Journal of Innovations in Computer Science and Engineering
Shahid Beheshti Unversity

January 2026, Volume 3, Issue 2

Feature-Oriented Model-Based GUI Test Generation with App-Specific Oracles for Mobile Applications

Ali Asghar Yarifard, **CODE ORCID:** 0000-0002-0238-0022
Department of Computer Engineering, Ferdowsi University of Mashhad, Iran. a_yarifard@yahoo.com
Saeed Araban✉, **CODE ORCID:** 0000-0002-8205-1631
Department of Computer Engineering, Ferdowsi University of Mashhad, Iran. araban@um.ac.ir
Samad Paydar, **CODE ORCID:** 0000-0002-3370-2151
Department of Computer Engineering, Ferdowsi University of Mashhad, Iran. samadpaydar@gmail.com
Vahid Garousi, **CODE ORCID:** 0000-0001-6590-7576
Queen's University, Belfast, UK, Azerbaijan Technical University, Baku, Azerbaijan. v.garousi@qub.ac.uk

Abstract

Context: Mobile application functional features are implemented through GUI event handlers, whereas user-visible behaviors emerge from meaningful sequences of these handlers. Detecting defects, especially non-crashing functional ones, requires test cases enriched with app-specific oracles capable of validating both individual handlers and higher-level user workflows, called targeted test cases. Although numerous automated GUI testing techniques have achieved high structural coverage, systematically generating targeted test cases with tailored functional oracles remains a significant challenge.

Objective: This study aims to bridge this gap by introducing a systematic framework that generates targeted test cases to exercise both isolated event handlers and complex, feature-level user scenarios.

Method: We propose a feature-oriented, model-based GUI testing approach that models the System Under Test (SUT) via introducing two complementary models: the Event Model (EM), which captures individual GUI handlers and their hierarchical dependencies, and the Event Relation Model (ERM), which formalizes higher-level functional user workflows. The test suite generated from these models are enriched with GUI-level invariants as app-specific oracles for runtime verification. We evaluate the effectiveness of the generated suite using both code coverage metrics and mutation analysis.

Results: An empirical evaluation on five open-source mobile applications shows that the generated test suite achieve, on average, 85% statement coverage and a 97.3% mutation score on the evaluated benchmarks. Furthermore, the integration of GUI-level invariants enhances the mutation score by 19% to 32%.

Conclusion: The findings demonstrate that combining feature-oriented behavioral modeling with app-specific test oracles enhances fault-detection effectiveness and code coverage on small-to-medium benchmarks. However, broader industrial evaluations remain necessary to fully assess fully assess generalizability across large-scale, complex applications.

CC BY

KEYWORDS:
Mobile app testing; GUI testing; Model-based testing; Feature-oriented testing; Test oracle; GUI invariants; Targeted test cases

1. Introduction

Mobile applications ("apps" in short) have evolved from rudimentary utilities into sophisticated, feature-rich software systems that serve as primary gateways for digital services across diverse domains such as communication, finance, healthcare, and enterprise operations [1]. This rapid evolution has been accompanied by an unprecedented growth in their adoption and usage worldwide, with over 230 billion downloads in 2022 alone [2], highlighting the scale, diversity, and criticality of the mobile software ecosystem.

As mobile apps continue to grow in complexity and societal importance, ensuring their reliability and quality has become a critical challenge [3–6]. Despite this importance, many mobile apps are still inadequately tested prior to release due to diverse usage scenarios, rapid release cycles, device fragmentation, and limited testing resources [6, 7]. As a consequence, released mobile apps frequently suffer from defects that negatively affect their functionality, usability, and user experience [6, 8]. Addressing these challenges requires scalable and effective automated testing solutions.

In response to this challenge, Graphical User Interface (GUI) testing has become a key approach for improving quality assurance, as the GUI constitutes the primary interface for interactions with mobile apps [9]. While notable research efforts have been devoted to automating GUI testing [10, 11], recent studies indicate that this field faces critical challenges [6, 12–14], including effective test case generation [13, 14], app-specific test oracle problem [6, 15], deeper understanding of app behavior [13], test minimization and prioritization [13], usability [14], maintainability [16], and appropriate bug reporting [14].

Submit Date: 2026-07-28
Revise Date: 2026-10-05
Accept Date: 2026-10-28
✉ Corresponding author

Among these challenges, the automated generation of effective, semantically meaningful GUI test cases is particularly critical [17]. Detecting functional defects—especially non-crashing ones—requires not only exercising individual GUI event handlers [14], but also systematically exploring meaningful combinations of them that collectively realize app features, such as data-entry and data-manipulation workflows [11]. However, many existing tools struggle to generate such targeted and semantically meaningful test cases [11, 12]. Although recent efforts [18–21] have explored leveraging human knowledge from interaction histories [18], manual test artifacts [20], and behavioral patterns [19] to tackle this challenge, the integration of such knowledge into systematic model-driven test generation remains limited.

Furthermore, this limitation is compounded by the challenge of equipping generated test suite[1] with app-specific test oracles capable of revealing non-crashing functional defects [3, 6, 11, 12, 22]. Currently, most existing automated GUI testing approaches can only reveal crash-related failures due to using implicit test oracle [7, 22, 23], while methods for revealing non-crashing functional defects often demand intensive manual effort to define explicit test oracles [20, 27, 28] in the generated test cases or suffer from limited generalizability and high false positives [32–34].

Taken together, these limitations reveal a fundamental gap: existing approaches lack an integrated solution that (i) systematically generates targeted test cases to cover feature-level behaviors realized through interaction sequences, and (ii) enriches the generated test cases with app-specific test oracles to reveal non-crashing functional defects.

To address this gap, we propose a feature-oriented, model-based GUI test generation framework. The core insight is that since mobile app features are realized within GUI event handlers and their meaningful semantic combinations, systematically modeling these units enables the synthesis of coherent, feature-level test scenarios. To achieve this, our approach leverages two complementary abstract models: an *Event Model (EM)* to capture individual handler structures via static analysis, and an *Event Relation Model (ERM)* to formalize feature-level workflows by modeling meaningful relations among event handlers, leveraging human-defined test scenarios [11]. Moreover, GUI-level invariants, introduced and validated in our earlier work [31], are utilized to address the test oracle problem, by serving as app-specific oracles embedded directly into the generated test suites.

In our earlier study [31], our primary objective was dedicated to the *GUI oracle problem*—specifically, establishing the concept of GUI invariants and qualitatively evaluating their correctness and readability as test oracles. While [31] introduced high-level conceptual definitions of models (i.e., EM and ERM) to demonstrate how extracted invariants might be embedded in the generated test suite, it did not provide the algorithmic formulations, architectural mechanics, or automated code synthesis required for a full test generation system. In contrast, this study shifts the focus entirely to introduce the proposed framework, which formally defines EM and ERM models, describes mechanisms for automated Espresso test script generation, and conducts an extensive empirical evaluation covering code coverage, fault detection against baseline tools.

In summary, this study makes the following contributions:

- We design and formalize a feature-oriented, model-based test generation framework that systematically targets feature-level behaviors by capturing GUI event handlers and their semantic combinations.
- We provide explicit algorithmic formulations for automatically constructing abstract models (i.e., EM and ERM) to facilitate the semi-automated generation of targeted GUI test cases.
- We realize an automated test engine that utilize extracted models to generate executable, standalone Espresso test suites which are enriched with GUI-level invariants as app-specific oracles.
- We empirically evaluate the proposed framework on real-world mobile apps, assessing its effectiveness in terms of code coverage, mutation analysis, and fault-detection criteria against baseline tools.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the proposed framework. Section 4 describes the empirical evaluation and results. Section 5 discusses implications, limitations, and threats to validity. Section 6 concludes the paper.

2. Related work

Due to the inherent challenges of manual testing—such as being labor-intensive, time-consuming, and error-prone—numerous techniques and tools have been developed over the last decade to partially automate GUI testing for mobile apps. These efforts have primarily aimed to improve scalability, reduce human effort, and enhance defect detection effectiveness. Several studies have attempted to classify and compare these approaches; for instance, surveys such as [16, 17] introduced foundational taxonomies in this area. Building on these foundations, we extend their categorization to provide a more comprehensive overview of recent advances and to contextualize the contribution of our proposed approach.

2.1. Interaction-Driven Approaches

Interaction-driven approaches focus on facilitating direct interactions over the GUI and primarily encompass framework-based and capture–replay techniques. Framework-based approaches, such as Appium [28], Espresso [35], and UI Automator [36], provide standardized interfaces for interacting with GUI elements and executing user actions. While widely adopted in industry, they serve primarily as execution drivers rather than autonomous test generators; test logic, interaction sequences, and validation assertions must still be manually specified and maintained by developers, leading to substantial engineering overhead as the SUT evolves [19]. To reduce scripting effort, capture–replay techniques (e.g., Barista [18], Espresso Test Recorder [20], and RANDR [21]) record and replay user interaction traces [16]. However, the recorded scripts are tightly coupled to the specific GUI structure observed during recording; even minor changes in widget identifiers, layouts, or navigation structures can invalidate previous tests, rendering them fragile and costly to maintain [16]. Crucially, both framework-based and capture–replay techniques

[1] is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors.

lack automated mechanisms to systematically explore unseen GUI states or define robust, app-specific test oracles, limiting their ability to detect non-crashing functional defects.

2.2 Exploration-Driven Approaches

Exploration-driven approaches replace manual test-authoring effort with automated event exploration, systematically traversing the SUT's state space. They include random-, search-, and symbolic-based techniques. Random (or "monkey") testing approaches [22], exemplified by Android Monkey [37], generate random streams of UI events without requiring prior knowledge of the SUT. While highly scalable and effective for stress exploration and crash detection [38–40], their exploration strategy is semantically disconnected from the functional features [12, 14], exercising GUI event handlers in isolation rather than triggering coherent, feature-level workflows [41]. To guide this exploration, search-based testing approaches, such as AGRippin [37], EvoDroid [42], and Sapienz [43], formulate event generation as an optimization problem [37, 42–45], using metaheuristic algorithms (e.g., genetic algorithms) to optimize structural objectives such as code coverage or state exploration [44]. Although search-based techniques achieve superior coverage and efficiency compared to random testing [44], their reliance on structural heuristics still struggles to target the semantic combinations of GUI event handlers that correspond to real application features [42, 43].

Complementary to heuristic exploration, symbolic testing approaches (e.g., ACTEve [46], SymDroid [47], and ConDroid [49]) leverage symbolic inputs and constraint solving to explore program execution paths systematically [46–48]. While powerful for revealing subtle logic defects, symbolic execution prioritizes path feasibility over feature-level GUI workflows, and its practical scalability is severely impeded by the path explosion induced by asynchronous callbacks, complex lifecycle transitions, and dynamic UI interactions.

Crucially, across all exploration-driven paradigms—whether random, search-based, or symbolic—approaches predominantly rely on implicit test oracles (e.g., crashes or unhandled exceptions) [41–49]. They lack automated mechanisms to define and validate app-specific functional oracles, leaving non-crashing behavioral inconsistencies largely undetected.

2.3. Model-Driven Approaches

Model-Driven approaches, commonly referred to as Model-Based GUI Testing (MBGT), seek to address the limitations of exploration-driven approaches by leveraging behavioral models of the SUT. These models are typically constructed using various analysis techniques, including dynamic [50, 51], static [52], hybrid [53], or learning-based [54] techniques. By formalizing app behavior through structures such as Finite State Machines (FSMs) [50], Extended FSMs (EFSMs) [55], or Window Transition Graphs (WTGs) [52], MBGT approaches facilitate systematic test generation by traversing paths across the model to satisfy predefined coverage criteria. Furthermore, these formal abstractions enable the definition of state-aware test oracles, which can verify functional correctness against expected state transitions and GUI properties.

Despite these advantages, the effectiveness of MBGT approaches is fundamentally constrained by the fidelity and completeness of the underlying models. In practice, the state-explosion problem—exacerbated by the highly dynamic and event-driven nature of mobile platforms— frequently results in models that are either too coarse-grained to capture intricate business logic or too dense to traverse efficiently. Such fidelity bottlenecks lead to several critical challenges: (i) the generation of redundant or infeasible test cases [53, 56], (ii) degraded precision of test oracles, which frequently miss subtle functional defects and non-crashing bugs [6, 8, 15], and (iii) inadequate coverage of complex user workflows involving multi-step, long-tail event sequences [13, 14, 53, 54]. Moreover, as the SUT evolves, the substantial overhead associated with model updating and maintenance often diminishes the practical return on automation [53, 56].

2.4. Reuse-Driven Approaches

Reuse-driven approaches seek to mitigate manual test development by exploiting human testing knowledge and reusing artifacts across apps with similar functionality, primarily including migration- and pattern-based approaches. Migration-based approaches [57–61], such as AppTestMigrator [60], CraftDroid [59], TestMig [58], and TRASM [62], reuse existing test assets (e.g., recorded traces, input data, and test scripts) from a source app and transfer them to cover similar functionalities on the SUT by employing semantic matching [58] or event-mapping techniques [62]. In parallel, pattern-based approaches formalize testing expertise into reusable templates—termed UI Test Patterns (UITPs)—tailored to recurring GUI structures such as login screens or navigation menus [63–66]. Frameworks like PBGT [63] and iMPAcT [64] use this idea to detect predefined UI patterns on the SUT and execute the corresponding templates to verify expected behaviors.

Despite their potential to reduce effort, both paradigms suffer from critical limitations when applied in practice. Migration-based techniques are inherently constrained by the availability and quality of source tests [19], and cross-app variations in layout, widget hierarchies, and navigation flows frequently yield fragility or partially executable scripts [62]. Similarly, pattern-based testing requires substantial upfront cost to formalize sufficiently generic templates, and it is inherently restricted to standardized UI designs. Furthermore, migration- and pattern-based approaches are essentially complementary strategies; they can only replicate recurring or shared functionality and fail to exercise the app-specific functional logic and custom event-handler workflows essential to the target SUT. By prioritizing the transfer of interaction sequences over app-specific behavioral validation, these approaches lack the necessary oracles to detect non-crashing functional defects in custom features [58, 62].

2.5. AI-Driven Approaches

Motivated by recent advances in Artificial Intelligence (AI), AI-driven approaches leverage various methods such as Reinforcement Learning (RL) [67, 68], deep learning [69–71], and Large Language Models (LLMs) [72, 73] to automate GUI testing for mobile apps. Broadly, these approaches aim to generate test cases more systematically by leveraging prior knowledge

(e.g., test scripts, user interaction logs, bug reports, and requirements) and learning actionable patterns for exploration, input generation, and validation [74]. Like pattern-based frameworks and test migration, they seek to reduce manual effort by reusing and generalizing knowledge extracted from past testing activities.

Overall, AI-driven approaches can enable scalable, adaptive, and semantically informed test generation. Nevertheless, their practical adoption remains nascent. Key challenges include reproducibility (e.g., sensitivity to training data, environments, and model versions), explainability (understanding why particular actions, paths, or oracles are generated), and integration with established testing workflows and toolchains (e.g., CI/CD, regression suites, and maintenance under GUI evolution).

