch coverage of the generated test cases, along with the time taken for test case generation. Additionally, we performed ablation experiments by removing key components from KotSuite, specifically the genetic algorithm and test mocking modules, to assess their contribution to test

TABLE III
AVERAGE TIME PER FUNCTION TAKEN BY DIFFERENT APPROACHES FOR TEST CASE GENERATION (IN SECONDS)

Application	Module	KotSuite	KotSuite variant		
			Without GA	Without reuse	Without mocking
Ad-Free	ch.abertschi.adfree.detector	3.4	1.6	4.9	3.5
	ch.abertschi.adfree.model	4.2	1.4	5.0	3.9
GPSTest	com.android.gpstest.util	3.7	1.2	5.4	3.8
	com.android.gpstest.io	3.4	1.4	5.7	3.3
CineLog	com.ulicae.cinelog.data.dao	4.7	1.4	5.8	4.4
	com.ulicae.cinelog.utils	4.3	1.3	5.4	4.6
WiFiAnalyzer	com.vrem.wifianalyzer.wifi.graphutils	4.2	1.8	5.7	4.3
	com.vrem.wifianalyzer.wifi.model	3.8	2.1	4.9	4.1
Average	-	4.2	1.6	5.4	4.1

case coverage and efficiency.

The experimental setup consisted of three configurations. **KotSuite** uses both the genetic algorithm and test mocking components, representing the complete test generation pipeline in KotSuite; **KotSuite without GA** replaces the genetic algorithm with a simple random algorithm to generate test cases; **KotSuite without mocking** removes the test mocking module and new objects were instantiated directly in the generated test cases instead of using mock objects.

For each configuration, we measured the line coverage, branch coverage, and average time taken to generate test cases for the target methods. The results of the experiment are summarized in Table II and Table III. Fig. 5 shows the significance of both the genetic algorithm and test mocking components in improving test case coverage. For example, in the `com.vrem.wifianalyzer.wifi.graphutils` module, KotSuite achieved 75.5% line coverage and 73.2% branch coverage. However, when the genetic algorithm was removed, the coverage dropped significantly to 52.8% line coverage and 36.8% branch coverage. Similarly, when the test mocking module was removed, the coverage declined to 11.2% line coverage and 19.8% branch coverage. These findings suggest that both the genetic algorithm and test mocking module are essential in ensuring high coverage.

In contrast, the random algorithm used in the absence of the genetic algorithm resulted in lower coverage, as it lacked the optimization mechanisms that the genetic algorithm provides to prioritize high-coverage test cases. The test mocking component uses mock objects to replace real objects, such as internal Android objects. Since the unit test cases run on JVM and lack the Android runtime environment, the absence of the test mocking component resulted in the failure of most test cases. As a consequence, the coverage of the generated test cases was significantly reduced.

While the coverage was reduced when the genetic algorithm or test mocking module was removed, the time taken for test case generation improved. In the `com.vrem.wifianalyzer.wifi.graphutils` module, generating test cases with KotSuite took 4.2 seconds per method, while removing the genetic algorithm reduced the time to 1.8 seconds. When the test mocking module was removed, the time decreased to 4.3 seconds. These results demonstrate that while the genetic algorithm and test mocking contribute significantly to the coverage, they also introduce additional computational complexity. Removing these components leads to faster test case generation, albeit at the expense of reduced test coverage.

The experimental results confirm that both the genetic algorithm and test mocking components in KotSuite effectively improve the coverage of generated test cases. The genetic algorithm optimizes the test case selection process, ensuring that high-impact code paths are prioritized, while the test mocking component ensures that the generated test cases can execute correctly on JVM by substituting real objects with mock objects. While the removal of these components leads to faster test generation times, it comes at the cost of significantly reduced coverage. These findings validate the importance of these components in KotSuite for achieving a balance between high coverage and efficient test case generation.

C. RQ3. Does the test reuse algorithm in KotSuite enhance the efficiency of test case generation?

In this section, we evaluate the impact of the test reuse algorithm in KotSuite on the efficiency of test case generation. To address this research question, we conducted experiments using eight Android Kotlin modules. These modules were selected from a variety of applications to assess the overall performance of the test reuse algorithm in enhancing the efficiency of test case generation without affecting the coverage of the generated test cases.

The experiment was conducted using two configurations. **KotSuite** uses all components of KotSuite, including the test reuse algorithm, the genetic algorithm, and the test mocking module. **KotSuite without reuse** disables the test reuse module, leaving the genetic algorithm and test mocking module active. In this configuration, the genetic algorithm generates test cases without reusing previously generated high-quality test cases.

For each configuration, we measured the average time taken to generate test cases for the target methods, as well as the line and branch coverage of the generated test cases. The results of the experiment are summarized in Table II and Table III. The results show that removing the test reuse module from KotSuite did not lead to any significant change in the coverage of the generated test cases, and indicate that the test

reuse algorithm does not impact the effectiveness of test case generation in terms of coverage.

