CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Shrinkdroid: A Property-Based Testing Tool for Android Applications using Dynamic Element Detection and Test Case Minimisation

Master's Thesis in Software Engineering and Technology

Clara Simonsson, Fanny Sjöström

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's Thesis 2026

Shrinkdroid: A Property-Based Testing Tool for Android Applications using Dynamic Element Detection and Test Case Minimisation

Clara Simonsson, Fanny Sjöström

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Shrinkdroid: A Property-Based Testing Tool for Android Applications using Dynamic Element Detection and Test Case Minimisation
CLARA SIMONSSON
FANNY SJÖSTRÖM

Advisor: Oskar Giljegård, CPAC Systems AB
Supervisor: Haroon Elahi, Department of Computer Science and Engineering
Examiner: Christian Berger, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Shrinkdroid: A Property-Based Testing Tool for Android Applications using Dynamic Element Detection and Test Case Minimisation
CLARA SIMONSSON
FANNY SJÖSTRÖM
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents Shrinkdroid, a property-based testing tool for automated GUI testing of Android applications. Existing Android GUI testing approaches typically depend on manually written test scripts tied to fixed UI element identifiers, making them fragile to interface changes and limited in exploration coverage. Shrinkdroid addresses these limitations by dynamically detecting UI elements at runtime, generating event sequences through weighted random and guided exploration strategies, and validating application behaviour against a set of seven generalised GUI properties. When a property violation is detected, an integrated shrinking pipeline automatically reduces the failing event sequence to a smaller counterexample that still reproduces the failure, improving debugging efficiency. Shrinkdroid is evaluated on two industrial Android Automotive applications developed by CPAC Systems AB, and on three open-source mobile applications from the Themis benchmark. Across 184 test runs, explored using both Random Weighted Exploration and Guided Exploration, 26 property violations were detected, of which 17 were validated as real bugs. Ten of these bugs were previously unreported. Shrinkdroid's findings complemented those of the industrial partner's existing testing practices with no overlap in detected bugs, demonstrating its value as a complementary testing tool. Test case minimisation through shrinking was completed in 14 out of 26 cases, achieving sequence reductions in 13 of these, while preserving the original property violation. Graph-based shortest-path reduction proved most efficient, completing in under three minutes for deterministic traces, while greedy reduction of longer sequences could require several hours. These results demonstrate that property-based testing with integrated shrinking can effectively detect and localise GUI bugs in Android applications, even in dynamically changing interfaces, and across different Android environments and device types.

Keywords: Automated Testing, Property-Based Testing, GUI Testing, Android Applications, Test Case Minimisation, Shrinking

Acknowledgements

We would like to express our gratitude to CPAC Systems AB for giving us the opportunity to conduct this thesis project in collaboration with the company and for providing an engaging and supportive environment throughout the study. We would especially like to thank our supervisor, Oskar Giljegård, who initiated the idea behind this project, and for his support and guidance during the project.

We would also like to thank our academic supervisor, Haroon Elahi at Chalmers University of Technology, for his insightful feedback and academic perspective throughout the whole process from the initial proposal to the final stage of this work.

Clara Simonsson, Fanny Sjöström, Gothenburg, June 2026

Contents

List of Acronyms

ADB Android Debug Bridge.
AUT Application Under Test.

BFS Breadth-first-search.

DFS Depth-first-search.
DMF Data-Manipulation functionality.
DMFs Data-Manipulation functionalities.
DSL Domain-Specific Language.
DSRM Design Science Research Methodology.

GE Guided Exploration.
GUI Graphical User Interface.

MBT Model-based Testing.

OS Operating System.

PBT Property-based Testing.

RL Reinforcement Learning.
RWE Random Weighted Exploration.

SUT System Under Test.

UI User Interface.
UTG UI Transition Graph.

List of Figures

List of Tables

List of Algorithms

1

Introduction

Finding bugs in complex software applications is challenging, regardless of whether testing is performed manually or through automated approaches. Such bugs not only introduce security and privacy issues in these applications but also undermine user confidence [1]. Consequently, application testing has gained significant attention from the research community. In applications with a Graphical User Interface (GUI), particularly Android applications, User Interface (UI) testing is still largely a manual process in practice, where the testers manually interact with the entire System Under Test (SUT) or a specific Application Under Test (AUT). However, research is exploring the field of automated GUI testing, where tools simulate or specify interactions with UI elements [2].

While manual approaches give precise control, they limit exploration of the application, as test coverage depends heavily on the developer's prior knowledge of the system, creativity, and available testing time. Manual GUI testing is also error-prone, since the number of possible input combinations is too large for any human to cover completely. Repeating the same tests manually can cause fatigue and loss of focus, making it easy to overlook subtle bugs or edge cases. As a result, test execution and maintenance become a time-consuming and costly manual effort. Even automated GUI tests can produce false positives or ambiguous results that require manual analysis and interpretation [2]. Furthermore, automated GUI tests are often tightly coupled to specific interface components, making them fragile to frequent GUI changes in applications.

These limitations motivate the need for more general and scalable automated GUI testing strategies. In particular, approaches that dynamically detect interface components at runtime and generate tests at a higher level, meaning they are not tied to specific components, while still effectively finding bugs, would facilitate the challenges currently faced. At the same time, reported bugs by the automated tests should be easy to interpret and facilitate manual inspection by developers. By extracting the live state of the interface during test execution, a testing tool can generate interactions that are valid for the current state of the application without requiring manual interactions.

Property-based Testing (PBT) is a high-level, semi-automatic testing technique that has been proposed as an effective approach for testing GUIs of Android applications

[2], [3], [4]. Instead of defining individual test cases, PBT relies on executable properties that the system should satisfy, while automatically generating a large number of random or heuristic test inputs. When a property is violated, the generated failing input is reported. However, generating such large inputs in a non-systematic manner introduces noise and irrelevant values that do not contribute to the failure, making the test cases difficult to understand [5]. Large or complex failure traces reduce debugging efficiency.

To address the debugging complexity arising from large or complex traces, PBT frameworks typically apply test case minimisation techniques, known as shrinking or test case reduction, to minimise failing inputs and produce simpler counterexamples [5], [6]. Shrinking attempts to find a minimal failing case by iteratively removing input steps until further removals cause the test to pass. Applying this technique results in a concise counterexample that highlights only the essential steps of the failure. This facilitates debugging and tracing complex cases by automatically performing the first stage of diagnostics.

Shrinking has been successfully applied in other domains of property-based testing to minimise failing inputs [5], [6]. However, its application to Android GUI testing remains largely unexplored. In this context, shrinking could reduce long and complex interaction sequences to shorter sequences that still reproduce the failure, thereby simplifying bug localisation and debugging. This type of testing can enable a more thorough exploration of the application and the detection of unexpected behaviours, even in dynamically changing GUIs, by focusing on properties rather than the specific behaviours of predefined interactive elements.

1.1 Problem Description

In industrial settings, GUI testing is typically performed either by manually writing tests for UI elements or through manual interaction with the interface to verify the correctness of their intended behaviour and functionality [2]. These tests are commonly implemented using automation frameworks that simulate predefined user tasks and interaction scenarios [7]. Because these tests rely on fixed UI element identifiers or screen positions, any change to the application's interface or the behaviour of its elements requires manual updates to the test suite. As a result, both manual testing and test maintenance are often expensive, difficult to scale, and limited in their ability to explore application behaviour thoroughly [2]. Frequent interface changes also introduce a risk of human error, as developers may misinterpret the new behaviour or introduce common implementation errors by mistake when updating the test suites, leading to incorrect or incomplete tests.

Both manual and scripted approaches have another drawback of not being able to fully explore the application's behaviour. Manually written tests cover only what the developer anticipates, and changed application behaviour has the risk of not being reflected when maintenance is required. In summary, conventional GUI testing in industry is labour-intensive, brittle, and limited in coverage.

1.2 Purpose of the Study

The purpose of the study is to develop an Android GUI testing tool in collaboration with an industrial partner and compare it against the company's existing GUI testing approaches. In doing so, the study aims to assess whether property-based testing can (1) discover bugs more effectively and (2) reduce the manual testing work and produce more interpretable failures. The tool intends to address the problems identified in the previous section. Rather than relying on fixed UI identifiers, this tool aims to automatically detect UI components at runtime and systematically explore applications by generating event sequences. This study intends to integrate automated test case minimisation through shrinking [5], and producing concise failure traces that are easier to analyse. In this way, the proposed tool addresses key limitations of existing Android GUI testing tools, which often depend on static UI references and provide little or no support for minimising failing test cases. As a result, GUI testing can become time-consuming and difficult to maintain, especially when interfaces change, and failures produce long, complex test cases. By combining property-based testing, dynamic UI detection, and automated shrinking, the proposed tool aims to reduce the manual effort required to write, maintain, and debug GUI tests while improving bug detection.

To guide the tool development, the following research questions have been formulated and investigated:

RQ 1: *How do property-based testing tools affect the discovery of bugs in Android applications?*
Investigating the number and types of bugs across different Android applications helps to determine the influence of property-based testing in this environment. Android applications often involve complex event sequences and state-dependent behaviours, which makes fault-detection difficult with manually written tests. Understanding how property-based testing performs in uncovering such bugs provides insight into its practical effectiveness [2], [8], [9].

RQ 2: *How does property-based testing compare to existing testing strategies in the industry?*
By comparing behaviours and results across testing approaches, this study evaluates the impact of property-based testing on established industry practices. This research focuses on the testing approaches followed by our industrial partner, and the comparison helps identify whether property-based testing can complement or outperform commonly used testing strategies.