2.6. Synthesis and Research Gap

A critical review of the literature reveals a clear trajectory in mobile GUI testing, transitioning from manual scripting toward increasingly automated, model-based, and intelligent exploration strategies. Despite these contributions, existing paradigms focus primarily on surface-level behavior, optimization heuristics, or reuse mechanisms rather than considering the true semantic structure of the SUT. Table 1 systematically summarizes and compares these paradigms against our proposed approach.

The comparative analysis in Table 1 reveals a recurring systemic limitation across the mobile GUI testing landscape. While varying in their specific exploration heuristics, these paradigms share a fundamental reliance on black-box or coarse-grained models, such as Event Flow Graphs (EFGs) [51] and WTGs [50–52]. By treating the app's interface as a disconnected structure rather than a reflection of its underlying logic, existing techniques inherently lack the semantic awareness required to generate test cases that target semantic features or to derive precise test oracles for revealing non-crashing functional defects [13, 17, 31, 75].

Table 1. Comparison of mobile GUI testing paradigms.

Paradigm	Representative Tools	Core Idea	Oracle Mechanism	Key limitations
Interaction-Driven	Appium [28], Barista [18]	Manual scripting & capture–replay	Manual assertions	Fragile to UI updates; high maintenance; lacks automated exploration.
Exploration-Driven	Monkey [37], Sapienz [43]	Random or search-based input generation	Implicit oracle (e.g., crashes)	Ignores feature semantics; blind to non-crashing functional bugs.
Model-Driven	Stoat [23], AMOGA [53]	State exploration using screen-level graphs (EFGs/WTGs)	State reachability or structural GUI checks	Coarse abstractions omit handler dependencies; susceptible to infeasible paths.
Reuse-Driven	CraftDroid [59], PBGT [63]	Semantic UI mapping & test transfer	Replayed source checks	Limited to generic UI flows; fails on custom app-specific logic.
AI-Driven	GPTDroid [72]	Learning/LLM-guided event generation	Implicit oracle (e.g., crashes)	Prompt heuristics Non-deterministic; high latency; prone to hallucinations.
Proposed Approach		Code-informed modeling of event handlers (EM & ERM)	GUI-level invariants as app-specific oracles	Requires app source code for model extraction.

This highlights a clear research gap: current approaches do not provide an integrated GUI testing framework that: (i) systematically targets feature-level behaviors encoded within GUI event handlers [13] and their meaningful interaction workflows [17, 75], and (ii) enriches generated test suite with app-specific test oracles enable revealing non-crashing functional defects [31].

To address this, we propose a feature-oriented, oracle-aware extension of MBGT techniques. By introducing handler-centric and code-informed models (i.e., EM and ERM), we explicitly encode hierarchical dependencies and semantic relationships among GUI event handlers. This synergy between GUI interaction and implementation logic allows for the systematic generation of feature-aware test cases enriched with GUI-level invariants as app-specific oracles, effectively mitigating the limitations inherent in existing GUI testing approaches.

3. The Proposed Approach

This section presents our semi-automated, feature-oriented model-based GUI testing approach for mobile apps. The approach is designed to generate targeted GUI test cases that exercise both individual GUI event handlers and their semantically meaningful combinations, while also embedding GUI-level invariants as app-specific test oracles to validate runtime behaviors. To illustrate our approach, we use the ContactManager [76] app as a running example throughout this section.

In this work, a feature is defined as a user-visible functional capability whose observable behavior is triggered through a GUI event handler. Based on this view, feature-level GUI testing can be achieved by systematically testing: (i) individual GUI event handlers that implement atomic functional actions, and (ii) meaningful GUI event-handler sequences that collectively realize higher-level workflows (e.g., create–edit–delete).

The approach takes as input the source code of the SUT and a set of human-defined functional scenarios based on the extracted features, and produces executable GUI test cases enriched with GUI-level invariants as app-specific assertions to target individual features and meaningful workflows of them. To achieve this goal, the approach combines static analysis, guided GUI exploration, and human knowledge through three main modules: (1) ***GUI Exploration***, which derives GUI-level invariants from runtime executions; (2) ***Model Constructor***, which builds two complementary models, namely EM and ERM; and (3) ***Test Generator***, which traverses these models to produce targeted GUI test cases and embeds them with extracted GUI-level invariants as app-specific test oracles. Figure 1 illustrates the overall architecture of our approach.

The proposed approach builds upon our earlier work on extracting GUI-level invariants [31]. While that work focused on the extraction of GUI-level invariants in isolation, the present study integrates those invariants into the proposed model-driven test generation approach. As a result, the generated test cases are not only structurally guided and feature-oriented, but also oracle-aware, enabling the detection of both crashing and non-crashing functional defects—a key limitation of most existing GUI testing techniques [6, 15].

3.1. GUI Exploration

The GUI Exploration module is responsible for deriving runtime behavioral knowledge from the GUI cues of the SUT and transforming it into GUI-level invariants that later serve as app-specific test oracles in the generated GUI test cases.

This module builds on our earlier work on GUI-level invariant extraction [31], but here it is integrated as part of a unified test generation architecture. Its input is the source code of the SUT, and its output is a set of inferred

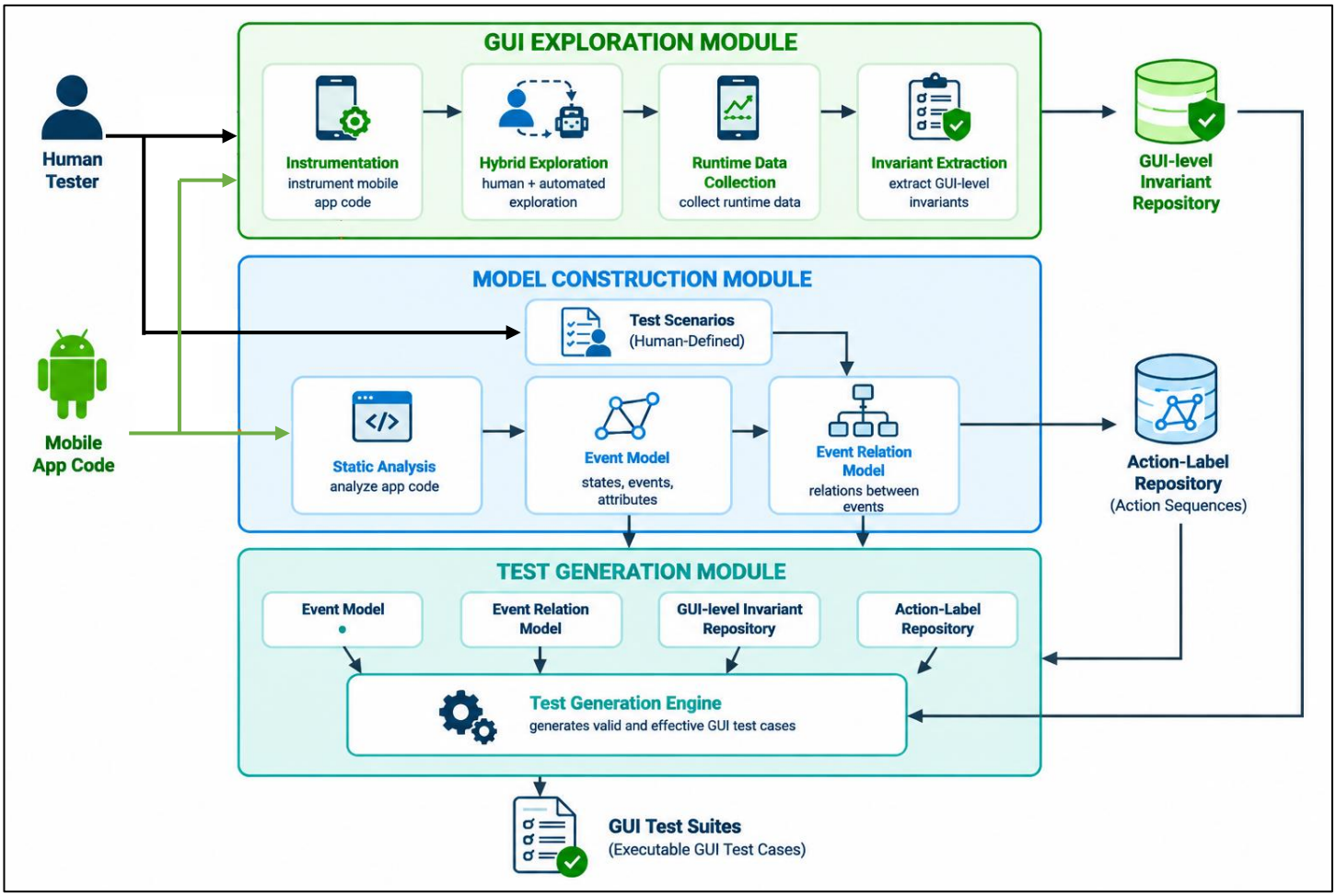


Figure 1: Overview of the proposed feature-based GUI test generation architecture

GUI-level invariants associated with the critical execution boundaries—specifically the entry and exit points of GUI event handlers and lifecycle initialization methods such as onCreate. The extraction process is structured as follows:

Instrumentation: The module first employs static analysis to parse the SUT source code, extracting GUI related classes, their associated layout widgets, and GUI event handlers. It then injects lightweight monitoring hooks at critical execution boundaries. These hooks are designed to capture the runtime values of functional GUI widget attributes, such as visibility, enabled states, and collection sizes. Furthermore, the module generates a schema file—containing structural descriptors for execution boundaries—which serves as a blueprint for interpreting the logged data at critical execution boundaries. By capturing raw execution flows according to these descriptors, the instrumentation provides the necessary foundation for subsequent GUI-level invariant inference. Figure 2 shows an example of an instrumented GUI event handler within the ContactManager app.

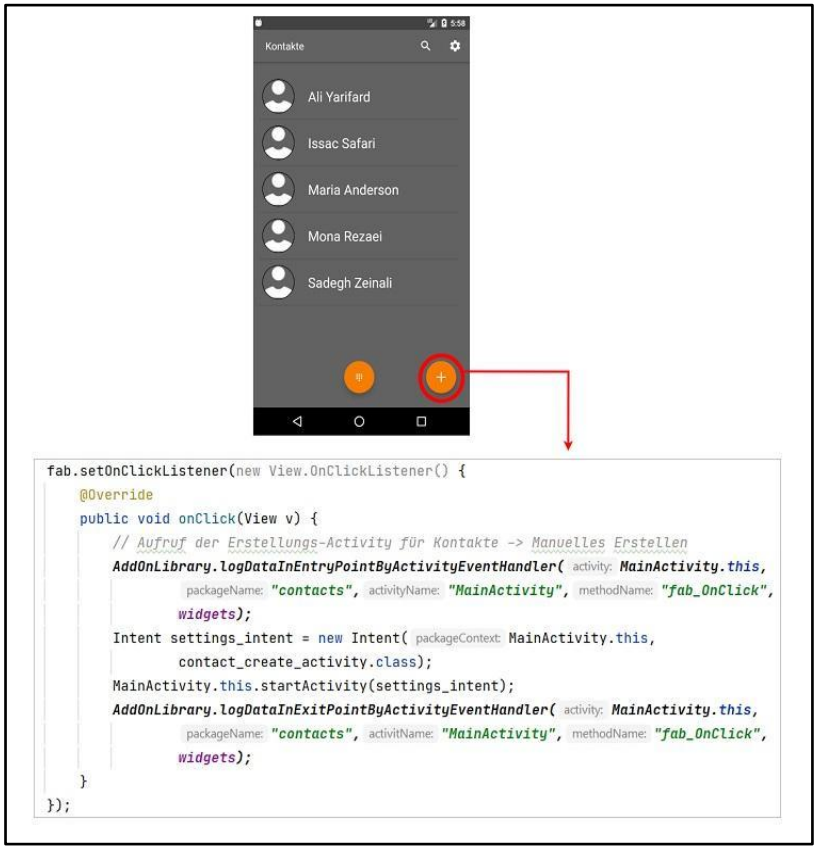


Figure 2: An instrumented GUI event handler within ContactManager app [76]

Dynamic Observation: The instrumented SUT is explored via a hybrid strategy: a human-guided phase to reach deep, semantically important GUI regions, followed by an automated random-event generation phase to expand the explored behavioral space and increase the diversity of observed execution traces [31]. During exploration, the injected monitoring hooks capture the runtime values of functional widget attributes into an execution trace file, strictly adhering to the structural descriptors defined in the schema file.

Invariant Extraction: The resulting execution trace file is interpreted using the schema file and processed by the Daikon tool [77] to infer candidate GUI-level invariants. Table 2 presents a selection of representative GUI-level invariants generated by this module for the ContactManager app.

Within the context of the present study, the main role of the GUI Exploration module is to supply the later test generation module with behaviorally meaningful and explainable oracle knowledge. In contrast to existing mobile GUI testing techniques that mainly rely on crash detection [71] or generic heuristic [6, 15], the extracted GUI-level invariants enable runtime validation of GUI-level functional correctness and improve the ability to detect non-crashing functional defects. Further implementation details regarding instrumentation and invariant inference are available in our earlier work [31].

Table 2: Representative GUI-level invariants extracted for ContactManager app [76]

Invariant Type	Point type	Label	GUI invariant	Description
Object Level	- --	MainPage	searchButton.enabled == settingButton.enabled	This GUI invariant specifies that the searchButton and settingButton must always be enabled whenever MainPage is displayed.
Object Level	- --	DeleteContact Dialog	dialogTitle == "Delete this contact?"	This invariant indicates that the dialog's title property equals "Delete an Item" upon successful instantiation and display.
GUI Event Handler Level	Entry Point	OpenContact ViewPage	ContactList.size >= 1	This GUI invariant indicates that the OpenContactViewPage is displayed only if there is at least one entry in the contactList.
GUI Event handler Level	Exit Point	DeleteContact _Confirm	ContactList.size == old(ContactList.Size) – 1	This invariant means that upon the successful deletion of a contact, the ContactList size must decrease by one.

3.2. Model Constructor

The Model Constructor module is responsible for deriving the abstract behavioral models that guide systematic and feature-oriented test generation. The design of this module is motivated by the observation that existing model-based GUI testing techniques predominantly rely on navigation-oriented abstractions, such as FSMs or WTGs. While effective for systematic exploration, such models are limited in their ability in representing the SUT from the perspective of functional feature-level and semantic relations between them.

To address this limitation, the module constructs two complementary models: (i) EM and (ii) ERM. Together, these models capture both the individual behavior of GUI event handlers and their meaningful combinations which are required in complex functional test scenarios. The idea behind the model extraction process is driven by the observation that mobile apps are inherently event-driven systems, in which functional features are primarily implemented within GUI event handlers. Accordingly, the identification of the functional features of the SUT is reduced to identifying its GUI event handlers and the execution paths that lead to them.

Therefore, modeling the SUT from the perspective of both individual GUI event handlers and their meaningful combinations constitutes the suitable foundation for meaningful GUI test generation. In this module, we address this challenge through a combination of static analysis of the SUT source code using JavaParser[2] library and guided human input.

3.2.1. GUI Event Handler Extraction

The extraction of GUI event handlers is the foundational step of the model construction process, as both the EM and ERM are built upon the identified GUI event handlers. This step is performed automatically using static analysis techniques.

