International Journal of Environmental Sciences
ISSN: 2229-7359
Vol. 12 No. 2, 2026
https://theaspd.com/index.php

A GA-Based Approach to Automatic Test Data Generation for Data-Flow Testing of Android Applications

Marwa A. Mohammed[1], Moheb R. Girgis[2], Enas Elgeldawi[3]

[1,2,3] Department of Computer Science, Faculty of Science, Minia University, 61511, Minia, Egypt.

Abstract

Automated test data generation is a critical yet challenging task in software testing, particularly when aiming to satisfy specific adequacy criteria. Within the field of Search-Based Software Testing (SBST), test data generation is commonly formulated as an optimization problem, where metaheuristic techniques such as Genetic Algorithms (GAs) are employed to produce effective test cases. Although GA-based approaches have been applied to data-flow testing in traditional software systems, they do not adequately address the distinctive characteristics of Android applications, including event-driven execution and complex user interface (UI) interactions. To overcome these limitations, this paper proposes a GA-based automated test data generation approach and a supporting tool specifically designed for data-flow testing of Android applications. The proposed method exploits data-flow dependencies within the Application Under Test (AUT) to generate test data that satisfy the all-uses coverage criterion. A key innovation is the chromosome representation, in which each individual is encoded as a sequence of UI control objects (genes) organized into logical activity blocks. To enhance the search process, two specialized genetic operators are introduced: block crossover and control-based mutation. The proposed GA generates new test data by recombining and evolving previously successful test cases, thereby maximizing data-flow coverage. The accompanying tool takes as input an instrumented version of the AUT, a set of definition-use (def-use) pairs to be covered, and information about input controls. As output, it produces a suite of test cases, identifies the def-use pairs covered by each test case, and reports any remaining uncovered pairs. The paper includes a case study that demonstrates the practical application of the proposed approach on real-world Android applications. Experimental results show that the proposed method effectively generates test data satisfying the all-uses criterion while efficiently exposing faults in Android applications. This work represents the first integrated solution that combines GA-based optimization, Android application instrumentation, static analysis, and UI-aware chromosome representations to systematically generate high-quality test cases that maximize data-flow coverage and improve fault detection in Android applications.

Keywords: *Search-Based Software Testing, Data-Flow Testing, Automatic Test Data Generation, Genetic Algorithms, Android Application Testing.*

1. INTRODUCTION

Software testing is a critical activity in the software development lifecycle, aimed at ensuring the reliability, correctness, and overall quality of software systems. Despite its importance, testing remains one of the most resource-intensive phases of development. Empirical studies report that testing may consume up to 50% of total development cost and approximately 40% of development time [1]. Consequently, enhancing the efficiency and automation of testing processes continues to be a major research challenge. Two fundamental components of software testing are test data generation and test adequacy criteria. Test data generation focuses on producing input values that exercise the software under test, while test adequacy criteria determine whether the generated test cases sufficiently satisfy the intended testing objectives. Manual generation of test data to meet such criteria is both time-consuming and error-prone, underscoring the need for automated solutions.

The rapid expansion of Android applications across business, social, and governmental domains has intensified the demand for highly reliable and robust mobile software. Android applications are expected to operate continuously and correctly under diverse usage scenarios. Insufficient testing can lead to functional failures, security vulnerabilities, and degraded user experience, ultimately resulting in loss of user trust. Therefore, effective and automated testing techniques are essential for ensuring the quality of Android applications [2], [3].

Random test data generation provides a simple and fast approach to automation; however, it offers limited guarantees regarding coverage and fault detection capability. To overcome these limitations, Search-Based Software Testing (SBST) formulates test data generation as an optimization problem and

applies metaheuristic search techniques to systematically explore the input space [4], [5]. Genetic Algorithms (GAs) have been widely adopted in SBST due to their ability to efficiently search large and complex solution spaces [6], [7].

This paper proposes a GA-based automated test data generation approach and a supporting tool for data flow testing of Android applications. The approach is guided by the data flow dependencies in the application under test, to search for test data to fulfil the all-uses criterion [8]. It aims to maximize the coverage of definition-use (def-use) pairs through a customized chromosome representation and genetic operators tailored to Android user interface interactions.

The main contributions of this paper are summarized as follows:

1. A novel GA-based automated test data generation approach for Android applications: To the best of our knowledge, this is the first study to apply GAs to automated test data generation for data-flow testing of Android applications.
2. A UI-aware chromosome representation and specialized genetic operators: The paper introduces a new chromosome representation based on Android UI interactions, along with two dedicated genetic operators block crossover and control-based mutation specifically designed for event-driven mobile applications.
3. An integrated automated testing framework and tool: An automated tool is developed to implement the proposed approach by integrating GA-based optimization, Android application instrumentation, static analysis, and UI-aware chromosome representations.
4. Generation of high-quality test cases: The proposed approach systematically generates effective test cases that maximize data-flow coverage and enhance fault detection capabilities in Android applications.
5. Comprehensive empirical evaluation: A case study and extensive experiments on real-world Android applications demonstrate the effectiveness of the proposed approach in satisfying the all-uses coverage criterion and detecting application faults efficiently.

The rest of the paper is organized as follows: Section 2 reviews some of the research work in the field of search-based test data generation and Android applications testing; Section 3 provides an overview of data-flow testing; Section 4 presents the basic concepts of GAs, and some utilities for automated testing, namely Espresso and Genetic Algorithm Framework (GAF); Section 5 describes a proposed GA-based test data generation approach for data-flow testing of Android applications; Section 6 provides a case study to demonstrate how the proposed GA-based testing tool automatically generates test data to cover the def-use pairs of an Android application; Section 7 presents the experimental results; finally, Section 8 presents the paper conclusion and future work.

2. RELATED WORK

As this paper focuses on GA-based automatic test data generation for Android applications, this section reviews some of the research work in the field of search-based test data generation and Android applications testing.

Automated test data generation has been widely studied within SBST, where testing activities are formulated as optimization problems and metaheuristic algorithms are employed to generate effective test cases [4], [5]. Among these techniques, GAs have gained significant attention due to their ability to efficiently explore large and complex search spaces while optimizing diverse coverage criteria [9]. Early work by Girgis [9] demonstrated the effectiveness of GA-based approaches in achieving data-flow coverage for traditional software systems. Similarly, Akhter et al. [10] introduced a parallel multi-population GA targeting path coverage. Several other studies have also explored GA-based and search-based test data generation for traditional software systems. Sharma et al. [4] provided a comprehensive survey of GA applications in software testing, showing that these techniques can enhance both test coverage and fault-detection effectiveness. Setiadi et al. [11], [12] investigated search-based approaches for concurrent programs. Takamatsu et al. [13] extended the Seeker tool [14] to support multiple testing targets in object-oriented programs using GA-based optimization, addressing the combinatorial explosion associated with method-sequence generation. Alternative search-based techniques, such as Novelty Search, have also been proposed to improve exploration in large search spaces where traditional fitness functions may become trapped in local optima [15]. Scalabrino et al. [16] introduced OCELOT, a tool integrating

instrumentation with search-based techniques for C programs, demonstrating the practical value of combining evolutionary algorithms with automated testing frameworks. Elgendy et al. [17] presented an automated test data generation approach based on GA for data flow testing of ASP.NET web applications. Auer et al. [18] introduced an approach to integrate surrogate models for testing Android applications. The surrogate model is an abstraction of the state-based behavior of the graphical user interface, and can predict traces for already explored behavior, thus avoiding costly test executions. They implemented this approach as an extension of the search-based test generator MATE (Mobile Accessibility Testing) for Android [19].

Testing Android applications introduces additional challenges compared to traditional software systems. Android apps are inherently event-driven and characterized by multiple activities, asynchronous callbacks, lifecycle management, and complex navigation flows. These characteristics complicate automated test data generation and require specialized testing strategies. Alshahwan and Harman [20] applied SBST techniques to Android applications, focusing on branch coverage. Gao et al. [21] extended this line of work by employing evolutionary algorithms to generate automated test inputs.

Despite these advancements, most existing approaches either target non-Android platforms or focus on structural coverage criteria such as statement, branch, or path coverage rather than data-flow coverage. Recent surveys [3] highlight that despite progress in SBST for mobile applications, data-flow-oriented testing for Android remains underexplored.

GA-based automated test data generation approaches for data flow testing of traditional software systems, such as the approach proposed by Girgis [9], or web applications, such as the approach proposed by Elgendy et al. [17], have not addressed the unique challenges of Android applications, including event-driven behavior and complex UI interactions. Likewise, SBST approaches for Android [20], [21] have primarily emphasized general coverage or fault detection without explicitly targeting data-flow adequacy.

To the best of our knowledge, no prior work has integrated GA-based test data generation specifically for data-flow testing of Android applications. This gap motivates the proposed research, which combines GA optimization with Android-specific instrumentation, static analysis, and UI-aware chromosome representations to systematically generate test cases that maximize data-flow coverage and fault detection in Android applications.

3. DATA FLOW TESTING

Data Flow Testing (DFT) is a white-box testing technique that emphasizes tracking how data moves through a program. Its main goal is to detect anomalies in variable usage for instance, variables that are declared but never used, or accessed before being initialized by leveraging data-flow analysis [22], [23].

The core idea of data flow analysis is to examine the sequence of actions performed on variables along a program's execution path. It emphasizes the relationship between variable definitions (defs) and their subsequent uses. DFT involves executing test paths, which are chosen based on definition-use (def-use) chains, where a def-use chain, also called def-use pair, represents the path from a variable's def to its point of use, without any intervening redefinitions. Various DFT techniques [24], [25], [26] have been developed to help identify program errors. These techniques rely on the program's data flow information to guide the selection of test cases.