RQ 3: *How does the application of shrinking methods affect the efficiency and quality of property-based testing suites?*
Shrinking aims to simplify failing test cases, making bugs easier to understand and reproduce [5], [8]. Evaluating its impact clarifies whether these simpler failures improve test quality and comprehension without adding extra overhead compared to standard property-based testing.

Together, these questions aim to provide insights into both the practical applicability of the tool in industrial settings and the benefits of integrating optimisation techniques to improve test effectiveness, reproducibility, and debugging of Android applications.

1.3 Contributions of the Study

This research contributes to both practitioners and researchers of automated GUI testing by demonstrating how property-based testing and automated shrinking can be applied to Android GUI testing. The main contributions include:

- **A PBT Tool for Android GUIs:** The tool uses property definitions for validation of the GUI and heuristic exploration to generate GUI event sequences. It dynamically detects UI components at runtime, avoiding brittle element bindings.
- **Integrated Shrinking for GUI Event Sequences:** The tool systematically reduces failing event traces to minimal reproducible sequences. This generator-guided minimisation ensures that reduced test cases remain valid and are easier to analyse.
- **Industrial Case Study:** An empirical evaluation with an industrial partner is performed, where the tool is applied to real Android applications developed by them, enabling comparison with existing GUI testing practices and analysis of the effectiveness of property-based GUI testing.

The results aim to contribute to further studies of the performance and usability of PBT in GUI testing, filling the gaps in current research around this topic. Additionally, introducing a shrinking methodology for reducing GUI event sequences aims to contribute to GUI tests with improved coverage and more interpretable test results.

2

Background

This section explains the fundamentals of software testing, property-based testing, testing of graphical user interfaces, testing of Android application graphical user interfaces, and test case minimisation.

2.1 Software Testing

Software testing is the process of evaluating that software is performing accurately against specified requirements and behaves as intended [10]. Rather than guaranteeing correctness, testing exposes problems and inconsistencies in the software design, allowing testers to address these gaps that the software seems to have. However, as software testing grows in complexity, ensuring enough testing coverage becomes increasingly difficult. This motivates the need for systematic approaches for design, generation and automation. To understand software testing, it is essential to distinguish between faults, errors, and failures [11]. A *Fault* is a latent defect that occurs only when specific conditions or events are triggered. When a fault is triggered, it may cause an *Error*, which is an incorrect internal state of the software. Errors can arise from incorrect logic or syntax and are illustrated by the difference between the expected and actual results of the program, and as the errors expand, they can induce a *Failure*, which is the observable incorrect behaviour that testing aims to detect. The Fault-Error-Failure chain helps to identify key points where correct actions should be taken to minimise the risk of failures, while also ensuring software quality.

Software verification and validation processes provide foundational context for levels and techniques of testing [12]. Verification asks the question "Are we building the system correctly?", and ensures that the correctness and consistency of the software artefacts are aligned with specified standards without having to execute the software. Validation, on the other hand, asks "Are we building the correct system?", and it determines whether the software meets the requirements and expectations of the stakeholders and users once the software has been developed and delivered. This distinction clarifies the significance of testing practices as verification relies on static inspection while validation requires actual execution against input, making the selection and generation of that input a critical and non-trivial concern.

To realise the value of verification and validation in practice, concrete testing artefacts are required. Software testing is performed through the execution of *test cases*. These are a set of actions, input data, and expected results that together define a specific scenario to be evaluated [13]. To determine whether a test case has passed, *test oracles* are required. Test oracles are mechanisms that judge whether the observed behaviour of the system is correct for a given input, typically derived from a defined specification and requirements set. However, since determining the correct behaviour is non-trivial, it introduces what is known as the *oracle problem* [11], [14]. It is considered a relatively hard problem as it stems from several technical and practical problems. While automation in test input has made significant progress, checking the expected behaviour against the automated test input remains less progressed, which in turn constrains the overall effectiveness of test automation. This arises due to the fact that the expected output of a system depends on its specification, which is often incomplete, ambiguous or unavailable in practice [14]. It leaves the human tester as the main source of oracle information. They are required to manually determine outputs based on the expectations and domain knowledge, which makes the process slow, expensive, and daunting when dealing with large test suites. The oracle problem also faces challenges regarding the complexity of the system behaviour as software evolves. It makes it difficult to define the expected output for all possible states and interactions. Even when specifications exist, they may not foresee all possible inputs as well, leaving significant portions of input space without a defined expected output, which would result in automated testing approaches being too weak to detect crashes or exceptions. This is particularly relevant in GUI testing, as correct visual and interactive behaviour is challenging to state, and failure often manifests as incorrect state transitions rather than outright crashes [15]. The oracle problem, therefore, represents a fundamental bottleneck in automated testing. The primary challenge is not the generation or execution of test cases, but rather the ability to determine whether the observed behaviour of the system under test is correct. While modern tools can automatically produce and execute large numbers of tests, reliably verifying the correctness of their outcomes remains difficult. This limitation motivates the need for alternative approaches to oracle specification, which will be examined in the next section.

2.1.1 Levels of Testing

Software testing can be structured into distinct levels, each addressing different scopes and objectives, enabling the software to be fully covered [11], [16]. At each level, the focus of testing is the *System Under Test* (SUT), meaning the specific part of the software that is being evaluated, from a single component to the entire application. Unit testing validates individual components in isolation, while integration testing examines the interaction between components. System testing, on the other hand, evaluates the overall system behaviour against requirements, and acceptance testing determines whether the software satisfies stakeholder and user needs and is the last phase before deployment. This hierarchical structure ensures coverage across different abstraction levels. In addition to these hierarchical levels, a standard part of the maintenance of the software development is *regression testing*.

It refers to the process of retesting software after changes have been implemented to verify that existing functionality is preserved and that the modifications have not introduced new defects. Regression testing is particularly important in iterative and agile development settings, where frequent updates may induce new faults.

2.1.2 Testing Approaches

The design of test cases can be followed by two fundamentally different approaches depending on the level of knowledge of the system available to the tester [11], [17]. *Black-box testing* evaluates the system based on its external behaviour, without any knowledge about its internal structure or implementation. Test cases are therefore designed from the specification and requirements, making them applicable to all testing levels. It is specifically well-suited to validate functional correctness from the user's perspective, as it tests the expected output behaviour [18]. In contrast, *white-box testing* uses the knowledge of the internal structure of the software, examining flows and execution paths to detect logic-related faults. While white-box testing can provide deeper structural coverage, achieving full path coverage is often infeasible for non-trivial systems, meaning practical white-box testing stays incomplete [17].

2.1.3 Automated Testing

As a software system grows in scale and complexity, manual testing alone becomes insufficient. Since manual testing involves testers interacting directly with the system, observing its behaviour, and determining whether the output satisfies expected results, it offers flexibility and is well-suited to exploratory and usability testing. However, it can become time-consuming, difficult to reproduce consistently and does not scale as the number of possible inputs and execution paths grows [19]. The labour intensity of manual testing also increases the risk of overlooking defects, especially when frequent regression testing is required. Test automation addresses these scalability and repeatability limitations by using scripts, frameworks or tools, minimising the manual effort and allowing tests to be executed repeatedly and consistently. In regression testing, this is particularly useful as it ensures that previously functional functionality remains the same. However, automation alone is not sufficient. The effectiveness of automated testing is constrained by the quality of test scripts, the oracle specification, and the coverage of the input space [14], [20]. These constraints imply that even well-automated test suites may fail to detect failures if the inputs do not adequately exercise the relevant system behaviour, motivating the need for more systematic approaches to input generation and oracle specification.

2.2 Property-Based Testing

Property-based Testing (PBT) [21] is a technique where tests are designed to validate general properties or invariants of a system. These properties or invariants are assertions that are expected to hold for all valid inputs, regardless of the input size or content. Originating from the tool QuickCheck, developed by Claessen and Hughes for the programming language Haskell, it is a form of random semi-automated,

black-box testing. By specifying these properties, the SUT can be evaluated based on the extent to which the properties are satisfied [3], [21]. Since its introduction, it has commonly been used in functional programming, but has become a stable testing technique for testing both functional and non-functional aspects of a software system, with tools for various programming languages such as Python, Java, C#, among others [6], [22], [23]. PBT contains three cornerstones: *properties*, *generators*, and *shrinking*, and will be introduced further below.

2.2.1 Properties

Properties are a high-level, universal rule or assertion that the software must satisfy for a wide range of inputs, rather than just specific input-output that are used in traditional testing [21]. It is a formal specification used as a Boolean function, meaning that if a property returns True for every possible input, the property is considered to hold. Unlike example-based tests that verify specific scenarios, properties describe general behavioural rules that must hold across all valid inputs, making them a more expressive and scalable form of test specification.

Properties in PBT can be categorised into several common patterns. The patterns are mostly tied to algebraic laws, which describe mathematical laws that control how functions behave when executed multiple times. These include commutativity, where the order of arguments does not affect the result ($f(a, b) = f(b, a)$), associativity, where the grouping of operations does not change the result ($f(a, f(b, c)) = f(f(a, b), c)$), and idempotence, where applying an operation multiple times yields the same result as applying it once ($f(f(a)) = f(a)$). Other patterns include round-trip properties, which assert a transformation can be reversed; for example, reversing a list twice should return the original list.

$$\forall xs : reverse(reverse(xs)) = xs$$

Invariant properties assert conditions that must always hold regardless of the sequence of operation, such as the length of a list remaining unchanged after sorting.

$$\forall xs : length(sort(xs)) == length(xs)$$

Lastly, comparison properties relate two implementations of the same function, asserting that they produce equivalent output for all inputs.