Given a context class— which represents an activity[3] class in Android apps— the module parses its source code using the JavaParser library to create its Abstract Syntax Tree (AST). The AST is then systematically traversed to identify method declarations corresponding to GUI event handlers— such as callbacks associated with user interactions (e.g., button clicks). This identification relies on structural cues in the AST, including method signatures, implemented listener interfaces, and associations with GUI widgets. Each extracted event handler is considered as an atomic functional feature representing a concrete entry point for triggering application logic. For each identified GUI event handler, the module tries to generate a meaningful name from its context information (e.g. description messages) using Apache Natural Language Processing (NLP) library[4], and collects GUI widgets used in its implementation, and encapsulates them as metadata which is used in future steps.

The resulting set of event handlers, along with their metadata form the shared input for constructing both the EM and ERM models. This separation of concerns ensures traceability from source code to model elements while avoiding redundancy in subsequent modeling steps.

3.2.2. Hierarchical Event-Handler Model (HEHM)

The Hierarchical Event-Handler Model (HEHM), referred to as the Event Model (EM) for brevity, provides a fine-grained structural representation of the SUT by modeling GUI behavior at the level of individual GUI event handlers and their execution contexts. Unlike traditional navigation-oriented models that primarily represent GUI windows and surface-level event sequences, the EM captures the internal functional structure of GUI contexts by explicitly modeling the GUI event handlers through which the SUT features are implemented, together with the hierarchical dependencies that govern their execution. This modeling perspective enables the generation of tests that are feature-aware rather than merely navigation-aware.

Formally, the EM is defined as a 6-tuple $G = \langle S, s_0, H, T, L, \delta \rangle$, where its components are defined as follows:

- $S = \{s_0, s_1, \dots, s_n\}$ is a finite set of GUI contexts which are identified through static analysis of the SUT source code. Each $s \in S$ represents a primary GUI context class, typically corresponding to an Android Activity class, which acts as a navigation container in which a set of GUI event handlers can be triggered during execution.
  Modern mobile interfaces may contain additional UI components such as fragments, dialogs, menus, and pop-ups. In the abstraction adopted by the EM, these elements are not modeled as independent GUI contexts because they are instantiated and executed within the lifecycle of an enclosing context class rather than introducing a separate navigation scope at the level targeted by our static analysis. Instead, their behavior is captured through the hierarchical relations among event handlers defined within the corresponding context. In particular, event handlers implemented in fragments are associated with their hosting context, while dialogs and menu interactions are represented through hierarchical dependencies between handlers.
  Each context $s \in S$ is therefore characterized by a finite set $N_s \subseteq H \times (H \cup \{\emptyset\}) \times L$, where each tuple $\langle h_i, p_i, l_i \rangle \in N_s$ represents an occurrence of a GUI event handler within context s and is defined as follows:
  - $h_i \in H$ denotes a GUI event handler associated with context s. Each handler corresponds to a concrete functional unit implemented in the SUT source code, typically defined as a callback responding to user interactions such as button clicks, long clicks, or menu selections.
  - $p_i \in H \cup \{\emptyset\}$ denotes the parent handler of h_i. If $p_i \neq \emptyset$, the execution of h_i becomes conditionally reachable only after the execution of p_i. This hierarchical dependency captures nested interaction structures commonly observed in GUI interfaces, such as menu–submenu relations, dialog confirmation actions, or context-specific interaction flows that become available only after a preceding user action.
  - $l_i \in L$ is the interaction label associated with handler h_i, serving as a unique key to retrieve the concrete UI interaction sequence required to trigger the handler from the Actions-Label repository.
- $s_0 \in S$ denotes the initial entry context of the SUT. It corresponds to the SUT's launch context (e.g., the *Main Activity*) extracted from the project metadata during static analysis.
- H represents the finite set of all GUI event handlers extracted from the SUT source code: $H = \bigcup_{s \in S} \{h | \exists p, l \; such \; that \; \langle h, p, l \rangle \in N_s\}$. Each handler $h \in H$ represents a unique functional execution unit in the SUT implementation (e.g., onClick, onLongClick, or onItemSelected). While a handler has a unique identity in H,

[2] This library provides various useful visitor APIs to travel on the AST nodes and extract information stored on them and available at https://javaparser.org/

[3] An activity in Android apps is equivalent to a window in desktop apps, in which the app draws its UI.

[4] https://opennlp.apache.org/

its contextual occurrence and hierarchical relationships are captured by the corresponding tuple $\langle h, p, l \rangle$ within the set N_s of its associated context. This distinction allows the model to separate handler identity from its structural embedding within a particular GUI context.

- $T \subseteq S \times H \times S$ is the navigation transition relation. A transition $(s, h, s') \in T$ exists if the execution of event handler h within context s triggers a navigation-related operation that transfers the SUT to another context s'. Such transitions are typically detected through static analysis of navigation-related API calls (e.g., startActivity). This relation captures the inter-context navigation structure of the SUT, where the hierarchical relations represented in N_s, describe intra-context interaction dependencies.
- L is the finite set of labels associated with GUI event handlers. Each label uniquely identifies an entry in the Action-Label repository, which stores the concrete UI interaction sequences required to invoke the corresponding handler.
- $\delta: L \rightarrow (interactions)^+$ is the interaction mapping function. For each label $l \in L$, $\delta(l)$ returns a non-empty sequence of concrete UI interactions (e.g., typeText, longClick) required to trigger the corresponding event handler. This mapping bridges the gap between abstract model elements and executable GUI actions during the test generation phase.

Unlike traditional black-box or navigation-oriented models such as EFGs [51] and WTGs [52]—which primarily capture surface-level event sequences and window-level reachability—the EM is designed as a handler-centric, code-informed abstraction. By leveraging static analysis to represent GUI contexts as structured collections of internal event handlers and their hierarchical dependencies, the EM bridges the gap between high-level navigation and the underlying functional logic implemented in the SUT source code. This structural precision helps mitigate the semantic limitations of purely exploration-based testing [51] and the abstraction mismatch of navigation-oriented models [52], providing a formal foundation for generating feature-aware test cases capable of detecting non-crashing functional defects.

While Figure 3 represents a simplified version of the extracted EM Model for ContactManager app, Algorithm 1 details its automated construction process. To enhance readability, we have abstracted the detailed handler names in Figure 3 into unique identifiers (e.g., MH_01). The complete mapping of these identifiers is available in the supplementary material [30].

As outlined in Algorithm 1, the model construction process begins by collecting the SUT project metadata (Line 4) and extracting its main context class (Line 5). The corresponding initial state is then created (Line 6), and both the EM model and the processing queue are initiated with this state (Line 7-8). While the processing queue is not empty (Line 8), the algorithm iteratively selects an unvisited state, marks it as visited (Line 12), and applies static analysis using JavaParser library to parse the source code of the corresponding context class to extract its GUI event handlers (Line 14).

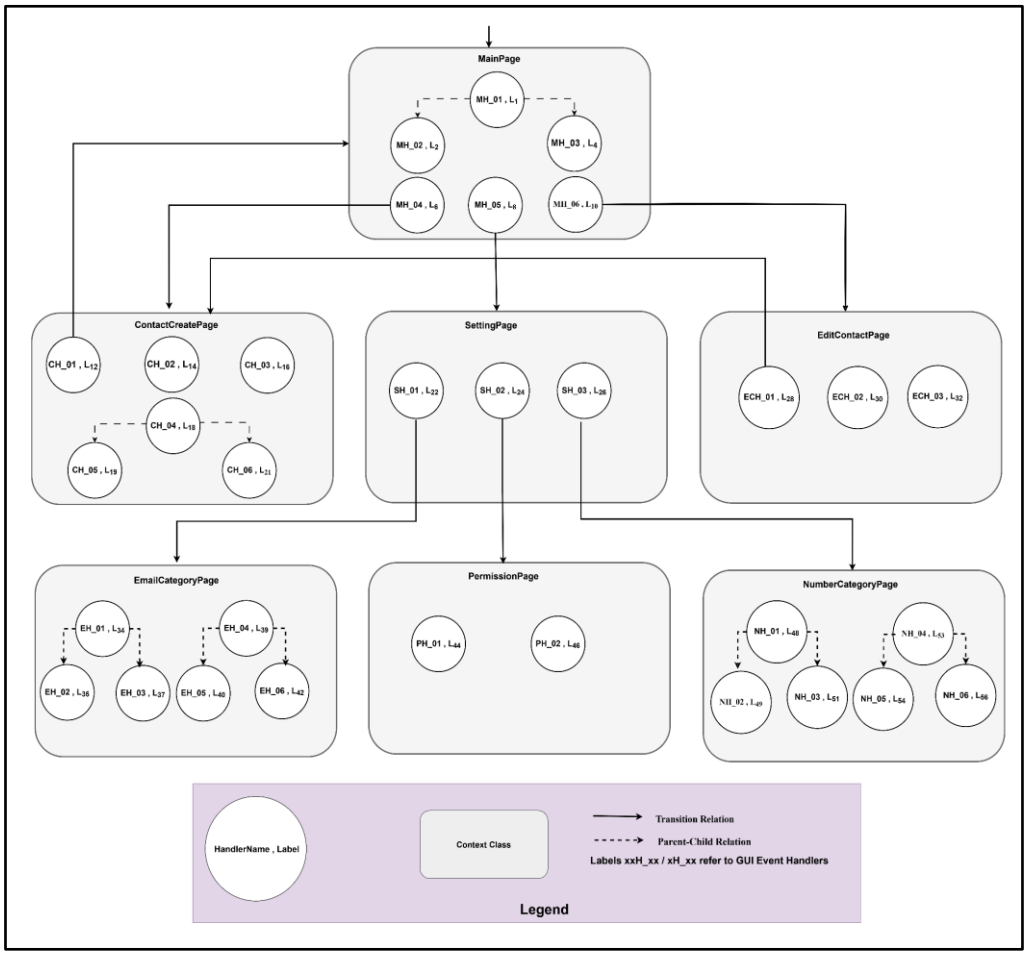


Figure 3: The simplified version of Event Model (EM) for the ContactManager app. The complete mapping of GUI event handler identifiers is available in the supplementary material [30]

For each detected GUI event handler, the algorithm generates its UI actions required to trigger its invocation (Line 16) using the metadata collected during the event-handler extraction phase. The generated interaction sequence is stored in the Action-Label repository (Line 17), and its corresponding label together with handler metadata is attached to the corresponding state within the EM (Lines 22 or 25).

Moreover, the algorithm statically analyzes the handler's source code to identify navigation-related operations and determine its target GUI context, if any (Line 19). When a new target context is detected and is not yet present in the EM, the algorithm creates the corresponding state (Line 21), adds a transition from the current context to the target context (Line 22), and enqueues the newly discovered context for subsequent analysis (Line 23). The construction process terminates once all reachable contexts have been visited and the processing queue becomes empty.

Algorithm 1: Event Model Construction process

```
Algorithm 1: Event Model Construction
   Input: MobileAppProject
   Output: EventModel
 1 EventModel ← Ø
 2 LabelRepository ← Ø
 3 VisitedState ← Ø
 4 projectMetaData ← collectMetaData(MobileAppProject)
 5 mainContext ← getMainContext(projectMetaData)
 6 currentState ← createState(mainContext)
 7 EventModel ← currentState
 8 Queue.enqueue (currentState)
 9 while (Queue != Ø) do
10   currentState ← Queue.dequeu()
11   if (!VisitedState.contains(currentState)) then
12     VisitedState.add(currentState)
13     currentState.EventList ← Ø
14     eventList ← extractGUIEventHandlers(currentState)
15     foreach event ∈ eventList do
16       actions ← generateUIActions(currentState, event)
17       key ← LabelRepository.append(actions)
18       if (trigerOtherContext(event)) then
19         targetContext ← getTargetContext(event)
20         if (!EventModel.contain(targetContext)) then
21           newState ← createState(targetContext)
22           currentState.EventList.add(event, key, newState)
23           Queue.enqueue(newState);
24         else
25           currentState.EventList.add(event, key, currentState)
```

3.2.3. Event Relation Model (ERM)

While the EM provides a fine-grained structural representation of the SUT by modeling individual GUI event handlers and their execution contexts, many non-crashing functional defects reveal only when multiple GUI event handlers execute in a meaningful order [78]. Therefore, testing individual handlers in isolation may fail to reveal defects that arise from violations of feature-level workflows (e.g., create–edit–delete operations on application data) [10, 11].

Although static analysis enables the systematic extraction of handler-level structure and navigation transitions (Section 3.2.1 and Section 3.2.2), automatically inferring higher-level semantic relations among handlers remains challenging. Source code typically exposes structural dependencies and navigation instructions, but it rarely encodes domain knowledge regarding how the SUT features are intended to be used in realistic user workflows. Therefore, purely automated approaches struggle to reconstruct meaningful feature-level workflows from code or execution traces alone [79].

To address this limitation, we introduce the Event Relation Model (ERM), a high-level behavioral abstraction constructed on top of the EM. In contrast to the EM—which is extracted automatically through static analysis— the ERM is constructed using human-defined functional test scenarios. These scenarios encode domain knowledge about realistic user workflows and therefore provide valuable guidance for constructing feature-oriented test cases [72, 78].

Formally, given the EM model defined as $G = \langle S, s_0, H, T, L, \delta \rangle$, the ERM is defined as an annotated directed tree $G' = \langle S', s_0', T', L', \delta', \mu' \rangle$ where:

- $S' = \{S_0',\ S_1', \ldots, S_n'\}$ is a finite set of nodes representing functional milestones derived from human-defined test scenarios. Each node $s' \in S' \setminus \{s_0'\}$ corresponds to a specific GUI event handler extracted in the EM. This association is captured through a mapping function μ'.
- $s_0' \in S'$ denotes the root node of G' and corresponds to the entry context of the SUT (i.e., the initial context S_0 in the EM). The root represents the starting point from which feature-level interaction sequences originate.

- $\mu' : S' \rightarrow H$ is a mapping function that associates each ERM node with the corresponding GUI event handler $h \in H$ defined in the EM. This mapping preserves traceability between the semantic representation in the ERM and the handler-level structure captured in the EM.
- $T' \subseteq S' \times L' \times S'$ is the set of semantic transitions. Each transition $t' = \langle src, l'_e, des \rangle$ represents a meaningful ordering relations with annotated label l'_e between two functional milestones (e.g., event handlers) in a human-defined test scenario. In other words, it specifies that the event handler associated with node src is followed by the handler associated with node des in a meaningful feature-level workflow.
- L' is the set of transition labels associated with the ERM edges. Each $l'_e \in L'$ identifies the navigation sequence required to move between the corresponding handlers.
- $\delta' : L' \rightarrow (interactions)^+$ is the interaction mapping function. For each label $l' \in L', \delta(l')$ returns a non-empty concrete GUI interactions required to navigate from $\mu'(src)$ to $\mu'(des)$. These interaction sequences are derived from the navigation information encoded in the EM, thereby ensuring that each semantic relation represented in the ERM corresponds to an executable GUI path in the SUT.