To determine when testing is complete, a test-adequacy criterion is required [27]. In this work, we adopt the all-uses criterion introduced by Rapps and Weyuker [8]. This criterion requires that, for every def of a variable, a def-clear path leading to each of its uses must be executed. The def-clear paths needed to meet this criterion, referred to as du-paths, are constructed from the defs and uses of program variables using the technique described in [28].

In an early work [29], we developed a novel approach to DFT, and a supporting automated tool, called AndroidDFT, specifically designed for Android applications, which extends traditional data-flow analysis to account for Android's distinctive structure and lifecycle.

In the present work, we developed an automated test data generation method based on a GA for data flow testing of Android applications, and we incorporated it into AndroidDFT to enhance its ability to generate automated test data that meets the all-use criterion for the Android application under test.

4. GENETIC ALGORITHM CONCEPTS AND UTILITIES

This section provides basic concepts of GAs, and some utilities used for automated testing, namely Espresso and Genetic Algorithm Framework (GAF)

Genetic Algorithms (GA)

The concept of the Genetic Algorithm (GA) was first introduced by Holland [30] as an optimization method inspired by the principles of natural evolution. GA belongs to the class of population-based search algorithms that emulate the process of natural selection, where individuals with higher fitness are more likely to survive and contribute to subsequent generations. Through iterative evolution, the algorithm aims to discover high-quality solutions to complex optimization problems [7].

In a GA, each candidate solution is encoded as a chromosome, typically represented as a string that resembles the structure of biological DNA. A collection of these chromosomes forms a population, with each chromosome corresponding to a potential solution. During every generation, the fitness of each individual is evaluated using a predefined objective function. Based on these fitness values, more promising individuals are selected to produce offspring through genetic operators such as crossover and mutation. The crossover operator combines genetic material from two parent chromosomes to generate new solutions, whereas mutation introduces small random changes to maintain genetic diversity and reduce the likelihood of premature convergence. By repeatedly applying these evolutionary operators, the population gradually evolves toward solutions with improved fitness. The overall procedure of a standard GA is summarized in Algorithm 1 [6].

```
Algorithm 1: Basic GA
1 begin
2  initialize population;
3  evaluate population;
4  while termination criterion not reached do
5     select solutions for next population;
6     perform crossover and mutation;
7     evaluate population;
8  end
9 end
```

The termination criterion of the algorithm can be either reaching a solution to the problem, or reaching the maximum number of iterations, meaning that a solution cannot be found given the available resources.

Espresso and GAF

Automated testing of Android applications often requires driving the application through its user interface (UI) to verify correct behavior. Espresso [31], [32] is a widely used testing framework for Android that allows developers to create coded UI tests, automating interactions such as clicks, text input, and navigation between activities. In the proposed approach, Espresso is utilized to execute the Android application with the test inputs generated by the GA. This enables the evaluation of each test case in terms of both execution feasibility and coverage achievement.

The Genetic Algorithm Framework (GAF) [33] is a .NET/Mono library that facilitates the implementation of GA-based solutions in C#. GAF supports chromosomes composed of binary, integer, real, or object-based genes, allowing the representation of complex entities, such as UI controls, as individual genes. The GAF allows the definition of any fitness function as well as a termination function, which causes the run to stop. It also provides many genetic operators, such as: crossover and mutation, and supports adding new custom genetic operators. Using GAF, the proposed approach constructs chromosomes where each gene represents a UI control object. The GA then evolves these chromosomes using custom genetic operators to generate effective test data. The combination of Espresso for execution and GAF for evolutionary search enables an automated pipeline from test data generation to coverage evaluation in Android applications.

5. THE PROPOSED APPROACH

This section describes a proposed GA-based test data generation approach for data-flow testing of Android applications. In an early work [29], the authors presented AndroidDFT for automated data flow testing of Android applications, where the Android application under test is instrumented and analyzed to identify the def-use pairs of relevant variables. This paper extends that work by using GA for generating test cases automatically to cover identified def use pairs.

Representation

The chromosome is constructed from information relating to UI controls provided by the static analysis report generated by AndroidDFT [29] for the Android application under test (AUT). Every UI control is considered as a gene in the chromosome. Figure 1 shows the chromosome structure, where A1, A2, ..., An represent the activities of the AUT. These Android activities are added to the chromosome one by one. Within each activity, its UI controls are added in a certain order. Three groups of controls may exist within each activity: Input controls (such as textboxes, or spinners), clickable controls (such as radioButtons, checkboxes, or buttons), and hyper link controls (including linkButtons). Within the clickable controls block, the buttons are grouped at the end of the block. The three groups within each Android activity are shown in Figure 1.

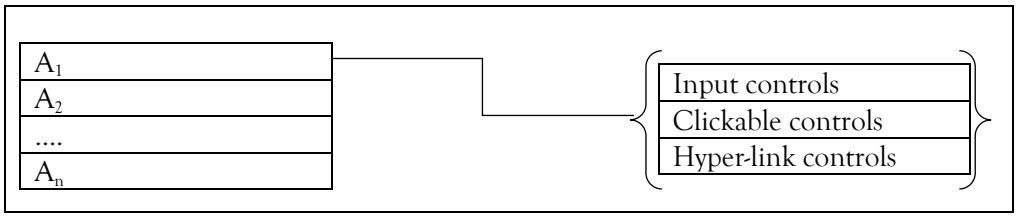


Fig. 1. The chromosome structure, where three control groups are included within each activity.

The chromosome is designed to reflect the sequence of interactions that a user typically performs within the code-behind (Activity Class) file. First, genes corresponding to input controls are assigned values, ensuring that any data required by the associated activity logic is available. Next, the chromosome specifies the order in which non-navigational clickable controls are activated. Finally, a hyperlink is selected to initiate navigation to another activity. This sequence is repeated iteratively until a complete chromosome has been generated.

Unlike traditional GA implementations that encode chromosomes as binary strings, the adopted GAF framework [33] supports object-oriented chromosome representations. Consequently, each gene corresponds directly to a user interface (UI) control object, enabling a more natural representation of application interactions. This chromosome design necessitates customized crossover and mutation operators capable of manipulating object-based genes rather than binary values.

To initialize the search process, the first population is generated randomly. Input controls receive randomly selected values, clickable controls are assigned a random execution order, and one navigation hyperlink is chosen at random. Before executing the algorithm, the tester must define the application's initial activity from which testing begins. In addition, the tester specifies the expected data type for each text input field (e.g., textual or numeric). For numeric inputs, valid value ranges must be provided, whereas for textual inputs, the required string length is defined.

Genetic Operators

As previously described, the chromosome is represented as a collection of UI control objects rather than a binary-encoded string, where each UI control corresponds to a single gene. Consequently, conventional genetic operators cannot be applied directly. To accommodate this object-oriented representation, specialized crossover and mutation operators were designed. These operators, namely block crossover and control-based mutation, are adapted from the techniques originally proposed for ASP.NET web applications by Elgendy et al. [17].

Block Crossover

To preserve the logical structure of the Android application, each chromosome is divided according to its constituent Android activities. During the crossover process, two parent chromosomes exchange genetic material on an activity-by-activity basis. For every activity, one of the three predefined blocks is selected randomly and exchanged between the two parents. As an illustration, consider an application containing three Android activities. Two example chromosomes are presented in Figure 2(a). Suppose that block 2 is randomly selected for the first activity, block 1 for the second activity, and block 3 for the third activity. After performing the crossover operation, the resulting offspring chromosomes are obtained as illustrated in Figure 2(b).

Control-Based Mutation

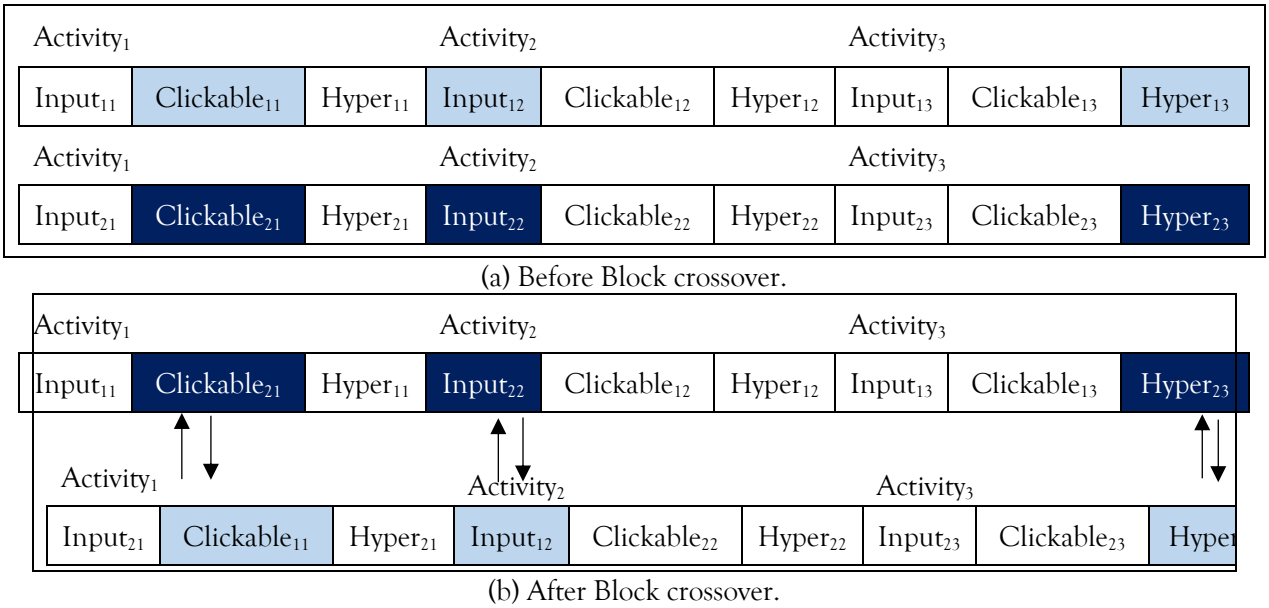
Because the chromosome consists of UI control objects instead of binary genes, the traditional mutation operator is not applicable. Instead, the mutation strategy is customized according to the type of UI control being modified. For input controls, mutation is achieved by generating a new value randomly within the range specified by the tester. For clickable controls, the mutation operator reverses the current state (clicked to unclicked, or unclicked to clicked). In the case of button controls, mutation changes the execution sequence of button-click events. For hyperlink controls, mutation is performed by randomly selecting an alternative hyperlink.