One important requirement for properties to be useful is that the right conditions must be clearly defined beforehand. A precondition is a predicate that must hold before a property is evaluated. It constrains the input space to valid inputs for which the property is expected to hold [8]. If a precondition is not satisfied by a generated input, the input is discarded rather than treated as a failure. To define what must be true after the property has been evaluated is done by a postcondition. Together, preconditions and postconditions give properties a clear structure: given that these conditions hold before execution, these conditions must hold after execution.

What also makes properties powerful is that, as executable specifications, they can be used as documentation of expected behaviour. As they define the correctness of

the SUT, they can be seen as test oracles evaluated automatically over input that has been generated [8], [14]. Since the oracle problem refers to the difficulty of determining the correct output, properties can serve as partial test oracles by specifying general behavioural rules rather than relying on manually defined expected outputs.

As stated, properties define behaviour that must always hold, leading to the goal of PBT being to falsify a SUT, which is to find an input that proves that the property was violated. This input, called a counterexample, represents a failed property. When a counterexample is found, it is reported back to the developer with the specification on what property was broken and with what input that triggered the failure. However, since input is generated randomly, the input is often large and complex, making it difficult to interpret and debug [24]. The framework, therefore, attempts to reduce the size of the input, if possible, to find the minimal input that violates the property [21], while preserving the failure. This challenge introduces the third cornerstone of PBT, which will be further examined in section 2.2.3.

2.2.2 Generators

The second cornerstone is the generators. A generator is a mechanism that automatically produces input data from a specific domain for properties to be tested against, and the effectiveness of PBT depends on the quality of its generators [8]. Modern PBT frameworks provide predefined generators for primitive types such as integers, strings, or booleans [6], [21], [23], making it straightforward to test properties over simple data types. However, as properties become more complex and involve more domain-specific types, the need for custom generators becomes necessary. Writing custom generators that satisfy all constraints while maintaining diversity in input is a time-consuming and error-prone task [25]. Generators are closely related to the preconditions of properties in PBT, and since preconditions define which inputs are valid for a property, generators must produce inputs that satisfy those preconditions. If preconditions are too restrictive or sparse, most input will be discarded due to not meeting the specified condition [21], [26]. This makes the alignment between generator design and property preconditions a critical concern in practice, as poorly designed generators may introduce bias or restrict exploration of the input space. It can complicate the detection of defects as it fails to trigger complex program behaviour, which reduces the likelihood of detecting bugs. Well-defined generators, in contrast, are more likely to increase the detection of defects by generating diverse and appropriate inputs, as it broadens the input space [27].

Since generators produce inputs randomly, the counterexample that first triggers a property failure is often large and complex, which shows the arbitrary nature of random generation rather than a minimal input to reproduce the failure. This makes randomly discovered counterexamples difficult to interpret and debug, motivating the third cornerstone of PBT: shrinking.

2.2.3 Shrinking

Shrinking was first introduced in QuickCheck [21] as a method to find a smaller counterexample once a property violation has been found. This is to reduce the time needed to understand the identified bug. Today, this idea is generally known as test-case reduction, or *shrinking*, as it is phrased in frameworks such as Hypothesis [6] and FsCheck [23]. The process works iteratively, with the failing input being systematically reduced by removing or simplifying components. After each reduction, the property is evaluated again to determine whether the failure persists. This continues until no further reduction is possible, resulting in the minimal failing input that still violates the property [6], [21]. By using shrinking, it improves the debugging of PBT by transforming a large, randomly generated counterexample into a human-readable minimal case which shows the root cause of the failure. Built-in generators in modern PBT frameworks typically include default shrinkers for primitive types, while custom generators require a custom shrinking implementation that defines what simpler means for that specific input type. However, shrinking relies on heuristic reduction strategies to simplify failing inputs, and therefore, the resulting counterexample might not be the smallest possible failing input [24].

In section 2.5, test case reduction will be covered more.

2.2.4 Stateful Property-Based Testing

While PBT is effective when testing the output of a function depending on its input, without modifying the state, real-world systems are usually stateful. In such systems, behaviour depends not only on the present input but also on the sequence of previously executed operations and the resulting internal state [28]. For example, a login system behaves differently depending on whether a user has been authenticated, and a shopping cart behaves differently if it has content or not. Testing such a system requires generating and validating sequences of operations rather than specific input. Stateful property-based testing extends traditional property-based testing by generating sequences of commands being executed against the system. QuickCheck has shown that it can be used to test stateful systems by modelling them as state machines, where generated sequences of commands are executed and validated against an abstract model [21], [29], [30]. In this approach, system behaviour is referred to using an abstract model that represents relevant aspects of the system state. It does not necessarily replicate the full internal implementation, but instead captures the expected behaviour of the system from the test's perspective. The model evolves as commands are executed, allowing the expected state to be tracked throughout the execution of a generated test sequence. Here, precondition and postconditions are also central. Each command represents an operation that can be performed on the system, which includes a precondition specifying whether the operation is valid in the current model state. During execution, generated command sequences are applied to both the model and the SUT. After each operation, the postconditions are evaluated based on the model transitions to a new abstract state. The postconditions ensure that the system's behaviour aligns with the expected model behaviour. However, writing stateful properties is a more challenging

task than generalising test cases that test isolated, non-stateful functions [30]. State machines are also hard to work with for non-experts of the system, and most developed machines are left abandoned as they are too troubling to maintain as the system evolves. This showcases one of the main challenges in stateful testing, and in PBT in general, where formulating appropriate properties and maintaining the associated specification can be difficult in practice.

Stateful PBT shares similarity with other automated testing approaches, such as fuzzing and Model-based Testing (MBT) [8], [31]. Fuzzing and PBT resemble each other as they both rely on automated generation of random input to explore a large portion of the input space. However, unlike traditional fuzzing, generated inputs are evaluated against defined properties that describe the expected behaviour. At the same time, PBT integrates elements of specification-based testing, since properties act as a form of specification. When extended with a state machine model, stateful property-based testing becomes more related to model-based testing. Unlike PBT, model-based testing evaluates the system behaviour based on the tests generated from a model. The abstract model can be a state machine, decision table or UML, and the model includes a description of valid states, transitions and outputs. The tools or algorithms that use MBT generate test cases that explore the model, trying to cover the states and transitions of the software [31]. PBT, on the other hand, defines properties that must always hold, and the framework generates random input.

By modelling systems as state machines, and automatically generate sequence of commands, stateful PBT enables systematic exploration of a complex system behaviour that unfolds from sequences of interactions [32]. Stateful property-based testing is particularly relevant for interactive and event-driven systems, such as GUI systems, where correctness depends on sequences of actions rather than individual function calls.

2.3 Testing the Graphical User Interface of Applications

GUI testing of applications checks whether the visible elements behave correctly and consistently, from the user's perspective. This kind of testing has become a standard step in the software development cycle for applications [33]. GUI tests can be performed either manually or automatically using dedicated tools and frameworks. GUI testing encompasses both functional verification (e.g., ensuring that clicking a button produces the expected response) and visual or usability verification (e.g., ensuring consistency and correct layout across the interface) [34].

Manual GUI testing typically follows a predefined test procedure that consists of a sequence of actions performed on the GUI, followed by validation steps to verify the expected behaviour [33]. Additionally, exploratory testing can be used, which emphasises creativity and experience-based approaches when designing, executing, and iteratively refining test sessions [34]. Although manual testing provides testers

with greater control and insight into the testing process, it is costly, time-consuming, and heavily dependent on tester expertise and structured approaches [20], [33], [35], [36], [37]. The process is also error-prone and does not scale well as the number of possible event sequences increases. This becomes particularly problematic when frequent regression testing for recurring event sequences is required for new application versions, where automation is more suitable.

Automated GUI testing has become increasingly popular due to its effectiveness in executing predefined scenarios and testing application behaviours without manual interaction. In automated GUI testing, testers implement scripts that simulate user interactions such as clicks, inputs, swipes, and gestures using automated testing tools [35]. Modern tools and frameworks often work across several operating systems and web browsers, enabling testing of cross-platform functionality and version upgrades, commonly referred to as regression scenarios. The automated tests can be run repeatedly, making them ideal for regression testing. Although the tools automate the actual test execution, they still need to be written manually and maintained to reflect any GUI changes. The automated tests exercise the GUI of an application through predefined interaction sequences and scenarios, making the process more time-efficient compared to manual testing since setup and interaction are handled through the tools. An alternative approach is capture-replay testing, where GUI interaction scenarios are recorded, and corresponding test scripts are automatically generated for repeated execution [37], [38]. Although automated GUI testing is more efficient, it is still time-consuming to define all interaction scenarios and tests, as well as updating any tests due to GUI changes or the removal of obsolete elements [36]. Other challenges in automated GUI testing include changes in the application that break test execution, synchronisation between tests and SUT, as well as robust identification of GUI elements [39].

Beyond execution strategies, there is a focus on automated test case generation for GUI testing. Several methodologies for automated software test case generation have been developed, which is one of the most demanding tasks in software testing and significantly impacts the effectiveness and efficiency of the testing process [20]. Some of the most prominent approaches are: 1) model-based testing, 2) random testing (and adaptive random testing), and 3) search-based testing. These will be further covered in 2.4.1.

While automated GUI testing improves efficiency and supports regression testing, it introduces challenges related to maintenance, robustness, and test case generation. These challenges have motivated extensive research into improved generation techniques and more structured testing approaches, with a majority of the research focusing especially on Android applications.

