International Journal of Environmental Sciences
ISSN: 2229-7359
Vol. 12 No. 1, 2026
https://theaspd.com/index.php

An Android Applications Data Flow Testing Approach

Marwa A. Mohammed[1], **Moheb R. Girgis**[2]
[1,2]Department of Computer Science, Faculty of Science, Minia University, 61511, Minia, Egypt.

Abstract
Motivation: Testing mobile applications is both critical and challenging. Conventional testing methods and tools often fail to address the particular requirements of mobile applications. Effective testing is essential because application failures can lead to expensive consequences. Problem Statement: Traditional data flow testing (DFT) techniques are designed for conventional programs and do not account for the structural and behavioral differences inherent in Android applications. As a result, there is a lack of specialized DFT methods tailored for Android applications. Proposed Approach: This paper presents a novel approach to DFT specifically designed for Android applications. The approach builds a data flow model that reflects the unique structure and execution semantics of Android applications. Our testing strategy operates at four levels: Method-level, within individual methods; Interprocedural, across method calls; Activity-level, within a single Android activity; and Inter-Activity, across multiple activities within the application. Technique: At each level, the approach identifies definition-use (def-use) chains for relevant variables. Once def-use chains are established, test inputs are generated to ensure coverage of all identified variable uses thereby satisfying the "all-uses" criterion. Implementation: We implemented the approach in an automated tool called AndroidDFT. Evaluation: The paper includes a case study to demonstrate the practical application of our technique on real Android applications. Experimental results show that AndroidDFT effectively uncovers errors, highlighting its error-detection capability and effectiveness in real-world testing scenarios. Contributions: This research delivers the first comprehensive DFT solution tailored for Android applications. By extending traditional data flow analysis to account for Android's distinctive structure and lifecycle, it fills an important gap in mobile applications testing research.
Keywords: *Software testing, Android applications, mobile application testing, data-flow testing, Android application data-flow model, Android application testing techniques.*

1. INTRODUCTION

Mobile applications continue to grow rapidly in number and complexity, and as they evolve to support increasingly sophisticated functionalities and heavier user loads, the demands on their performance continue to rise. Mobile applications are now widely deployed in safety- and time-critical domains such as digital payments, m-government services, military applications, and mobile health systems [1, 2]. This highlights the importance and relevance of mobile application testing. Testing mobile applications is both challenging and crucial challenging because traditional testing methods and tools are inadequate for mobile environments, and crucial because failures can result in significant consequences.

Given the dominance of the Android platform, this paper focuses on data-flow testing for Android applications. Although the proposed approach is designed for Android, the underlying concepts are general enough to be adapted to other mobile platforms.

Android applications exhibit a unique architecture that differs significantly from traditional software systems. An Android application typically consists of XML layout files that define the user interface, Java classes that implement application logic, and persistent data storage mechanisms such as SharedPreferences or SQLite databases [3]. As Android applications expand into more critical domains, they become increasingly challenging to develop, test, and validate [4]. Prior research highlights several open challenges in mobile application testing, including device diversity, event-driven execution, and the lack of mature automated testing tools [1].

Mobile application testing refers to the process of evaluating an application's functionality, performance, and behavior against specified requirements across different devices, platforms, and network conditions [5, 6]. Ensuring the quality of a mobile application is essential for its success, and this quality can only be achieved through systematic and effective testing [5]. Verification and validation activities for mobile applications are typically conducted within controlled testing environments that simulate real-world usage scenarios [6].

Data-flow testing (DFT) is a white-box testing technique that examines how data moves through a

program, specifically how variables are defined, used, and eventually killed (overwritten or going out of scope). By applying data flow analysis, it can detect anomalies in variable usage, such as uninitialized variables, unused definitions, and incorrect data manipulations [7, 8]. Unlike control-flow testing, which focuses on execution paths, DFT emphasizes the correct handling of data throughout the program, thereby supporting the development of more reliable, maintainable, and error-free software.
This paper introduces a DFT approach tailored specifically for Android applications, which differ structurally and behaviorally from conventional programs. The proposed approach constructs a data-flow model that supports effective analysis and testing of Android applications at four levels: method, interprocedural, activity, and inter-activity. At each level, def-use (def-use) chains are identified, and test data are generated to ensure their coverage, thereby satisfying the "all-uses" criterion.
To the best of our knowledge, no prior research has addressed data-flow testing specifically for Android applications. This work therefore represents a significant advancement in mobile application testing.

The key contributions of this paper are as follows:
1. A novel data-flow testing approach specifically designed for Android applications, addressing the limitations of traditional DFT techniques when applied to mobile environments.
2. A comprehensive data-flow model that supports analysis at the method, interprocedural, activity, and inter-activity levels, reflecting Android's unique architecture and lifecycle.
3. Systematic identification of def-use chains across all levels to support rigorous all-uses coverage.
4. An automated tool, AndroidDFT, implementing the proposed DFT approach and test-generation process.
5. A case study and experimental evaluation demonstrating the effectiveness and strong error-detection capability of the proposed approach in real-world Android applications.

The remainder of this paper is organized as follows. Section 2 reviews related work on Android application testing. Section 3 provides an overview of data-flow analysis and the testing criterion adopted in this study. Section 4 introduces the proposed data-flow model for Android applications. Section 5 presents the proposed DFT approach. Section 6 describes the supporting automated tool, AndroidDFT. Section 7 provides a case study demonstrating the application of the approach. Section 8 reports the experimental results. Section 9 concludes the paper.

2. RELATED WORK

Research on mobile application testing remains relatively limited compared to traditional software testing. This section reviews the most relevant studies and highlights the gaps that motivate the need for a dedicated data-flow testing approach for Android applications.
Takala et al. [9] reported their experience applying model-based testing to Android applications. Their work discusses model construction, automated test execution, and the challenges of applying model-based techniques to mobile environments. In [10], the authors proposed an evolutionary testing framework for Android applications, where test cases evolve according to search-based heuristics to maximize code coverage. Their approach addresses limitations commonly encountered when applying evolutionary algorithms to system-level testing.
A/B testing has also been explored in the context of mobile applications. Adianta et al. [11] applied A/B testing to mobile application evaluation, addressing challenges related to UI element composition, variant delivery, and network connectivity. Similarly, Clemens and Patrick [12] conducted A/B testing using a multivariate testing tool.
Muccini et al. [1] conducted an informal review of mobile application testing challenges and identified several research gaps in areas such as mobile services testing, test automation, and test integration. Their study emphasized the unique characteristics of mobile applications, which necessitate specialized research in mobile applications testing.
Concolic testing has also been applied to mobile software. Anand et al. [13] introduced Contest, a concolic testing technique designed to mitigate the path-explosion problem by generating event sequences and eliminating redundant paths through subsumption checks.
Mahmood et al. [14] proposed EvoDroid, an evolutionary system-testing approach for Android applications that uses input and event mutations to maximize coverage.
Cloud-based testing has gained attention as well. Starov et al. [15] surveyed cloud services for mobile testing and classified them into device clouds, application lifecycle management services, and testing technique-specific tools. Their study highlighted the benefits of cloud-based testing such as enhanced

collaboration, reduced testing time, and lower development costs.

Model-based GUI testing has also been explored. Janicki et al. [16] examined the challenges and opportunities of applying model-based GUI testing to mobile applications, emphasizing the need for automated test-generation techniques.

Joorabchi et al. [17] conducted a qualitative study on real-world challenges faced by mobile developers, identifying issues such as slow emulators, platform fragmentation, and the lack of mature automated testing tools. Their findings indicate that manual testing remains the dominant practice over automated approaches. Moreover, test engineers are often required to conduct separate testing processes for each platform, and most unit testing frameworks lack support for mobile-specific features such as GPS and sensors.

Gao et al. [18] reviewed testing infrastructures and approaches for native and web-based mobile applications, analyzing the strengths and limitations of emulation-based, device-based, cloud-based, and crowd-based testing solutions. They also discussed state-of-the-art tools and the challenges faced by mobile test engineers.

In a related study, Gao et al. [19] discussed Mobile Testing as a Service (MTaaS). They proposed a TaaS (Testing as a Service) infrastructure to enable cloud-based mobile testing through two approaches: (i) mobile device test clouds and (ii) emulation-based test clouds. The study aimed to address three key challenges in mobile application testing: (i) the high cost of existing testing environments, (ii) limited scalability support, and (iii) the complexity introduced by the diversity of devices, platforms, and environments.

Search-based testing has also been applied to mobile applications. Amalfitano et al. [20] introduced an approach that combines genetic algorithms with hill-climbing techniques for mobile applications testing.

Girgis et al. [21, 22] proposed an approach for testing Android application user interfaces (UIs) involving multiple activities. In this approach, the application under test is first statically analyzed to extract activities, views, and events. Testing is then conducted at two levels: the activity level, where each activity is tested independently to ensure that it functions as expected, and the application level, where the entire application is tested to verify that all activities interact correctly to accomplish the intended tasks.

The Android development environment includes a robust testing framework [23] built on JUnit. Robolectric [24] extends this by allowing test cases to run independently of a device or emulator through library file references. However, while these frameworks automate test execution, the test cases must still be manually written by engineers.

The preceding review of mobile testing research indicates that existing studies do not address data-flow testing for Android applications. Therefore, the approach proposed in this paper fills a significant gap by introducing the first data-flow testing methodology specifically tailored for Android applications.

3. DATA FLOW ANALYSIS

In software testing, two fundamental aspects are involved: generating test data and applying an adequacy criterion. Test data generation refers to the methods used to create test cases, while an adequacy criterion defines the conditions under which testing can be considered complete [25]. Various test data adequacy criteria have been introduced in the literature, including those based on control flow and data flow.

This section presents the DFT technique adopted in our proposed approach for testing Android applications. 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. Test paths are chosen based on definition-use (def-use) chains, where a def-use chain represents the path from a variable's definition to its use, without any intervening redefinitions. A def occurs when a variable is assigned a value, while a use occurs when the variable is referenced. Variable uses are generally classified into two categories:

Computation uses (c-uses): where a variable participates in a computation or assignment.

Predicate uses (p-uses): where a variable appears in a conditional expression that determines control flow.

For c-uses, the def-use chain extends from the statement that defines the variable to the statement where it is used in a computation. For p-uses, the chain extends from the defining statement to each successor of the predicate in which the variable is used [26].

Data flow testing is a technique that tests individual procedures using their flow graphs. It involves executing paths that connect a variable's def to its points of use. This technique is also applied across interacting procedures, where interprocedural data flow analysis is used to identify def-use chains for variables defined in one procedure and used in another.

Various DFT techniques [27, 28, 29] 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. Traditional data flow analysis techniques [30], which use a program's control flow graph representation, are employed to determine def-use associations.

A program's control flow can be modeled using a directed graph, known as the control flow graph (CFG), consisting of nodes and edges. Each node corresponds to a program statement, while the edges represent possible transfers of control between nodes. A path is defined as a finite sequence of nodes connected by edges. A complete path begins at the start node and ends at an exit node. A path is considered def-clear for a variable if it does not include a redefinition of that variable.

In a CFG, defs and c-uses are associated with nodes, while p-uses are associated with edges. Data flow analysis relies on these defs and uses to construct def-use chains. A def-use chain is expressed as a tuple <(v, s), u>, where the value of variable v defined in statement s is subsequently used in statement or edge *u*.