Fitness Function

The quality of each candidate solution is assessed using a fitness function based on its achieved coverage. Every chromosome (ch) is first translated into a corresponding set of test inputs, which constitutes the candidate solution. The instrumented Android application is then executed using this generated test data. During execution, a trace file containing the traversed execution path is produced. This execution path is subsequently compared with the predefined set of def-use pairs, and the chromosome fitness is computed according to the following equation:

$$f(ch) = \frac{number\ of\ newly\ covered\ def\ -\ use\ pairs}{total\ number\ of\ def\ -\ use\ pairs}$$

where f(ch) is the fitness of the chromosome ch. A test case which is represented by the chromosome ch is considered ***effective*** if its fitness value f(ch) > 0. In each generation, every solution in the current population is run and its fitness is calculated.



(a) Before Block crossover.

(b) After Block crossover.

Fig. 2. Two possible chromosomes (a) before and (b) after the Block crossover operation.

In the traditional GA approach, the population evolves until a single individual emerges that represents the final solution. In our context, this would mean that one group of data items achieves maximum application coverage specifically, by traversing all the def-use pairs within the application. Although this may be feasible for some applications, most cannot be fully covered by a single group of data items (i.e., a single test case). Instead, achieving the desired level of testing typically requires multiple groups and several executions of the application. To address this, the population is allowed to evolve until a combined subset of individuals collectively reaches the required coverage. This is accomplished by tracking which def-use pairs are covered by each individual. The evolutionary process stops once a group of individuals together covers all def-use pairs of the application, if such coverage is possible. In this scenario, the optimal solution is the set of individuals that collectively achieve complete coverage [9].

Following the current population evaluation, it is evolved by applying the block crossover and control-based mutation operators, resulting in a new generation of candidate solutions. This newly generated population is then evaluated using the same fitness assessment procedure. Unlike an independent evaluation of each generation, the algorithm maintains the cumulative coverage achieved by the best solutions from previous generations, allowing newly discovered coverage to be added to the existing results. To demonstrate this process, consider an Android application containing a total of 50 def-use pairs. Suppose that the best chromosome in the initial generation covers 20 def-use pairs, yielding a coverage percentage of 40%. This accumulated coverage is retained when evaluating the subsequent generation. If a chromosome in the next generation successfully covers an additional five previously uncovered def-use pairs, the total number of covered pairs increases to 25. Consequently, the accumulated coverage rises to 50%. The evolutionary process continues in this manner, with each generation contributing newly covered def-use pairs whenever possible, until either complete coverage is obtained or the predefined maximum number of generations has been reached.

The Overall GA Algorithm

Algorithm 2 outlines the workflow of the proposed genetic algorithm, (GATestDataGenerator). The algorithm requires several inputs, including the instrumented version APP' of the Android application under test APP, the collection of def-use pairs targeted for coverage, and the information associated with the input controls (type and range for each input control). In addition, the algorithm receives the starting address of the application from which the testing process begins, the population size Pop_Size, and the maximum number of generations Max_Gen.

The algorithm generates three outputs: a set of test cases, the def-use pairs covered by each generated test case, and the remaining uncovered def-use pairs, if any. To monitor coverage throughout the search process, a data structure referred to as 'the def-use coverage table' is maintained. Each entry in this table represents a single def-use pair. At the beginning of execution, all entries are marked as uncovered, the final test suite S is initialized as an empty set, and the initial population is created randomly.

In line 11, the algorithm invokes the EvaluatePopulation() procedure, whose details are presented in Algorithm 3 and discussed in the following subsection. This procedure evaluates all candidate solutions and returns the best-performing solution, consisting of the effective chromosomes within the current population. The returned solution is subsequently examined in line 12 to determine whether it improves the cumulative coverage achieved so far. If an improvement is detected, the solution is incorporated into the final test suite S, the accumulated coverage value is updated accordingly, and the def-use coverage table is modified to reflect the newly covered pairs.

Afterward, the generation counter is incremented, and a new population is produced by applying the block crossover and control-based mutation operators. This evolutionary cycle continues until either complete def-use coverage is obtained or the predefined maximum number of generations Max_Gen has been reached. Upon termination, the algorithm returns the final test suite S and produces a report summarizing the def-use pairs covered by each generated test case together with any def-use pairs that remain uncovered.

Algorithm 2: GATestDataGenerator - A GA algorithm to automatically generate test cases for a given Android application.
Input: Instrumented version APP' of the Android application to be tested APP;

List of def-use pairs to be covered; Input controls related information;
Population size (Pop_size); Maximum no. of generations (Max_Gen);
Probability of crossover P_c; Probability of mutation P_m; and Starting address of APP';
Output: Set of test cases for APP, def-use pairs coverage percentage, the set of def-use pairs covered by each test case; and List of uncovered def-use pairs, if any;

```
1  begin
2    Step 1: Initialization
3      Initialize the def-use coverage table to uncovered;
4      Create Initial_Population;
5      Cur_population = Initial_Population;
6      Set of test cases for APP, S = ∅;
7      Acc_Coverage_Percent = 0;
8      No_Of_Generations = 0;
9    Step 2: Generate test cases
10     while Acc_Coverage_Percent ≠ 100 and No_Of_Generations ≤ Max_Gen do
11       best = EvaluatePopulation(Cur_population);
12       Cur_Coverage = checkCoverage(best);
13       if Cur_Coverage > Acc_Coverage_Percent then
14         Acc_Coverage_Percent = Cur_Coverage;
15         S = S ∪ best;
16         Update the def-use coverage table;
17       end if
18       Create New_Population using block crossover and control-based mutation operators;
19       Cur_population = New_Population;
20       Increment No_Of_Generations;
21     end while
22   Step 3: Produce output
23     Return S, the set of def-use pairs covered by each test case, and Acc_Coverage_Percent;
24     Report uncovered def-use pairs, if any;
25 end.
```

Running Tests

In order to evaluate the fitness of the current population, the method EvaluatePopulation() (Algorithm 3) is used. In this algorithm, each chromosome in the population is decoded into a set of test data and run automatically on the Android application APP' using Espresso. This requires generating a coded UI file based on the current chromosome, and the file is compiled and run automatically to perform the test on the Android application. Running the instrumented Android application APP' produces a text file that contains the line numbers of the traversed path, which is used to check the fitness of the chromosome as explained before. We use a data structure named activityList to track the start and end indices of every Android activity and its internal blocks within the chromosome. The activityList is used to retrieve information about the positions of the UI controls of the Android activities. Also, we use an array of Boolean values, named visited, to keep track of the status of the Android activities. The value true means an Android activity is visited, false otherwise.

Implementation

An automated tool has been developed in Java to implement the proposed GA-based test data generation approach for Android applications. Figure 3 shows the UI of the proposed GA-based testing tool. The test data generation process begins by allowing the tester to select the Android application to be tested. Subsequently, the tester clicks the "Run Analysis & Instrumentation Module" button, which submits the Android AUT to the DFTAndroid tool [29] for analysis and instrumentation through its Static Analysis & Instrumentation Phase. During this phase, the tool identifies variables, their definitions and uses across application activities, detects the associated UI controls, and extracts all def-use pairs. An instrumented version of the AUT is also generated.

By clicking the "Show UI Controls" button, the tester can view all extracted UI controls in the table displayed beneath this button. The tester then provides the information required by the GA to facilitate test case generation. For each input control, the tester specifies the data type (e.g., text or numeric, with numeric selected by default) and the corresponding range of valid values. For text inputs, the specified range represents the allowable length of the input string, while the characters themselves are generated randomly. For list controls, the range determines the indices that may be selected during test generation. In addition, the tester configures the GA parameters, including the population size and the maximum number of generations.