The ERM construction process begins by processing human-defined test scenarios expressed as ordered sequences of the extracted GUI event handlers. The model is built incrementally in order to reuse common prefixes among scenarios. When multiple scenarios share the same initial interaction steps, these prefixes are represented by shared paths in the ERM, which improves model compactness and avoids redundant representations of identical interaction sequences. Moreover, each consecutive pair of handlers in a scenario induces a semantic transition in the ERM. To ensure executability, the EM is queried to derive the concrete navigation path between the corresponding handlers. The resulting sequence of UI actions is then stored as the label of the ERM edge connecting the corresponding nodes. In this way, the ERM maintains semantic information derived from human knowledge while remaining consistent with the structural constraints of the underlying execution model.

As defined above, the ERM forms a directed tree whose root corresponds to the entry point of the SUT, while the remaining nodes represent GUI event handlers participating in feature-level workflows. The edges capture semantic ordering relations between handlers and are annotated with navigation labels that encode the concrete interaction sequences required to traverse between them in the SUT. Figure 4 presents the ERM model constructed for the ContactManager app, while Algorithm 2 describes the automated procedure used for its construction.

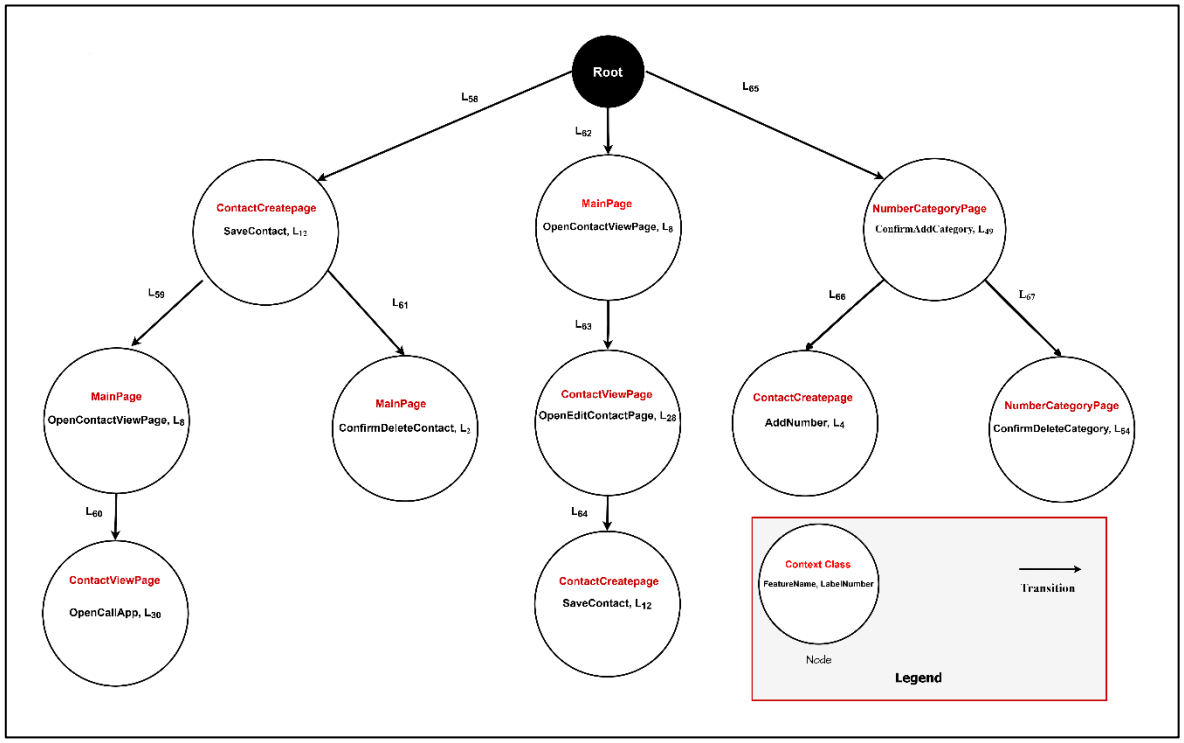


Figure 4: The Event Relation Model (ERM) for the ContactManager app.

Algorithm 2: Event Relation Model (ERM) Construction process

Algorithm 2: Event Relation Model Construction

Input: MobileAppProject, EventModel, LabelRepository, testScenarios
Output: EventRelationModel

```
1  EventRelationModel ← Ø
2  projectMetaData ← collectMetaData(MobileAppProject)
3  mainContext ← getMainContext(projectMetaData)
4  EventRelationModel ← createInitialState(mainContext)
5  foreach (testScenario ∈ testScenarios) do
6      eventSet ← splitTestScenarioBasedOnEvent(testScenario)
7      currentState ← EventRelationModel.getRootState()
8      foreach (event ∈ eventSet) do
9          if (!currentState.hasAdjacency(event)) then
10             nextState ← createState(event, LabelRepository)
11             actions ← generateActionsFor-
                 Path(EventModel,LabelRepository,currentState,nextState)
12             key ← LabelRepository.append(actions)
13             currentState.appendAdjacency(nextState,key)
14             currentState ← nextState
15         else
16             currentState ← currentState.getAdjacency(event)
```

As shown in Algorithm 2, the ERM construction begins by collecting the SUT's project metadata and extracts its main context, which is used to create the root state of the model (Lines 2–4). The algorithm then iterates over the human-defined test scenarios (Line 5), each of which is decomposed into an ordered sequence of GUI event handlers (Line 6), and processed starting from the root state (Line 7).

For each event handler in the sequence (Line 8), the algorithm checks whether a corresponding outgoing relation is already encoded in the current state (Line 9). If not, a new state is created and the EM is queried to derive an executable GUI navigation path between the previous state and the new state (Line 11). The resulting UI actions are stored in the Action-Label repository and its returned label is used to annotate the corresponding edge (Lines 12–13), after which the traversal proceeds to the newly created state (Line 14). Otherwise, the existing relation is reused and the traversal continues accordingly (Line 16).

By structuring human-defined test scenarios into a formal model, the ERM preserves domain knowledge in a reusable and analyzable representation. The separation between EM and ERM enables a complementary modeling strategy: the EM ensures systematic and automated coverage of individual GUI event handlers, whereas the ERM captures realistic feature-level workflows that involve meaningful combinations of handlers. Together, these models provide the foundation for generating effective GUI test cases.

3.3. Test Generator

The Test Generator module utilizes the extracted models, GUI invariants derived from the SUT's runtime behavior, and Action-Label repository to generate GUI test cases enriched with app-specific oracles. The generated tests— expressed in the Espresso framework syntax— are designed to hit defect-sensitive execution paths— namely individual GUI event handlers and their meaningful compositions —within the SUT. To do this, the module generates two types of test cases: simple test cases— which hit individual GUI event handlers— and complex test cases —which cover meaningful combinations of GUI event handlers.

To generate simple test cases, the module systematically traverses the EM model from its initial state. Each visited state is mapped to a corresponding test class, while each internal node within that state is transformed into individual test cases. Each generated test case is conceptualized as a macro event defined as $\phi =\prec Pre, E, Post \succ$, where E denotes UI actions required to test the corresponding GUI event handler, and Pre and Post define the assertions checked before and after executing E, respectively. The UI actions are retrieved from the Action-Label repository using the label stored in the corresponding GUI event handler node, as generated during the construction of the EM model.

Furthermore, the module retrieves the prioritized GUI invariants associated with the event handler's entry and exit points using its metadata stored in the corresponding internal node and presents them to the human tester. Leveraging domain expertise [59, 60, 64, 80, 81] alongside the generated event sequence, the tester selects the most suitable GUI invariants to instantiate the pre- and post-conditions. This guided human involvement ensures that the resulting assertions remain semantically aligned with the intended functional objective of each test case.

The complex test cases are generated using the ERM model— which encodes complex test scenarios as root-to-leaf paths in an annotated tree. In this process, the module first creates a dedicated test class for complex test scenarios, and then systematically transforms each root-to-leaf path on the ERM model into individual complex test cases. As described before, each such path consists of nodes representing GUI event handlers involved in the corresponding test scenario and annotated edges encoding the navigation required to transition between consecutive nodes in the EM. Accordingly, a complex test case is synthesized by composing the macro events associated with the involved GUI event handlers and the navigation macro-event sequences encoded along the ERM path, following their defined order. This composition enables the execution of semantically related functionalities within a single test case, facilitating the detection of non-crashing functional defects arising from interactions and side effects among multiple GUI event handlers.

The final output of this module is a suite of targeted GUI test cases that collectively hit both isolated GUI event handlers and their semantic compositions as multi-step functional scenarios. Each test case is augmented with app-specific GUI oracles derived from inferred GUI invariants, enabling reliable detection of subtle non-crashing defects.

3.4. Tool-Support

To operationalize the proposed approach, its workflow was implemented as two plugins for Android Studio which are publically available open-source at [30]**.** The first plugin, called *Instrumenter*, performs static analysis of the SUT to identify GUI contexts and event handlers, injects monitoring hooks at their entry and exit points, and generates the necessary runtime data for extracting GUI invariants. The second plugin, called *GUITestGenerator*, constructs the EM and ERM from the SUT's source code and automatically produces executable test cases in Espresso format, enriched with the derived GUI invariants as app-specific test oracles. Integrating the developed plugins within the IDE enables systematic instrumentation, model construction, and test generation directly over the project structure of Android apps.

4. Empirical Evaluation

This section presents the empirical evaluation process conducted to assess the quality of the test suite generated by the proposed approach. This process begins with the presentation of the research objective, followed by the formulation of research questions (RQs) that guide the design of the empirical studies. Next, we define evaluation metrics used to measure outcomes and objectively address the RQs. Finally, we describe the evaluation process, including empirical study setup, the characteristics and preparation of subjects, the description of empirical studies, data collection procedure, and analysis of the results, providing evidence for the quality of the generated test cases and the practicality of the proposed approach.

4.1. Goals & Research Questions

Following the Goal-Question-Metric (GQM) template [82], the goal of this study is to evaluate the quality of the generated test cases by the proposed approach with respect to a key software quality attribute, namely effectiveness from the tester's perspective, defined in the ISO/IEC25010 [83]. Based on this goal, the following research questions (RQs) have been formulated:

- ***RQ 1****- How does the code coverage achieved by the generated test cases compare with state-of-the-art GUI testing tools?*
  Code coverage is a fundamental indicator of test effectiveness, as test cases that explore a larger portion of the codebase are more likely to reveal hidden defects. This RQ evaluates the extent to which the generated test cases achieve code coverage—measured through statement coverage—relative to the state-of-the-art GUI testing tools.
- ***RQ 2-*** *How effective are the generated test cases in revealing potential defects?*
  Detecting defects, especially non-crashing ones, is a critical goal of GUI testing. This RQ assesses how effectively the generated test cases reveal potential defects in the SUTs. To quantify this, we use mutation score—an established standard indicator for evaluating the fault-revealing power of test suites.
- ***RQ 3****- How effective are the derived GUI-level invariants, when used as app-specific test oracles, in revealing defects during test execution?*
  This question evaluates the defect-revealing capability of GUI-level invariants, which is a critical indicator of their practical usefulness. The assessment is performed using mutation testing to quantify defect detection.
- ***RQ 4****- How does the proposed approach for generating test cases compare with the state-of-the-art GUI testing tools in terms of fault-detection effectiveness?*
  This question evaluates the defect detection effectiveness of test cases generated by the proposed approach, comparing them with state-of-the-art mobile testing tools using mutation testing [10] to quantify their practical defect-revealing capability.

4.2. Metrics

To answer the study's RQs, we employ a set of quantitative and qualitative metrics, commonly used in GUI testing research and aligned with the software quality attribute defined in the ISO/IEC25010 [83]. These metrics collectively evaluate (i) the behavioral exploration capability of the generated GUI test cases, (ii) their fault-detection effectiveness, and (iii) the practical usefulness of the extracted GUI-level invariants as app-specific test oracles. In accordance with the GQM paradigm, Figure 5 presents the GQM tree linking research questions to evaluation metrics.

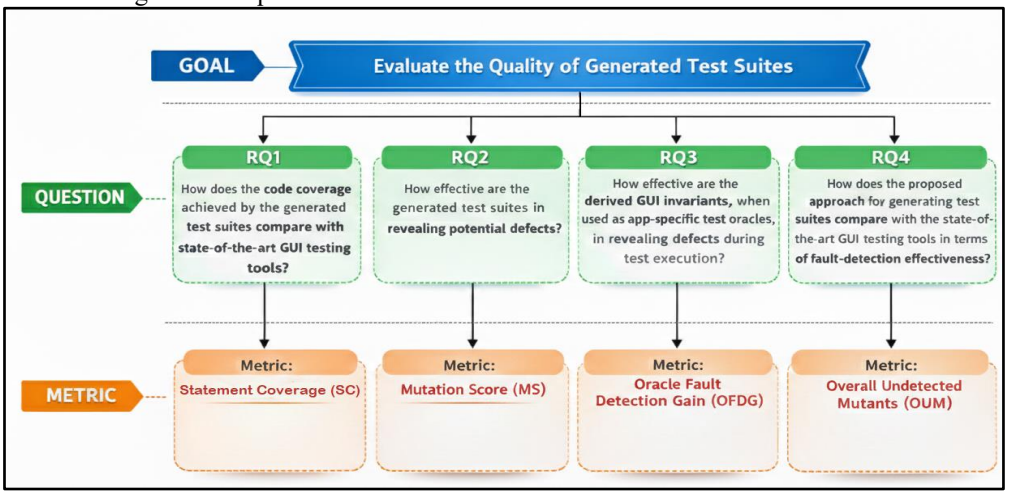


Figure 5: GQM tree showing the relationship of the study's RQ and the selected metrics

Statement Coverage (RQ1): Statement coverage measures the proportion of executable code statements exercised by the generated test cases, which is defined as follows:

$$Statement\ Coverage(\%) = \frac{Executed\ Statements}{Total\ Executable\ Statements} \times 100 \quad \text{(Eq. 1)}$$

Although statement coverage originates from structural testing (i.e., unit-level testing), it remains highly relevant in GUI-level testing of event-driven systems such as mobile apps. In such systems, system-level testing is conducted via GUI interactions, as the GUI constitutes the primary gateway for triggering event-handler methods that implement the underlying functional logic. Consequently, the extent to which generated GUI test cases execute program statements directly reflects the degree of behavioral exploration achieved through user-level interactions. As such, statement coverage serves as an objective proxy for assessing the depth of functional execution reachable via GUI-based test cases. Furthermore, it has been widely adopted in prior GUI testing research [53, 71, 84], facilitating fair and reproducible comparison with state-of-the-art approaches.

Mutation Score (RQ 2): The Mutation Score (MS) evaluates the fault-detection capability of the generated test cases by injecting artificial defects (mutants) into the SUT, which is defined as follow:

$$MS = \frac{Killed\ Mutants}{Total\ Mutants} \quad \text{(Eq.2)}$$

This metric is widely recognized as the most reliable approximation for real fault-detection capability, particularly when real defects are unavailable or limited [85]. Unlike coverage metrics, which measure exploration depth, mutation score directly evaluates the ability of test cases to reveal defects. In GUI testing, structural coverage alone does not guarantee fault detection, as many defects reveal only under specific event sequences or state conditions. Therefore, mutation score provides a stronger effectiveness indicator by assessing whether the generated GUI test cases not only execute code but also detect incorrect behavior triggered through GUI interactions.