2.4 Testing of Android Application Graphical User Interfaces

Android is a mobile operating system developed by Google and first released in 2008. Its graphical user interface is structured as a hierarchical arrangement of *Views*, which represent individual UI elements, and *ViewGroups*, which act as containers that organise other views within layouts [40]. Views represent UI components such as text fields, images, buttons, and tables. This hierarchical structure facilitates systematic GUI testing, as UI elements can be programmatically identified and interacted with. Central in the Android UI architecture is the concept of *Activity*, which is a single focused screen with a user interface that the user can interact with [41]. An Android application usually consists of multiple activities, with navigation between them representing the transition state of the application. This architecture design, therefore, decides how user interactions are received and processed with the applications, leading to an interaction model that forms the basis for automated GUI testing.

Android applications are event-driven, meaning that test inputs typically consist of events such as clicks, scrolls, and text entries [15]. Android testing tools exercise the GUI by automatically generating user events and applying various exploration strategies, simulating manual interaction. While some tools focus solely on exercising application-level behaviour, others also generate system-level events such as adjusting volume or modifying system settings, thereby expanding the scope of testing. One approach addressing this challenge is a GUI exploration approach that dynamically acquires information about the UI elements' characteristics [42]. It also considers how runtime action may affect other elements within the interface. This approach aims to simulate the dynamic nature of Android applications GUI more accurately during automated testing. More generally, however, existing GUI testing tools and frameworks typically require manual configuration and ongoing maintenance [43].

As the GUIs of Android applications are constantly evolving, even automated GUI testing approaches remain sensitive to changes and require continuous manual effort to keep test suites up to date. Furthermore, the dynamic nature of Android application GUIs increases the fragility of test scripts, particularly when UI components are modified across versions.

A central component of Android GUI testing is the strategy used to generate input events and explore the application under test (AUT). Since Android applications are event-driven, the effectiveness and robustness of testing largely depend on how user interactions are generated and sequenced. Consequently, it has placed significant emphasis on exploration and input generation techniques for Android applications [39], [44], [45], [46].

While many strategies focus on exploring the application to dynamically discover existing UI components and exercise all their widgets [47], another important aspect

is to consider and properly test runtime changes to widgets [42].

2.4.1 GUI Exploration and Input Generation of Android Applications

Numerous Android testing tools prioritise the exploration of the SUT, and various strategies have been developed for automated test input generation to maximise the exploration coverage of the application [15]. Modern approaches combine static analysis of application code with dynamic GUI exploration to guide input generation more effectively, for example, by identifying reachable activities or event handlers before runtime exploration. Due to the large number of possible event sequences and dynamically changing GUI states, Android applications are subject to a state explosion problem, making exhaustive exploration infeasible in practice. Therefore, different exploration strategies aim to strike a balance among coverage, efficiency, and computational cost. Existing approaches include random input generation, model-based exploration, search-based testing, and machine learning-based techniques such as reinforcement learning [15], [33], [42], [47], [48], [49], [50], [51].

Although random input generation requires little setup effort and is easy to use, it is limited in the type of events it generates and tends to generate invalid and redundant events [15], [52]. For example, different events can test the same element several times by slightly changing the coordinates in the input, but still be within the element coordinate boundaries, where these events are considered different inputs or events.

Model-based exploration approaches build on the idea of generating a GUI model of the application with activities as states and events as transitions between these states [15]. This approach usually encourages exploring new states rather than exercising the same behaviour, but is sometimes limited in defining what constitutes a new stage since it is based on GUI changes and may miss internal state changes which affect the application. Those state changes may lead to different application behaviour, but are not further exercised since they are considered the same state as before, according to the unchanged look of the GUI.

In search-based test input generation, test event sequences are generated and evaluated based on different fitness functions when executed in the application. The event sequences that have the best fitness measures survive and evolve into better ones, which are further tested and evaluated, and this process continues throughout the test session [48], [53]. Although the search-based approach does not need a pre-created model and avoids wasted exploration in random events, it is more computationally expensive than random strategies and requires careful design of the fitness functions.

Reinforcement Learning (RL) generates event inputs based on feedback obtained during runtime testing and dynamically optimises the exploration strategy by prioritising actions that are likely to reach new states of transitions [49], [50], [51]. In this context, the testing process is formulated as a state-action decision problem,

where action selection is typically guided by Q-values, representing the expected reward of performing a specific action in a given state. In tabular RL approaches, individual state-action pairs are evaluated independently, and rewards are assigned based on the novelty of the resulting state compared to the previously explored state [51]. The estimated values are stored in a Q-table, which is updated and prioritised dynamically throughout testing to guide future exploration. Deep RL extends this idea by using neural networks to approximate value functions, enabling generalisation across similar states and actions, rather than updating state-action pairs independently, as in tabular RL [49].

2.5 Test Case Minimisation and Reproducibility

Automated GUI exploration techniques often generate long and complex event sequences to achieve high coverage. However, when such sequences expose failures, they may be difficult to reproduce, analyse, and debug. Many Android input generation tools do not automatically produce concise and reproducible failing test cases [15]. Test case minimisation aims to reduce failing inputs while preserving the failure behaviour, thereby facilitating debugging and improving reproducibility.

2.5.1 Classical Test Case Reduction

Test case reduction, also known as simplification, is an iterative process in which parts of a failing input are systematically removed and re-evaluated to determine whether they are necessary for the failure to persist [54], [55, pp. 117–139]. One important aspect of this is the creation of *bug report*, as it needs to be as specific as possible, while a test case should be as simple as possible in terms of minimising the test input. This can be done by only including a set of single-input entities, where removing any of these would cause the failure to disappear.

To do this, delta debugging, first introduced by Zeller and Hildebrandt [54], is a common algorithm which processes failing inputs until a reduced test case is obtained that still preserves the failure. Delta debugging has been applied, tested and evaluated on typical scenarios since 2002, and lies as a fundamental ground to event sequence simplification in the context of GUI testing of application [56], [57], [58], [59], [60]. Although reduction can be performed manually, automated approaches are strongly preferred, as they remove the repetitive manual effort and can be executed iteratively and efficiently. Although delta debugging has proved effective in reducing the test input in this context, there are still major limitations in execution time, where it can take several hours or even days to reduce a sequence of events [60]. The reason behind this limitation is that the process requires re-execution of the simplified test case input to validate the new behaviour and determine whether a specific set of steps is needed or not. This becomes computationally expensive as a result of the required execution time and is also sensitive to environmental instability and dependent on a correctly configured testing environment.

2.5.2 Event Sequence Reduction in Android GUI Testing

As Android applications often exhibit complex state-dependent behaviour, replaying all possible event subsequences during reduction can be computationally expensive in both resources and time [59], [60]. Consequently, practical reduction strategies benefit from preprocessing steps that filter out clearly irrelevant events or state transitions before replay-based validation [57], [58], [59], [60]. Such preprocessing reduces the number of candidate sequences that must be executed, thereby narrowing the search space while still ensuring correctness through final failure validation.

In property-based testing, similar challenges arise when large, automatically generated inputs expose failures. To facilitate debugging, shrinking mechanisms are employed to systematically derive smaller counterexamples while preserving the observed property violation. In the context of Android GUI testing, shrinking can be viewed as a structured strategy for generating candidate reductions, which are then validated through replay to ensure failure preservation.

3

Related Work

In this section, an overview of the current state of the art will be provided, highlighting how approaches are established and which benefits and limitations have been identified. By analysing prior research across these areas, this review establishes a foundation for evaluating the applicability and impact of property-based testing techniques on the graphical user interfaces of Android applications.

3.1 Property-Based Testing in Practice

Since the introduction of PBT in 2000, the technique has been adopted beyond its original context in Haskell to many languages and domains, such as Hypothesis[61] for Python [6], and FsCheck[62] for .NET [23]. Empirical studies confirm PBT's strengths and weaknesses. Ravi and Coblenz evaluated a set of Python programs using Hypothesis to identify common categories that proved to be more efficient to test using PBT compared to regular unit testing [9]. Their study found that PBT tests (with Hypothesis) detected far more mutants than traditional unit tests, especially tests related to checking exceptions, collection membership, and type invariants. Goldstein et al. recently conducted an industrial case study at Jane Street [8], where the result reported that developers value PBT for testing complex code and increasing confidence, but struggle with writing properties and data generators. In particular, users cited shrinking as crucial yet cumbersome, highlighting the need for improving interfaces for shrinking. They requested more automated and informative shrinkers by incorporating internal test-case reduction guided by the generators. It would mean that the generators are more involved in the shrinking process, where it is possible.

These findings align well with targeted PBT research presented by Löscher and Sagonas, where their target framework was implemented into the tool PropEr [63] as an extension, guiding input generation by search strategies rather than pure randomness [3]. They show how their tool was effective in finding counterexamples in three case studies. In summary, PBT is powerful for diverse applications, but tooling could be improved by tighter integration of generation and shrinking [8]. Our work builds on these insights by applying PBT to GUI event sequences and integrating shrinkers to reduce test cases, addressing gaps identified in prior studies.

3.2 Property-Based Testing for Android Applications

The availability of PBT tools for Android applications is limited and not widely established in practice, with a few developed tools bringing PBT to Android GUIs. For instance, ChimpCheck [4] embeds ScalaCheck into Android's Espresso framework. It provides a high-level Domain-Specific Language (DSL) for generating UI event sequences and lets testers mix scripted and random actions. ChimpCheck can express dense, customised and useful testing patterns but still relies on manual scripts for complex interaction and does not inherently simplify long failing event traces. Sun et al. introduce PBFDroid [64], which is a more recent tool that applies property-based fuzzing to Data-Manipulation functionalities (DMFs) in Android applications. It uses user-defined models of CRUD (create, read, update, delete) operations to generate random event sequences, aiming to find inconsistencies across these operations. PBFDroid find many logical bugs in practice, but its model-based approach requires manual DMF specification and does not attempt to shrink failing sequences. Additionally, the authors also mention that it struggles with properties that involve dynamic or data-driven GUI changes. KEA, proposed by Xiong et al. [2], provides a custom property descriptive language and combines random and guided exploration. KEA proves to work well for identifying functional bugs in Android applications, and has shown in evaluations that it is able to cover 94.5% of known bugs, which outperforms the earlier tools. However, like the others, KEA neither minimises the length of discovered failing event sequences nor automates replaying scripts, making debugging difficult.