Algorithm 3: EvaluatePopulation(Pop).
Input: The population Pop; The activity List data structure activityList;
Output: The set of effective chromosomes in Pop;

```
1 begin
2   activityIndex = FindStart(Pop, Starting address of APP');
3   for each chromosome ch in Pop do
4     Decode every gene into a UI control object uic;
5     Initialize values of the array visited to false;
6     Create an empty coded UI file (CUIF);
7     startIndex = activityList[activityIndex].start;
8     endIndex = activityList[activityIndex].end;
9     visited[activityIndex] = true;
10    for every UI control object uic in ch from the startIndex to endIndex do
11      Write coded UI command into CUIF based on the type of the uic object to perform
        the corresponding action on this UI control;
12      if uic.Type = hyperlink then
13        targetactivity = uic.Target;
14        targetIndex = search(targetActivity, activityList);
15        if targetIndex ≠ -1 and !visited[targetIndex] then
16          startIndex = activityList[targetIndex].start;
17          endIndex = activityList[targetIndex].end;
18        else
19          close CUIF;
20          break;
21        end if
22      end if
23    end for
24    Compile and run CUIF to automatically run APP' with the generated test data;
25    Calculate f(ch) based on the percentage of covered def-use pairs by the traversed path;
26  end for
27   Return The set of effective chromosomes in Pop;
28 end.
```

After all, required information has been provided, the tester clicks the "Run GA" button. The GA then automatically generates test data for the AUT and evaluates each execution based on coverage criteria. As mentioned above, the GA operates on the instrumented version of the AUT. Upon completion, the tool presents a summary of the GA run results at the bottom of the interface, including the actual number of generations executed, the total number of generated test cases, and the achieved du-paths coverage percentage.

For detailed execution information, the tester can click the "GA Report" button, which displays a comprehensive report in the text area located below the UI controls table. The interface also provides access to all generated test cases through the "Generated Test Cases" button, which replaces the report view with the list of generated test cases. The tester may then select a specific test case by entering its number and clicking the "Select & Run Test Case" button to execute the AUT using the chosen test case.

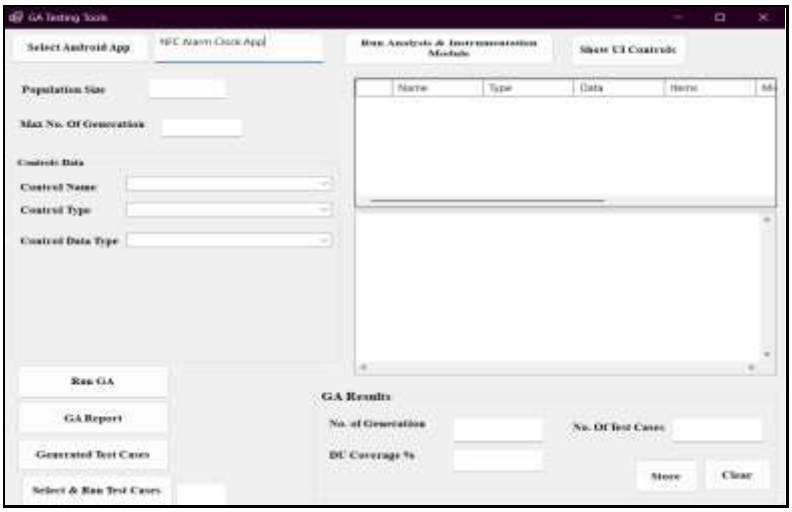


Fig. 3. The user interface of the proposed GA-based testing tool

6. CASE STUDY

This section presents a case study to demonstrate how the proposed GA-based testing tool automatically generates test data to cover the def-use pairs of an Android application. The application used in the case study is called "NFC Alarm Clock-App"[1]. This application is a customizable and user-friendly alarm clock designed to help users wake up on time and maintain daily organization. It consists of two activities, named "Activity1" and "Activity2". Figure 4 shows the screens of these two activities. The first screen is the Main Logo Screen, which serves as the opening screen of the application, and the second one is the alarm management screen, which enables users to store and manage alarm details and executes the actual alarm functionality. The UI controls contained in these activities form the chromosome representation used by the GA, as shown in Figure 5.

The AUT was instrumented and analyzed using our DFTAndroid tool [29]. The static analysis phase of DFTAndroid identified the variables, their definitions, and their corresponding uses across the two activities, detected the UI controls in both activities, and extracted all def-use pairs. The static analysis showed that the AUT has a total of 240 statements, 37 variables, 56 def-use pairs, and 5 UI controls. This information was then used to construct the chromosome structure and guided the GA-based test data generation process.

The following information was passed to the proposed GA-based testing tool: the instrumented version of the AUT; the list of extracted def-use pairs; the list of UI controls in the AUT activities; the input ranges for numeric and text fields; GA parameters (Pop_Size, Pc, Pm, and Max_Gen); the starting activity from which test execution begins. After passing this information, the GA started running, generating populations of chromosomes in successive generations, where each chromosome represented a test case, as mentioned before. Figure 6 shows an example chromosome and its corresponding test case. In each generation, the chromosomes were evaluated and the best chromosomes were kept.

Fig. 4. The Screens for Activity 1 and Activity 2 of the NFC Alarm Clock-App

[1] https://github.com/gabeg805/NFC-Alarm-Clock

	Activity1		Activity2		
Control	tvAlarmStatus	ImageView	timePicker	alarm_list	btnSetAlarm
Type	TextView	Button	Button	Button	Button

Fig. 5. The Chromosome structure for the NFC Alarm Clock App

A chromosome	**The corresponding test case**
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 12	Type 12 in TextBox tvAlarmStatus in Activity1
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kjg.png	Click on Button imageView in Activity1
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker	Click on Button timePicker in Activity2
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm	Click on Button btnSetAlarm in Activity2
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list	Click on Button alarm_list in Activity2

Fig. 6. An example chromosome and the corresponding test case

A chromosome was evaluated by decoding it into a test case and the instrumented AUT was executed automatically with it, using Espresso. The execution resulted in a text file containing the line numbers of the traversed path, which was then matched against the list of def-use pairs. The current test case was evaluated in terms of percentage coverage of def-use pairs, which in turn represented the fitness of the current chromosome.

Across generations, the GA progressively improved coverage by evolving chromosomes that exercised additional def-use pairs. Whenever a chromosome increased the accumulated coverage, it was added to the final set of test cases. This process continued until either full coverage was achieved or the maximum number of generations was reached. Figure 8 shows a screenshot of the UI of the proposed tool after the completion of running it with the AUT, displaying a portion of the GA report and the GA results. As shown in Figure 7, the tool managed to generate 10 test cases that achieved an accumulated coverage of 100 % in three generations.

The complete GA report after the completion of executing the tool with the AUT includes for each generation: the current population; the best chromosomes that were evaluated as effective; their corresponding test cases; current def-use coverage and accumulated def-use coverage; traversed path; the def-use pairs covered by this path; and those not covered yet (if any), the result of performing crossover and mutation operations on the effective chromosomes of the current population, and the resulted new population. At the end, the report lists all the generated test cases.

Figure 8 presents a portion of the GA report after the completion of running the tool with the NFC Alarm Clock App, showing only the first generation and the final test cases that were generated.

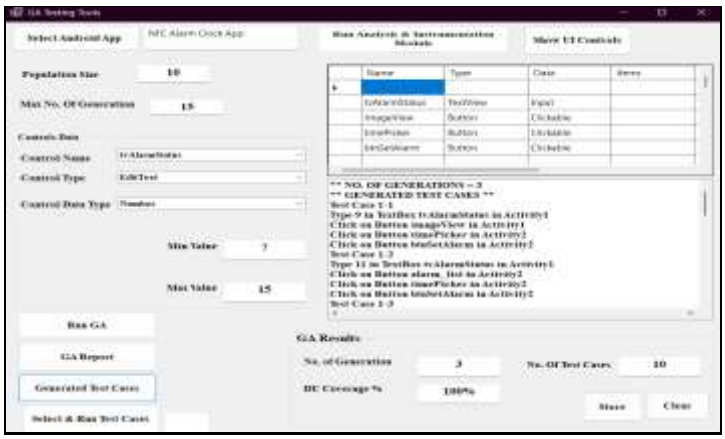


Fig. 7. A Screenshot of the UI Of the Proposed Tool After the Completion of Running It with the AUT, Displaying A Portion of the GA Report and the GA Results.

The generated test cases can be used by the tester to test the AUT and identify potential errors in it. The tester can select a specific test case by entering its number and clicking the "Select & Run Test Case" button to execute the AUT with the chosen test case, as shown in Figure 9.

Case Study1 (NFC Alarm Clock App)

Population Size: 10
Maximum no. of generations (Max_Gen): 15
CROSSOVER AND MUTATION PROBABILITIES: 0.8, 0.15

** GA STARTED **

** INITIAL POPULATION **

Chromosome 1:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 9
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kjg.png
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Chromosome 2:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 11
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Chromosome 3:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 7
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Chromosome 4:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 13
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_opl.png
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Chromosome 5:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 8
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Chromosome 6:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 10
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Chromosome 7:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 9
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Chromosome 8:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 14
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kklm.png
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Chromosome 9:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 12
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kjg.png
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Chromosome 10:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 15

Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list

_____
Chromosome 1: ********** SELECTED **********
Corresponding test case
Type 9 in TextBox tvAlarmStatus in Activity1
Click on Button imageView in Activity1
Click on Button timePicker in Activity2
Click on Button btnSetAlarm in Activity2
DEF-USE COVERAGE: 44.4%
ACCUMULATED DEF-USE COVERAGE: 44.4%
Traversed Path: 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