The most significant difference between the two configurations was observed in the time taken for test case generation. As shown, the average time to generate test cases for a single target method without the test reuse algorithm is 5.4 seconds. With the test reuse algorithm enabled, this time is reduced to 4.2 seconds, representing a 22% improvement in efficiency. For example, in the `com.vrem.wifianalyzer.wifi.graphutils` module, the time taken to generate test cases with KotSuite was 4.2 seconds, while the time was increased to 5.7 seconds when the test reuse module was removed. This increase in time demonstrates that the test reuse algorithm contributes to the efficiency of test case generation by reusing previously generated high-quality test cases, thus avoiding redundant iterations in the genetic algorithm. By reusing test cases, KotSuite reduces the need for repeated exploration of the same code paths, speeding up the overall test generation process.

The experimental results confirm that the test reuse algorithm in KotSuite enhances the efficiency of test case generation by reducing the time needed to generate test cases. While removing the test reuse module did not affect the coverage of the generated test cases, it led to a noticeable increase in the time taken for test case generation. These findings validate the effectiveness of the test reuse algorithm in speeding up the test case generation process by reusing high-quality test cases from previous iterations. This demonstrates the importance of the test reuse module in KotSuite for improving the efficiency of automated test generation in real-world Android Kotlin projects.

VI. Threats to Validity

We present the threats to the validity of our work.

Construct validity. A potential source of bias in the evaluation results is the lack of consideration for the impact of different parameter settings on the algorithm performance. In our experiments, we fixed certain parameters, such as the mutation rate, crossover rate, and maximum number of iterations, without exploring how these settings might influence the efficiency and effectiveness of test case generation. The genetic algorithm is highly sensitive to these parameters. Further studies should conduct experiments with different parameter configurations to better understand their impact on performance.

External validity. The limited generalization of KotSuite is a potential threat. The reason is that the tool currently operates exclusively on Android applications built with Kotlin and Java. This lack of cross-platform applicability means that KotSuite may not effectively generate test cases for applications written in other languages or designed for non-Android environments. Techniques in KotSuite can be expanded to support other platforms and programming languages in future.

VII. Related Work

Unit test generation based on program analysis. These tools focus on leveraging static or dynamic analysis of source code or bytecode to automatically generate test cases. Notable tools in this area include EvoSuite [8] and Randoop [9], which have been widely adopted for automated test generation in Java-based projects. EvoSuite is a popular test case generation tool that uses evolutionary algorithms to generate test suites with high code coverage. It generates high-coverage unit tests by first randomly creating a population of tests and then iteratively evolving them using genetic algorithms. Randoop is another widely used tool, generates unit tests by systematically invoking methods on the target classes and inferring input values. It uses feedback from execution results to refine the generated tests, often focusing on producing tests that satisfy specific coverage criteria such as method, branch, or exception coverage. The application of EvoSuite and Randoop in Android Kotlin projects has been limited due to language-specific features and Android runtime environment. Tools like EvoSuite and Randoop do not support Kotlin-specific constructs or the Android framework, which complicates the test generation process for modern Android applications.

Unit test generation based on deep learning. These approaches leverage advances in natural language processing and machine learning to generate test cases that are context-aware and tailored to the underlying logic of the program [31]. A3Test [11] is a notable deep-learning-based test generation tool that utilizes a PLBART model [32] to automatically generate unit tests for software applications. By training on a large corpus of existing test cases and program code, A3Test learns to predict the most likely test inputs and corresponding outputs for given functions. ChatUniTest [12] is a deep learning approach that focuses on using large language models to generate unit tests by interpreting program code in a manner similar to how chat bots generate conversational text. While deep-learning-bashes like A3Test and ChatUniTest show potential for improving the quality and relevance of generated tests, they are still in the early stages of development and are not yet as widely applicable or reliable as traditional program-analysis-based methods. Furthermore, they often require large datasets for training, and their effectiveness can vary significantly depending on the domain and the complexity of the software being tested.

VIII. Conclusion and Future Work

In this paper, we have introduced KotSuite, an automated tool of unit test generation for Kotlin programs in Android projects. The evaluation demonstrates that KotSuite can help save the time cost of manually writing test cases. The implementation of KotSuite provides and an Android Studio plugin and a command-line tool.

In future work, we aim to extend the capability of KotSuite. First, we plan to enhance the test case generation process by incorporating adaptive mutation and crossover strategies. Second, we intend to explore the integration of KotSuite with other programming languages and platforms, broadening its applicability beyond Android and Kotlin.

REFERENCES

[1] Statista, "Number of apps available in leading app stores as of augest 2024," 2024. [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[2] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 429–440. [Online]. Available: https://doi.org/10.1109/ASE.2015.89

[3] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 738–748. [Online]. Available: https://doi.org/10.1145/3238147.3240465

[4] B. Beizer, *Software testing techniques (2. ed.)*. Van Nostrand Reinhold, 1990.