To determine when testing is complete, a test-adequacy criterion is required [25]. In this work, we adopt the all-uses criterion introduced by Rapps and Weyuker [26]. 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 [31].

This traditional data-flow analysis forms the foundation for our Android-specific extensions, which adapt these concepts to the component-based, event-driven, and lifecycle-managed nature of Android applications.

4. THE PROPOSED ANDROID APPLICATION DATA FLOW MODEL

The initial step in the proposed DFT approach for Android applications is to extract data flow information by performing data flow analysis. The first task in this analysis is constructing a data flow model of the target application. The proposed model incorporates four types of flow graphs: CFG, ICFG (interprocedural control flow graph), ACFG (Activity Control Flow Graph), and CCFG (composite control flow graph). Each graph captures a different dimension of control and data interactions within and across Android components.

To construct these graphs, each statement (node) in the target application is assigned a unique sequence number, and directed edges are created to represent possible control flow between nodes. The following subsections describe the construction rules for each of the four flow graphs:

Control Flow Graph (CFG)

The CFG represents the data flow information of a given method, as described in Section 3. To capture this information, the CFG is annotated with the defs and uses of variables, enabling the extraction of def-use chains for the variables of interest. For instance, a def-use chain <(v, i), j> exists if a du-path connects the definition of variable *v* at node *i* to its use at node *j*.

Interprocedural Control Flow Graph (ICFG)

The ICFG captures data flow information that spans across multiple methods. It combines the CFGs of calling and called methods into a unified single-entry, single-exit CFG. This integration allows the extraction of def-use chains for variables defined in one method and used in another.

Building on this interprocedural foundation, the analysis must next account for the interaction between XML-defined UI components and the Activity code that manipulates them.

Activity Control Flow Graph (ACFG)

An Android Activity primarily consists of two parts: the XML layout, which specifies the UI components declarations, and the corresponding Activity class, which implements the logic that interacts with these components. The ACFG models the data flow information between these two parts. Specifically, it integrates the CFG of the XML layout with the ICFG of the corresponding Activity class into a single-entry, single-exit CFG. This integration enables the extraction of def-use chains for variables defined in one part of the activity and used in the other. For example, a Button declared in XML constitutes a def, and its use occurs when it is referenced in code via findViewById() or when its properties (e.g., text, listeners) are accessed or modified.

Composite Control Flow Graph (CCFG)

While the ACFG captures data flow within a single Activity, Android applications frequently pass data across Activities, which requires an additional abstraction.

Unlike traditional programs, Android Activities do not have direct calling relationships between their

internal methods, and therefore the ICFG cannot capture such data interactions. To address this limitation, the CCFG is introduced to model the data flow between interacting Activities, enabling the extraction of def-use chains that span across Activity boundaries. For example, a variable may be defined in one Activity and used in another. This extension is essential because data in Android often flows between Activities through Intents, Intent extras, or user-triggered navigation events.

The following code snippet demonstrates how to configure an activity that sends text data and responds to requests from other activities to perform this action [32]:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
<intent-filter>
<action android:name ="android.intent.action.SEND"/>
<category android:name = "android.intent.category.DEFAULT"/>
<data android:mimeType = "text/plain"/>
</intent-filter>
</activity>
```

In this example, the <action> element specifies that the Activity is responsible for handling send operation. The <category> element, declared as DEFAULT, enables the Activity to respond to standard launch requests. The <data> element defines the type of data the Activity can send. The following code snippet illustrates how to invoke the Activity described above:

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.setType("text/plain");
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
startActivity(sendIntent);
```

This code creates an Intent configured to send plain-text data. The ACTION_SEND action specifies that the Activity intends to share data, while EXTRA_TEXT attaches the text message to be transmitted. The startActivity(sendIntent) call launches an appropriate Activity capable of handling the send request.

The CCFG is constructed by linking the corresponding ACFGs of interacting Android Activities. Specifically, for each Activity's ACFG, an edge is created from its exit node to the entry node of the ACFG of another Activity. In this way, the CCFG enables the extraction of def-use chains across interacting Activities.

5. THE PROPOSED ANDROID APPLICATIONS DATA FLOW TESTING APPROACH

This section presents the proposed DFT approach for Android applications. The process begins by performing data-flow analysis on the target application to identify the defs and uses of its variables. Using this information, together with the constructed data-flow model and the data flow testing technique, described in [31], the def-use chains for all relevant variables are derived.

Given the unique structure of Android applications, the proposed approach performs testing across four levels: Method level, Interprocedural level, Activity level and Inter-Activity level. At each level, def-use chains are identified, and corresponding test data are generated to achieve coverage according to the all-uses criterion whenever possible.

The following subsections describe the data-flow analysis of Android Activities and the four testing levels:

Data Flow Analysis of Android Activities

Data-flow analysis focuses on ensuring the correct use of data within a program. In traditional software, data is stored primarily in program variables. However, Android applications introduce additional data elements that must be considered, including: UI components defined in XML layouts, Instance variables of Activities, Data passed between Activities through Intents, Persistent storage mechanisms such as SharedPreferences or SQLite, and Objects of built-in Android classes (e.g., adapters, database helpers).

Therefore, data-flow analysis techniques must be adapted to account for these Android-specific data objects, which form the core building blocks of any Android application.

In the proposed approach, we address the defs and uses of five categories of data objects commonly found in Android applications: Traditional program variables and arrays; Instance variables of the Activity class; Simple and complex Android UI components (Views) and their properties; Implicit state objects, such as Intent extras, savedInstanceState Bundles, and SharedPreferences; Objects of built-in Android classes, such as SQLite database helpers, adapters, and custom UI components. Definitions and uses for traditional variables follow the rules described in [31]. The remaining Android-specific data objects, which require additional considerations, are described below.

Defs and uses of UI components and their properties
In Android, UI components defined in XML layout files (e.g., Button, TextView, EditText, Spinner, RecyclerView) are treated as global variables accessible within the corresponding Activity class. A def occurs when the component is declared in XML, or instantiated in code using findViewById(). A use occurs when the component or one of its properties is accessed, modified, or interacted within code. Once identified, these defs and uses are used to compute def-use chains and generate appropriate test cases. Table 1 summarizes the def and use actions for selected Android UI components.

Table 1. The Def and Use Actions of Android UI Components

Android Data Item	Action	Statement	Location
Data control GridView, ImageView, ScrollView, RecyclerView	Def	Ex: <GridView android:id="@+id/simpleGridView" android:layout_width="match_parent" />	XML file
		Ex: GridView simpleGrid= (GridView) findViewById (R.id.simpleGridView);	Code-behind class
	Use	Ex: CustomAdapter customAdapter = new CustomAdapter (); simpleGrid.setAdapter(customAdapter);	Code-behind class
Data control property getCount(), getItemId(), getView(),setImageResource()	Use	Ex: simpleGrid.getItem(int i); simpleGrid.getView(int I, View view, ViewGroup group);	Code-behind class
DataSource setAdapter() (binds the data)	Use	Ex: private ArrayAdapter <String> adapter = new ArrayAdapter<>(this, android.R.layout.simple_lis, nameList); listView.setAdapter(adapter);	Code-behind class
UI control EditText, TextView, Button, RadioButton,CheckBox, ImageButton	Def	Ex: <ControlType android:id="@+id/ControlName" android:layout_width="wrap_content" />	XML file
	Use	Ex: ControlType controlName = findViewById(R.id.ControlName);	Code-behind class

UI control property
(setText, getText)
	Def	Ex: EditText editText;
		editText = findViewById(R.id.editText);
		editText.setText("Convert");
			Code-behind class

	Use	Ex: String inputText = editText.getText().toString();	
List-based control ListView, Spinner, RadioGroup, ImageView	Def	Ex: < ListType android:id= "@+id/ ListName>	XML file
		Ex: ListType listName = findViewById(R.id.ListName);	Code-behind class
	Use	Ex: listView.setAdapter(arrayAdapter); String selected = listView.getItemAtPosition(0).toString();	Code-behind class
List control property setAdapter, getAdapter, getSelectedItem, getSelectedItemPosition	Def	Ex: ListView listView = findViewById(R.id.listView); Ex: Spinner spinner = findViewById(R.id.spinner);	Code-behind class
	Use	String selectedItem = spinner.getSelectedItem().toString();	

Adapters serve as bridges between UI components and their underlying data sources. They populate data into views such as ListView, GridView, Spinner, and RecyclerView. A use action is assigned to the data-related UI component and its properties whenever the component is accessed, as illustrated in the last row of Table 1.
The following examples demonstrate some of the def and use actions for Android UI components defined in Table 1.
Example 1: Defs and uses of a Button control
The following definition element contains a def for a Button control, named simpleButton:
<Button android:id="@+id/simpleButton" android:layout_width="wrap_content"

```
android:layout_height="wrap_content" android:text="Convert"/>
```

and the following header of the Click event of simpleButton control contains a use for that control:

```
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
// Find button by ID
simpleButton1 = (Button) findViewById(R.id.simpleButton);
// Set click listener
simpleButton1.setOnClickListener(new View.OnClickListener()
{
@Override
public void onClick (View view)
{
// Show a toast message
Toast.makeText(MainActivity.this, "Button Clicked!", Toast.LENGTH_SHORT).show();
}}
```

Here, the call to findViewById() represents a use of the UI component, while attaching a click listener represents a use of its event-handling property.

Example 2: Defs and uses of a TextView control and its properties

The following definition element contains a def for a TextView control, named simpleTextView:

```
<TextView android:id="@+id/simpleTextView" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:text="Currency Converter " />
```

To illustrate how its properties are manipulated, the following code contains a def for the property Text of simpleTextView control:

```
TextView textView = (TextView) findViewById(R.id.simpleTextView);
textView.setText("Currency Converter"); //set text for text view
```

and the following statement represents a use of this property:

```
String resText = textView.getText().toString();
```

Example 3: Defs and uses of an EditText control and its properties

The following definition element contains a def for an EditText control, named simpleEditText:

```
<EditText android:id="@+id/simpleEditText" android:layout_height="wrap_content"
android:layout_width="match_parent"/>
```

The following statement contains a def for the property Text of simpleEditText control:

```
EditText simpleEditText = (EditText)findViewById(R.id.simpleEditText);
```

and the following statement contains a use of this property:

```
String editTextValue = simpleEditText.getText().toString();
```

Example 4: Defs and uses of a Spinner control

The following definition element contains a def for a Spinner control, named simpleSpinner:

```
<Spinner android:id="@+id/simpleSpinner" android:layout_width="fill_parent"
android:layout_height="wrap_content" />
```

The following code adds items to the simpleSpinner control and represents a def for that component:

```
String[] bankNames= {"BOI", "SBI", "HDFC", "PNB", "OBC"};
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
Spinner spin = (Spinner) findViewById (R.id.simpleSpinner);
spin.setOnItemSelectedListener(this);
ArrayAdapter aa = new ArrayAdapter(this,android.R.layout.simple_spinner_item, bankNames);
aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
spin.setAdapter(aa);
String selected = spin.getSelectedItem().toString();
```

The following assignment statement, which refers to the currently selected item of simpleSpinner control, contains a use for the property SelectedValue of that control:

```
string language = simpleSpinner.SelectedValue.ToString();
```

Example 5: Defs and Uses of GridView and RecyclerView controls and their Adapters

(i) The following definition element contains a def for a RecyclerView control, named dataGridUsers:

```
<androidx.recyclerview.widget.RecyclerView android:id="@+id/dataGridUsers"
   android:layout_width="match_parent" android:layout_height="wrap_content"/>
```

The following setLayoutManager() defines how items are arranged (linear list, grid, staggered grid), and setAdapter() binds data to the RecyclerView:

```
dataGridUsers = findViewById(R.id.dataGridUsers);
dataGridUsers.setLayoutManager(new LinearLayoutManager(this));
ArrayList userList = new ArrayList<>();
userList.add(new User("Ahmed Ali", "ahmed@example.com"));
UserAdapter adapter = new UserAdapter(userList);
dataGridUsers.setAdapter(adapter);
```

This code contains defs of the variables dataGridUsers, adapter and LinearLayoutManager. There are uses of dataGridUsers and adapter because it binds the control to the specified data source through the adapter. setAdapter() attaches the adapter to the RecyclerView for the first time.