[THE DU-PATHS COVERED BY THIS PATH]:
DEF-USE PAIR <4, 60> For VARIABLE: timePicker
DEF-USE PAIR <7, 62> For VARIABLE: btnSetAlarm
DEF-USE PAIR <11, 65> For VARIABLE: tvAlarmStatus
DEF-USE PAIR <70, 72> For VARIABLE: bundle
DEF-USE PAIR <3, 75> For VARIABLE: alarm_list
DEF-USE PAIR <76,77> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <2,80> For VARIABLE: imageView
DEF-USE PAIR <87,83> For VARIABLE: mActionMode
DEF-USE PAIR <234, 83> For VARIABLE: mActionMode
DEF-USE PAIR <81, 88> For VARIABLE: view
DEF-USE PAIR <87,93> For VARIABLE: mActionMode
DEF-USE PAIR <116, 117> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <121, 123> For VARIABLE: requestCode
DEF-USE PAIR <121,132> For VARIABLE: requestCode
DEF-USE PAIR <121,134> For VARIABLE: resultCode
DEF-USE PAIR <116,136> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <121, 136> For VARIABLE: data
DEF-USE PAIR <116,137> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <121, 141> For VARIABLE: requestCode
DEF-USE PAIR <76, 143> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <3, 70> For VARIABLE: Alarm_list
DEF-USE PAIR <91, 99> For VARIABLE: view
DEF-USE PAIR <68, 77> For VARIABLE: ImageView
DEF-USE PAIR <114, 116> For VARIABLE: view

[THE DU-PATHS NOT COVERED YET]:
DEF-USE PAIR <151, 153> For VARIABLE: item
DEF-USE PAIR <156,157> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <76,174> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <147, 174> For VARIABLE: menu
DEF-USE PAIR <147, 175> For VARIABLE: menu
DEF-USE PAIR <147, 176> For VARIABLE: menu
DEF-USE PAIR <182, 189> For VARIABLE: index
DEF-USE PAIR <181, 191> For VARIABLE: info

DEF-USE PAIR <195, 197> For VARIABLE: v
DEF-USE PAIR <199, 200> For VARIABLE: checkBox
DEF-USE PAIR <209, 210> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <214, 216> For VARIABLE: actionMode
DEF-USE PAIR <214, 218> For VARIABLE: menu
DEF-USE PAIR <217, 218> For VARIABLE: inflater
DEF-USE PAIR <179, 181> For VARIABLE: item
DEF-USE PAIR <206, 208> For VARIABLE: view
DEF-USE PAIR <206, 209> For VARIABLE: position
DEF-USE PAIR <206, 210> For VARIABLE: id
DEF-USE PAIR <222, 225> For VARIABLE: actionMode
DEF-USE PAIR <222, 226> For VARIABLE: menu
DEF-USE PAIR <227, 229> For VARIABLE: actionMode
DEF-USE PAIR <227, 230> For VARIABLE: menuItem
DEF-USE PAIR <232, 233> For VARIABLE: actionMode
DEF-USE PAIR <182, 183> For VARIABLE: index
DEF-USE PAIR <76, 185> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <185, 186> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <63, 171> For VARIABLE: ABOUT_ACTIVITY
DEF-USE PAIR <66, 177> For VARIABLE: CONTEXT_MENU_DUPLICATE
DEF-USE PAIR <76, 191> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <121,125> For VARIABLE: resultCode
DEF-USE PAIR <116,127> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <121, 127> For VARIABLE: data
DEF-USE PAIR <116,128> For VARIABLE: mCurrentAlarm

_____

Chromosome 2: ********** SELECTED **********
Corresponding test case
Type 11 in TextBox tvAlarmStatus in Activity1
Click on Button alarm_list in Activity2
Click on Button timePicker in Activity2
Click on Button btnSetAlarm in Activity2
DEF-USE COVERAGE: 11.1%
ACCUMULATED DEF-USE COVERAGE: 55.6%
Traversed Path: 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

[THE DU-PATHS COVERED BY THIS PATH]:
DEF-USE PAIR <151, 153> For VARIABLE: item
DEF-USE PAIR <156,157> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <121,125> For VARIABLE: resultCode
DEF-USE PAIR <116,127> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <121, 127> For VARIABLE: data
DEF-USE PAIR <116,128> For VARIABLE: mCurrentAlarm

[THE DU-PATHS NOT COVERED YET]:
DEF-USE PAIR <76,174> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <147, 174> For VARIABLE: menu
DEF-USE PAIR <147, 175> For VARIABLE: menu

DEF-USE PAIR <147, 176> For VARIABLE: menu
DEF-USE PAIR <182, 189> For VARIABLE: index
DEF-USE PAIR <181, 191> For VARIABLE: info
DEF-USE PAIR <195, 197> For VARIABLE: v
DEF-USE PAIR <199, 200> For VARIABLE: checkBox
DEF-USE PAIR <209, 210> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <214, 216> For VARIABLE: actionMode
DEF-USE PAIR <214, 218> For VARIABLE: menu
DEF-USE PAIR <217, 218> For VARIABLE: inflater
DEF-USE PAIR <179, 181> For VARIABLE: item
DEF-USE PAIR <206, 208> For VARIABLE: view
DEF-USE PAIR <206, 209> For VARIABLE: position
DEF-USE PAIR <206, 210> For VARIABLE: id
DEF-USE PAIR <222, 225> For VARIABLE: actionMode
DEF-USE PAIR <222, 226> For VARIABLE: menu
DEF-USE PAIR <227, 229> For VARIABLE: actionMode
DEF-USE PAIR <227, 230> For VARIABLE: menuItem
DEF-USE PAIR <232, 233> For VARIABLE: actionMode
DEF-USE PAIR <182, 183> For VARIABLE: index
DEF-USE PAIR <76, 185> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <185, 186> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <63, 171> For VARIABLE: ABOUT_ACTIVITY
DEF-USE PAIR <66, 177> For VARIABLE: CONTEXT_MENU_DUPLICATE
DEF-USE PAIR <76, 191> For VARIABLE: mAlarmListAdapter
_____

Chromosome 3: ********* NOT SELECTED **********
_____
Chromosome 4: ********** SELECTED **********
Corresponding test case
Type 13 in TextBox tvAlarmStatus in Activity1
Click on Button imageView in Activity1
Click on Button btnSetAlarm in Activity2
Click on Button alarm_list in Activity2
DEF-USE COVERAGE: 11.1%
ACCUMULATED DEF-USE COVERAGE: 66.7%
Traversed Path: 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

[THE DU-PATHS COVERED BY THIS PATH]:
DEF-USE PAIR <76,174> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <147, 174> For VARIABLE: menu
DEF-USE PAIR <147, 175> For VARIABLE: menu
DEF-USE PAIR <147, 176> For VARIABLE: menu
DEF-USE PAIR <63, 171> For VARIABLE: ABOUT_ACTIVITY
DEF-USE PAIR <66, 177> For VARIABLE: CONTEXT_MENU_DUPLICATE

[THE DU-PATHS NOT COVERED YET]:

DEF-USE PAIR <182, 189> For VARIABLE: index
DEF-USE PAIR <181, 191> For VARIABLE: info
DEF-USE PAIR <195, 197> For VARIABLE: v
DEF-USE PAIR <199, 200> For VARIABLE: checkBox
DEF-USE PAIR <209, 210> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <214, 216> For VARIABLE: actionMode
DEF-USE PAIR <214, 218> For VARIABLE: menu
DEF-USE PAIR <217, 218> For VARIABLE: inflater
DEF-USE PAIR <179, 181> For VARIABLE: item
DEF-USE PAIR <206, 208> For VARIABLE: view
DEF-USE PAIR <206, 209> For VARIABLE: position
DEF-USE PAIR <206, 210> For VARIABLE: id
DEF-USE PAIR <222, 225> For VARIABLE: actionMode
DEF-USE PAIR <222, 226> For VARIABLE: menu
DEF-USE PAIR <227, 229> For VARIABLE: actionMode
DEF-USE PAIR <227, 230> For VARIABLE: menuItem
DEF-USE PAIR <232, 233> For VARIABLE: actionMode
DEF-USE PAIR <182, 183> For VARIABLE: index
DEF-USE PAIR <76, 185> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <185, 186> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <76, 191> For VARIABLE: mAlarmListAdapter

_____

Chromosome 5: ********* NOT SELECTED ************

Chromosome 6: ********* NOT SELECTED ************

_____

Chromosome 7: ************ SELECTED ************
Corresponding test case
Type 9 in TextBox tvAlarmStatus in Activity1
Click on Button alarm_list in Activity2
Click on Button timePicker in Activity2
DEF-USE COVERAGE: 13.0%
ACCUMULATED DEF-USE COVERAGE: 79.6%
Traversed Path: 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

[THE DU-PATHS COVERED BY THIS PATH]:
DEF-USE PAIR <182, 189> For VARIABLE: index
DEF-USE PAIR <181, 191> For VARIABLE: info
DEF-USE PAIR <76, 191> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <182, 183> For VARIABLE: index
DEF-USE PAIR <76, 185> For VARIABLE: mAlarmListAdapter
DEF-USE PAIR <185, 186> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <179, 181> For VARIABLE: item

[THE DU-PATHS NOT COVERED YET]:

DEF-USE PAIR <195, 197> For VARIABLE: v
DEF-USE PAIR <199, 200> For VARIABLE: checkBox
DEF-USE PAIR <209, 210> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <214, 216> For VARIABLE: actionMode
DEF-USE PAIR <214, 218> For VARIABLE: menu
DEF-USE PAIR <217, 218> For VARIABLE: inflater
DEF-USE PAIR <206, 208> For VARIABLE: view
DEF-USE PAIR <206, 209> For VARIABLE: position
DEF-USE PAIR <206, 210> For VARIABLE: id
DEF-USE PAIR <222, 225> For VARIABLE: actionMode
DEF-USE PAIR <222, 226> For VARIABLE: menu
DEF-USE PAIR <227, 229> For VARIABLE: actionMode
DEF-USE PAIR <227, 230> For VARIABLE: menuItem
DEF-USE PAIR <232, 233> For VARIABLE: actionMode