3.3 Automated GUI Testing of Android Applications

Considerable research has been conducted on methods for efficiently testing the user interfaces of Android applications [2], [34], [42], [46], [52], [65].

A range of tools and frameworks have been developed specifically for exercising Android applications, such as Monkey [66], Robotium [67], Appium [68], and UIAutomator2 [69]. These tools enable the simulation of user interactions and automate GUI test execution[2], [42], [43], [46], [52], [65]. As an example of one testing tool, Appium enables automated GUI testing and simulation of user interactions in mobile applications across several operating systems, including Android [7]. Like similar tools, it supports code reuse at various testing stages, which can improve test effectiveness and reusability [46].

Examples of established Android GUI input generation tools include Dynodroid[70], Droidbot[71], and Sapienz[48], each employing different strategies for generating test inputs. Dynodroid and several other tools that apply the random exploration strategy have attempted to mitigate inefficient and redundant application exploration, making the tool more efficient by storing transition models and guiding event gen-

eration more intelligently [15], [52], [70]. They also incorporate system events in their generation, which allows for broader coverage and testing of the application. Droidbot constructs a model of the AUT by systematically exploring the GUI and generating event sequences using exploration strategies such as Depth-first-search (DFS) or Breadth-first-search (BFS), allowing developers to control how the application state space is traversed and tested [71]. Sapienz, on the other hand, integrates a multi-objective search-based testing approach to automatically generate test input and explore applications, re-framing it as an optimisation problem [48]. The three optimisation objectives in this tool are: code coverage, sequence length, and number of crashes found. Another example of a search-based tool is STGFA-SMOG, whose objectives additionally focus on quality- and performance-related attributes [53].

Many studies have examined automated GUI testing for Android applications more generally as well. The GUI in an Android system can involve hundreds of interactive elements, so effective testing is crucial to prevent event sequence explosion. In the past decade, a substantial number of tools have been proposed (e.g., random exploration, model-based or search-based approaches), as explained in section 2.4. Many of these tools show improved crash-detection compared to their predecessors [72]. While Su et al. find that many tools detect more crash bugs than their predecessors, their large-scale evaluation simultaneously reveals significant gaps in effectiveness [72]. In their study, they constructed the Themis benchmark of 52 real-world, reproducible Android application crash bugs and ran six state-of-the-art GUI testing tools on them. They found a remarkable gap in the effectiveness of finding bugs; 18 of the 52 real bugs were not detected by any tool. Their analysis identified common challenges for these tools, such as tools struggling with deep use-case scenarios, specific user inputs, required login or system settings, and other complex interactions. All these challenges are preventing many testing tools from finding these failures.

Choudhary et al. concluded, in their comparison of input generation tools, that a lot of tools exist that achieve high application exploration coverage and properly identify failures in the application, while also being easy to use and requiring little setup effort [15]. Conversely, their study highlights further improvements for these kinds of tools, where one major point is the limitations in reproducing identified failures in an easy way. When a failure was identified, no reproducible test cases were generated to later be rerun and reproduce the failure, which is an important feature facilitating debugging. Nie et al. raise an important limitation of existing tools covering test minimisation and prioritisation [33]. They highlight that many tools today have difficulties in reaching a specific point or target in the application, and thus, the test cases can become very large. Research on test case minimisation and streamlining of event sequences remains an open challenge, particularly in the context of dynamically generated GUI tests.

Other systematic studies confirm this verdict. Nass et al. note that, unlike unit testing, automated GUI testing is still not showing convincing results and remains restricted by technical challenges like application changes and synchronisation between tests and SUT [39]. Kong et al. likewise emphasise that Android GUI test-

ing still faces unresolved issues, such as dealing with application updates, growing application complexity, and Android ecosystem fragmentation, that limit the effectiveness [44]. Practitioners report that automated GUI tests are often fragile or non-reproducible, and failure reports can be hard to interpret without full context. While Android GUI testing tools have shown to work successfully, empirical analyses highlight their limitations in fault coverage, reproducibility, and usability [39], [72]. These findings motivate the need for this work, intending to make automated Android GUI tests more thorough (via property-based test generation) and more manageable (by applying shrinking techniques).

3.4 Reduction and Shrinking in Practice

Shrinking is one of the main practical enablers of property-based testing, turning initially large failing test inputs into smaller counterexamples that are easier to reproduce, inspect, and debug. Android GUI testing faces the same fundamental problem, but with inputs expressed as event sequences rather than structured values. Consequently, much of the Android GUI testing literature has studied event sequence reduction, methods used to search for shorter failure-preserving subsequences [57], [58], [59], [60]. These methods often use delta debugging variants and additional domain knowledge about the system to reduce the size of event traces. From the perspective of PBT, these techniques can be seen as shrinking of event traces by incorporating domain knowledge and systematic processes.

At the same time, there is a workflow-level mismatch where event sequence reduction is commonly treated as a post-processing step performed once a failing execution has been obtained, whereas PBT conceptualises shrinking as part of the testing loop. Shrinking is tightly coupled to input generation and guided by a property oracle and systematic re-execution of candidate inputs. This section positions prior Android event sequence reduction work as evidence that minimisation through shrinking is viable for Android GUI failures, and motivates why embedding minimisation into property-based testing for Android GUIs remains a meaningful and still largely unresolved step despite the success of stand-alone reduction tools.

3.4.1 Event Sequence Reduction

As mentioned, many studies on event sequence reduction for GUI testing have been conducted, including studies primarily focusing on Android applications [57], [58], [59], [60]. Yan et al. present an extension of event sequence reduction with delta debugging by introducing their tool CHARD, which acts as a pre-processing model for event sequence reduction. They describe how their tool uses an ineffective-event-directed sequence reduction approach, which evaluates each event in the sequence and checks against a pre-defined set of rules whether they are ineffective and should be removed from the event sequence, thus reducing the overall input size. As a later step, classical delta debugging is applied, which replays the reduced event sequence in its traditional manner and reduces it further if necessary.

Jiang et al. introduce their tool SimplyDroid, which utilises delta debugging but adds a layer by incorporating the hierarchical structures of test cases and applications [58]. They structure the event sequence hierarchically based on user interaction sessions and introduce three hierarchical delta debugging algorithms to reduce the event sequence of the trace representation. The algorithms are introduced as hierarchical Delta Debugging (HDD), Balanced Hierarchical Delta Debugging (BHDD), and Local Hierarchical Delta Debugging (LHDD). Each algorithm applies a different reduction strategy, and LHDD is reported as the most efficient among them.

In addition to event-directed and three hierarchical reduction strategies, state-based approaches have also been proposed. Sui et al. [59] introduce a reduction tool called Echo that analyses differences between GUI states along a failing execution trace. Instead of focusing solely on removing events, the approach identifies state transitions that are irrelevant to reproducing the failure. By comparing GUI states before and after each event, the tool determines whether an event contributes to a state change that is necessary for triggering the failure.

This differential state analysis enables more informed event sequence reduction as events that do not affect the GUI state leading to the failure can be removed. Compared to purely replay-based delta debugging approaches, the technique leverages structural information about events' impact on GUI state transitions, thereby improving event sequence reduction effectiveness for Android applications.

CHARD, SimplyDroid, and ECHO all demonstrate that incorporating domain-specific knowledge improves the efficiency of traditional delta debugging when reducing Android GUI event traces. Collectively, these approaches illustrate three complementary reduction perspectives: event-directed filtering, hierarchical structural decomposition, and state-differential analysis. However, failure-preserving reduction will always require validation through replay of the reduced event sequences, since the only reliable way to confirm that a failure is preserved is to re-execute the sequence.

4

Implementation of Shrinkdroid

This chapter presents the design and implementation of Shrinkdroid. The tool is structured around five components: a test generator, executor, invariant checker, artefact recorder, and test shrinker. Their overall design and architecture, and the component implementation details, are described in the sections that follow.

4.1 System Design and Architecture

Figure 4.1 illustrates the overall data flow of Shrinkdroid. The system consists of a Test Generator (based on Hypothesis RuleBasedStateMachine), a Test Executor (using UIAutomator2), an Invariant Checker (verifying GUI invariants), an Artefact recorder, and a Test Shrinker (minimising failing test cases). Once the device and application are set up, UIAutomator2 detects elements in the current application state, which are then converted to actionable events and executed on a connected Android device via UIAutomator2. During initialisation and after each executed event, the tool captures the screen/UI state and checks specified invariants (assertions about expected behaviour). Upon a failure, the Test Shrinker pre-processes the failing test by removing irrelevant events before re-executing for validation. The shrinking pipeline and its techniques are described in detail in Section 4.4.

The exploration and shrinking are intentionally separated into two phases to allow static pre-processing of the generated event sequences and validation through execution of the processed event sequence.

4.1.1 Tools, Environment and Configurations

Shrinkdroid was implemented in Python 3.12.3 using Pytest 9.0.2, which was already established in the company's testing infrastructure. This ensures its compatibility with the current structure, reducing integration effort and allowing the tool to be adopted without major changes. The tool is also implemented using the Hypothesis Property-based Testing (PBT) library [6], version 6.151.5.