(ii) The following definition element contains a def for a GridView control, named gridViewUsers:

```
<GridView android:id="@+id/gridViewUsers" android:layout_width="match_parent"
android:layout_height="wrap_content"
   android:numColumns="auto_fit" android:verticalSpacing="8dp" android:horizontalSpacing="8dp" />
```

The following code contains defs of the variables gridViewUsers and customAdapter. There are uses of gridViewUsers and customAdapter because it binds the control to the specified data source through the adapter.

```
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 gridViewUsers = findViewById(R.id.gridViewUsers);
 ArrayList usersList = new ArrayList<>();
 usersList.add(new User("Ahmed Ali", "ahmed@example.com"));
 CustomAdapter customAdapter = new CustomAdapter(this,usersList);
 gridViewUsers.setAdapter(customAdapter);
}
```

Defs and uses of implicit SharedPreferences objects

In Android, implicit session/state objects such as Intent extras, the savedInstanceState Bundle, SharedPreferences and the application-level state act as containers for storing name/value pairs. These objects allow data to be passed between activities and preserved across configuration changes, or stored persistently for later retrieval. The stored values can be accessed or modified by different Android components, enabling data sharing within a user session and supporting inter-activity communication. Below, we describe the def and use actions associated with certain implicit session/state objects, accompanied by illustrative examples.

ViewModelState property

Android applications commonly maintain UI-related state using ViewModel. These components allow an activity to observe state changes and update the UI accordingly. The state object (often an enum or data class) acts as a container for name-value pairs representing the current UI state, and it can be defined or used by different lifecycle methods within the activity.

Example 6: This example includes a def and use for a ViewModelState property named state.

<table>
<tr>
<td>1. public class MainActivity extends
AppCompatActivity
2. {
3. private MyViewModel viewModel;
4. private ProgressBar progressBar;
5. private TextView textView;
6. protected void onCreate (Bundle
savedInstanceState)
7. {
8. super.onCreate(savedInstanceState);
9. setContentView(R.layout.activity_main);
10. progressBar = findViewById(R.id.progressBar);</td>
<td>15. switch (state)
16. {
17. case LOADING:

progressBar.setVisibility(View.VISIBLE);
18. textView.setVisibility(View.GONE);
19. break;
20. case SUCCESS:
progressBar.setVisibility(View.GONE);
21. textView.setVisibility(View.VISIBLE);
22. textView.setText("Data Loaded
Successfully!");</td>
</tr>
</table>

```
11.  textView = findViewById(R.id.textView);
12.  viewModel = new

ViewModelProvider(this).get(MyViewModel.class);
         viewModel.getViewState().observe (this,
this: updateUI);
         // Start loading data
      viewModel.loadData();
      }
13.  private void updateUI (ViewModelState state)
14.  {
```

```
23.    break;
24.   case ERROR:
progressBar.setVisibility(View.GONE);
25.    textView.setVisibility(View.VISIBLE);
26.    textView.setText("Error Loading Data!");
27.    break;
28.   case EMPTY:
progressBar.setVisibility(View.GONE);
29.    textView.setVisibility(View.VISIBLE);
30.    textView.setText("No Data Available");
31.    break;
32.  }}}
```

At line 13, there is a def for the ViewModelState property state, and at line 15, there is a use for it.

SharedPreferences as Cookie

Cookies give Android developers a way to customize user interactions within Android applications. SharedPreferences allows activities to define and use name/value pairs across the application, similar to how cookies store client information in web applications. A value is defined when it is written to SharedPreferences using an Editor, and used when it is retrieved through SharedPreferences methods such as getString() or getAll().

The following example demonstrates the def and use actions for SharedPreferences.

Example 8: This example includes excerpts from the code-behind classes of two Android activities, OptionsActivity and RecommendationsActivity.

```
4  public class OptionsActivity extends
AppCompatActivity
5  {
6    private Map<String, String> books = new
HashMap<>();
7    protected void onCreate(Bundle
savedInstanceState)
8    {
9    super.onCreate(savedInstanceState);
10   books.put("AndroidApps", "0-13-606305-X");
11   books.put("Java", "0-13-606322-X");
12   Button simpleButton2 =

findViewById(R.id.simpleButton2);
13   simpleButton2.setOnClickListener(new
                    View.OnClickListener() {
14   @Override
15   public void onClick(View view) {
16   String language =

languagesList.getSelectedItem().toString();
17   String isbn = books.get(language);
18   SharedPreferences prefs =
     getSharedPreferences("AppCookies",
MODE_PRIVATE);
19   SharedPreferences.Editor editor = prefs.edit();
20   editor.putString(language, isbn);
21   editor.apply(); // stored persistently
22    }
23    });
24  }
25 }
```

```
28    public class RecommendationsActivity
extends

AppCompatActivity
29    {
30    private List<String> booksList = new
ArrayList<>();
31    protected void onCreate(Bundle
savedInstanceState)
32    {
33    super.onCreate(savedInstanceState);
34    SharedPreferences prefs =
      getSharedPreferences("AppCookies",
MODE_PRIVATE);
35    Map<String, ?> cookies = prefs.getAll();
36    if (!cookies.isEmpty())
37    {
38    for (String key : cookies.keySet())
39    {
40    String value = cookies.get(key).toString();
41    booksList.add(key + " How to Program.
ISBN: " + value);
42       }
43    }
44    ListView booksListView =

findViewById(R.id.booksListView);
45    ArrayAdapter<String> adapter = new
         ArrayAdapter<>(this,
android.R.layout.simpleListItem1,

booksList);
46    booksListView.setAdapter(adapter);
```

	47 } ...

In OptionsActivity, the application writes a key/value pair (language/ISBN) into the SharedPreferences container. This writes operation represents a def action. Specifically, at line 20, the statement editor.putString(language, isbn); stores the selected language and its corresponding ISBN into SharedPreferences. The subsequent call to editor.apply() at line 21 commits this definition persistently.
In RecommendationsActivity, the stored data is later retrieved and processed, representing use actions. SharedPreferences is accessed at line 34 using getSharedPreferences(), and all previously stored entries are obtained at line 35 via prefs.getAll(). Additional use actions occur at lines 38, 40, and 41, where the stored keys are iterated over, their associated values are retrieved, and the resulting strings are added to the booksList collection for display.

SharedPreferences as Session property
SharedPreferences can be used as a Session object to store key/value pairs that persist across activities and can be accessed throughout the application. A SharedPreferences object is treated as a global variable. The following example demonstrates the def and use actions that can be performed on SharedPreferences objects.
Example 7: Consider a ListView control named SimpleListView, defined in the XML layout file of an activity as shown below:

```
<ListView android:id="@+id/SimpleListView" android:layout_width="fill_parent"
android:layout_height="wrap_content"
   android:divider="@color/material_blue_grey_800" android:dividerHeight="1dp"/>
```

At this element, there is a def for SimpleListView control.
Suppose the following code is defined in the corresponding Activity class:

```
1  private SharedPreferences session;
2  protected void onCreate (Bundle savedInstanceState)
3 {
4   super.onCreate(savedInstanceState);
5   session = getSharedPreferences("AppSession",  MODE_PRIVATE);
6   // determine whether session contains any information
7   if (session.Count != 0)
8   {
9     if (SimpleListView.SelectedItem! = null)
10      // add name-value pair to session
11      session.setItems(SimpleListView.SelectedItem.Text, SimpleListView.SelectedItem.Value );
12    }}
```

At line 5 there is a def for the SharedPreferences object session, and at line 7 there is a use for it. At lines 9 and 11, there are uses for the property SelectedItem of SimpleListView control, and at line 11 there is a use for session, as a key/value pair is added to it, and there is a use for SimpleListView control.
Additionally, suppose a Spinner control, named simpleSpinner, is defined in the XML file of another activity in the same Android application as follows:

```
<Spinner android:id="@+id/simpleSpinner" android:layout_width="wrap_content"
android:layout_height="wrap_content"
  android:layout_centerHorizontal="true" android:layout_marginTop="100dp"/>
```

At this element, there is a def for simpleSpinner control.
Now, suppose the corresponding Activity class includes the following code:

1 private Session session; 2 protected void onCreate (Bundle savedInstanceState) 3 { 4 super.onCreate(savedInstanceState); 5 session = new Session(GetDataAdapter.this); 6 String [] bankNames ={"BOI","SBI","HDFC","PNB","OBC"}; 7 setContentView(R.layout.activity_main);	12 ArrayAdapter aa = new ArrayAdapter(this,android.R. layout.simple_spinner_item,bankNames); 13 aa.setDropDownViewResource(android.R. layout.simple_spinner_dropdown_item); 14 //Setting the ArrayAdapter data on the Spinner 15 spin.setAdapter(aa);

```
8   //Getting the instance of Spinner and
applying
    // OnItemSelectedListener on it
9   Spinner spin = (Spinner)

findViewById(R.id.simpleSpinner);
10    spin.setOnItemSelectedListener(this);
11    //Creating the ArrayAdapter instance
having bank name list
```

```
16    // determine whether Session contains any
information
17    if (session.Count! = 0)
18    {
19    // display Session's name-value pairs
20    foreach (string keyName in session.Keys)
21    simpleSpinner.setItems (keyName +"
ISBN#: " +

session[keyName]);
22    }// end if
23  }//end onCreate
```

There is a def for the Session property session at line 5, and uses at lines 17, 20 and 21, as it is referenced in session.Count, session.Keys and session[keyName], respectively. There is a def for Spinner control spin at line 9, as an item is added to it, and uses at lines 10 and 15.

Defs and uses of instance variables of the code-behind class

In object-oriented programming (OOP), a class's instance variables can be defined, accessed, and modified by the methods within that class. Android activities follow the same fundamental principles: instance variables declared in the activity's code-behind class are accessible to all of its lifecycle methods. Although Android XML layout files cannot reference Java instance variables directly, they can define resource values, such as strings, identifiers, and layout attributes, that are subsequently accessed within the activity through the automatically generated class. These XML-defined resources function as globally accessible constants and participate in def-use relationships when retrieved and manipulated in the activity's code.