_____

Chromosome 8: ********** NOT SELECTED **********

_____

Chromosome 9: ********** SELECTED **********
Corresponding test case
Type 12 in TextBox tvAlarmStatus in Activity1
Click on Button imageView in Activity1
Click on Button timePicker in Activity2
Click on Button btnSetAlarm in Activity2
Click on Button alarm_list in Activity2
DEF-USE COVERAGE: 3.7%
ACCUMULATED DEF-USE COVERAGE: 83.3%
Traversed Path: 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

[THE DU-PATHS COVERED BY THIS PATH]:
DEF-USE PAIR <195, 197> For VARIABLE: v
DEF-USE PAIR <199, 200> For VARIABLE: checkBox

[THE DU-PATHS NOT COVERED YET]:
DEF-USE PAIR <209, 210> For VARIABLE: mCurrentAlarm
DEF-USE PAIR <214, 216> For VARIABLE: actionMode
DEF-USE PAIR <214, 218> For VARIABLE: menu
DEF-USE PAIR <217, 218> For VARIABLE: inflater
DEF-USE PAIR <206, 208> For VARIABLE: view
DEF-USE PAIR <206, 209> For VARIABLE: position
DEF-USE PAIR <206, 210> For VARIABLE: id
DEF-USE PAIR <222, 225> For VARIABLE: actionMode
DEF-USE PAIR <222, 226> For VARIABLE: menu
DEF-USE PAIR <227, 229> For VARIABLE: actionMode
DEF-USE PAIR <227, 230> For VARIABLE: menuItem
DEF-USE PAIR <232, 233> For VARIABLE: actionMode

_____

Chromosome 10: ********* NOT SELECTED ***********

_____
PARENT SELECTION
Parent 1: Chromosome 1:
Parent 2: Chromosome 2:
Parent 3: Chromosome 4:
Parent 4: Chromosome 7:
Parent 5: Chromosome 9:
********** CROSSOVER OPERATION **********
CROSSOVER 1:
Parents used: Parent 1 ~~ Parent 3
Child 1:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 13
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kjg.png
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Child 2:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 9
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_opl.png
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
CROSSOVER 2:
Parents used: Parent 2 ~~ Parent 3
Child 3:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 11
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_opl.png
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Child 4:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 13
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
CROSSOVER 3:
Parents used: Parent 3 ~~ Parent 4
Child 5:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 9
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_opl.png
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Child 6:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 13
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
CROSSOVER 4:
Parents used: Parent 3 ~~ Parent 5
Child 7:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 13
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kjg.png
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker

Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Child 8:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 12
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_opl.png
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
CROSSOVER 5:
Parents used: Parent 4 ~~ Parent 5
Child 9:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 9
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kjg.png
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Child 10:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 12
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
********** MUTATION OPERATION **********
MUTATION 1:
Child 1:
txtInput TestCase: 13 ~~ 10
Mutated Child 1:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 10
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kjg.png
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
MUTATION 2:
Child 10:
txtInput TestCase: 12 ~~ 14
Mutated Child 10:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 14
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
****
** NEW POPULATION **
Chromosome 1:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 10
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kjg.png
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Chromosome 2:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 9
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_opl.png
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Chromosome 3:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 11
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_opl.png
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Chromosome 4:

Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 13
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Chromosome 5:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 9
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_opl.png
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Chromosome 6:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 13
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Chromosome 7:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 13
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kjg.png
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Chromosome 8:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 12
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_opl.png
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Chromosome 9:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 9
Name: imageView | Type: Button | Data: Clickable | TestCase: Click_kjg.png
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
Name: btnSetAlarm | Type: Button | Data: Clickable | TestCase: Click_btnSetAlarm
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Chromosome 10:
Name: tvAlarmStatus | Type: TextView | Data: Input | Min: 7 | Max: 15 | TestCase: 14
Name: alarm_list | Type: Button | Data: Clickable | TestCase: Click_alarm_list
Name: timePicker | Type: Button | Data: Clickable | TestCase: Click_timePicker
****

** GA TERMINATED **

** NO. OF GENERATIONS = 3
** GENERATED TEST CASES **
Test Case 1-1
Type 9 in TextBox tvAlarmStatus in Activity1
Click on Button imageView in Activity1
Click on Button timePicker in Activity2
Click on Button btnSetAlarm in Activity2
Test Case 1-2
Type 11 in TextBox tvAlarmStatus in Activity1
Click on Button alarm_list in Activity2
Click on Button timePicker in Activity2
Click on Button btnSetAlarm in Activity2
Test Case 1-3
Type 13 in TextBox tvAlarmStatus in Activity1
Click on Button imageView in Activity1
Click on Button btnSetAlarm in Activity2
Click on Button alarm_list in Activity2

Test Case 1-4
Type 9 in TextBox tvAlarmStatus in Activity1
Click on Button alarm_list in Activity2
Click on Button timePicker in Activity2
Test Case 1-5
Type 12 in TextBox tvAlarmStatus in Activity1
Click on Button imageView in Activity1
Click on Button timePicker in Activity2
Click on Button btnSetAlarm in Activity2
Click on Button alarm_list in Activity2
Test Case 1-6
Type 10 in TextBox tvAlarmStatus in Activity1
Click on Button imageView in Activity1
Click on Button timePicker in Activity2
Click on Button btnSetAlarm in Activity2
Test Case 1-7
Type 13 in TextBox tvAlarmStatus in Activity1
Click on Button alarm_list in Activity2
Click on Button timePicker in Activity2
Click on Button btnSetAlarm in Activity2
Test Case 1-8
Type 12 in TextBox tvAlarmStatus in Activity1
Click on Button imageView in Activity1
Click on Button btnSetAlarm in Activity2
Click on Button alarm_list in Activity2
Test Case 1-9
Type 13 in TextBox tvAlarmStatus in Activity1
Click on Button imageView in Activity1
Click on Button timePicker in Activity2
Click on Button btnSetAlarm in Activity2
Test Case 1-10
Type 12 in TextBox tvAlarmStatus in Activity1
Click on Button imageView in Activity1
Click on Button timePicker in Activity2
Click on Button btnSetAlarm in Activity2
Click on Button btnDot in Activity1

Fig. 8. A portion of the GA report after the completion of running the tool with the NFC Alarm Clock App, showing only the first generation and generated test cases.

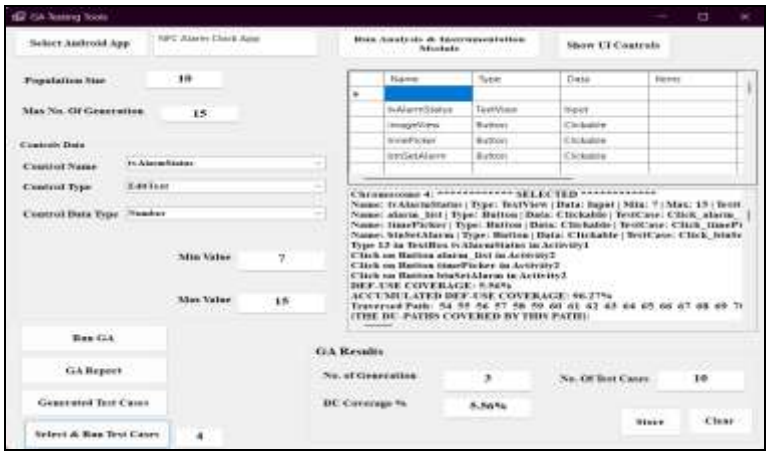


Fig. 9. A Screenshot of the UI of the Proposed Tool After Selecting Test Case 4 And Executing the AUT With It, Showing the Test Case and the Corresponding DU-Paths Coverage Percentage

7. EXPERIMENTAL EVALUATION

Two sets of experiments were conducted to rigorously evaluate the proposed GA-based automatic dataflow test data generation approach for Android applications and its supporting tool. The first set examined the ability of the GA-generated test data to achieve comprehensive coverage of DU-paths (def-use pairs), thus fulfilling the all-uses criterion for the tested applications. The second set evaluated the effectiveness of this test data in detecting errors within the Android applications. The materials used in these experiments were nine Android applications[2]. The GA parameters used were: Pop_size = 10, Max_Gen = 15, Pc = 0.8, and Pm = 0.15. All these parameters were experimentally determined.

In the first set of experiments, the proposed GA-based tool was applied to each of the nine Android applications. The execution of the tool with each application resulted in a GA report that containing detailed information about each GA generation and concluding with a list of the generated test cases for this application, as mentioned in Section VI. Table 1 shows, for each of the nine Android applications, the no. of DU-paths, the no. of generated test cases, the actual no. of generations that achieved maximum coverage, the cumulative DU-paths coverage after the application's execution with each test case. For example, the first row in this table shows that the first application has 56 DU-paths, 10 test cases were generated for it across 3 generations, and the cumulative DU-paths coverage achieved with these 10 test cases were 44.4%, 55.6%, 66.7%, 79.6%, 83.3%, 90.71%, 96.27%, 97.12%, 98.15%, and100%, respectively. The 10 test cases generated for this application are listed in the GA report shown in Figure 8.

As shown in Table 1, the tool generated test cases that achieved 100% cumulative coverage for all applications in just a few generations. These results demonstrate the effectiveness of the proposed GA-based approach in fulfilling the all-uses criterion.

Table 1. Empirical Evaluation of the Def-Use Coverage of the Test Data Generated by Proposed GA Approach.