[5] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test. Verification Reliab.*, vol. 14, no. 2, pp. 105–156, 2004. [Online]. Available: https://doi.org/10.1002/stvr.294

[6] B. Souza and P. D. L. Machado, "A large scale study on the effectiveness of manual and automatic unit test generation," in *34th Brazilian Symposium on Software Engineering, SBES 2020, Natal, Brazil, October 19-23, 2020*, E. Cavalcante, F. Dantas, and T. Batista, Eds. ACM, 2020, pp. 253–262. [Online]. Available: https://doi.org/10.1145/3422392.3422407

[7] Google, "Android's kotlin-first approach," 2019. [Online]. Available: https://developer.android.com/kotlin/first

[8] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[9] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds. ACM, 2007, pp. 815–816. [Online]. Available: https://doi.org/10.1145/1297846.1297902

[10] G. Developers, "Fundamentals of testing android apps," 2024. [Online]. Available: https://developer.android.com/training/testing/fundamentals

[11] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3test: Assertion-augmented automated test case generation," *Inf. Softw. Technol.*, vol. 176, p. 107565, 2024. [Online]. Available: https://doi.org/10.1016/j.infsof.2024.107565

[12] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "Chatunitest: a chatgpt-based automated unit test generation tool," *CoRR*, vol. abs/2305.04764, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.04764

[13] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002. [Online]. Available: https://doi.org/10.1109/4235.996017

[14] JetBrains, "Kotsuite plugin for android studio," 2024. [Online]. Available: https://plugins.jetbrains.com/plugin/22699-kotsuite

[15] ——, "Intellij platform sdk." [Online]. Available: https://plugins.jetbrains.com/docs/intellij/welcome.html

[16] RedHat, "What is CI/CD?" 2023. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd

[17] J. Campos, Y. Ge, N. M. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Inf. Softw. Technol.*, vol. 104, pp. 207–235, 2018. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.08.010

[18] R. Vallée-Rai, P. Lam, C. Verbrugge, P. Pominville, and F. Qian, "Soot (poster session): a java bytecode optimization and annotation framework," in *Addendum to the 2000 Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2000, Minneapolis, MN, USA, October 15-19, 2000*, J. Haungs, Ed. ACM, 2000, pp. 113–114. [Online]. Available: https://doi.org/10.1145/367845.368008

[19] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*, E. Bodden, L. J. Hendren, P. Lam, and E. Sherman, Eds. ACM, 2012, pp. 27–38. [Online]. Available: https://doi.org/10.1145/2259051.2259056

[20] K. Sen, "DART: directed automated random testing," in *Hardware and Software: Verification and Testing - 5th International Haifa Verification Conference, HVC 2009, Haifa, Israel, October 19-22, 2009, Revised Selected Papers*, ser. Lecture Notes in Computer Science, K. S. Namjoshi, A. Zeller, and A. Ziv, Eds., vol. 6405. Springer, 2009, p. 4. [Online]. Available: https://doi.org/10.1007/978-3-642-19237-1_4

[21] A. Arcuri, M. Z. Z. Iqbal, and L. C. Briand, "Random testing: Theoretical results and practical implications," *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 258–277, 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.121

[22] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[23] Y. Zhang, M. Harman, and S. A. Mansouri, "The multi-objective next release problem," in *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, H. Lipson, Ed. ACM, 2007, pp. 1129–1137. [Online]. Available: https://doi.org/10.1145/1276958.1277179

[24] EclEmma, "Jacoco java code coverage library," 2024. [Online]. Available: https://www.eclemma.org/jacoco/

[25] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, M. B. Dwyer and F. Tip, Eds. ACM, 2011, pp. 353–363. [Online]. Available: https://doi.org/10.1145/2001420.2001463

[26] MockK, "mocking library for kotlin." [Online]. Available: https://mockk.io/

[27] JMockK, "A java adapter for mockk library." [Online]. Available: https://mvnrepository.com/artifact/io.github.Maple-pro/JMockK

[28] F-Droid. [Online]. Available: https://f-droid.org/

[29] GitHub. [Online]. Available: https://github.com/

[30] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013. [Online]. Available: https://doi.org/10.1109/TSE.2012.14

[31] X. Chen, X. Hu, Y. Huang, H. Jiang, W. Ji, Y. Jiang, Y. Jiang, B. Liu, H. Liu, X. Li, X. Lian, G. Meng, X. Peng, H. Sun, L. Shi, B. Wang, C. Wang, J. Wang, T. Wang, J. Xuan, X. Xia, Y. Yang, Y. Yang, L. Zhang, Y. Zhou, and L. Zhang, "Deep learning-based software engineering: progress, challenges, and opportunities," *SCIENCE CHINA Information Sciences*, vol. 68, no. 1, p. 111102, 2025.

[32] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tür, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds. Association for Computational Linguistics, 2021, pp. 2655–2668. [Online]. Available: https://doi.org/10.18653/v1/2021.naacl-main.211