To illustrate how XML-defined resources can be defined in XML files and subsequently used within an activity's methods (and how additional instance variables may be defined and used within the activity itself), consider the following excerpts from AndroidManifest.xml, strings.xml, and MainActivity.java:

```
1. <manifest xmlns:android=
“http://schemas.android.com/apk/res/android"
package= "com.example.hp_lap.free_dictionary">
2. <application android:allowBackup="true"
android:label=
“@string/app_name" android:supportsRtl ="true"
android:theme= “@style/Theme.AppCompat.
Light.NoActionBar">
3. <activity android:name=".MainActivity">
4. <intent-filter>
5. <action
android:name="android.intent.action.MAIN"/>
6. <category android:name=
“android.intent.category.LAUNCHER" />
</intent-filter>
7. </activity>
8. </application>
9. </manifest>
```

```
10.  <resources>
11.  <string name="app_name">Free
Dictionary</string>
12.  <string name="welcome_message">
MyApplication
</string>
13.  </resources>
14.  package
com.example.hp_lap.free_dictionary;
15.  import android.os.Bundle;
16.  import android.widget.Toast;
17.  import
androidx.appcompat.app.AppCompatActivity;
18.  protected void onCreate (Bundle
savedInstanceState)
19.  {
20.  super. onCreate(savedInstanceState);
21.  String message=
getString(R.string.welcome_message);
22.  String title = "Welcome" + message;
23.  Toast.makeText(tilte).show();
24.  }
```

In this example, the presentation files (XML resources) define values, such as strings, that are later accessed within the activity's code-behind class through the generated class. Although XML files do not contain instance variables in the OOP sense, their resource definitions participate in def-use relationships when referenced in the activity. The string resource welcome_message has a def at line 12 and a corresponding

use at line 21. Within the onCreate method, the variables message and title also exhibit def-use behavior: message is defined at line 21 and used at line 22, while title is defined at line 22 and used at line 23 in the output statement.

Objects of built-in classes

In Android applications, objects from built-in classes (such as SQL Server classes) as well as custom controls (such as button widgets) can be created and used within the code-behind class. The following subsections describe how def and use actions are applied to objects of these classes.

SQL database access in Android applications

Android applications frequently retrieve or store data using relational databases. This functionality is typically provided through SQLite or through custom helper classes that encapsulate database operations. When an Android application connects to an external SQL database (e.g., MySQL or SQL Server), this is typically performed using JDBC or a dedicated API wrapper.

In such cases, objects such as Connection, Statement, and ResultSet participate in def-use relationships analogous to those found in server-side environments. Likewise, Android UI components, such as ListView, TextView, and adapters, interact with the retrieved data through well-defined def-use actions.

The following example illustrates these relationships in the context of an Android activity that connects to a remote SQL database, retrieves data, and displays the results using a ListView. In Android, database interaction is typically performed through JDBC objects such as Connection, Statement, and ResultSet, or through helper classes that encapsulate these operations. A use action is associated with each of these objects whenever one of their methods is invoked, for example, when a Statement executes a query or when a ResultSet iterates over returned rows. These method invocations represent uses of the previously defined objects and their properties, as demonstrated in Table 2.

Example 9:

```
1.  <LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
2.  <ListView
        android:id="@+id/listView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="8dp"/>
3.  <TextView
        android:id="@android:id/text1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
4.  <TextView
        android:id="@android:id/text2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
    </LinearLayout>
  ...
9.  protected void onCreate(Bundle savedInstanceState)
10. {
11. super.onCreate(savedInstanceState);
12. setContentView(R.layout.activity_main);
13. listView = findViewById(R.id.listView);
14. dbHelper = new MSSQLHelper();
```

```
15. Class.forName("com.mysql.jdbc.Driver");
16. Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/sonoo","root","root");
18. con.open();
19. Statement stmt = con.createStatement();
20. ResultSet rs = stmt.executeQuery("select * from emp");
21. while (rs.next()) {
22. System.out.println(rs.getInt(1) + " " + rs.getString(2) + " "
        + rs.getString(3)); }
23. MSSQLHelper dbHelper = new MSSQLHelper();
25. ArrayList<HashMap<String, String>> userList =
        dbHelper.getUsers();
26. if (userList.isEmpty())
27. {
28.  Log.e("Database", "No data found");
29.  }
30.  else
31. {
32.  SimpleAdapter adapter = new SimpleAdapter(this,
        userList,
    android.R.layout.simple_list_item_2, new
        String[]{"ID", "Name"}, new
```

	int[]{android.R.id.text1, android.R.id.text2}); 33. listView.setAdapter(adapter); 34. con.close(); 35. }}}

In this example, several Android UI components and database-related objects participate in clearly defined def-use relationships. There are defs for TextView controls, text 1 and text 2, at line 3 and 4, and use at line 32 when the SimpleAdapter maps database fields to these view identifiers.
There is a def for the savedInstanceState parameter at line 9 as part of the activity lifecycle method and use at line 11 when it is passed to the superclass implementation.
There is a def for the ListView control listView in the XML layout at line 2 and use at line 13 when it is retrieved through findViewById. There is a def for the variable listView at line 13 and use at line 33 when the adapter is assigned to it.
There is a def for the database helper object dbHelper at line 14 and use at line 25 when the getUsers() method is invoked to retrieve data from the database. There is a def for the database connection object con at line 16 and use at lines 18, 19, 20, and 34 during connection opening, statement creation, query execution, and connection closure, respectively.
There is a def for the Statement object stmt at line 19 and use at line 20 to execute the SQL query. There is a def for the ResultSet object rs at line 20 and use at lines 21 and 22 during row iteration and data extraction.
There is a def for the userList collection at line 25 and use at lines 26 and 32 when checking whether the list is empty and when binding the retrieved data to the adapter. Finally, there is a def for the adapter object at line 32 and use at line 33 when it is assigned to the ListView for display.

Table 2. The Def and Use Actions of the Objects of Some JDBC Classes

JDBC Class Description	Action	Affected Class Object	Example
Connection (Represents a session with a specific database. Used to establish connectivity and manage transactions)	def	Object created using DriverManager.getConnection()	Connection conn = DriverManager.getConnection(dbURL, username, password); - def of conn
	use	Object used to bind parameters and execute queries	ps.setString(1, "MM"); ps.executeUpdate(); - use of ps
Statement (Used to execute static SQL queries against the database)	def	Object created using the createStatement() method	Statement stmt = conn.createStatement(); - def of stmt
	use	Object used to execute SQL commands	ResultSet rs = stmt.executeQuery("SELECT * FROM users"); - use of stmt
PreparedStatement (Used to execute parameterized SQL queries, improving performance and security)	def	Object created using prepareStatement()	PreparedStatement ps = conn.prepareStatement("INSERT INTO users VALUES (equals, 230)"); - def of ps
	use	Object used to bind parameters and execute queries	ps.setString(1, "Add"); ps.executeUpdate(); - use of ps
ResultSet (Represents the result of a SQL query. Used to read data returned from the database)	def	Object returned by executing a query	ResultSet rs = stmt.executeQuery("SELECT * FROM users"); - def of rs
	use	Object used to iterate over query results	while(rs.next()) { String name = rs.getString("name"); - use of rs

Custom button controls

In Android, custom button controls can be defined in XML layouts and programmatically configured in the activity's code-behind class. Android uses the Button class directly, and customization is achieved through XML attributes and method bindings. A def action is set when the Button object is instantiated via findViewById() and its properties, such as text, background, and padding, are assigned values. A use action occurs when the button is added to the layout and its behavior is defined through event listeners. To demonstrate how def and use actions are applied to a custom button in Android, consider the following example:

Example 10:

```
<Button android:id="@+id/customButton" android:layout_width ="wrap_content"
android:layout_height ="wrap_content" android:text="Custom Button" android:textColor= "#FFFFFF"
android:textStyle="bold" android:background ="@drawable/custom_button" android:padding="12dp"/>
...
7. public class MainActivity extends AppCompatActivity
8. {
9.   protected void onCreate (Bundle savedInstanceState)
10. {
11.  super.onCreate(savedInstanceState);
12.  setContentView(R.layout.activity_main);
13.  Button customButton = findViewById(R.id.customButton);
14.  // Use: assign behavior through a click listener
15.  customButton.setOnClickListener(new View.OnClickListener() {
16.  @Override
17.  public void onClick(View v)
18.  {
19.  // Show a toast message when the button is clicked
20.  Toast.makeText(MainActivity.this, "Custom Button Clicked!", Toast.LENGTH_SHORT).show();
21.   } });
23.  }}
```

A def action occurs at line 13 when the customButton object is instantiated using findViewById(). A use action is then established at lines 15-21, where the button is referenced to assign an OnClickListener and to execute associated behavior when clicked. Within the listener, the invocation of Toast.makeText() represents the runtime use of the previously defined button. This example demonstrates how def and use actions are associated with Android UI components through their creation and subsequent interaction in the activity.

Data Flow Testing Levels of Android Applications

An Android Activity consists of two primary components: XML layout files (e.g., activity_main.xml), which define user interface elements such as Button, TextView, EditText, Spinner, and RecyclerView; and the Activity class (e.g., MainActivity.java), which contains the logic, event handlers, and data used to control and interact with these UI elements. In the proposed DFT approach for Android applications, testing is performed across four levels: Method level, Interprocedural level, Activity level, and Inter-Activity level. These levels are described below.

Method-level testing

Method-level testing focuses on variables whose def-use chains are confined to a single method within an Activity or supporting Java class. Each method is treated as an independent unit, and defs and uses of local variables, method parameters, and instance variables accessed within the method are identified. Using the method's CFG, def-use chains are extracted, and test cases are generated to cover them.

This level resembles traditional data-flow testing; however, Android introduces additional considerations. Methods frequently interact with: lifecycle callbacks (e.g., onCreate(), onStart(), onResume()), event handlers (e.g., button click listeners), and UI components accessed via findViewById() or view binding. These interactions must be incorporated into the CFG to ensure complete coverage.

Interprocdural-level testing

Interprocdural-level testing extends the analysis across method boundaries. In Android applications, methods often interact through: event callbacks, listener interfaces, helper or utility classes, database access classes, and adapter classes (e.g., RecyclerView.Adapter). The ICFG integrates the CFGs of calling and called methods into a unified graph. This enables the extraction of def-use chains where a variable is

defined in one method and subsequently used in another. For example, a variable may be defined in onCreate() and later used inside a click listener or in a helper method invoked by that listener. The ICFG captures these relationships, enabling comprehensive testing of data interactions across method boundaries.

Activity-level testing

Activity-level testing considers the combined behavior of: the XML layout (UI declarations), the Activity class (implementation logic), event handlers, and lifecycle methods. The ACFG integrates the CFG of the XML layout, which captures the UI component declarations, and the ICFG of the Activity class (representing the code that manipulates these components). This integration enables the extraction of def-use chains that span both UI definitions and Java code. Examples include: a Button defined in XML (def) and referenced in code via findViewById() (use), an EditText defined in XML (def) and its text retrieved in code (use), a Spinner defined in XML (def) and its selected item accessed in code (use). Activity-level testing ensures that all UI-related data flows are thoroughly analyzed and validated.

Inter-activity level testing

Inter-activity level testing is applied when Android applications transfer data between Activities using Intents and Intent extras. A variable may be defined in one Activity, packaged into an Intent, and then used in another Activity.