Android App Name	**# of DU-Paths**	**# of Generated Test Cases**	**# of Actual Generations**	**Cumulative DU-Paths Coverage for the Generated Test Cases**
NFC Alarm Clock	56	10	3	44.4% - 55.6% - 66.7% - 79.6% - 83.3% - 90.71% - 96.27% - 97.12% - 98.15% - 100%
Free Dictionary	19	5	1	57.89% - 68.42% - 78.95% - 94.74% - 100%
Currency Converter	14	4	2	35.71% - 57.14% - 78.57% - 100%
Photo Editor	37	10	3	40.4% - 59.6% - 66.7% - 75.6% - 87.3% - 90.71% - 95.27% - 97.12% - 98.12% - 100%
Voice Recorder	48	10	4	30.4% - 49.6% - 66.7% - 75.6% - 87.3% - 90.71% - 95.27% - 97.12% - 98.15% - 100%
Calculator	49	9	2	17.86% - 42.86% - 57.14% - 64.29% - 71.43% - 85.71% - 89.28% - 92.85% - 100%

[2] https://github.com/gabeg805/NFC-Alarm-Clock, https://github.com/yamin8000/freeDictionaryApp, https://github.com/DevGoyalG/Currency-Converter-App, https://github.com/burhanrashid52/PhotoEditor, https://github.com/SimpleMobileTools/Simple-Voice-Recorder, https://github.com/SimpleMobileTools/Simple-Calculator, https://github.com/todotxt/todo.txt-android, https://github.com/topics/android-stopwatch, https://github.com/SimpleMobileTools/Simple-Notes

To-Do List	29	8	2	20.00% - 30.00% - 33.33% - 46.67% - 66.67% - 86.67% - 93.33% - 100.00%
Stopwatch	47	7	2	38.78% - 51.02% - 59.18% - 67.35% - 85.71% - 89.80% - 100%
Easy Note	48	11	3	27.08% - 37.50% - 41.67% - 45.83% - 46.80% - 54.17% - 58.33% - 60.42% - 77.08% - 89.58% - 100%

In the second set of experiments, various errors were seeded, one at a time, into each of the nine Android applications, after which the tool was applied to these faulty versions. The seeded errors, representative of faults that commonly occur in Android applications, were grouped into five categories: computation errors (incorrect variable assignments), domain errors (control-flow faults), object-oriented errors (issues related to classes, objects, or methods), event errors (faults in event-handling logic), and presentation errors. The first four categories correspond to faults in the Activity's source code, while the last category pertains to the XML layout file. The first column of Table 2 lists the application levels, and the subsequent columns enumerate the specific error types within each category along with their corresponding codes.

The actual output produced during execution was compared with the expected output obtained by running the original, correct version of the application using the same test cases generated by the proposed GA-based tool. In addition, the static and dynamic reports generated by the tool were examined. If an error was detected, either through a deviation in the output or through messages in the dynamic report, this indicated that the generated test cases satisfying the all-uses criterion had successfully exposed the fault.

To establish a comparative baseline, the GA-based tool was evaluated against Appium [34], a widely adopted open-source testing framework that supports automated testing across Android, iOS, and Windows platforms. Appium allows test scripts to be written in multiple programming languages and is considered a leading commercial-grade solution. Both tools were applied to the same set of nine faulty Android applications used in the experiments.

Table 3 and Figures 10 and 11 summarize the results of evaluating the error-detection capability of the test cases generated by the proposed GA-based tool, that satisfy the all-uses criterion, and compare these results with those obtained using Appium. The total number of errors seeded in the nine applications was 223. Table 3 reports both the number and percentage of detected errors for each tool. The results show that the proposed GA-based tool identified 91% of the seeded errors, whereas Appium detected only 42%.

Figure 10 illustrates the percentage of detected errors across the different error categories for both tools. As shown, the proposed GA-based tool detected 91% of computation errors, 93% of domain errors, 87% of object-oriented errors, 91% of event errors, and 95% of presentation errors. In contrast, Appium detected 52% of computation errors, 38% of domain errors, 49% of object-oriented errors, 26% of event errors, and 33% of presentation errors.

Figure 11 presents the percentage of detected errors at each application level (XML and Activity source code). The figure shows that the proposed GA-based tool detected 90% of the errors seeded in Activity source-code files and 95% of those seeded in XML layout files. By comparison, Appium detected 44% of the source-code-level errors and 33% of the XML-level errors.

The results demonstrate that the proposed GA-based tool consistently outperformed Appium in terms of error-detection capability. Specifically, test data generated through the GA-based approach achieved higher def-use coverage and revealed a greater percentage of faults compared to Appium. These findings highlight the advantage of integrating evolutionary search with data-flow analysis, confirming that maximizing def-use coverage directly enhances fault detection in Android applications.

Table 2. Classification of Seeded Error Types in the Java Activity Class and XML Layout Files

Application Level	Error Category	Error Type	Error code

Java Activity Class	Computation Errors	Wrong variable definition	C1
		Wrong arithmetic operator	C2
		Wrong variable reference	C3
		Wrong constant value	C4
		Statement wrongly placed	C5
		Missing statement	C6
	Domain Errors	Wrong relational operator	D1
		Wrong arithmetic operator	D2
		Wrong variable reference	D3
		Wrong constant value	D4
		Statement wrongly placed	D5
		Wrong logical operator	D6
		Missing condition check	D7
	Object-Oriented Errors	Incorrect class instance creation	O1
		Wrong object name	O2
		Wrong actual parameter	O3
		Incorrect actual parameters order	O4
		Incorrect formal parameters order	O5
		Wrong method call	O6
		Incorrect access modifier	O7
		Incorrect inherited class	O8
		Null Pointer Exception	O9
	Event Errors	Missing event implementation	E1
		Event swap	E2
		Message-Event Contradiction	E3
		Event Replacement	E4
XML File	Presentation Errors	Incorrect inherited class	P1
		Element wrongly placed	P2
		Missing element	P3
		Wrong attribute value	P4

Table 3. A Comparison of the Number and Percentage of Errors Detected by the Proposed GA Tool And by Appium

Error code	# of Seeded Errors	Proposed GA Tool		Appium	
		# of Detected Errors	% of Detected Errors	# of Detected Errors	% of Detected Errors
C1	10	10	100%	7	70%
C2	14	12	86%	6	43%
C3	7	7	100%	3	43%
C4	5	5	100%	1	20%
C5	8	8	100%	6	75%
C6	10	7	70%	5	50%
D1	11	11	100%	3	27%
D2	2	2	100%	2	100%
D3	4	3	75%	3	75%
D4	5	5	100%	2	40%

Error code	# of Seeded Errors	Proposed GA Tool		Appium	
		# of Detected Errors	% of Detected Errors	# of Detected Errors	% of Detected Errors
D5	3	1	33%	1	33%
D6	5	5	100%	1	20%
D7	12	12	100%	4	33%
O1	3	3	100%	1	33%
O2	4	4	100%	4	100%
O3	13	11	85%	4	31%
O4	8	7	88%	5	63%
O5	3	3	100%	1	33%
O6	19	18	95%	8	42%
O7	4	0	0%	0	0%
O8	4	4	100%	4	100%
O9	3	3	100%	3	100%
E1	5	5	100%	1	20%
E2	10	10	100%	3	30%
E3	5	3	60%	0	0%
E4	3	3	100%	2	67%
P1	6	6	100%	3	50%
P2	12	11	92%	0	0%
P3	6	6	100%	6	100%
P4	19	18	95%	5	26%
Total	**223**	**203**	91%	**94**	42%

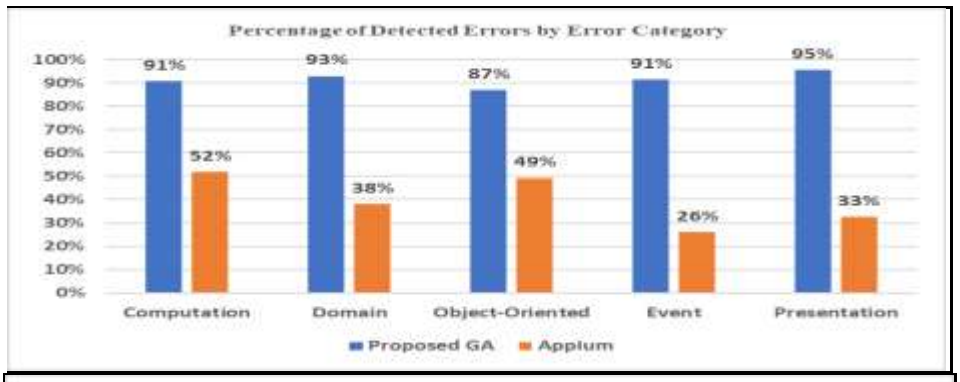


Fig. 10. Percentage of detected errors by error category

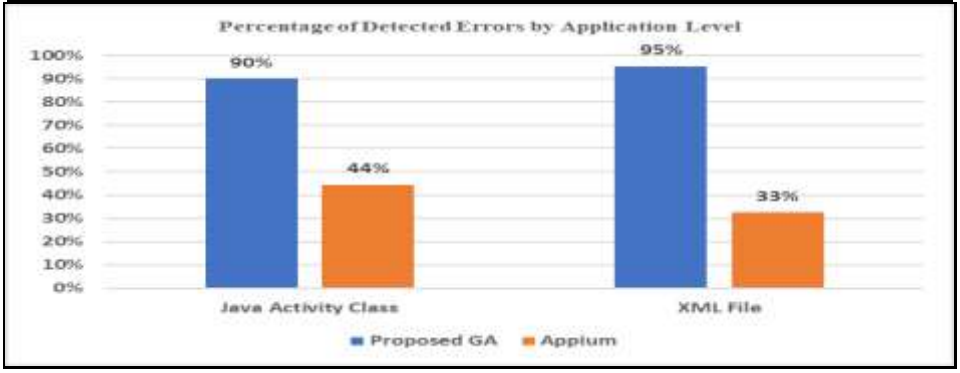


Fig. 11. Percentage of detected errors by application level

8. CONCLUSION

This paper introduced a proposed GA-based approach, supported by a dedicated tool, for automatic dataflow test data generation for Android applications. The proposed tool accepts the AUT as input, instruments it, and performs static analysis to compute the set of def-use pairs. Within the proposed GA framework, each gene in a chromosome represents a UI control object. To effectively manipulate such chromosomes, two novel genetic operators, namely block crossover and control-based mutation, were designed and implemented. During the evolutionary process, each candidate solution in the population is evaluated, and the solution that increases the accumulated def-use pair coverage percentage is selected as a test case. This iterative process continues until the maximum accumulated coverage is reached. The tool outputs a comprehensive set of generated test cases, the def-use pairs covered by each test case, and a list of any uncovered pairs.