Oracle Fault Detection Gain (RQ 3): To measure the gain of using GUI-level invariants as test oracles in fault detection rate, we define Oracle Fault Detection Gain (OFDG) as the improvement in mutation score obtained when invariant-based oracles are incorporated into the generated test cases:

$$OFDG_i = \left[MS_i^{inv} - MS_i^{no-inv}\right] \times 100 \quad \text{(Eq. 3)}$$

Where MS_i^{inv} and MS_i^{no-inv} denote the mutation score for the i^{th} app with and without GUI-level invariants as test oracles, respectively.

Unlike traditional GUI testing approaches that rely primarily on generic failure oracles (e.g., crash detection or uncaught exceptions), OFDG metric explicitly quantifies the additional fault-detection capability attributable to the inclusion of the extracted GUI-level invariants within the generated test cases. By isolating the difference in mutation score with and without invariant-based oracles, this metric captures the incremental effectiveness contributed specifically by semantic validation mechanisms. Therefore, a positive OFDG value therefore indicates that invariant-based oracles enhance the fault-detection strength of the generated test cases beyond structural exploration alone, demonstrating their practical value in improving testing effectiveness.

Overall Undetected Mutants (RQ4): To compare the fault-detection of each testing approach, we define Overall Undetected Mutants (OUM) to measure the proportion of injected defects (mutants) that remain undetected during testing— which is defined as follow:

$$OUM\ (\%) = \left(\frac{\sum_{i=1}^{N} FN_i}{\sum_{i=1}^{N} TM_i}\right) \times 100 \quad \text{(Eq. 4)}$$

Where N denotes the number of subjects (apps), FN_i represents the number of undetected (survived) mutants in the i^{th} app, and TM_i denotes the total number of mutants injected into the i^{th} app.

Unlike per-application mutation score, which evaluates fault-detection effectiveness locally, OUM metric provides an aggregate indicator of residual fault-detection weakness across the entire benchmark set. By summing the undetected mutants over all subjects and normalizing by the total number of injected mutants, this metric enables direct comparison of the overall effectiveness of competing testing approaches. Lower OUM values indicate that fewer injected defects remain unrevealed, reflecting stronger and more consistent fault-detection effectiveness across diverse applications.

4.3. Empirical Procedure and Setup

This section describes the empirical setup and experimental protocols designed to evaluate the proposed approach. The overall evaluation workflow encompasses subject application selection, invariant extraction, model generation (EM and ERM), and the generation of test cases enriched with app-specific oracles (i.e., GUI-level invariants) based on the extracted model. To systematically answer the research questions, two empirical studies were conducted: (1) a code coverage analysis addressing RQ1, and (2) a mutation-based fault-detection study addressing RQ2–RQ4.

4.3.1. Baseline Tools and Selection Rationale.

We compared the proposed approach with four baseline testing tools. These tools have been selected as baseline tools, since they (i) have frequently served as benchmarks in prior studies, (ii) represent different families of test-generation strategies—namely random, model-based, and learning-based respectively—and (iii) reflect current and relevant advancements in automated mobile testing. We selected Monkey [40] as the representative for random testing, Stoat [23] for stochastic model-based exploration, and DinoDroid [71] as a representative of deep-reinforcement-learning-based strategies. These tools primarily rely on implicit oracles and thus serve as baselines for both code coverage (RQ1) and fault detection (RQ4). Additionally, we included REGDROID [6], a differential testing tool, as an explicit-oracle baseline. REGDROID was deployed exclusively in the fault-detection study (RQ4) to benchmark the effectiveness of the proposed approach against non-crashing functional defects.

4.3.2. Configuration and Instrumentation.

The experimental environment consisted of a workstation running Ubuntu 16.04 (Intel Core i7 8th Gen, 16 GB RAM) and

Android Studio 2020 utilizing Nexus 6 emulators. For the coverage analysis, all subject apps were instrumented using JaCoCo [93] to facilitate the collection of statement-level coverage data. For the fault-detection study, we utilized DroidMutator [96] to inject a diverse set of faults, covering both standard Java-level logic and Android-specific constructs. All baseline tools were configured strictly according to their official documentation and the parameters recommended in recent comparative studies [71, 92]. Specifically, Monkey was configured with a standard 200ms event delay to avoid event interference and ensure environment stability, Stoat's budget was divided equally between its exploration and generation phases, and for DinoDroid, we utilized its publicly available, pre-trained model [71].

4.3.3. Execution Protocol and Bias Mitigation.

A critical requirement for empirical software engineering is the mitigation of execution bias arising from the non-deterministic nature of stochastic tools. Following established evaluation protocols [71, 92], each exploration baseline tool (i.e., Monkey, Stoat, and DinoDroid) was allotted a runtime budget of one hour per subject app for the coverage study. For the fault-detection study, an event budget of 5,000 events was enforced across all baseline tools, strictly adhering to the official evaluation setup and operational guidelines of REGDROID (i.e., generating 50 test cases, each containing 100 events) [6]. This budget provides a robust baseline configuration against which the proposed approach's compact test suites (containing at most 752 events per app) were evaluated.

To account for stochastic variance in baseline tools, each baseline tool was executed three independent times per subject app for RQ1 and RQ4, and we report the sample maximum (upper bound) across these executions to evaluate baseline tools under their most favorable conditions. In contrast, although the proposed approach generates a deterministic test suite driven by handler-aware models, we also executed our generated test suite three independent times per app, with cache clearing and app re-installation between executions to rule out environmental flakiness (e.g., UI rendering delays or emulator stalls). All three recorded coverage values for the proposed approach were identical within each app, indicating stable results under the evaluated conditions.

4.3.4. Statistical Analysis

We used exact two-sided Wilcoxon signed-rank tests [104] to compare paired app-level outcomes, because each baseline tool and the proposed approach were evaluated on the same five subject apps. Each app constituted one comparison unit (N = 5). For the primary coverage analysis, the three recorded measurements were averaged within each app and tool. This mean summarizes performance across the recorded executions and reduces reliance on an exceptionally favorable run. The best-of-three reported values were evaluated separately in an exploratory sensitivity analysis.

The signed-rank null hypothesis was that the distribution of paired differences was symmetric about zero. Absolute nonzero differences were ranked, with average ranks assigned to ties; zero-difference pairs were excluded. Exact two-sided p-values were obtained by enumerating every sign assignment of the nonzero ranked differences. Holm's procedure [105] controlled the family-wise error rate across all seven primary comparisons, comprising three coverage comparisons and four mutation-score comparisons, at a significance threshold of 0.05. The three exploratory best-of-three coverage comparisons were adjusted separately using Holm's procedure and did not contribute to the primary significance decisions. We report the smaller of the positive and negative rank sums (W), mean and median paired differences in percentage points (pp), and the matched rank-biserial effect size, calculated as the difference between the positive and negative rank sums divided by their sum. Positive effect sizes favor the proposed approach. An effect size of 1.00 indicates that all nonzero paired differences favor the proposed approach; it does not represent a 100% improvement. Pooled mutation scores and the overall undetected-mutant rate are descriptive aggregates; inference uses app-level scores.

4.4. Systems Under Test (SUTs): Objects under study

To evaluate the proposed approach, we selected a set of subject apps by exploring well-known open-source repositories, including GitHub and F-Droid, based on the following criteria: (1) the app must be a native Android app developed in Java; (2) the source code must be publicly accessible; (3) the app must employ a widget-based GUI—namely excluding canvas-based or game-like interfaces; (4) the app must have been used as a subject in prior studies; and (5) the apps must be vary in size and category. Based on these criteria, we chose five Android apps and downloaded their source code for the following empirical studies, as detailed in Table 3.

Table 3: Summarized information of the subject apps

App subject	Description	#LOC	# of Activities (UI screens)	Category	Used in	URL
ContactManager	Contacts manager	2,495	7	Business	[31, 86]	[76]
Mileage	A mileage tracker	44,631	8	Finance	[31, 87]	[88]
MultiSMS	Sending a message to multiple users	1,943	6	Communication	[53, 87]	[89]
Notepad	Note taking app	332	3	Text Editor	[31, 87]	[90]
ToDoManger	Daily task manager	323	2	Productivity	[31, 53]	[91]

We intentionally used this benchmark because it was also employed in our previous work [31] published in the *Information and Software Technology* (IST) journal, which supports direct comparability and reproducibility across studies. At the same time,

we acknowledge that the benchmark is relatively small, with only 2 to 8 UI screens per app, and therefore may not fully reflect the complexity of larger commercial Android apps. This limitation should be considered when interpreting the empirical results.

4.5. RQ1: Coverage analysis

This study evaluates the effectiveness of the test suite generated by the proposed approach in terms of statement coverage [53, 71, 92], as it reflects the extent to which code is exercised during testing. Since unexecuted or untested code paths are frequent sources of crashing and non-crashing defects, higher coverage is generally associated with a greater fault-detection potential. Accordingly, this study addresses RQ1 by benchmarking the proposed approach against state-of-the-art testing tools

4.5.1. Process and data preparation

Following the experimental setup detailed in Section 4.3, we compared the code coverage of the proposed approach against three baseline tools: Monkey [40], Stoat [23], and DinoDroid [71].

To do this study, each subject app was executed against the test suite generated by the proposed approach and the baseline tools. Code coverage achieved during execution was recorded via JaCoCo [93]. In accordance with our bias mitigation protocol (Section 4.3.3), the best coverage across three independent runs is reported for the baseline tools [71]. The paired analysis (described in Section 4.3.4) uses within-app means of the three recorded measurements.

4.5.2. Results and discussion

Figure 6 shows the best-of-three code coverage achieved across all subject apps, and Figure 7 summarizes the corresponding app-level distribution.

For the primary statistical analysis, code coverage was first averaged across the three recorded measurements within each app. As shown in Figure 7, on average, the proposed approach achieves 85% code coverage, outperforming Monkey (47%), Stoat (62%), and DinoDroid (70%). The observed gains were 38%, 23%, and 15%, respectively. Furthermore, all five differences were positive for each baseline tool, giving $W = 0$ and a matched rank-biserial effect size of 1.00, indicating that all nonzero paired differences favored the proposed approach. Each exact two-sided test yielded $p = 0.0625$; Holm adjustment across the seven primary comparisons yielded $p = 0.4375$. The observed code coverage gains were therefore consistent across these apps, but did not establish statistically significant superiority at the 0.05 level. The best-of-three sensitivity analysis produced the same unadjusted p-values; Holm adjustment within its separate three-comparison family yielded $p = 0.1875$ for each baseline tool.

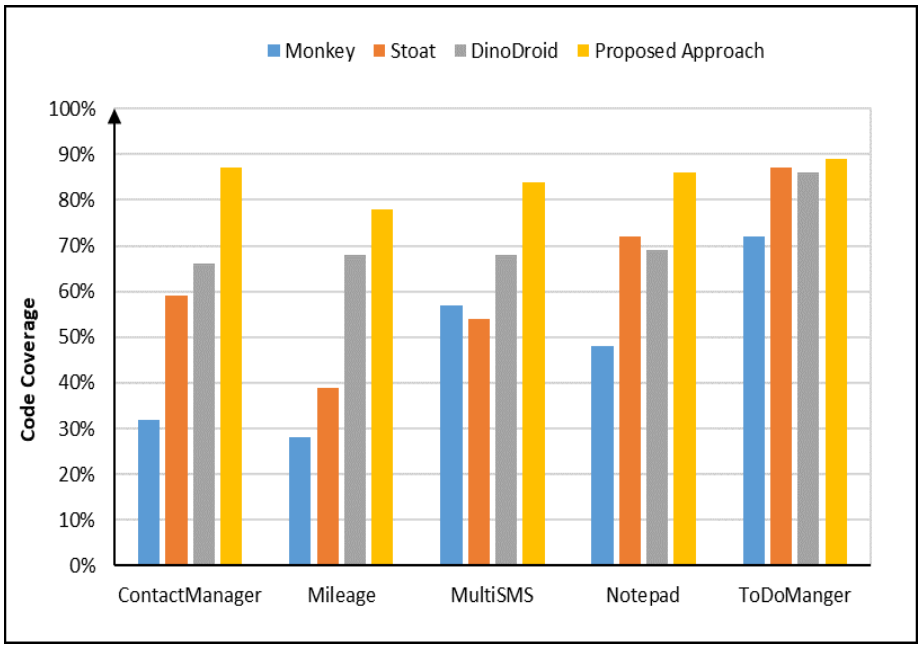


Figure 6: Code coverage achieved by the proposed approach and comparative tools

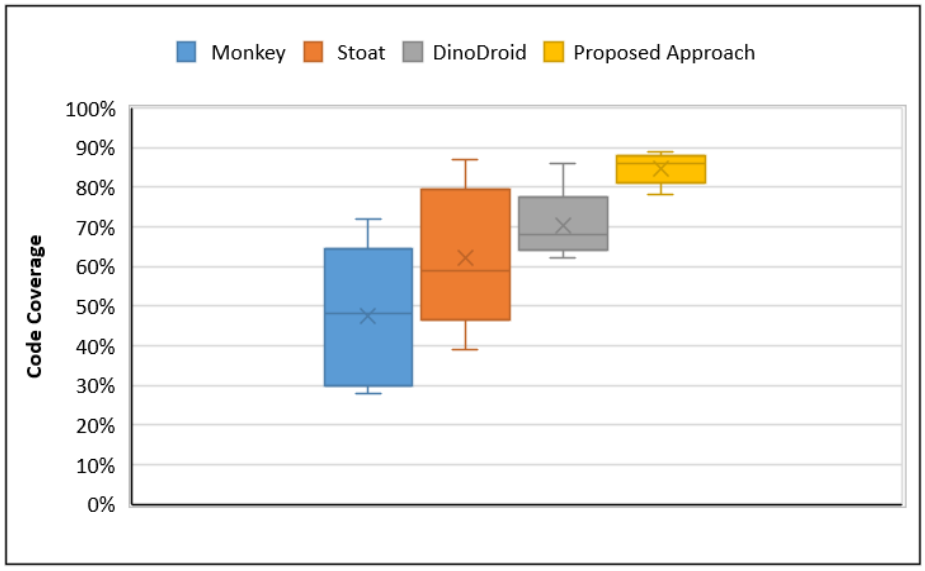


Figure 7: Comparative statistical analysis of code coverage

A detailed analysis across individual subject apps shows that the performance of the baseline tools degrades as architectural complexity increases—such as in apps with deeper functional layers or a larger number of interacting activities. For instance, while Monkey exhibits reasonable coverage on simpler apps like *ToDoManager* [91], its effectiveness declines sharply on complex apps such as *Mileage* [88] and *ContactManager* [76]. This degradation stems directly from its purely random event-generation strategy, which is unable to generate meaningful event sequences or supplying the valid input data necessary to traverse deep functional features [67, 71].

Stoat achieves higher coverage than Monkey by employing dynamic exploration models to guide event sequence generation. Nevertheless, because its underlying behavioral models are constructed through stochastic exploration, they are often incomplete, leaving complex functionalities unexercised [92]. DinoDroid, which leverages deep Q-network reinforcement learning [94], improves upon Stoat by generating more sophisticated event sequences capable of reaching deeper functionalities. However, its inherently non-deterministic behavior prevents it from generating consistently targeted event sequences to hit the functional features in the complex subject apps. So, it cannot guarantee coverage of all relevant behaviors. DinoDroid also struggles to exercise data-dependent branches due to the absence of a mechanism for generating valid test data.