Traditional interprocedural analysis cannot capture these interactions because Activities do not invoke each other through direct method calls. Instead, navigation occurs through the Android framework. To address this, the CCFG, which links the ACFGs of interacting Activities, is analyzed to extract the def-use chains across Activity boundaries, and test cases are then generated to cover them. Examples of inter-Activity data flows include: passing a username from a login Activity to a dashboard Activity, sending a selected item from a list Activity to a details Activity, transferring text or images using implicit Intents. Inter-Activity-level testing ensures that all such cross-component data flows are covered and validated.

6. THE TOOL DESCRIPTION

This section presents AndroidDFT, the automated tool developed to implement the proposed data-flow testing approach for Android applications. Figure 1 shows a screenshot of the AndroidDFT user interface. Developed in Java, AndroidDFT operates in two main phases: the Static Analysis & Instrumentation Phase and the Testing Phas*e*. The following subsections detail each of these phases.

The Static Analysis & Instrumentation Phase

In this phase, the XML layout files and the corresponding Java source files of the Android Activities under test serve as input. The tool analyzes the statements within each Activity and constructs the corresponding data-flow model, which incorporates four types of flow graphs, namely CFG, ICFG, ACFG, and CCFG, as described in Section 4. After building the model, the tool inserts probes into the application to record the line numbers of the execution path during each test run. Two types of probes are used: one inserted into XML-related statements and the other inserted into Activity source-code statements, as illustrated in the following examples.

```
15. <activity android:name=".LicensesActivity" android:exported="false" />
    <%XMLGETPT(sw , 15);%> // Probe for XML, where sw is a StreamWriter.
60. preferences = new Preferences(this);
     streamWriterOut.Write("60"); // Probe for code-behind
```

The output of this phase includes the instrumented version of the Android application, the generated edge list, the du-path list, and a static analysis report summarizing key structural and data-flow elements of the Activity or Activities under test. This phase consists of two modules: one responsible for processing the XML layout file(s) and another responsible for processing the corresponding Activity source file(s). Figure 2 illustrates the structure of the Static Analysis & Instrumentation Phase.

International Journal of Environmental Sciences
ISSN: 2229-7359
Vol. 12 No. 1, 2026
https://theaspd.com/index.php

Fig. 1. A screenshot of the AndroidDFT user interface

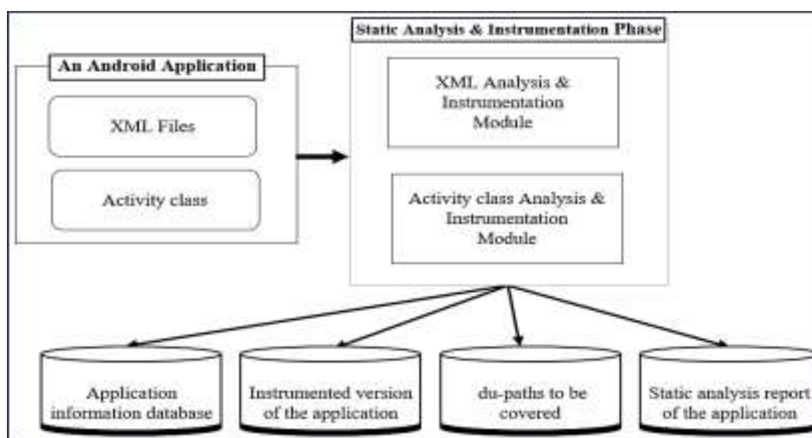

Fig. 2. The structure of the Static Analysis & Instrumentation Phase

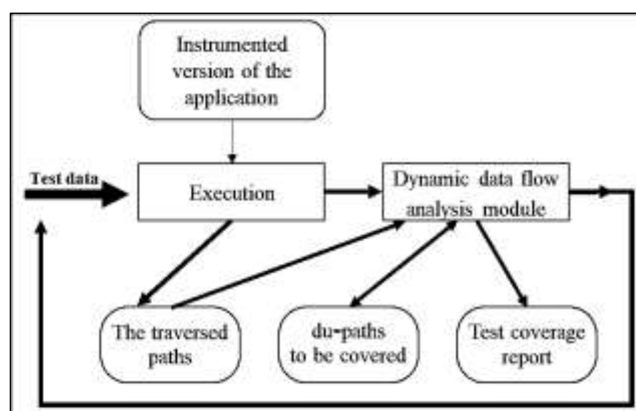

Fig. 3. The structure of the Testing Phase

The Testing Phase

This phase takes as input the instrumented version of the application and the list of du-paths to be covered. Its outputs include a dynamic test-coverage report that identifies the execution paths traversed during runtime, the du-paths covered by those paths, and any du-paths that remain uncovered. Figure 3 illustrates the structure of this phase. The Testing Phase consists of two main tasks:

Executing the instrumented Android application, produced during the Static Analysis & Instrumentation Phase, using test data and recording the corresponding execution path.

Using the recorded execution path together with the target du-path list, the dynamic data-flow analysis module generates the test-coverage report, indicating which du-paths were successfully covered and those that remain uncovered in the current execution. The tester reviews the coverage report after each run and repeats the process as needed until all du-paths are covered, whenever possible.

7. CASE STUDY

To demonstrate the practical applicability of the proposed data-flow testing approach, a case study was conducted on a sample Android application titled "Calculator Application for a Simple Personal Agenda."[1]

[1] https://github.com/Mehedi61/Calculator-App

Figure 4 shows selected parts of the application's XML layout file and its associated Activity class.

```
// The activity_main xml file of the Calculator Activity
0 <?xml version="1.0" encoding="utf-8"?>
1 <androidx.fragment.app.FragmentContainerView
xmlns:android=http://schemas.android.com/apk/res/android
    xmlns:tools=http://schemas.android.com/tools android:layout_width="match_parent"
android:layout_height="match_parent"
    android:id="@+id/main" tools:context="com.example.hp_lap.calculator.MainActivity"
tools:layout="@layout/fragment_main">
2 <Button android:id="@+id/simpleButton1" android:layout_width="fill_parent"
android:layout_height="wrap_content"
    android:layout_centerHorizontal="true" android:layout_marginTop="100dp" android:background="#00f"
    android:drawableRight="@drawable/ic_launcher" android:hint="=" android:padding="5dp"
    android:textColorHint="#fff" android:textSize="20sp" android:textStyle="bold|italic" />
3 <EditText android:id="@+id/simpleEditText" android:layout_width="fill_parent"
android:layout_height="wrap_content"
    android:layout_centerInParent="true" android:text="  " android:textSize="25sp" />
4 </androidx.fragment.app.FragmentContainerView> // The AndroidManifest xml file of the Calculator Activity
5 <?xml version="1.0" encoding="utf-8"?>
6 <manifest xmlns:android=http://schemas.android.com/apk/res/android
xmlns:tools="http://schemas.android.com/tools.calculator">
7 <uses-permission android:name="android.permission.VIBRATE" />
8 <application android:name=".App" android:allowBackup="true"
android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules" android:icon="@mipmap/ic_launcher"
android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round" android:supportsRtl="true"
android:theme="@style/color_0" tools:targetApi="31">
9 <service android:name=".MyQSTileService" android:exported="true" android:label="@string/app_name"
    android:icon="@drawable/tile_icon"
android:permission="android.permission.BIND_QUICK_SETTINGS_TILE">
10  <meta-data android:name="android.service.quicksettings.ACTIVE_TILE" android:value="true" />
11  <intent-filter>
12  <action android:name="android.service.quicksettings.action.QS_TILE" />
13  </intent-filter>
14  </service>
15  <activity android:name=".LicensesActivity" android:exported="false" />
16  <profileable android:shell="true" tools:targetApi="29" />
17  <activity android:name=".MainActivity" android:exported="true"
android:windowSoftInputMode="stateAlwaysHidden">
18  <intent-filter>
19  <action android:name="android.intent.action.MAIN" />
20  <category android:name="android.intent.category.LAUNCHER" />
21  </intent-filter>
22  </activity>
23  <activity android:name= “. AboutActivity" android:exported="false" />
24  <activity android:name=".settings.SettingsActivity" android:exported="false" />
25  </application>
26  </manifest>
```

```
// The Java code-behind class of the Calculator Activity
51  public class MainActivity extends AppCompatActivity
52 {
53    private ActivityMainBinding binding;
54    private Preferences preferences;
55    private Button simpleButton1;
56    private EditText simpleEditText;
```

International Journal of Environmental Sciences
ISSN: 2229-7359
Vol. 12 No. 1, 2026
https://theaspd.com/index.php

```
57   @Override
58   protected void onCreate(Bundle savedInstanceState)
59   {
60     preferences = new Preferences(this);
61     Config.init();
62     preferences.getTheme(),
63     preferences.getColor(),
64     preferences.getDynamicColor(),
65     preferences.getGroupingSeparatorSymbol(),
66     preferences.getDecimalSeparatorSymbol(),
67     preferences.getNumberPrecision(),
68     preferences.getMaxScientificNotationDigits(),
69     preferences.getSwipeHistoryAndCalculator(),
70     preferences.getSwipeDigitsAndScientificFunctions(),
71     preferences.getAutoSavingResults(),
72     preferences.getVibration(),
73     preferences.getSoundEffects()
74   );
75   CalculatorViewModel.init(preferences.getDegreeMod());
76   simpleButton1 = (Button) findViewById(R.id.simpleButton1);
77   simpleEditText = (EditText) findViewById(R.id.simpleEditText);
78   UnitConverterFragment.setPhysicalQuantity(preferences.getPhysicalQuantity());
79   UnitConverterFragment.setUnit(preferences.getUnit());
80   if (!preferences.getDynamicColor())
81   {
82     setTheme(getResources());
83   }
84   else
85   {
86     setTheme(R.style.dynamicColors);
87   } .....
```

Fig. 4. Selected parts of the XML layout files and the Java activity class of the sample Android application

In the proposed approach, the first step involves analyzing the data flow of the target Activity. This step consists of two main tasks: constructing the Activity's data-flow model as described in Section 4, and identifying the defs and uses of the Activity's variables as outlined in Section 5. Figure 5 presents a screenshot of the AndroidDFT user interface displaying portion of the static analysis report generated for the sample Android application, which shows the defs and uses of two variables.

Fig. 5. A Screenshot of the AndroidDFT user interface displaying a portion of the static analysis report generated for the sample Android application

Figure 6 presents the CFG of the Activity's components, annotated with the corresponding defs and uses. Based on the extracted data-flow information, the def-use chains of the Activity's variables are derived, followed by the construction of the du-paths, using the technique described in [31]. Table 3 lists the resulting du-paths. Each du-path is defined by three elements: the def-node, where the variable is defined; the use-node, where the variable is used; and the killing nodes, which contain redefinitions of the same variable and therefore must not appear along the path. A value of (-1) in the killing-node column indicates that the du-path has no killing nodes.

After identifying the du-paths for the application under test, the tester executes its instrumented version using appropriate test data. During each execution, the traversed path is recorded. At the end of the run, the execution path is examined to determine whether it covers any of the constructed du-paths, and a report is generated indicating which du-paths remain uncovered. A path is considered to cover a du-path if it contains a subpath that begins at the def-node and ends at the use-node without passing through any killing nodes [31]. The tester then re-executes the application with different test data until all feasible du-paths are covered. Some du-paths may be infeasible and therefore cannot be covered by any test data. Figure 7 shows a screenshot of the AndroidDFT user interface displaying a portion of the coverage report generated for the sample Android application, whereas Figure 8 shows the complete coverage report after three successful test runs. For each run, the report displays the execution path traversed, the coverage percentage, the du-paths that were covered, and the du-paths that remain uncovered. As shown, all required du-paths were successfully covered within the three test runs.