Hypothesis was selected due to its well-established integration among PBT libraries for Python, natural integration with Pytest, and its built-in *RuleBasedStateMachine* support for stateful testing. This class enables custom tuning of the test generator for

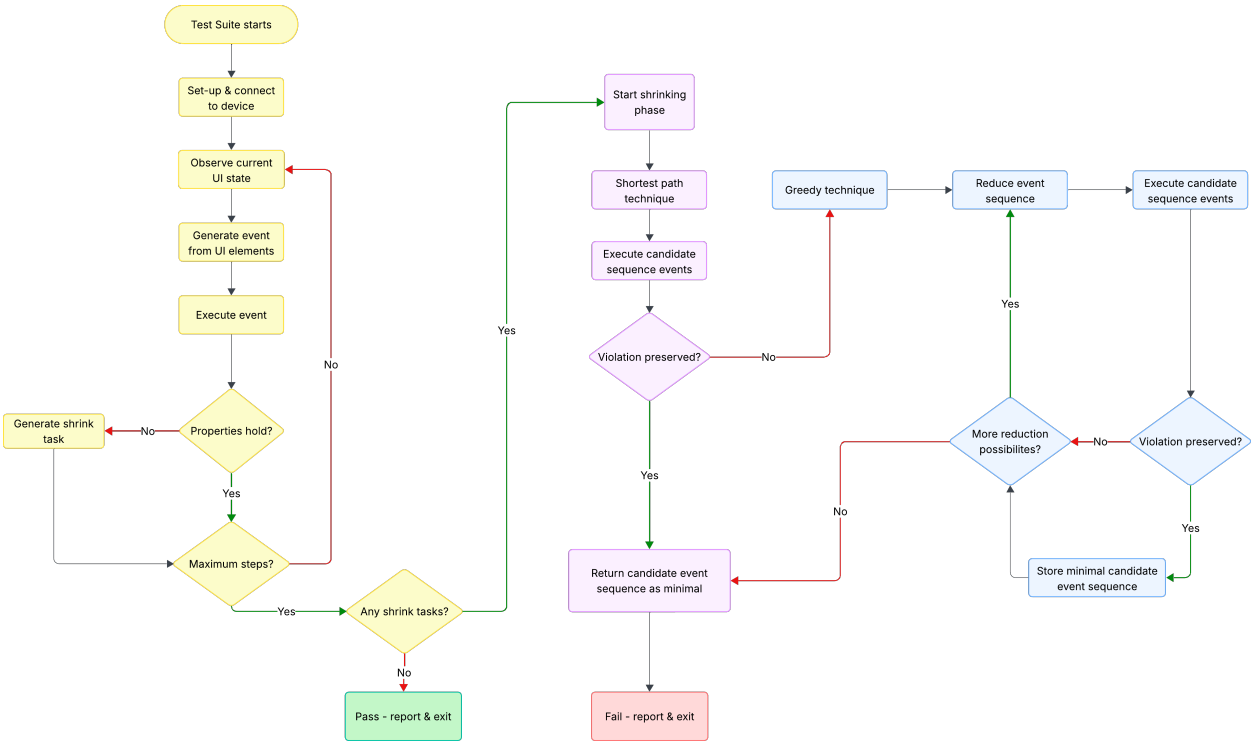

Figure 4.1: Flowchart of Shrinkdroid data flow

the GUI testing. Additionally, a custom shrinking implementation was required due to Hypothesis's lack of such methods for event-driven applications. Since Hypothesis already has the capabilities of shrinking in other settings, it naturally facilitates support and structure for the test case minimisation contribution of this study. For the testing tool to be a stand-alone tool, the implementation was conducted separately from the company's existing implementations and was not developed to target a specific application.

The Python library UIAutomator2, version 3.5.0, was used to observe and control the Android GUI. It is more flexible than legacy tools and allows interactions with the system UI while focusing solely on the Android Operating System (OS), which is suitable for use in the Android Automotive setting for industry and general Android development. UIAutomator2 enables the programmatic simulation of human interactions with the screen, eliminating the need for manual interaction. The tool was also already integrated with the company's infrastructure, which made it a good fit as well.

For the exploration implementation, inspiration was drawn from DroidBot's [71] concept of weighted random crawling of the application's GUI, where states and transitions are stored to generate a model of the AUT and SUT. To generate the model as a graph representation, the Python library NetworkX version 3.6.1 was used. It is a library useful for creating a graph by nodes and edges, with possibilities to include metadata for the components. For Shrinkdroid, this was used to generate a UI Transition Graph (UTG), with screens represented as nodes and actions as

edges, representing the transitions between different screens in the application.

4.2 State Representation and Exploration Mechanisms

This section describes the implementation of the core components used for modelling and exploring Android GUIs. It outlines how screens are represented as structured states, how UI elements are extracted and interpreted from runtime observations, and how these states are connected through a transition graph to model the application and system. Furthermore, it presents the strategies used to explore applications, building the base for bug detection and potential shrinking.

4.2.1 Screen Observation and State Representation

Screen observation is performed by dumping the hierarchy of the UI at runtime using UIAutomator2. The dumped hierarchy returned an XML file representing the elements' nested nodes with semantic, interaction and geometric/display attributes. The dumping is done both before and after an action has been performed to capture how the screen changes. All the retrieved nodes are split into application and system-UI windows, where both are stored respectively, so that the system controls relevant to navigation and recovery are not lost.

The complete observed screen is stored as a `ScreenState` object. It contains the current activity, identified elements, and application package, as shown in Table 4.1.

Table 4.1: List of attributes for a ScreenState.

ScreenState	Type
activity	string
elements	Tuple[“UIElements”, ..]
signature	string
package_name	string

Each of the XML nodes is recursively parsed into a `UIElement` object. A UIElement is a data class which stores each node's attributes retrieved from the XML, along with the parent and child relations to preserve the original hierarchy structure. Table 4.2 shows the attribute variables and their respective type. To reduce the action space, non-actionable elements are grouped and linked to their closest actionable ancestor. This preserves the descriptive and visual information available but excludes non-interactive nodes from action selection.

To enable a reliable screen comparison during exploration and shrinking, each UIElement and ScreenState are assigned a signature in the form of a structural hash. For the UIElement, the signature is based on the stable attributes, such as resource identification and content description, providing a consistent identifier for element tracking across screen observations. The ScreenState hash is based on the screen's

Table 4.2: List of attributes for a UIElement.

UIElement	Type
resource-id	string
text	string
class_name	string
package_name	string
content_desc	string
checkable	bool
checked	bool
clickable	bool
enabled	bool
scrollable	bool
selected	bool
long_clickable	bool
visible_to_user	bool
bounds	tuple
display_id	int
drawing_order	int
children	List[“UIElement”]
parent	Optional[“UIElement”]
signature	string

hierarchical structure of the UI elements. Together, these structural hashes provide distinct identifiers for each UIElement and complete screen states.

As an additional validation and traceability mechanism, screenshots are stored for each observed screen state. The screenshots help reduce false positive matches between visually dissimilar states, such as day and night mode variants, while also providing a visual history of the exploration process that can later be inspected during debugging, replay, shrinking, and analysis.

To further enhance distinguishing states, for example, when screens are semantically identical but differ in dynamic content, a multimodal semantic embedding model can be used to check for similarities. For each captured screenshot, the model generates a semantic embedding vector representing the visual content of the screen. When a new screen state is observed, its embedding vector is compared against previously stored screen embeddings using a similarity metric. States with highly similar embeddings are considered semantically equivalent. Using the embedding model for state matching is optional and disabled by default due to computational and cost overhead, but can be activated through environment configuration. Structural signatures are therefore used as the primary state-matching mechanism when embedding-based comparison is inactive.

4.2.2 UI Exploration

The UI exploration is integrated as part of the test, supported by Shrinkdroid and is dependent on the dynamic UI element detection to navigate the application's interface. To enable the tool to explore the application in longer sessions and detect possible bugs, several exploration strategies have been developed.

All strategies follow the same pattern, with inheritable methods from a common abstract Python base class, enforcing a uniform structure for action selection. This class utilises abstract methods to ensure a consistent structure for every exploration strategy, which can then use the Shrinkdroid instance to determine which actions to execute. The action selection is determined based on the selected strategy, where each has a different implementation to follow its intended strategy. The UML class diagram for the strategy and inherited exploration strategies can be seen in Fig. 4.2.

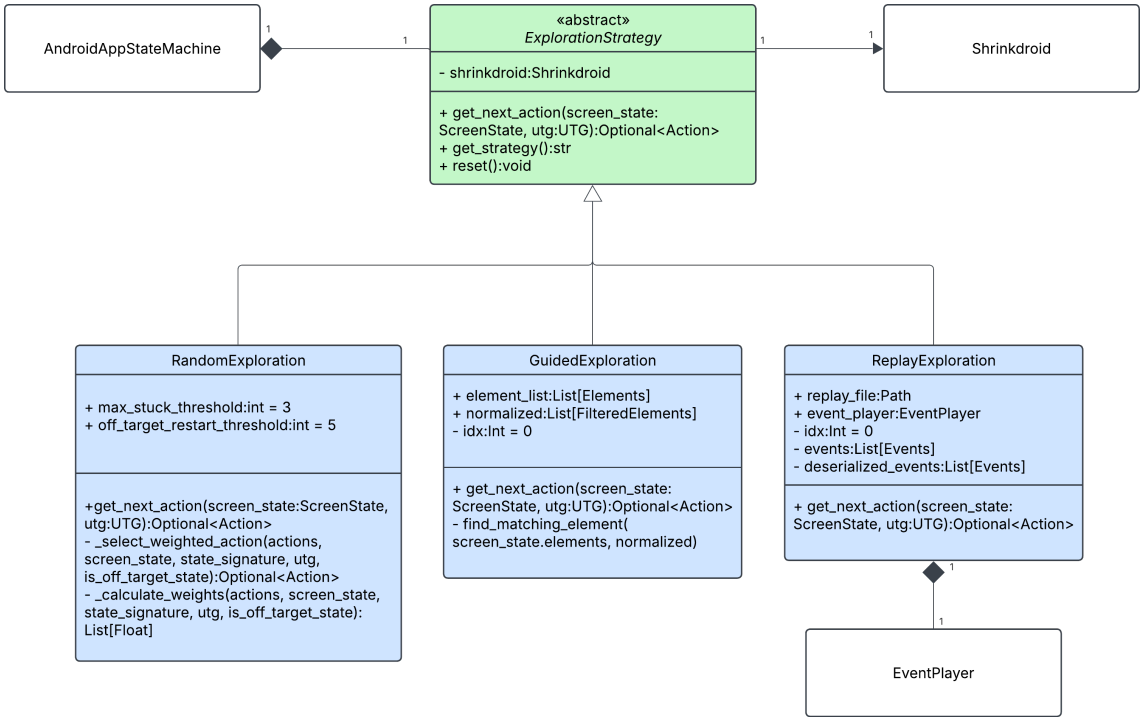

Figure 4.2: UML class diagram for exploration strategies.