In contrast, the proposed approach systematically overcomes these limitations by coupling abstract behavioral models (EM and ERM) as prior knowledge with a multi-entry-point strategy [95]. This strategy allows its test engine to generate targeted event sequences that traverse functional hierarchies directly, rather than relying on chance state discovery. The minor coverage limitations observed in complex apps like *Mileage* and *MultiSMS* are primarily due to limited coverage of conditional branches rather than unexercised functional features.

Answer to RQ1: The obtained results indicate that the proposed approach achieved an average code coverage of 85.0% across all subject apps. Based on within-app means, it achieved substantial mean paired gains of 38% , 23%, and 15% over Monkey, Stoat, and DinoDroid, respectively, with all non-zero differences favoring our approach (r=1.00). While the exact Wilcoxon signed-rank tests did not reach nominal significance at the α=0.05 level following Holm adjustment due to the discrete sample size (N=5, unadjusted p=0.0625), the large effect sizes and consistent gains demonstrate the effectiveness of handler-aware behavioral modeling in systematically exercising mobile app execution paths.

4.6. RQ2-RQ4: Mutation analysis

This empirical study evaluates the (fault-detection) effectiveness of the proposed approach using mutation testing technique [10]. Specifically, we aim to (1) measure the overall fault-detection effectiveness of the proposed approach (RQ2); (2) quantify the specific contribution of GUI-level invariants as app-specific GUI oracles in revealing defects (RQ3); and (3) conduct a comparative benchmark against baseline tools—Monkey, Stoat, Dinodroid, and REGDROID [6] —in detecting both crashing and non-crashing (functional) defects (RQ4).

4.6.1. Process and data preparation

The evaluation began by generating synthetic faults (mutants) for each subject apps using DroidMutator [96], injecting defects across general Java-level logic, Android component lifecycle, and event-handling constructs [6]. Mutants identified as undetectable (e.g., equivalent mutants) were manually inspected and discarded. Consistent with standard mutation-testing protocols facing computational constraints [97, 98], a representative sample of approximately 30% of the non-equivalent mutants was randomly retained across all subject app. Table 4 details the distribution of generated and selected mutants across operator types.

Table 4: Detailed statistics of generated and selected mutants for the subjects
#GM (displays the number of generated mutants), #SM (indicates the number of selected mutants).

`Mobile app	#SM / #GM	Generated and selected mutants per each mutation operator type	
		Java-specific #SM / # GM	Android-specific #SM / #GM
ContactManager	300 / 989	162 / 693	138 / 296
Mileage	270 / 884	209 / 708	61 / 176
MultiSMS	150 / 448	101 / 392	49 / 56
Notepad	51 / 97	17 / 45	34 / 52
ToDoManager	40 / 80	19 / 54	21 / 26
Total	**811 / 2498**	**508 / 1892**	**303 / 606**

We conducted three mutation-based evaluations to assess the (fault-detection) effectiveness of the proposed approach. In the first study, the generated test suites for each subject app by the proposed approach were executed on the selected mutants to compute the overall mutation score (MS metric, Eq. 2) and address RQ2.
In the second study, we evaluated the contribution of GUI-level invariants used as test oracles in the generated test suite to defect detection. To do this, the generated test suite was executed under two distinct configurations—with GUI-level oracles enabled and with oracles disabled (relying solely on crash detection)—to calculate the Oracle Fault Detection Gain (OFDG metric, Eq. 3) and address RQ3.
In the third study, we compared the fault-detection effectiveness of the proposed approach with the baseline tools to address RQ4. We distinguish between two categories of baselines tools: implicit-oracle-based tools (i.e., Monkey, Stoat, and Dinodroid) and an explicit-oracle-based tool (i.e., REGDROID). The implicit-oracle-based tools provide a comparison based on crash detection, while REGDROID, which uses differential testing [99], provides the explicit-oracle-based comparison for detecting non-crashing functional defects.
For the RQ4 comparison, a randomly selected subset of mutants were used for each subject app, consisting of no fewer than 10 mutants and no more than 10% of its total mutants, in the preceding studies. Each tool was evaluated on the same subset within each app. Since, REGDROID exploits differential regression oracles across consecutive versions, the original version of each subject app served as the reference, and its mutant version served as the target. The comparative mutation-score analysis treats each app as one paired observation, following Section 4.3.4, and the application-level killed/tested ratios were reported.

1.1.1.1 4.6.2. Results and discussion

1.1.1.2 4.6.2.1. RQ2: Fault-detection effectiveness (mutation scores)

Table 5 summarizes the mutation testing results across all subject applications. Across all subject apps, the proposed approach successfully detected 789 mutants out of 811 mutants, achieving an aggregate detection rate of 97.3% and an average application-level mutation score of 95.6%. The undetected mutants (approximately 2.7% overall) occurred exclusively within Java-specific mutation operators whose internal state alterations did not propagate to observable GUI properties. Crucially, the detected faults were evenly distributed across both Java-specific logic mutants (477 killed) and Android-specific lifecycle and event-handling mutants (312 killed). These findings confirm that the proposed test-generation strategy, namely coupling formal behavioral abstractions (EM and ERM) with the multi-entry-point strategy, yields comprehensive fault-revealing test suites capable of exposing both java-specific and Android-specific defects.

Table 5. Mutation testing results for the proposed approach.
#GM (generated mutants), #DM (detected mutants), and MS (mutation score).

Mobile app	#GM	#DM	Java-Specific	Android-Specific	Overall MS
			#DM	#DM	
ContactManager	300	300	162	138	1.00
Mileage	270	270	209	61	1.00
MultiSMS	150	132	73	59	0.88
NotePad	51	51	17	34	1.00
ToDoManager	40	36	16	20	0.90
Total	**811**	**789**	**477**	**312**	**0.96**

Answer to RQ2: The obtained results show that the proposed approach achieved high fault-detection effectiveness, killing 789 out of 811 evaluated mutants (an aggregate mutation score of 97.3% and an average app-level score of 95.6%). Moreover, the utilized test oracles contribute in the detection of both Java-specific and Android-specific defects, underscoring their utility in revealing non-crashing functional defects in the subject apps.

1.1.1.3 4.6.2.2. RQ3: Fault-Detection Gain of Test Oracles

Figure 8 illustrates the explicit contribution of model-derived GUI assertion oracles in detecting faults compared to implicit crash oracles alone.

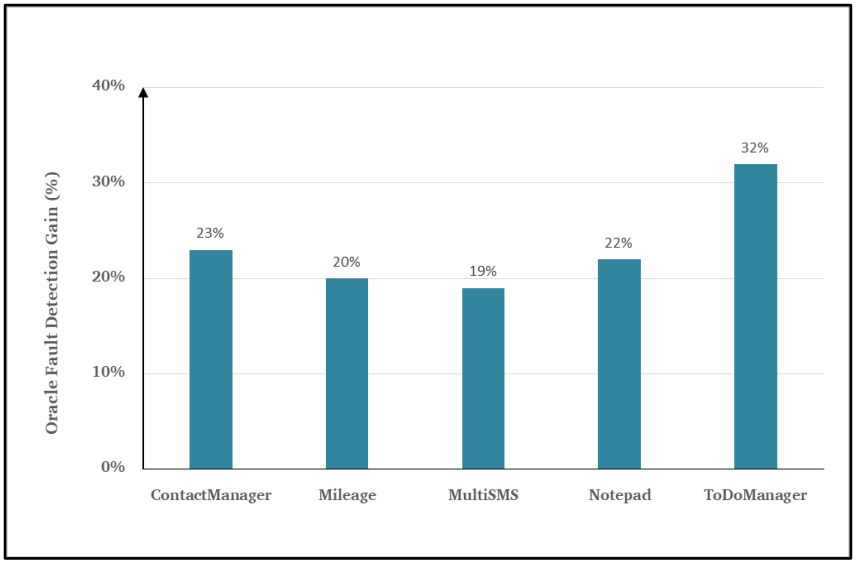


Figure 8: The contribution of the generated oracle in fault-detection effectiveness

Integrating GUI-level invariants as app-specific oracles into the test suite generated by the proposed approach yielded a substantial improvement in fault revelation, providing an OFDG ranging from 19% to 32% (mean: 23%) across all subject apps. For instance, in *ContactManager*, 23% of the injected defects were detected exclusively through oracle evaluation. These defects did not trigger runtime exceptions or application crashes and were identified strictly through explicit oracle violations. This finding underscores the essential role of test oracle in identifying non-crashing functional defects during GUI testing.

Answer to RQ3: The findings indicate that incorporating the derived GUI-level invariants as test oracles into the generated test suite improves their fault-detection effectiveness by 19–32% (mean: 23%).

1.1.1.4 4.6.2.3. RQ4: Comparative Analysis

Table 6 summarizes the comparative fault-detection results across the baseline tools for both crashing and non-crashing defects. As observed, the proposed approach achieves the highest overall effectiveness, detecting all 36 crashing defects (100%) and 54 out of 57 non-crashing defects (94.7%), resulting in an aggregate detection rate of 96.8% (90/93). The undetected defects (OUM =3.2%) were either outside the behavioral scope of the generated test suite or not observable through the employed test oracles. In contrast, DinoDroid, Stoat, and Monkey detect 33, 30, and 28 crashing defects, respectively, while detecting zero non-crashing defects. Meanwhile, REGDROID detects 30 crashing defects and 44 non-crashing defects.

Table 6. Comparative results of the proposed approach, and the baseline tools in detecting crashing (#C) and non-crashing defects (#NC)

Apps	Mutants used			Killed by Monkey			Killed by Stoat			Killed by DinoDroid			Killed by REGDROID			Killed by proposed approach		
	Total	#C	#NC	Total	#C	#NC	Total	#C	#NC	Total	#C	#NC	Total	#C	#NC	Total	#C	#NC
ContactManager	30	16	14	12	12	0	12	12	0	14	14	0	23	13	10	30	16	14
Mileage	**27**	**9**	**18**	**7**	**7**	**0**	**8**	**8**	**0**	**7**	**7**	**0**	**22**	**7**	**15**	**27**	**9**	**18**
MultiSMS	**15**	**3**	**12**	**2**	**2**	**0**	**2**	**2**	**0**	**3**	**3**	**0**	**11**	**3**	**8**	**13**	**3**	**10**
NotePad	**11**	**5**	**6**	**4**	**4**	**0**	**5**	**5**	**0**	**5**	**5**	**0**	**9**	**4**	**5**	**11**	**5**	**6**
ToDoManager	**10**	**3**	**7**	**3**	**3**	**0**	**3**	**3**	**0**	**3**	**3**	**0**	**9**	**3**	**6**	**9**	**3**	**6**
Total	**93**	**36**	**57**	**28**	**28**	**0**	**30**	**30**	**0**	**33**	**33**	**0**	**74**	**30**	**44**	**90**	**36**	**54**

To interpret these outcomes critically, defect detection must be viewed through two prerequisites: (i) generating event sequences that reach and trigger fault-sensitive states, and *(ii)* employing test oracles capable of observing the resulting behavioral deviations. Evaluating the baselines along these two dimensions helps the observed performance of the proposed approach:

- Monkey, Stoat, and DinoDroid rely on implicit crash oracles, so their zero detection of non-crashing defects is expected under this configuration. Their crashing-defect counts also provide an observed comparison of the executed test suites: the proposed approach detected 36 crashing defects, compared with 33 for DinoDroid, 30 for Stoat, and 28 for Monkey. This pattern is consistent with the targeted, handler-aware event sequences used by the proposed approach and the higher observed code coverage reported for RQ1.
- Against REGDROID, the proposed approach achieved higher observed detection counts for both crashing (36 vs. 30) and non-crashing defects (54 vs. 44). These outcomes are consistent with the complementary roles of targeted test generation and app-specific GUI invariants. The proposed test-generation strategy targets functional features and event-handler dependencies, while the GUI-level invariants provide explicit

oracles for GUI-visible behaviors.

The observed gains involve both crashing and non-crashing defects; Table 7 reports the exact paired comparisons. The proposed approach and REGDROID achieved mean app-level mutation scores of 95.33% and 80.66%, respectively, with mean and median gains of 14.67% and 18.18%. Four apps favored the proposed approach; ToDoManager was tied. Excluding this zero difference yielded exact $p = 0.125$. Against Monkey, Stoat, and DinoDroid, all five differences were positive; mean gains were 66.21%, 63.65%, and 61.72%, respectively, with exact $p = 0.0625$. All mutation comparisons yielded $W = 0$ and a matched rank-biserial effect of 1.00. Every primary comparison had Holm-adjusted $p = 0.4375$ and was non-significant at the 0.05 level.

Table 7. Exact paired statistical comparisons across five subject apps

Outcome	Baseline	Nonzero pairs / N	Mean Δ (%)	Median Δ (%)	W	Matched rank-biserial effect	Exact p	Holm p
Coverage	Monkey	5/5	42.00	38.67	0	1.00	0.0625	0.4375
Coverage	Stoat	5/5	27.27	34.67	0	1.00	0.0625	0.4375
Coverage	DinoDroid	5/5	14.20	16.33	0	1.00	0.0625	0.4375
Mutation	Monkey	5/5	66.21	63.64	0	1.00	0.0625	0.4375
Mutation	Stoat	5/5	63.65	60.00	0	1.00	0.0625	0.4375
Mutation	DinoDroid	5/5	61.72	60.00	0	1.00	0.0625	0.4375
Mutation	REGDROID	4/5	14.67	18.18	0	1.00	0.1250	0.4375

Each comparison contains five paired apps; the pair's column gives the nonzero count relative to this total. Coverage uses within-app means; mutation uses Table 6 ratios. Differences are proposed minus baseline. Mean and median gains include the tied app; W and the matched rank-biserial effect exclude it. An effect of 1.00 indicates positive nonzero differences, not a 100% gain. Holm adjustment covers seven primary comparisons. Minimum attainable exact two-sided p-values are 0.0625 for five nonzero pairs and 0.125 for four. Non-significance does not establish equivalence.

Answer to RQ4: The comparative analysis demonstrates the effectiveness of the proposed approach in revealing both crashing and non-crashing defects compared to the baseline tools, achieving OUM = 3.2%, markedly lower than the baseline rates ranging from 20.4% to 69.9%. Its app-level mutation scores exceeded those of the implicit-oracle baselines on all five apps and REGDROID on four apps, with one tie. The observed gains included crashing and non-crashing defects; the exact paired comparisons remained non-significant after Holm correction.

5. Discussions

This section provides a comprehensive discussion of the proposed approach from other aspects beyond its quantitative evaluation results. In particular, we discuss its manual effort required to generate test cases, scalability, practical considerations for adoption in real-world development environments, inherent limitations, and the key threats that may affect the validity of our findings. Together, these analyses offer a broader understanding of the strengths and constraints of the proposed approach and its applicability in practice.

5.1. Manual Effort and Cost-Benefit Analysis

In the proposed approach, the required human effort is limited to an initial setup phase. This involves a few key tasks: collecting execution traces through lightweight exploratory interaction, reviewing and selecting appropriate GUI-level invariants from a system-generated list, and describing complex functional workflows [10, 11]. Based on our study, the time needed for this setup phase scaled directly with app size and structural complexity, ranging from 35 to 205 minutes, with an average of approximately two hours per app.