8. EXPERIMENTAL EVALUATION

Experiments were conducted to evaluate the effectiveness and error-detection capability of the proposed Android data-flow testing approach, as implemented in the AndroidDFT tool. The materials used in these experiments were thirteen Android applications[2].

In the experiments, various errors were seeded, one at a time, into each of the thirteen 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 4 lists the application levels, and the subsequent columns enumerate the specific error types within each category along with their corresponding codes.

[2] https://github.com/mohanramphp/Currency-Converter, https://github.com/Mehedi61/ NFC Alarm Clock-App, https://github.com/ayush221b/Voice-Recorder, https://github.com/himanshusharma89/Dictionary-App, https://github.com/burhanrashid52/PhotoEditor
https://github.com/azharimm/NotesApp, https://github.com/akshay2211/ToDoApp-Android, https://github.com/Sudhirkrish/Stopwatch
https://github.com/karan1105/Unit-Converter-Android-App, https://github.com/smarteist/Android-Quiz-App
https://github.com/technolifestyle/Weather-App-Android, https://github.com/nisrulz/android-examples/tree/master/Flashlight
https://github.com/shaon2016/Firebase-Login-Android

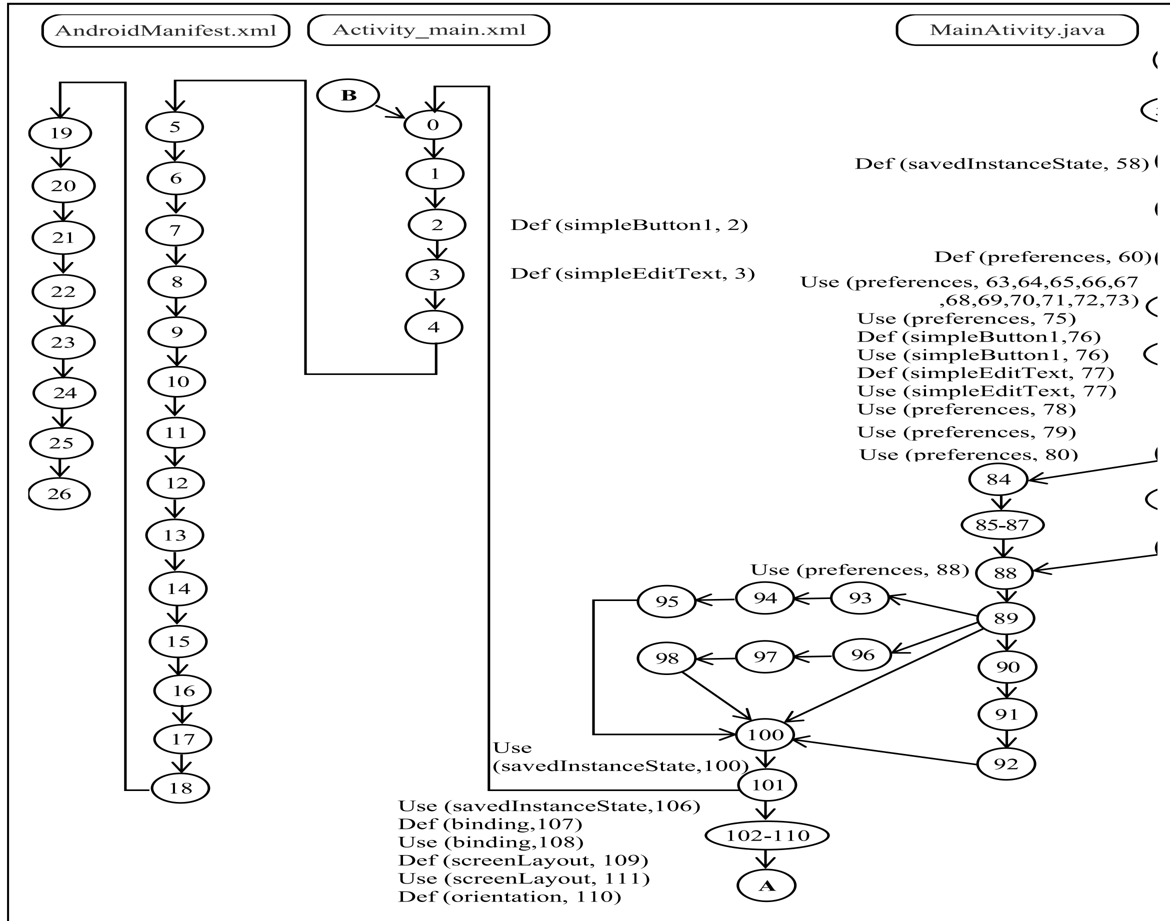

Fig. 6. The CFGs of the components of sample Android application

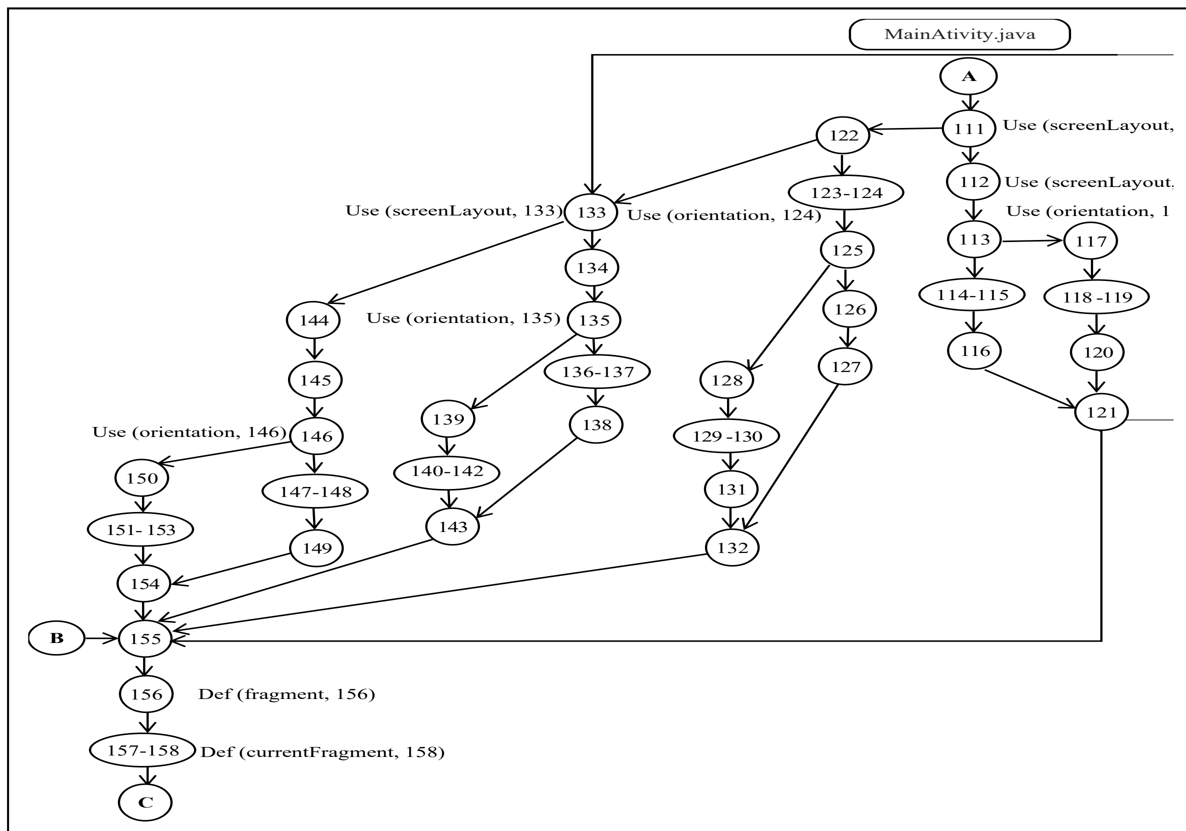

Fig. 6. The CFGs of the components of sample Android application (Continued)

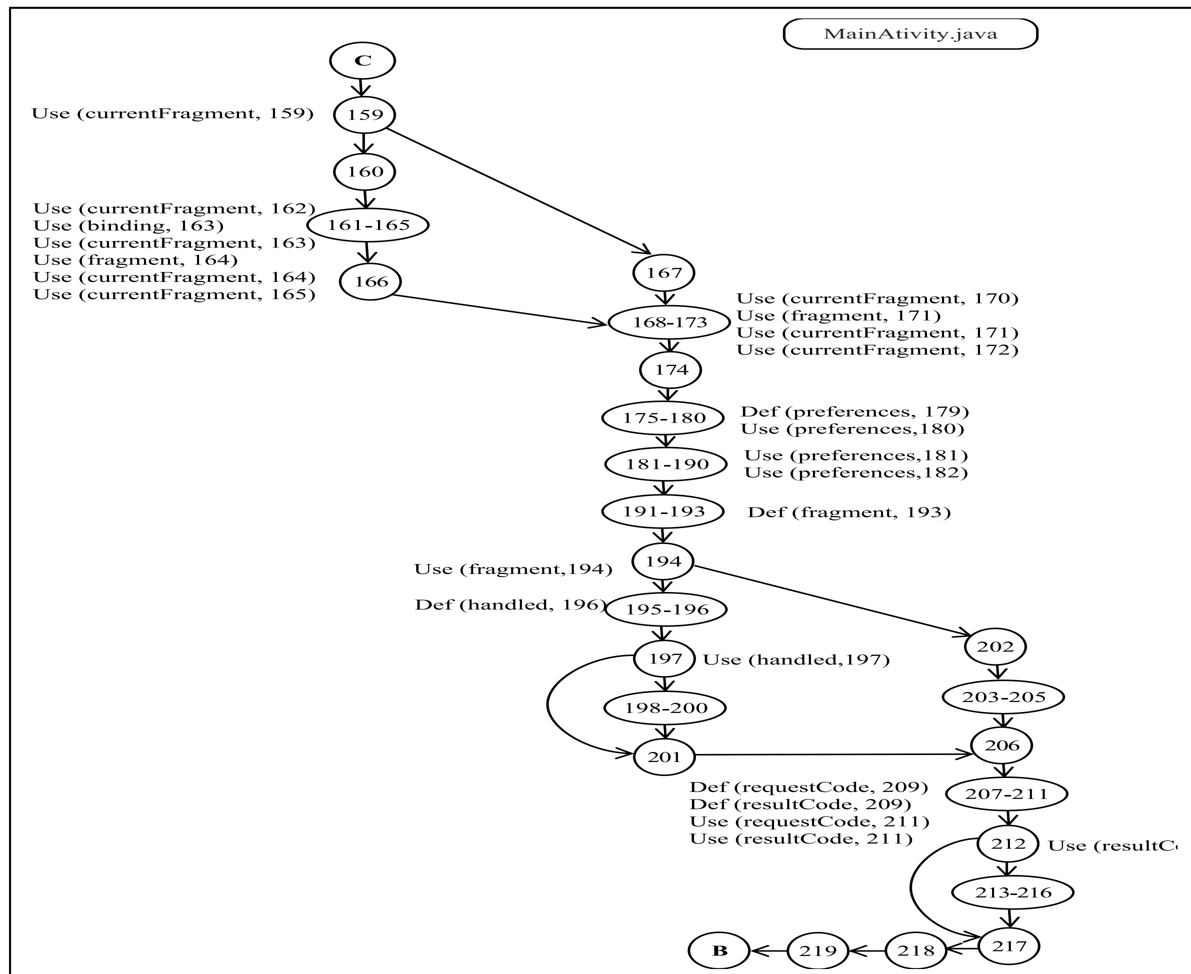

Fig. 6. The CFGs of the components of sample Android application (Continued)