The paper presented a case study that was conducted to demonstrate the practical application of the proposed approach on real-world Android applications. It also presented the results of the empirical evaluation of the effectiveness of the generated test data in satisfying the all-uses criterion. Finally, the paper reported the experimental results of evaluating the error detection capability of the proposed GA-based approach for Android applications and its supporting tool, in comparison with Appium, one of the leading commercial Android testing frameworks. The findings showed that the proposed approach detected 91% of all seeded errors, whereas Appium detected only 42%. These results highlight the superior effectiveness and strong error-detection capability of the proposed GA-based approach for Android application. The experimental results indicate a direct correlation: higher def-use coverage percentages consistently led to increased error detection rates.

The proposed GA-based approach has demonstrated its effectiveness in fulfilling the all-uses criterion and in exposing application faults more efficiently than existing tools such as Appium. By leveraging a customized chromosome representation and novel genetic operators, the proposed tool achieves higher def-use coverage and improved fault detection rates.

Beyond its immediate contributions, this work carries broader implications for the future of mobile application testing. First, the methodology can be extended to hybrid mobile applications (e.g., those built with frameworks such as React Native or Flutter), where complex UI interactions and cross-platform behaviors present additional challenges for test adequacy. Second, the integration of the proposed approach into Continuous Integration/Continuous Deployment (CI/CD) pipelines would enable automated, coverage-driven testing as part of the regular development workflow, thereby reducing regression risks and improving release quality. Finally, the adaptability of the GA-based framework suggests potential applications in other domains of SBST, including performance testing and security validation.

In summary, the proposed approach not only advances automated data-flow testing for Android applications but also lays the groundwork for scalable, cross-platform, and pipeline-integrated testing solutions that can significantly enhance the reliability of modern mobile software systems.

REFERENCES

[1] Ali, H.M., Hamza, M.Y., Rashid, T.A.: A comprehensive study on automated testing with the software lifecycle. Empirical Software Engineering. Springer Nature (2026). doi:10.1007/s10664-026-10234-5.

[2] Li, Y., Feng, Y., Guo, C., Chen, Z., Xu, B.: Crowdsourced test case generation for Android applications via static program analysis. Automated Software Engineering 30(26). Springer Nature (2023). doi:10.1007/s10515-023-00367-9.

[3] Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T.F., Klein, J.: Automated testing of Android apps: A systematic literature review. IEEE Transactions on Reliability 68(1), 45-66 (2019). doi:10.1109/TR.2018.2882683.

[4] Sharma, C., Sabharwal, S., Sibal, R.: A survey on software testing techniques using genetic algorithm. International Journal of Computer Science Issues (IJCSI) 10(1), 381 (2013).

[5] Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T.F., Klein, J.: Automated testing of Android apps: A systematic literature review. IEEE Transactions on Reliability 68(1), 45-66 (2019). doi:10.1109/TR.2018.2882683.

[6] Alhijawi, B., Awajan, A.: Genetic algorithms: theory, genetic operators, solutions, and applications. Evolutionary Intelligence 17, 1245-1256 (2024). doi:10.1007/s12065-023-00789-4.

[7] Srinivas, M., Patnaik, L.M.: Genetic algorithms: A survey. IEEE Computer 27(6), 17-26 (1994).

[8] Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Trans. Software Engineering 11(4), 367-375 (1985).

[9] Girgis, M.R.: Automatic test data generation for data flow testing using a genetic algorithm. Journal of Universal Computer Science 11(6), 898-915 (2005).

[10] Akhter, R., Babar, M.A., Shah, S.M.A.: Parallel multi-population genetic algorithm for path coverage in software testing. International Journal of Computer Applications 975(8887), 1-7 (2012).
[11] Setiadi, T.E., Ohsuga, A., Maekawa, M.: Efficient execution path exploration for detecting races in concurrent programs. IAENG International Journal of Computer Science 40(3), 143-163 (2013).
[12] Setiadi, T.E., Ohsuga, A., Maekawa, M.: Efficient test case generation for detecting race conditions. IAENG International Journal of Computer Science 41(2), 112-130 (2014).
[13] Takamatsu, H., Sato, H., Oyama, S., Kurihara, M.: Automated test generation for object-oriented programs with multiple targets. IAENG International Journal of Computer Science 41(3), 198-203 (2014).
[14] Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., Su, Z.: Synthesizing method sequences for high-coverage testing. ACM SIGPLAN Notices 46(10), 189-206 (2011).
[15] Boussaa, M., Barais, O., Sunyé, G., Baudry, B.: A novelty search approach for automatic test data generation. In: Proc. 8th IEEE/ACM Int. Workshop on Search-Based Software Testing, pp. 40-43 (2015).
[16] Scalabrino, S., Grano, G., Di Nucci, D., Guerra, M., De Lucia, A., Gall, H., Oliveto, R.: OCELOT: A search-based test-data generation tool for C. In: Proc. 33rd ACM/IEEE Int. Conf. Automated Software Engineering, pp. 868-871 (2018).
[17] Elgendy, I.T., Girgis, M.R., Sewisy, A.A.: A GA-based approach to automatic test data generation for ASP.NET web applications. IAENG International Journal of Computer Science 47(3) (2020).
[18] Auer, M., Adler, F., Fraser, G.: Improving search-based Android test generation using surrogate models. In: Proc. 14th Int. Symp. Search-Based Software Engineering (SSBSE 2022), LNCS vol. 13711, pp. 51-66. Springer, Singapore (2022). doi:10.1007/978-3-031-15020-7_4.
[19] Eler, M.M., Rojas, J.M., Ge, Y., Fraser, G.: Automated accessibility testing of mobile apps. In: Proc. 11th IEEE Int. Conf. Software Testing, Verification and Validation (ICST), pp. 116-126 (2018).
[20] Alshahwan, N., Harman, M.: Automated Android application testing using search based software engineering. In: Proc. 26th IEEE/ACM Int. Conf. Automated Software Engineering (ASE 2011), pp. 3-12 (2011).
[21] Gao, X., Zhang, L., Mei, H.: Automated test input generation for Android apps using evolutionary algorithms. Journal of Systems and Software 172, Art. no. 110862 (2021). doi:10.1016/j.jss.2020.110862.
[22] Fosdick, L.D., Osterweil, L.J.: Data flow analysis in software reliability. Computing Surveys 8(3), 305-330 (1976).
[23] Osterweil, L.J.: The detection of unexecutable program paths through static data flow analysis. In: Proc. IEEE COMSAC (1977).
[24] Frankl, P.G., Weyuker, E.J.: An applicable family of data flow testing criteria. IEEE Trans. Software Engineering 14(10), 1483-1498 (1988).
[25] Korel, B., Laski, J.: A tool for data flow-oriented program testing. ACM Software Proceedings, 35-37 (1985).
[26] Ntafos, S.C.: An evaluation of required element testing strategies. In: Proc. 7th Int. Conf. Software Engineering, pp. 250-256 (1984).
[27] Frankl, P.G., Weiss, S.N.: An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Transactions on Software Engineering 19(8), 774-787 (1993).
[28] Girgis, M.R.: Using symbolic execution and data flow criteria to aid test data selection. Software Testing, Verification and Reliability 3, 101-112 (1993).
[29] Mohammed, M.A., Girgis, M.R.: An Android applications data flow testing approach. International Journal of Environmental Sciences 12(1), 385-412 (2026). Available: https://theaspd.com/index.php/ijes/article/view/12598/8819.
[30] Holland, J.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Application to Biology, Control and Artificial Intelligence. MIT Press (1975).
[31] Armando, M. et al.: Espresso: Programmatically interacting with mobile apps. In: Proc. 11th ACM/IEEE Int. Conf. Automated Software Engineering (ASE) (2016).
[32] Wang, X., Xie, T.: Automated GUI testing for Android apps using Espresso. In: Proc. IEEE/ACM Int. Conf. Mobile Software Engineering and Systems (MOBILESoft) (2017).
[33] GAF-Genetic Algorithm Framework. Available: https://bitbucket.org/johnnewcombe/gaf/wiki/Home (2019).
[34] Appium Tutorial for Mobile Application Testing. Available: https://www.browserstack.com/guide/appium-tutorial-for-testing.