To understand the value of our approach, we need to compare its cost and effectiveness with the baseline tools. Our study (RQ4, Table 6) shows that the baseline tools (such as Monkey, Stoat, and Dinodroid) are quick to set up, but have significant limitations. They struggle to reveal all crashing defects because they don't explore the app deeply enough, and importantly, they cannot identify non-crashing functional defects since they rely solely on implicit oracles. On the other hand, manual testing, for example using Espresso, is accurate but very time-consuming, often requiring 10 to 16 person-hours for medium-sized apps to achieve similar test coverage [100]. Furthermore, although REGDROID avoids some manual scripting, it introduces distinct challenges:

it requires running two versions of the app and comparing results, which can lead to substantial computational overhead and frequent false alarms due to minor, harmless changes in the app's layout (layout drift). In contrast, as demonstrated in RQ1 and RQ4, the proposed approach provides superior effectiveness —uncovering 94.7% of non-crashing functional mutants and 100% of crashes, alongside 85% statement coverage—compared to existing baselines. The initial two-hour effort is significantly less than purely manual testing, while completely avoiding the diagnostic shortcomings and high post-test analysis burden associated with fully automated or differential techniques.

Moreover, the initial investment time is justified because it is largely a one-time cost. Once the necessary invariants and workflow descriptions are defined, they are saved and can be reused within the Continuous Integration and Continuous Delivery (CI/CD) pipeline. This means that for subsequent app versions, human intervention is significantly mitigated, primarily involving the verification and potential adaptation of previously defined test scenarios and selected oracles. This reuse dramatically reduces the long-term effort and cost associated with maintenance, establishing a sustainable pathway toward highly reliable testing with minimal recurring manual effort [101].

5.2. Scalability

Scalability is an essential consideration when applying the proposed approach to large commercial apps with rich and deeply nested GUIs. Several components rely on widely used tools—such as JavaParser, Daikon, and Monkey—that have been shown to scale effectively in prior research [31, 40, 102]. Excluding these external components, the computational complexity of our proposed approach can be expressed as:

$$Complexity = \begin{cases} O\,(NumOfwidgets^2), & NumOfWidgets \gg NumOfFeatures \\ O(NumOfFeature), & else \end{cases} \quad \text{(Eq. 5)}$$

It indicates that the overall computational cost remains manageable even for large apps. Also, human involvement—such as selecting invariants or describing complex test scenarios— does not pose a scalability bottleneck by automated recommendations that reduce effort as the app size grows.

Furthermore, the runtime data stored—such as trace files—has negligible storage requirements (e.g., the largest trace file in our experiments occupied only 16 KB). Moreover, collecting execution trace data requires minimal human participation, ensuring that the scalability of the proposed approach is not hindered by manual intervention.

Consequently, the theoretical and component-level analysis suggests that the proposed approach can be applied effectively to large-scale Android apps. Nevertheless, while the underlying algorithms are designed for scalability, the empirical validation in this study was conducted on a benchmark of relatively small apps. Further empirical studies on large-scale industrial apps are required to fully evaluate the practical performance and any potential interaction bottlenecks in such complex environments.

5.3. Practical Considerations

The practicality of the proposed approach is reflected in its integration design and compatibility with existing development workflows. Implementing the proposed approach as Android Studio plugins facilitates its integration seamlessly without requiring significant configuration effort [103]. This integration supports the automatic extraction of the information needed to derive GUI-level invariants and generate test cases, thereby improving efficiency.

Despite the proposed approach developed originally for Android Studio, it can be adapted to other development environments with minimal effort. Moreover, the generated test cases are fully compatible with the Espresso testing framework, ensuring straightforward integration into existing testing infrastructures [103]. Also, storing the generated test cases within the project under test improves accessibility, allowing development and testing teams to review, execute, and maintain them based on the project needs [103].

Finally, the semi-automated nature of the proposed approach aligns well with modern CI/CD practices [103]. Its integration into CI/CD pipelines enables automated execution of test cases, promotes rapid feedback cycles, and consistent high-quality software delivery [103]. So, the proposed approach enhances agility, reliability, productivity, and efficiency within contemporary development workflows [103].

5.4. Limitations and Potential Threats to validity

Despite its advantages, our proposed approach has some limitations. Firstly, It depends on access to the SUT's source code— which restricts its use in scenarios where source code is unavailable. Nevertheless, this requirement does not prevent its integration into CI/CD pipelines in typical development environments. Secondly, its current implementation supports only native Android apps developed in Java; however, extending it to other languages such as Kotlin is straightforward.

Thirdly, it is also intended for widget-based GUIs and does not currently address canvas-based apps— such as many game interfaces. Since widget-based GUIs dominate most mobile apps domains, this limitation does not substantially affect its practicality. Finally, although the heuristic approach used to prioritize GUI invariants simplifies the GUI invariant selection process, developing a more structured prioritization strategy could further improve its effectiveness and usability.

Construct Validity. A key threat to construct validity concerns the reliability of the derived GUI-level invariants— which depend on the execution traces collected from the SUTs. To mitigate this concern, we employed a hybrid data-collection strategy that combines human-driven interactions with automated event generation using Monkey. Human interactions enabled exploration of functional scenarios requiring complex event sequences or valid input data, whereas the Monkey tool was used to cover additional execution paths and increase the diversity of collected trace logs. Moreover, GUI-level invariants were inferred using Daikon— a well-established invariant inference tool— to improve the correctness and consistency of the extracted invariants.

Another construct-related threat arises from the characteristics of certain Android apps that may hinder the extraction of required information from source code— such as predefined Android activities (e.g., preference activities) whose implementation logic is inaccessible. Although the utilized subject apps did not contain such activities, we note that their presence could limit the applicability of our proposed approach. To mitigate this threat in practice, development teams can be encouraged to avoid such components when testability is a priority. Additionally, variation in programming styles across apps may complicate source-

code analysis. Although we utilized several normalization steps to standardize code structures for information extraction in the proposed approach, establishing coding-style guidelines in real development environments would further minimize this issue.

Internal Validity. Two main threats to internal validity were identified in evaluating the effectiveness of the generated test cases and GUI oracles. The first threat relates to the evaluation methodology. To mitigate this threat, we utilized mutation testing—a widely accepted and rigorous technique— to measure the fault-detection capability of the generated test cases and GUI oracles. Moreover, using a standard metric— mutation score— for measuring the quality of the generated test cases in fault-detection capability, carefully selection of a mutation testing tool to ensure coverage of both crashing and non-crashing functional defects, ignoring equivalent or invalid mutants to prevent bias in the results, we tried to the robustness and reliability of the evaluation process.

Another threat arises from the baseline tools which are used in comparative analysis experiments. To mitigate this, we consider some criteria for selecting the baseline tools. First, they have been utilized as a benchmark tool in the previous studies. Second, they are representative of various test generation strategies, e.g., random-based, model-based, and etc. Third, they use various oracle mechanisms. Finally, they were accessed publicly. So, based on the mentioned criteria, we chose and used the utilized baseline tools in all experiments.

External Validity. Threats to external validity concern the generalizability of our findings. To mitigate these threats, we selected a diverse set of open-source, widget-based Android apps from well-known repositories such as GitHub and F-Droid. The selected apps vary in size, category, and structural complexity, and have been used in prior empirical studies, making them recognized subjects for empirical study. Although the proposed approach is currently limited to widget-based Android apps, this category still covers a substantial portion of mobile apps used in practice.

However, a threat to generalizability remains due to the scale of the subject apps, which do not exceed 8 screens. While these subject apps are representative real-world mobile apps and our approach is technically capable of handling larger apps, the current results may not fully generalize to complex commercial apps. We acknowledge this as a limitation and plan to address larger-scale industrial apps in future extensions of this work.

6. Conclusions

This study presented a semi-automated, feature-oriented model-based approach for generating targeted GUI test cases for mobile apps. By integrating two complementary behavioral models (i.e., EM and ERM), GUI-level invariant-based oracles, and multi-entry-point exploration strategy, our approach addresses critical limitations in existing GUI testing techniques, such as shallow exploration, the lack of effective oracles, and the difficulty of exercising complex feature-level interactions.

Empirical evaluation conducted on five open-source Android applications demonstrates the effectiveness of the proposed framework. The generated test suites achieved an average statement coverage of 85% and an average mutation score of 97.3%, confirming their high capability in revealing faults. Notably, incorporating GUI-level invariants as app-specific oracles in the generated test suite improved its mutation score by 19–32% compared with test execution without invariant-based validation. Moreover, the results demonstrate the role of integrating invariants in improving fault-detection effectiveness of the proposed approach against the baseline tools by revealing both crashing and non-crashing functional defects.

While these results are promising, it is important to interpret them within the study's scope. The evaluation was conducted on small-to-medium open-source apps (ranging up to 8 screens and ~44K LOC). Although these subject apps incorporate critical real-world functional patterns (e.g., CRUD operations, persistent storage, and background processing), the observed performance may vary in monolithic, large-scale enterprise systems. Furthermore, the proposed approach currently requires limited human involvement to encode semantic test scenarios. Future work will focus on scaling the empirical evaluation to industrial-scale apps, increasing the level of automation to reduce human effort, and extending support to multi-platform environments.

Data Availability Statement

The source code of the implemented plugins and all empirical data and results are available in [30].

References

[1] A. K. Y. Tang, "A systematic literature review and analysis on mobile apps in m-commerce: Implications for future research," *Electron. Commer. Res. Appl.*, vol. 37, p. 100885, Sep. 2019, doi: 10.1016/j.elerap.2019.100885.

[2] "App - Worldwide | Statista Market Forecast." https://www.statista.com/outlook/dmo/app/worldwide (accessed Nov. 02, 2023).

[3] "Japan's COVID-19 app failed to pass on some contact warnings | Reuters." https://www.reuters.com/article/us-health-coronavirus-japan-app-idUSKBN2A31BA/ (accessed Dec. 05, 2023).

[4] "E-Commerce App Loses 'Tens of Millions' From Coupon Glitches." https://www.sixthtone.com/news/1003483 (accessed Dec. 05, 2023).

[5] "App Uninstall Statistics 2019: 25% App Abandonment | Localytics." https://uplandsoftware.com/localytics/resources/blog/25-of-users-abandon-apps-after-one-use/ (accessed Dec. 05, 2023).

[6] Y. Xiong *et al.*, "An empirical study of functional bugs in android apps," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, Jul. 2023, pp. 1319–1331, doi: 10.1145/3597926.3598138.

[7] T. Su *et al.*, "Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–31, Oct. 2021, doi: 10.1145/3485533.

[8] J. Johnson, J. Mahmud, T. Wendland, K. Moran, J. Rubin, and M. Fazzini, "An Empirical Investigation into the Reproduction of Bug Reports for Android Apps," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2022, pp. 321–322, doi: 10.1109/SANER53432.2022.00048.

[9] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1679–1694, Oct. 2013, doi: 10.1016/j.infsof.2013.03.004.

[10] P. Tramontana, D. Amalfitano, N. Amatucci, and A. R. Fasolino, "Automated functional testing of mobile applications: a systematic mapping study," *Software Qual. J.*, vol. 27, no. 1, pp. 149–201, Mar. 2019, doi: 10.1007/s11219-018-9418-6.

[11] J. Sun, T. Su, J. Jiang, J. Wang, G. Pu, and Z. Su, "Property-Based Fuzzing for Finding Data Manipulation Errors in Android Apps," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, Nov. 2023, pp. 1088–1100, doi: 10.1145/3611643.3616286.

[12] M. Linares-Vasquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 613–622, doi: 10.1109/ICSME.2017.47.

[13] L. Nie, K. S. Said, L. Ma, Y. Zheng, and Y. Zhao, "A systematic mapping study for graphical user interface testing on mobile apps," *IET Softw.*, vol. 17, no. 3, pp. 249–267, Jun. 2023, doi: 10.1049/sfw2.12123.

[14] P. H. Kuroishi, J. C. Maldonado, and A. M. R. Vincenzi, "Towards the definition of a research agenda on mobile application testing based on a tertiary study," *Inf. Softw. Technol.*, p. 107363, Nov. 2023, doi: 10.1016/j.infsof.2023.107363.

[15] J. Wang *et al.*, "Detecting non-crashing functional bugs in Android apps via deep-state differential analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, Nov. 2022, pp. 434–446, doi: 10.1145/3540250.3549170.

[16] M. Linares-Vasquez, K. Moran, and D. Poshyvanyk, "Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 399–410, doi: 10.1109/ICSME.2017.27.

[17] K. S. Said, L. Nie, A. A. Ajibode, and X. Zhou, "GUI testing for mobile applications: objectives, approaches and challenges," in *12th Asia-Pacific Symposium on Internetware*, New York, NY, USA, Nov. 2020, pp. 51–60, doi: 10.1145/3457913.3457931.

[18] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A technique for recording, encoding, and running platform independent android tests," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Mar. 2017, pp. 149–160, doi: 10.1109/ICST.2017.21.

[19] L. Ardito, R. Coppola, S. Leonardi, M. Morisio, and U. Buy, "Automated test selection for android apps based on APK and activity classification," *IEEE Access*, vol. 8, pp. 187648–187670, 2020, doi: 10.1109/ACCESS.2020.3029735.

[20] S. Negara, N. Esfahani, and R. Buse, "Practical Android Test Recording with Espresso Test Recorder," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 193–202, doi: 10.1109/ICSE-SEIP.2019.00029.

[21] O. Sahin, A. Aliyeva, H. Mathavan, A. Coskun, and M. Egele, "RANDR: record and replay for android applications via targeted runtime instrumentation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2019, pp. 128–138, doi: 10.1109/ASE.2019.00022.

[22] "UI/Application Exerciser Monkey | Android Studio | Android Developers." https://developer.android.com/studio/test/other-testing-tools/monkey (accessed Dec. 06, 2023).

[23] T. Su *et al.*, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, New York, New York, USA, Sep. 2017, pp. 245–256, doi: 10.1145/3106237.3106298.

[24] "Improve your code with lint checks | Android Studio | Android Developers." https://developer.android.com/studio/write/lint.html (accessed Dec. 06, 2023).

[25] "FindBugs™ - Find Bugs in Java Programs." https://findbugs.sourceforge.net/ (accessed Dec. 06, 2023).

[26] "Infer Static Analyzer | Infer | Infer." https://fbinfer.com/ (accessed Dec. 06, 2023).

[27] "Robotium." https://robotium.org/ (accessed Dec. 06, 2023).

[28] "Appium Documentation - Appium Documentation." http://appium.io/docs/en/2.2/ (accessed Dec. 06, 2023).

[29] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, May 2015, doi: 10.1109/TSE.2014.2372785.

[30] "Yarifard/Feature-oriented-model-based-Test-Generation." https://github.com/Yarifard/Feature-oriented-model-based-Test-Generation (accessed Jun. 07, 2026).

[31] A. A. Yarifard, S. Araban, S. Paydar, V. Garousi, M. Morisio, and R. Coppola, "Extraction and empirical evaluation of GUI-level invariants as GUI Oracles in mobile app testing," *Inf. Softw. Technol.*, vol. 177, p. 107531, Jan. 2025, doi: 10.1016/j.infsof.2024.107531.