Table 3. The list of du-paths and killing nodes of the sample Android application

No.	Variable	Def node	Use node	Killing Nodes	No.	Variable	Def node	Use node	Killing Nodes
0	Preferences	60	62	-1	25	screenLayout	109	122	-1
1	Preferences	60	63	-1	26	Orientation	110	124	-1
2	Preferences	60	64	-1	27	screenLayout	109	133	-1
3	Preferences	60	65	-1	28	Orientation	110	135	-1
4	Preferences	60	66	-1	29	Orientation	110	146	-1
5	Preferences	60	67	-1	30	currentFragment	158	159	-1
6	Preferences	60	68	-1	31	currentFragment	158	162	-1
7	Preferences	60	69	-1	32	Binding	107	163	-1
8	Preferences	60	70	-1	32	currentFragment	158	163	-1
9	Preferences	60	71	-1	33	Fragment	156	164	-1
10	Preferences	60	72	-1	34	currentFragment	158	164	-1
11	Preferences	60	73	-1	35	currentFragment	158	165	-1

12	Preferences	60	75	-1	36	currentFragme nt	158	170	-1
13	simpleButton1	2	76	-1	37	Fragment	156	171	-1
14	simpleEditText	3	77	-1	38	currentFragme nt	158	171	-1
17	Preferences	60	78	-1	39	currentFragme nt	158	172	-1
18	Preferences	60	79	-1	40	Preferences	179	180	-1
19	Preferences	60	80	-1	41	Preferences	179	181	-1
20	Preferences	60	88	-1	42	Preferences	179	182	-1
21	savedInstanceSta te	58	100	-1	43	Fragment	193	194	-1
22	savedInstanceSta te	58	106	-1	44	Handled	196	197	-1
22	Binding	107	108	-1	45	requestCode	209	211	-1
23	screenLayout	109	111	-1	46	resultCode	209	211	-1
24	Orientation	110	113	-1	47	resultCode	209	212	-1

Fig. 7. A screenshot of the AndroidDFT user interface displaying a portion of the coverage report generated for the sample Android application

[THE PATH NUMBER AND THE DEF-USE PARIRS THAT COVERED AND NOT COVERED]
------------------------- Run Number (1) -------------------------
[THE PATH NUMBER IS: 2]
The DU-PATH Coverage Percent: 81.25%
[51 52 53 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 88 89 96 97 98 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 121 155 156 157 158 159 160 161 162 163 164 165 166 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 206 207 208 209 210 211 212 213 214 215 216 217 218 219 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 51 52 53 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 88 89 96 97 98 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 121 155 156 157 158 159 160 161 162 163 164 165 166 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 206 207 208 209 210 211 212 213 214 215 216 217 218 219]
[THE DU-PATHS COVERED BY THIS PATH]:
DEF-USE PAIR <60,62> For VARIABLE: preferences
DEF-USE PAIR <60,63> For VARIABLE: preferences
DEF-USE PAIR <60,64> For VARIABLE: preferences
DEF-USE PAIR <60,65> For VARIABLE: preferences

International Journal of Environmental Sciences
ISSN: 2229-7359
Vol. 12 No. 1, 2026
https://theaspd.com/index.php

DEF-USE PAIR <60,66> For VARIABLE: preferences
DEF-USE PAIR <60,67> For VARIABLE: preferences
DEF-USE PAIR <60,68> For VARIABLE: preferences
DEF-USE PAIR <60,69> For VARIABLE: preferences
DEF-USE PAIR <60,70> For VARIABLE: preferences
DEF-USE PAIR <60,71> For VARIABLE: preferences
DEF-USE PAIR <60,72> For VARIABLE: preferences
DEF-USE PAIR <60,73> For VARIABLE: preferences
DEF-USE PAIR <60,75> For VARIABLE: preferences
DEF-USE PAIR <2,76> For VARIABLE: simpleButton1
DEF-USE PAIR <3,77> For VARIABLE: simpleEditText
DEF-USE PAIR <60,78> For VARIABLE: preferences
DEF-USE PAIR <60,79> For VARIABLE: preferences
DEF-USE PAIR <60,80> For VARIABLE: preferences
DEF-USE PAIR <60,88> For VARIABLE: preferences
DEF-USE PAIR <58,100> For VARIABLE: savedInstanceState
DEF-USE PAIR <110,113> FOR VARIABLE: orientation
DEF-USE PAIR <58,106> FOR VARIABLE: savedInstanceState
DEF-USE PAIR <107,108> FOR VARIABLE: binding
DEF-USE PAIR <109,111> FOR VARIABLE: screenLayout
DEF-USE PAIR <158,159> FOR VARIABLE: currentFragment
DEF-USE PAIR <158,162> FOR VARIABLE: currentFragment
DEF-USE PAIR <107,163> FOR VARIABLE: binding
DEF-USE PAIR <158,163> FOR VARIABLE: currentFragment
DEF-USE PAIR <156,164> FOR VARIABLE: fragment
DEF-USE PAIR <158,164> FOR VARIABLE: currentFragment
DEF-USE PAIR <158,165> FOR VARIABLE: currentFragment
DEF-USE PAIR <179,180> FOR VARIABLE: preferences
DEF-USE PAIR <179,181> FOR VARIABLE: preferences
DEF-USE PAIR <179,182> FOR VARIABLE: preferences
DEF-USE PAIR <193,194> FOR VARIABLE: fragment
DEF-USE PAIR <196,197> FOR VARIABLE: handled
DEF-USE PAIR <209,211> FOR VARIABLE: requestCode
DEF-USE PAIR <209,211> FOR VARIABLE: resultCode
DEF-USE PAIR <209,212> FOR VARIABLE: resultCode
[THE DU-PATHS NOT COVERED YET]:
DEF-USE PAIR <158,170> FOR VARIABLE: currentFragment
DEF-USE PAIR <156,171> FOR VARIABLE: fragment
DEF-USE PAIR <158,171> FOR VARIABLE: currentFragment
DEF-USE PAIR <158,172> FOR VARIABLE: currentFragment
DEF-USE PAIR <109,122> FOR VARIABLE: screenLayout
DEF-USE PAIR <110,124> FOR VARIABLE: orientation
DEF-USE PAIR <109,133> FOR VARIABLE: screenLayout
DEF-USE PAIR <110,135> FOR VARIABLE: orientation
DEF-USE PAIR <110,146> FOR VARIABLE: orientation
------------------------------ Run Number (2) ------------------------------
[THE PATH NUMBER IS: 3]
The DU-PATH Coverage Percent: 93.75%
[51 52 53 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
84 85 86 87 88 89 90 91 92 100 101 102 103 104 105 106 107 108 109 110 111 112 122 123 124
128 129 130 131 132 155 156 157 158 159 167 168 169 170 171 172 173 174 175 176 177 178 179
180 181 182 183 184 185 186 187 188 189 190 191 192 192 193 194 195 196 197 198 199 200 201

```
202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 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 51 52 53 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 84 85 86 87 88 89 90 91 92 99 100
101 102 103 104 105 106 107 108 109 110 111 122 123 124 125 126 127 132 155 156 157 158 159
167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189
190 191 192 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211
212 213 214 215 216 217 218 219]
[THE DU-PATHS COVERED BY THIS PATH]:
DEF-USE PAIR <158,170> FOR VARIABLE: currentFragment
DEF-USE PAIR <156,171> FOR VARIABLE: fragment
DEF-USE PAIR <158,171> FOR VARIABLE: currentFragment
DEF-USE PAIR <158,172> FOR VARIABLE: currentFragment
DEF-USE PAIR <109,122> FOR VARIABLE: screenLayout
DEF-USE PAIR <110,124> FOR VARIABLE: orientation
[THE DU-PATHS NOT COVERED YET]:
DEF-USE PAIR <109,133> FOR VARIABLE: screenLayout
DEF-USE PAIR <110,135> FOR VARIABLE: orientation
DEF-USE PAIR <110,146> FOR VARIABLE: orientation
------------------ Run Number (3) ------------------
[THE PATH NUMBER IS: 5]
The DU-PATH Coverage Percent: 100%
[51 52 53 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
84 85 86 87 88 89 90 91 92 100 101 102 103 104 105 106 107 108 109 110 111 112 122 133 134
135 136 137 138 143 155 156 157 158 167 168 169 170 171 172 173 174 175 176 177 178 179 180
181 182 183 184 185 186 187 188 189 190 191 192 192 193 194 195 196 197 198 199 200 201 202
203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 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 51 52 53 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 84 85 86 87 88 89 90 91 92 99 100 101 102
103 104 105 106 107 108 109 110 111 122 133 144 145 146 150 151 152 153 154 155 156 157 158
167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189
190 191 192 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211
212 213 214 215 216 217 218 219]
[THE DU-PATHS COVERED BY THIS PATH]:
DEF-USE PAIR <109,133> FOR VARIABLE: screenLayout
DEF-USE PAIR <110,135> FOR VARIABLE: orientation
DEF-USE PAIR <110,146> FOR VARIABLE: orientation
[THE DU-PATHS NOT COVERED YET]:
   NONE
```

Fig. 8. The Complete coverage report generated by AndroidDFT for the sample Android application after three successful test runs

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 data. 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 selected test data satisfying the all-uses criterion had successfully exposed the fault. If no error was detected in the initial run, the application was re-executed with additional test data to cover any remaining uncovered def-use chains, as indicated by the dynamic report. This process was repeated until the fault was revealed or all feasible def-use chains were covered without detecting the error.

Since no prior research has investigated data-flow testing in Android applications, there is currently no automated data-flow testing tool specifically designed for Android. Therefore, to evaluate the effectiveness of our proposed approach, we compared it with one of the leading commercial Android testing tools, namely Appium [33]. Appium is an open-source framework that supports automated testing across

International Journal of Environmental Sciences
ISSN: 2229-7359
Vol. 12 No. 1, 2026
https://theaspd.com/index.php

multiple platforms, including Android, iOS, and Windows, and allows test scripts to be written in various programming languages. We applied Appium to the same thirteen faulty Android applications used in our experiments.

Table 5 and Figures 9 and 10 summarize the results of evaluating the error-detection capability of the proposed tool, AndroidDFT, using test data that satisfy the all-uses criterion, and compare these results with those obtained using Appium. Table 5 reports both the number and percentage of detected errors for each tool. The results show that AndroidDFT identified 96% of the seeded errors, whereas Appium detected only 42%.