All strategies convert identified UI elements for a certain screen state into executable actions by Shrinkdroid based on their attributes. For instance, an element with attribute `clickable: "True"` is converted into a `ClickEvent`, which is added to a list of possible actions to execute. Based on the strategy, one of the actions in the possible action list is selected for execution in the end.

The Guided Exploration (GE) and Replay strategies are very similar in structure, where sequences are fetched from a JSON file. The GE strategy executes a developer-defined sequence of actions, whereas the Replay strategy replays a previously recorded event sequence. The Replay event sequence is processed from the file and deserialised to be executable and rerun by Shrinkdroid. The strategy tracks

which event in the list is in turn, and uses Shrinkdroid to execute them on the device. The Replay strategy is primarily used to automatically reproduce and validate known bug-triggering event sequences.

The GE strategy, on the other hand, executes actions based on element descriptions already defined in a list of UI elements to interact with instead of executing already logged actions. The event sequence is processed from the element list, where specific element attributes are defined to be identified and dynamically mapped to their corresponding observed elements during execution. This strategy also tracks which element in the list is in turn. It uses the observed screen elements and Shrinkdroid to identify the corresponding element and executes the corresponding actions one by one. The purpose of this strategy is to let developers define a set of steps leading to an already known bug or to test a certain flow with very specific interactions. This strategy has acted as a good basis when defining properties and developing the test setup by using already found bugs and verifying them when running the tool and executing the test.

The Random Weighted Exploration (RWE) strategy differs from the GE and Replay strategies, as it is primarily intended for exploratory test execution to detect previously unknown bugs and simulate real user interaction with the SUT. The strategy tracks visited screen states, avoids repeated exploration cycles, computes per-action weights, and selects actions using weighted random sampling. Features such as repeated state-action tracking, local cycle detection, off-target application handling, and learned "back-intent" behaviour are implemented to better simulate realistic navigation patterns. Off-target situations refer to cases where the exploration leaves the AUT package, for example, by entering system UI, settings, or launcher applications. The strategy computes weights for each interactive UI element and samples actions based on the resulting probability distribution.

The flow for the `get_next_action()` method for the RWE strategy is implemented as follows:

1. Validate state and finalise learning from previously executed actions.
2. Update off-target status and revisit state tracking to detect stuck application behaviour.
3. Get candidates from the action generator with executable actions on observed elements.
4. Either run explicit off-target recovery routing or weighted selection.
5. Record state-action history and cycle suppression context.

The implementation can be studied in more detail by looking at pseudocode for the algorithm in Alg. 1, describing how the RWE strategy implements `get_next_action()`.

Algorithm 1: Random Exploration with Weighted Selection

Data: Screen state s, UTG, target package P
Result: Selected action a^*

$actions \leftarrow$ get_available_actions(s);
if $actions = \emptyset$ **then**
  return `None`
end
`// Recovery mode: return to target app if off-target too long`
if *consecutive_off_target* $\geq$ *threshold* **then**
  return SelectRecoveryAction(actions, s);
end
`// Calculate weight for each candidate action`
foreach $a_i \in actions$ **do**
  $w_i \leftarrow 1.0$;
  $w_i \leftarrow w_i \times f_{\text{novelty}}(a_i)$; `/* prefer untried; decay repeated */`
  $w_i \leftarrow w_i \times f_{\text{package}}(a_i, P)$; `/* boost in-app elements */`
  $w_i \leftarrow w_i \times f_{\text{type}}(a_i)$; `/* click > long-click > scroll > ... */`
  $w_i \leftarrow w_i \times f_{\text{e\_signal}}(a_i)$; `/* boost element signals including text or resource-id attrs. */`
  $w_i \leftarrow w_i \times f_{\text{UTG}}(a_i, s)$; `/* penalize self-loops */`
  $w_i \leftarrow w_i \times f_{\text{escape}}(a_i)$; `/* boost back-intent when stuck */`
  $w_i \leftarrow w_i \times f_{\text{suppress}}(a_i)$; `/* suppress cyclic actions */`
  $w_i \leftarrow w_i \times \text{Uniform}(0.5, 1.5)$; `/* random jitter */`
end
$a^* \leftarrow$ WeightedRandomChoice(*actions*, *weights*);
return a^*;

The weight model for each candidate action starts at 1.0 and is multiplied depending on eight different key factors:

$$w = \prod_i f_i$$

where,

$$f_i$$

represent each factor up to i. To avoid zero-probability action selection, the resulting weight is lower-bounded by:

$$10^{-5}$$

The final weight is selected by the following formula:

$$w \leftarrow \max(w, 10^{-5})$$

The eight key factors are the following:

1. **Untried preference**: Giving higher weight to unexecuted state-action pairs.
2. **Target package bias**: In-target actions are boosted and off-target slightly penalised.

3. **Action type priors**: Click events are highly favoured, followed by long-click, scroll, drag, and lastly double-click events, with weights decreased in the same order.

4. **Element signal bonus**: Actions including elements that have a resource-id or text get extra multipliers.

5. **UTG effectiveness factor**: Pulls per-state and per-action statistics from the UTG and computes the effectiveness by looking at self-loop rate and unique target transitions, both divided by the number of attempts.

6. **Escape-route boost when stuck**: If the exploration keeps returning to the same screen state or an action self-loops repeatedly, back-intent actions are boosted via calculated heuristic back scores mapped to defined back-intended elements (e.g., elements with *cancel* or *back* as text attributes) and learned back action behaviour (e.g., previously executed actions that lead to new states).

7. **Short-term cycle suppression**: Recently cyclic state-action pairs get a small penalty for a few steps, where actions that do not have this penalty get a small weight multiplier instead.

8. **Random jitter**: The final weight is multiplied by,

$$\sim \mathcal{U}(0.5,\, 1.5)$$

   to keep the exploration runs diverse.

It is the product of these factors that gives the action's weight, where one of the actions with the highest weights gets selected for execution.

4.2.3 UI Transition Graph

To represent the AUT and SUT, the UI Transition Graph (UTG) models discovered ScreenStates and transitions between them as a directed graph. The UTG acts as a persistent runtime model used during exploration to track visited states, executed actions, and discovered transitions, and later during analysis and shrinking.

The following formula provides a formal representation of the UTG:

$$UTG = (S, A, T)$$

where:

- S = set of ScreenStates
- A = set of executable actions
- $T \subseteq S \times A \times S$ = set of transitions

ScreenStates are stored as nodes in the graph together with metadata such as state signatures, screenshot paths, and connected observations. Multiple observations may be associated with the same ScreenState to preserve screenshots and metadata for semantically equivalent states that differ only in dynamic or visual content. Transitions between different ScreenStates are stored as directed edges, where each edge represents a transition caused by an executed action. Edge metadata includes source and target state signatures, action signatures, transition counts, executed action types, and associated UIElements.

The UTG additionally stores global graph metadata, including the first and last observed state signatures, total number of nodes and edges, and total transition counts. An example of how nodes and edges are added to the UTG after each new screen observation can be seen in Alg. 2. The stored transition statistics are also used by the RWE strategy to guide action selection and avoid ineffective exploration paths.

4.3 Test setup with PBT

Hypothesis's RuleBasedStateMachine was used as the execution framework for coordinating stateful exploration and invariant evaluation. GUI interactions were modelled as rule invocations, while properties were implemented as invariants evaluated after each executed step. The number of execution steps and test runs was defined as part of Hypothesis general test setup, and could be changed depending on the desired configurations for the test execution. The complete setup of the property-based testing follows a defined execution sequence, described as:

1. Initialisation: Set up the application, test environment using fixtures, and Shrinkdroid.

2. Action Step: Execute one UI action per rule invocation via the chosen strategy.

3. Invariant Check: After each executed action, verify that all defined properties still hold.

4. Tear-down: Clean up after a test run completes.

The algorithm for the PBT and shrinking setup can be found in Alg. 3, giving a high-level overview of the testing flow and each execution sequence step in the process.