[32] W. Guo, Z. Dong, L. Shen, W. Tian, T. Su, and X. Peng, "iFixDataloss: a tool for detecting and fixing data loss issues in Android apps," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, Jul. 2022, pp. 785–788, doi: 10.1145/3533767.3543297.

[33] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2017, pp. 308–318, doi: 10.1109/ASE.2017.8115644.

[34] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl eyes: Spotting UI display issues via visual understanding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, Dec. 2020, pp. 398–409, doi: 10.1145/3324884.3416547.

[35] "Espresso | Android Developers." https://developer.android.com/training/testing/espresso/ (accessed Dec. 06, 2023).

[36] "Write automated tests with UI Automator | Android Developers." https://developer.android.com/training/testing/other-components/ui-automator (accessed Mar. 07, 2024).

[37] D. Amalfitano, N. Amatucci, A. R. Fasolino, and P. Tramontana, "AGRippin: a novel search based testing technique for Android applications," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile - DeMobile 2015*, New York, New York, USA, Aug. 2015, pp. 5–12, doi: 10.1145/2804345.2804348.

[38] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: are we there yet? (E)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 429–440, doi: 10.1109/ASE.2015.89.

[39] M. Mohammed, H. Cai, and N. Meng, "An empirical comparison between monkey testing and human testing (WIP paper)," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, New York, NY, USA, Jun. 2019, pp. 188–192, doi: 10.1145/3316482.3326342.

[40] S. Paydar, "An empirical study on the effectiveness of monkey testing for android applications," *Iran. J. Sci. Technol. Trans. Electr. Eng.*, vol. 44, no. 2, pp. 1013–1029, Jun. 2020, doi: 10.1007/s40998-019-00270-y.

[41] D. Amalfitano, N. Amatucci, A. R. Fasolino, P. Tramontana, E. Kowalczyk, and A. M. Memon, "Exploiting the saturation effect in automatic random testing of android applications," in *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, May 2015, pp. 33–43, doi: 10.1109/MobileSoft.2015.11.

[42] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: segmented evolutionary testing of Android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, Nov. 2014, pp. 599–609, doi: 10.1145/2635868.2635896.

[43] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, New York, New York, USA, Jul. 2016, pp. 94–105, doi: 10.1145/2931037.2931054.

[44] I. Arcuschin, J. P. Galeotti, and D. Garbervetsky, "An empirical study on how sapienz achieves coverage and crash detection," *J. Softw. Evol. and Proc.*, vol. 35, no. 4, Apr. 2023, doi: 10.1002/smr.2411.

[45] A. Rohella and S. Takada, "Testing Android Applications Using Multi-Objective Evolutionary Algorithms with a Stopping Criteria," in *Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering*, Jul. 2018, vol. 2018, pp. 308–353, doi: 10.18293/SEKE2018-084.

[46] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, New York, NY, USA, Nov. 2012, pp. 1–11, doi: 10.1145/2393596.2393666.

[47] J. Jeon, K. K. Micinski, and J. S. Foster, "SymDroid: Symbolic execution for Dalvik bytecode," *SymDroid: Symbolic execution for Dalvik bytecode*, 2012.

[48] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, New York, NY, USA, Jul. 2013, pp. 67–77, doi: 10.1145/2483760.2483777.

[49] J. Schutte, R. Fedler, and D. Titze, "Condroid: targeted dynamic analysis of android applications," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, Mar. 2015, pp. 571–578, doi: 10.1109/AINA.2015.238.

[50] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, Sep. 2015, doi: 10.1109/MS.2014.55.

[51] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Softw. Test. Verif. Reliab.*, vol. 17, no. 3, pp. 137–157, Sep. 2007, doi: 10.1002/stvr.364.

[52] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android (T)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 658–668, doi: 10.1109/ASE.2015.76.

[53] I.-A. Salihu, R. Ibrahim, B. S. Ahmed, K. Z. Zamli, and A. Usman, "AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing," *IEEE Access*, vol. 7, pp. 17158–17173, 2019, doi: 10.1109/ACCESS.2019.2895504.

[54] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of Android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, New York, NY, USA, Nov. 2018, pp. 31–37, doi: 10.1145/3278186.3278191.

[55] N. J. Alkhateeb and M. S. AbouTrab, "Developing a Framework for Testing Android Applications based on Reverse Engineering Techniques," *Journal of Digital Information Management*, vol. 17, no. 4, p. 189, Aug. 2019, doi: 10.6025/jdim/2019/17/4/189-200.

[56] Y. L. Arnatovich and L. Wang, "A systematic literature review of automated techniques for functional gui testing of mobile applications," *arXiv preprint arXiv:1812.11470*, 2018.

[57] F. Behrang and A. Orso, "Test Migration Between Mobile Apps with Similar Functionality," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2019, pp. 54–65, doi: 10.1109/ASE.2019.00016.

[58] X. Qin, H. Zhong, and X. Wang, "TestMig: migrating GUI test cases from iOS to Android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, Jul. 2019, pp. 284–295, doi: 10.1145/3293882.3330575.

[59] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2019, pp. 42–53, doi: 10.1109/ASE.2019.00015.

[60] F. Behrang and A. Orso, "AppTestMigrator: A tool for automated test migration for Android apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, New York, NY, USA, Jun. 2020, pp. 17–20, doi: 10.1145/3377812.3382149.

[61] J. Liang, S. Wang, X. Deng, and Y. Liu, "RIDA: Cross-App Record and Replay for Android," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2023, pp. 246–257, doi: 10.1109/ICST57152.2023.00031.

[62] S. Liu, Y. Zhou, T. Han, and T. Chen, "Test Reuse based on Adaptive Semantic Matching across Android Mobile Applications," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2022, pp. 703–709, doi: 10.1109/QRS57517.2022.00076.

[63] R. M. L. M. Moreira and A. C. R. Paiva, "PBGT tool: An integrated modeling and testing environment for pattern-based GUI testing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, New York, NY, USA, Sep. 2014, pp. 863–866, doi: 10.1145/2642937.2648618.

[64] I. C. Morgado and A. C. R. Paiva, "Mobile GUI testing," *Software Qual. J.*, vol. 26, no. 4, pp. 1–18, Sep. 2017, doi: 10.1007/s11219-017-9387-1.

[65] D. Amalfitano *et al.*, "Improving code coverage in android apps testing by exploiting patterns and automatic test case generation," in *Proceedings of the 2014 international workshop on Long-term industrial collaboration on software engineering*, New York, NY, USA, Sep. 2014, pp. 29–34, doi: 10.1145/2647648.2656426.

[66] M. Pan, Y. Lu, Y. Pei, T. Zhang, J. Zhai, and X. Li, "Effective testing of Android apps using extended IFML models," *Journal of Systems and Software*, vol. 159, p. 110433, Jan. 2020, doi: 10.1016/j.jss.2019.110433.

[67] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for Android GUI testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, New York, NY, USA, Nov. 2018, pp. 2–8, doi: 10.1145/3278186.3278187.

[68] Y. Koroglu and A. Sen, "Functional test generation from UI test scenarios using reinforcement learning for android applications," *Softw. Test. Verif. Reliab.*, vol. 31, no. 3, May 2021, doi: 10.1002/stvr.1752.

[69] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep Reinforcement Learning for Black-box Testing of Android Apps," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 1–29, Oct. 2022, doi: 10.1145/3502868.

[70] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2019, pp. 1070–1073, doi: 10.1109/ASE.2019.00104.

[71] Y. Zhao, B. Harrison, and T. Yu, "DinoDroid: Testing Android Apps Using Deep Q-Networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, pp. 1–24, Jun. 2024, doi: 10.1145/3652150.

[72] Z. Liu *et al.*, "Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, New York, NY, USA, Apr. 2024, pp. 1–13, doi: 10.1145/3597503.3639180.

[73] C. Wang, T. Liu, Y. Zhao, M. Yang, and H. Wang, "LLMDroid: Enhancing Automated Mobile App GUI Testing Coverage with Large Language Model Guidance," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, pp. 1001–1022, Jun. 2025, doi: 10.1145/3715763.

[74] Y. Lan *et al.*, "Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, New York, NY, USA, Feb. 2024, pp. 1–13, doi: 10.1145/3597503.3623344.

[75] C. Sun, Z. Zhang, B. Jiang, and W. K. Chan, "Facilitating monkey test by detecting operable regions in rendered GUI of mobile game apps," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Aug. 2016, pp. 298–306, doi: 10.1109/QRS.2016.41.

[76] "ContactManager | GitHub - Free and Open Source Repository." https://github.com/fraygeyst/Contactbook (accessed Dec. 15, 2023).

[77] M. D. Ernst *et al.*, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007, doi: 10.1016/j.scico.2007.01.015.

[78] Y. Xiong, T. Su, J. Wang, J. Sun, G. Pu, and Z. Su, "General and Practical Property-based Testing for Android Apps," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, Oct. 2024, pp. 53–64, doi: 10.1145/3691620.3694986.

[79] E. S. L. Lam, P. Zhang, and B.-Y. E. Chang, "ChimpCheck: property-based randomized test generation for interactive apps," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*, New York, New York, USA, Oct. 2017, pp. 58–77, doi: 10.1145/3133850.3133853.

[80] A. Rosenfeld, O. Kardashov, and O. Zang, "Automation of Android applications functional testing using machine learning activities classification," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, New York, NY, USA, May 2018, pp. 122–132, doi: 10.1145/3197231.3197241.

[81] G. Hu, L. Zhu, and J. Yang, "AppFlow: using machine learning to synthesize robust, reusable UI tests," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, Oct. 2018, pp. 269–282, doi: 10.1145/3236024.3236055.

[82] V. R. Basili, "Software modeling and measurement: the Goal/Question/Metric paradigm.," 1992.

[83] "ISO/IEC 25010:2023 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model." https://www.iso.org/standard/78176.html (accessed Feb. 12, 2026).

[84] R. Coppola and E. Alégroth, "A taxonomy of metrics for GUI-based testing research: A systematic literature review," *Inf. Softw. Technol.*, vol. 152, p. 107062, Dec. 2022, doi: 10.1016/j.infsof.2022.107062.

[85] D. Shin, S. Yoo, and D.-H. Bae, "A Theoretical and Empirical Study of Diversity-Aware Mutation Adequacy Criterion," *IEEE Trans. Software Eng.*, vol. 44, no. 10, pp. 914–931, Oct. 2018, doi: 10.1109/TSE.2017.2732347.

[86] S. Yu *et al.*, "Automated mobile app test script intent generation via image and code understanding.," *arXiv preprint arXiv:2107.05165*, 2021.

[87] B. Jiang, Y. Zhang, W. K. Chan, and Z. Zhang, "A Systematic Study on Factors Impacting GUI Traversal-Based Test Case Generation Techniques for Android Applications," *IEEE Trans. Rel.*, vol. 68, no. 3, pp. 913–926, Sep. 2019, doi: 10.1109/TR.2019.2928459.

[88] "Mileage | F-Droid - Free and Open Source Android App Repository." https://f-droid.org/en/packages/com.evancharlton.mileage/ (accessed Jan. 21, 2024).

[89] "Multi Sms | F-Droid - Free and Open Source Android App Repository." https://f-droid.org/en/packages/com.hectorone.multismssender/ (accessed Jan. 21, 2024).

[90] "NotePad | GitHub - Free and Open Source Repository." https://github.com/Abdallah-Abdelazim/mynotes-app (accessed Jan. 21, 2024).

[91] "ToDoManager | GitHub - Free and Open Source Repository." https://github.com/WestFR/ToDoList-Android-App (accessed Dec. 15, 2023).

[92] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of Android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, Jul. 2020, pp. 153–164, doi: 10.1145/3395363.3397354.

[93] "JaCoCo :Java Code Coverage Library." https://www.eclemma.org/jacoco/index.html (accessed Sep. 30, 2024).

[94] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 26–38, Nov. 2017, doi: 10.1109/MSP.2017.2743240.

[95] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry testing of Android applications by constructing activity launching contexts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, New York, NY, USA, Jun. 2020, pp. 457–468, doi: 10.1145/3377811.3380347.

[96] J. Liu, X. Xiao, L. Xu, L. Dou, and A. Podgurski, "DroidMutator: An effective mutation analysis tool for Android applications," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, New York, NY, USA, Jun. 2020, pp. 77–80, doi: 10.1145/3377812.3382134.

[97] R. Pitts, "Random mutant selection and equivalent mutants revisited," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2022, pp. 170–178, doi: 10.1109/ICSTW55395.2022.00040.

[98] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: an analysis and survey," vol. 112, Elsevier, 2019, pp. 275–378.

[99] W. M. McKeeman, "Differential testing for software.," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[100] M. Linares-Vasquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2017, pp. 613–622, doi: 10.1109/ICSME.2017.47.

[101] M. Brunetto, G. Denaro, L. Mariani, and M. Pezzè, "On introducing automatic test case generation in practice: A success story and lessons learned," *Journal of Systems and Software*, vol. 176, p. 110933, Jun. 2021, doi: 10.1016/j.jss.2021.110933.

[102] K. Pattabiraman and B. Zorn, "Dodom: leveraging DOM invariants for web 2.0 application robustness testing," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, Nov. 2010, pp. 191–200, doi: 10.1109/ISSRE.2010.17.

[103] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017, doi: 10.1109/ACCESS.2017.2685629.

[104] F. Wilcoxon, "Individual Comparisons by Ranking Methods," Biometrics Bulletin, vol. 1, no. 6, pp. 80–83, 1945, doi: 10.2307/3001968.

[105] S. Holm, "A Simple Sequentially Rejective Multiple Test Procedure," Scandinavian Journal of Statistics, vol. 6, no. 2, pp. 65–70, 1979.



Ali Asghar Yarifard received his Ph.D. in Software Engineering from Ferdowsi University of Mashhad, Iran. He is currently an Instructor in the Department of Computer Engineering at the University of Bojnord, Iran. His research interests include Software Quality Assessment and Engineering, Empirical Software Engineering, and Image Processing.



S. Araban has received his Ph.D. in Software Engineering from the University of Melbourne, Australia. He is currently an Assistant Professor in the Department of Computer Engineering at Ferdowsi University of Mashhad, Iran. His research interests include Software Quality Assessment and Engineering, Empirical Software Engineering and Service-Oriented Enterprise Architecture.



Samad Paydar received his Ph.D. in Software Engineering from Ferdowsi University of Mashhad (FUM), Iran. He served as an Assistant Professor at FUM from 2014 to 2022 and is currently a Software Engineer III at Laserfiche in Toronto, Ontario, Canada. His research interests include the Semantic Web and software testing.



Vahid Garousi is a Professor of Software Engineering and AI at Queen's University Belfast (QUB), UK. He received a PhD in software engineering from Carleton University in 2006. His MSc and BSc degrees were from the University of Waterloo in 2003, and Sharif University of Technology in 2000, respectively. He is also a registered professional engineer in Canada and an active IEEE/ACM contributor, his research focuses on Model-driven development, Software Testing and Quality Assurance, and Search-based software engineering.