Figure 9 illustrates the percentage of detected errors across the different error categories for both tools. As shown, AndroidDFT detected 96% of computation errors, 92% of domain errors, 96% of object-oriented errors, 100% of event errors, and 100% of presentation errors. In contrast, Appium detected 49% of computation errors, 36% of domain errors, 34% of object-oriented errors, 64% of event errors, and 55% of presentation errors.

Figure 10 presents the percentage of detected errors at each application level (XML and Activity source code). The figure shows that AndroidDFT detected 100% of the errors seeded in XML layout files and 95% of those seeded in Activity source-code files. By comparison, Appium detected 55% of the XML-level errors and 41% of the source-code-level errors.

Overall, these results demonstrate the effectiveness and strong error-detection capability of the proposed data-flow testing approach for Android applications and its supporting tool.

Table 4. 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

9. CONCLUSION

This paper introduced a proposed approach for data-flow testing of Android applications. The approach consists of three main steps. The first step involves constructing an Android application data-flow model to support subsequent analysis. This model integrates four types of flow graphs: CFG, ICFG, ACFG, and CCFG.

The second step involves performing data-flow analysis on the target Android application to collect information about variable defs and uses. In this work, five types of variables and data objects commonly found in Android applications were considered: traditional program variables and arrays, instance variables of the Activity class, simple and complex Android UI controls and their properties, implicit session or state variables, and objects of built-in classes such as database-related components and classes. The third step computes the def-use chains of the application variables by applying the collected data-flow information in conjunction with the constructed model and the data-flow testing technique described in [31].

Testing is conducted at four levels: method, interprocedural, activity, and inter-activity. At each level, the def-use chains of the relevant variables are identified, and test data is generated to ensure coverage according to the all-uses criterion.

The paper also presented AndroidDFT, the automated tool developed to implement the proposed approach. AndroidDFT operates in two main phases: the Static Analysis & Instrumentation Phase and the Testing Phase. In addition, the paper presented a case study in which the proposed approach was applied to a sample Android application, demonstrating its practical applicability. Finally, the paper reported the experimental results evaluating the proposed data-flow testing approach for Android applications and its supporting tool, AndroidDFT, in comparison with Appium, one of the leading commercial Android testing frameworks. The findings showed that the proposed approach detected 96% of all seeded errors, whereas Appium detected only 42%. These results highlight the superior effectiveness and strong error-detection capability of the proposed data-flow testing approach for Android application. Overall, the proposed approach makes a notable contribution to the field of mobile application testing, as no prior research has directly addressed data-flow testing in the context of Android application development.

We are currently developing an automated test data generation method based on a Genetic Algorithm for data flow testing of Android applications. This method will be incorporated into AndroidDFT to enhance its ability to generate automated test data that meets the all-use criterion for the Android application under test.

Table 5. A Comparison of the Number and Percentage of Errors Detected by the Proposed Tool and by Appium

Error code	**# of Seeded Errors**	**Proposed Tool**		**Appium**	
		# of Detected Errors	**% of Detected Errors**	**# of Detected Errors**	**% of Detected Errors**
C1	2	2	100%	2	100%
C2	11	11	100%	7	64%
C3	3	3	100%	2	67%
C4	6	6	100%	3	50%
C5	7	6	86%	2	29%
C6	16	15	94%	6	38%
D1	11	11	100%	6	55%
D2	3	3	100%	3	100%
D3	7	5	71%	3	43%
D4	8	8	100%	2	25%

D5	3	3	100%	0	0%
D6	4	4	100%	0	0%
D7	3	2	67%	0	0%
O1	3	3	100%	2	67%
O2	5	5	100%	4	80%
O3	7	7	100%	4	57%
O4	10	9	90%	0	0%
O5	2	2	100%	0	0%
O6	35	33	94%	10	29%
O7	3	3	100%	0	0%
O8	2	2	100%	2	100%
O9	3	3	100%	2	67%
E1	3	3	100%	1	33%
E2	1	1	100%	0	0%
E3	1	1	100%	1	100%
E4	6	6	100%	5	83%
P1	3	3	100%	1	33%
P2	2	2	100%	0	0%
P3	4	4	100%	3	75%
P4	13	13	100%	8	62%
Total	**187**	**179**	**96%**	**79**	**42%**

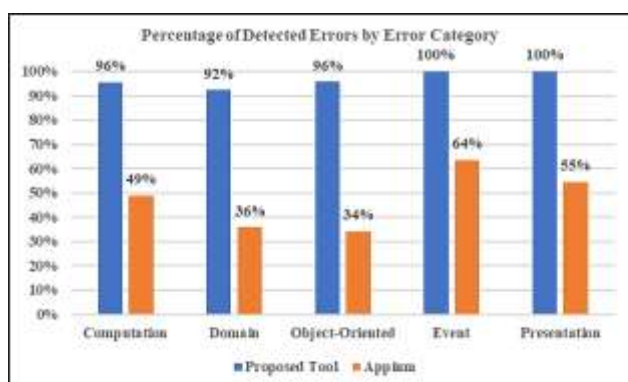

Fig. 9. Percentage of detected errors by error category

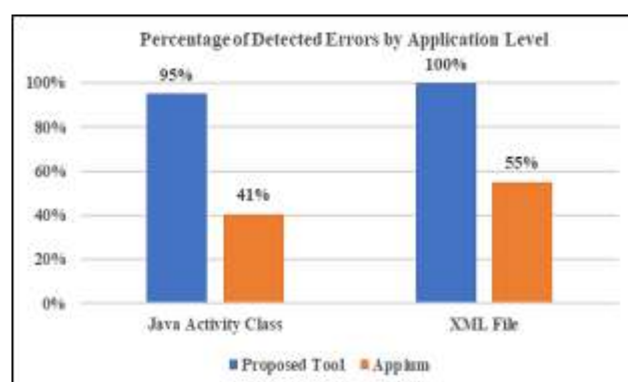

Fig. 10. Percentage of detected errors by application level

REFERENCES

[1] Muccini, H., Di Francesco, P., Esposito, R.: Software testing of mobile applications: Challenges and future research directions. In: Proc. 7th Int. Workshop on Automation of Software Test (AST), pp. 29-35 (2012). doi: 10.1109/AST.2012.6220014
[2] Payet, É., Spoto, F.: Static analysis of Android programs. Information and Software Technology 54, 1192-1201 (2012)
[3] Li, L., Bissyandé, T., Klein, J., Le Traon, Y.: An investigation into the use of common libraries in Android applications. In: Proc. 23rd IEEE Int. Conf. Software Analysis, Evolution, and Reengineering (SANER), pp. 1-10 (2016)
[4] Nagowah, L., Sowamber, G.: A novel approach of automation testing on mobile devices. In: Proc. Int. Conf. Computer & Information Science (ICCIS), pp. 924-930 (2012)
[5] Holl, K., Elberzhager, F.: Mobile application quality assurance. Advances in Computers, Elsevier (2018)
[6] Zein, S., Salleh, N., Grundy, J.: A systematic mapping study of mobile application testing techniques. Journal of Systems and Software 117, 334-356 (2016)
[7] Fosdick, L.D., Osterweil, L.J.: Data flow analysis in software reliability. Computing Surveys 8(3), 305-330 (1976)
[8] Osterweil, L.J.: The detection of unexecutable program paths through static data flow analysis. In: Proc. IEEE COMSAC (1977)
[9] Takala, R., Kämäräinen, T., Hämäläinen, T.D.: A systematic literature review on software test automation. In: Proc. 11th Int. Conf. Quality Software (QSIC), pp. 45-54 (2011). doi: 10.1109/QSIC.2011.11
[10] Lu, Y., Zulie, P., Jingju, L., Yi, S.: Android malware detection technology based on improved Bayesian classification. In: Proc. 3rd Int. Conf. Instrumentation, Measurement, Computer, Communication and Control (IMCCC), pp. 1338-1341 (2013)
[11] Muhammad, A., Inggriani, L.: A/B test tools of native mobile applications. In: Proc. Int. Conf. Data and Software Engineering (2014)

[12] Clemens, H., Patrick, H.: Multivariate testing of native mobile applications. In: Proc. Int. Conf. Advances in Mobile Computing and Multimedia (2014)
[13] Anand, A., Naik, M., Harrold, M.J., Yang, H.: Automated concolic testing of smartphone applications. In: Proc. ACM SIGSOFT Symp. Foundations of Software Engineering (FSE'12) (2012)
[14] Mahmood, R., Mirzaei, N., Malek, S.: EvoDroid: Segmented evolutionary testing of Android applications. In: Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering (2014)
[15] Starov, O.: Testing-as-a-Service for mobile applications: State-of-the-art survey. In: Dependability Problems of Complex Information Systems, Springer (2015)
[16] Janicki, M., Katara, M., Pääkkönen, T.: Obstacles and opportunities in deploying model-based GUI testing of mobile software: A survey. Software Testing, Verification and Reliability 22(5), 313-341 (2012)
[17] Joorabchi, M.E., Mesbah, A., Kruchten, P.: Real challenges in mobile application development. In: Proc. ACM/IEEE Int. Symp. Empirical Software Engineering and Measurement, pp. 15-24 (2013)
[18] Gao, J., Bai, X., Tsai, W.T., Uehara, T.: Mobile application testing: A tutorial. Computer, 46-55 (2014)
[19] Gao, J., Bai, X., Tsai, W.T., Uehara, T.: Mobile Testing-as-a-Service (MTaaS): Infrastructures, issues, solutions and needs. In: Proc. 15th Int. Symp. High-Assurance Systems Engineering (HASE), pp. 158-167 (2014)
[20] Amalfitano, D., Amatucci, N., Fasolino, A.R., Tramontana, P.: Agrippin: A novel search-based testing technique for Android applications. In: Proc. Int. Workshop on Software Development Lifecycle for Mobile (2015)
[21] Girgis, M.R., Abdel Latef, B.A., Akl, T.: A GUI testing strategy and tool for Android applications. International Journal of Computing 19(3), 355-364 (2020)
[22] Girgis, M.R., Abdel Latef, B.A., Akl, T.: A system for GUI testing of Android applications with multiple activities. International Journal of Computer Applications 174(19), 25-35 (2021)
[23] Android Testing Framework. Available: http://developer.android.com/guide/topics/testing/index.html
[24] Robolectric. Available: http://pivotal.github.com/robolectric/
[25] Frankl, P.G., Weiss, S.: An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Trans. Software Engineering 19(8), 774-787 (1993)
[26] Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Trans. Software Engineering 11(4), 367-375 (1985)
[27] Frankl, P.G., Weyuker, E.J.: An applicable family of data flow testing criteria. IEEE Trans. Software Engineering 14(10), 1483-1498 (1988)
[28] Korel, B., Laski, J.: A tool for data flow-oriented program testing. ACM Software Proceedings, 35-37 (1985)
[29] Ntafos, S.C.: An evaluation of required element testing strategies. In: Proc. 7th Int. Conf. Software Engineering, pp. 250-256 (1984)
[30] Aho, V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
[31] Girgis, M.R.: Using symbolic execution and data flow criteria to aid test data selection. Software Testing, Verification and Reliability 3, 101-112 (1993)
[32] Introduction to Activities. Available: https://developer.android.com/guide/components/activities/intro-activities
[33] Appium Tutorial for Mobile Application Testing. Available: https://www.browserstack.com/guide/appium-tutorial-for-testing