Algorithm 2: UI Transition Graph (UTG) Construction

Data: Sequence of observations and actions during exploration
Result: Directed graph $G = (V, E)$ serialized as `utg.json`

$G \leftarrow$ empty DiGraph;
$s_{\text{first}} \leftarrow$ `None`;
`// Called after each new screen observation`
Function `AddObservation`(*screen_state s, action a, previous state* s_p`)`**:**
  $sig \leftarrow$ signature(s) ; `/* hash of activity + element tree */`
  if $sig \notin V$ **then**
    add node *sig* with metadata: activity, package, screenshot path;
    if $s_{first} =$ ***None*** **then**
      $s_{\text{first}} \leftarrow s$
    end
  end
  if $a \neq$ ***None*** ***and*** $s_p \neq$ ***None*** **then**
    $sig_p \leftarrow$ signature(s_p);
    action_sig $\leftarrow$ signature(a);
    add or update edge (sig_p, *sig*) with *action_sig*, count;
    if $sig_p = sig$ **then**
      mark *action_sig* as ineffective (self-loop);
    else
      mark *action_sig* as effective;
    end
  end
`// Serialized structure of utg.json`
`utg.json` $\leftarrow$ {
  `meta`: {num_nodes, num_edges, first_state, last_state, activities, packages},
  `nodes`: [{id (signature), activity, package, observations, screen_paths}],
  `edges`: [{source, target, action_signature, action (serialized), count}]
};

Algorithm 3: Full Shrinkdroid Test Flow with PBT and Shrinking

Data: App under test (APK path), optional replay file, invariant set $\mathcal{I}$
Result: Detected violations with minimal reproducing sequences

config ← pytest fixtures configure app path, replay mode, optional app id;
for *each Hypothesis example* $i = 1, \ldots,$ *max_examples* **do**
  sm ← AndroidAppStateMachine() **while** *step_index* < *stateful_step_count* ***and*** ¬*abort* **do**
    ExecuteStepFromStrategy(*sm*) **foreach** *invariant* $\mathcal{I}_j \in \mathcal{I}$ **do**
      if $\mathcal{I}_j$ *is violated* **then**
        EmitDeferredShrinkTask(*sm*, $\mathcal{I}_j$);
        abort_current_example ← **true**;
      end
    end
  end
  sm.teardown()
end
RunShrinkTasks(*execution_session*) **if** |*detected_violations*| > 0 **then**
  fail test session with violated property summary;
end

Initialisation is conducted in two steps, using Pytest fixtures and the state machine constructor. Pytest fixtures handle input parameters (application path, replay file path) to prepare the testing session. The state machine constructor initialises objects like the Shrinkdroid runner, the selected exploration strategy, and the invariant-checker instances.

For the *action step*, Shrinkdroid is the event generator, where events are generated and selected according to the used exploration strategy and executed through UIAutomator2.

A logging and replay system was created to record and document each step of exploration and shrinking. During exploration, event streams and observations are recorded to thoroughly track what happens in the test session, which is later used during shrinking and manual inspection. The logger captures:

- Action/event metadata, such as which action was executed and which UI element it interacted with.
- Screen observations, including screenshots, extraction of interacted UI element and an overlay off the interacted UI element on the screenshot.
- UTG artefacts, including nodes, edges and metadata for both of them.

Replay mode processes a previously executed event file and replays the event stream following the recorded instructions.

The recorded artefacts are created to facilitate visualisation and help developers

understand what happens during test executions. The UTG is used as a base to automatically generate both a sequence diagram to visually present the order of the executed events and the resulting screen states, as well as a visual graph representation of the screen states and transitions among them (i.e. executed actions).

The *invariant check* is performed after each rule step and delegates to a property-checking implementation in the Invariant object. Upon violation, the current run schedules deferred shrink task creation. Instead of immediately stopping the whole test execution, it interrupts the current exploration session to mark that a violation has happened. This setup allows for early termination of failing runs when properties have been violated, but gives the ability to continue exploration from a fresh start and for a new test run.

The property set is defined using a combination of sources, including PBT literature, fundamental UI correctness assumptions, discussions with developers and testers at the industrial partner, and manual exploration of the target applications. These inputs are used to define properties that act as execution oracles for detecting incorrect behaviour.

Properties were refined through iterative discussions with the developers and testers who could provide insights about known bugs, expected behaviour, and common failures of the system.

During execution, each property violation is recorded together with information such as the violated property, execution steps, and how often the violation can be reproduced. Non-deterministic and non-reproducible violations are discarded and classified as flaky or invalid bugs.

Whenever a property violation is detected, a corresponding bug report is created in addition to the standard execution artefacts stored during testing. The generated report included information about the violated property, assertion or error message, timestamp, execution context, and relevant logcat information when applicable. The purpose of these reports was to provide the developer with details about the identified bug, along with relevant information for further analysis, for simpler debugging.

To test the expected behaviour of the system, multiple properties have been designed with a focus on generalisation to capture common application behaviour. The resulting properties are presented in Table 4.3.

Table 4.3: Description of implemented properties.

Property ID	Summary	Description	Dependencies
P1	Application should not crash	Incrementally scans logcat for application crash sections and reports detected fatal exceptions	Logcat file plus per-invariant byte-offset cursor for incremental reads
P2	Application should not throw NullReference-Exception	Scans logcat incrementally for NullReferenceException lines, filters to application-related lines (by package match or PID), captures short stack context, then raises and writes a bug report	Logcat file, incremental per-invariant byte-offset cursor, target package name and PID
P3	Toggle actions must follow expected checked-state behaviour	After a click, long-click or double-click action on a toggle/switch/radiobutton, it verifies that the `checked` attribute changed as expected. If confirmation is required, the confirmation-switch flow must commit only the visual state change after a proper Yes/No flow. Raised on invalid transition	Last transition context (before state, after state, action), action type, element attributes and signatures in before/after states, and confirmation data
P4	Child elements must stay within parent bounds	For visible elements in the current screen, ensure each child element `bounds` fit within its visible parent (or within screen for top-level nodes), reports child and parent elements for any violation in current screen	Current screen state with element hierarchy, device window dimensions
P5	Target application screen should be preserved after external detour	Tracks screen signature before leaving the application, and verifies signature remains on return, with skip rules for camera/day-night mode transitions	Transition context, screen signatures, camera state getter, day-mode getter, target package/activity detection

Property ID	Summary	Description	Dependencies
P6	Disabled elements must not change state	If the last action targeted an element with the attribute `enabled=False` before interaction, the screen signature must remain unchanged after the action and otherwise a violation is raised	Last transition context (before state, after state, action), targeted element data, and screen state signatures
P7	Selected element attribute implies enabled	No visible element should have `selected=True` and `enabled=False` simultaneously, as this is a contradictory UI state and is reported as a violation	Current screen state with its composed UI elements and their attributes

The test setup has both a *per-example tear-down* and an *end-of-session tear-down* to separate cleaning between exploration runs and final test session procedures. For the per-example tear-down, the runtime processes are closed, such as disconnecting the device, stopping the logger and saving the artefacts. The end-of-session tear-down is slightly different since that is where the potential shrinking phase begins if there are any queued shrink tasks from the example runs. The overall test session verdict is determined after the shrinking has completed, and is considered a failure if deferred violations existed and were correctly verified during shrinking.

4.4 Shrinking workflow

The shrinking phase is separated from the exploration to avoid interference with Hypothesis's built-in shrinking techniques and to manage the cost of replaying sequences. Testing of Android applications is a dynamic process with several dependencies between states, and re-execution of event sequences is very time-consuming and inefficient for validating violations.

When a violation is detected, a deferred shrink task containing exploration artefacts and metadata is created. These tasks are picked up and executed after exploration and always consist of three steps: 1) a reproducibility check, 2) minimisation, and 3) post-validation.

The *reproducibility pre-check* is a step to verify that the reported violation can be reproduced, to avoid spending time on shrinking event sequences and validation that are one-time occurrences.

The *minimisation* is the step where shrinking is happening. This is an iterative process where candidate event sequences are derived from the original failing event

sequence, and executed with Shrinkdroid to validate if the violation is preserved or the test passes. The validation step is important to verify that the removed events are irrelevant to triggering the failure and identify ineffective events. Before running the validation step for each candidate, the event sequence is statically validated based on the UTG to verify if it is a valid path. If not, that candidate is removed, and the iteration moves on. If it is valid, the sequence is executed with Shrinkdroid. Three different shrinking techniques are implemented and follow a prioritisation order based on computational cost in time. The minimisation algorithms are the following:

- **UTG Shortest Path reduction**: This uses the UTG to build the shortest navigation path from the initial screen state to the target state (last screen state when a violation was detected) and tests whether that path still violates the property. If it does, that path is minimal with respect to that UTG application path model. If it does not, the shrinker tries alternative short paths (k-shortest) and then falls back to other reducers.

- **K-Shortest Path reduction**: This algorithm also uses the UTG to generate the first, second, third, up to k shortest valid paths from the initial state to the target state. Each valid path is tested one at a time, and the first one that still reproduces the same violation is kept.

- **Greedy reduction**: Greedy removes one event at a time (and sometimes adjacent pairs), reruns the candidate sequence, and keeps the removal only if the same violation still occurs.

First, the UTG shortest path reduction is used to produce a candidate event sequence since preserving the violation while executing that event sequence provides a strong minimality guarantee within the UTG model. If the violation is reproduced, no shorter path in that graph representation can reach the same failure, and the result is returned immediately. If not, the k-shortest path reducer is invoked, which enumerates up to $k = 10$ alternative simple paths from the initial state to the target state using Yen's algorithm. Each path constitutes concrete event sequences by selecting edge actions ranked by execution frequency, and self-loop actions observed at intermediate states are included as additional variants since they may represent real visual transitions that were merged into a single state node by the screen signature function. Candidates are tested sequentially, and the first one that reproduces the violation is accepted.

If none of the graph-derived paths reproduces the violation, the shrinker falls back to greedy reduction, generating candidate event sequences based on the original recorded event sequence. Greedy reduction iterates over events from front to back, removes one event at a time, statically validates the candidate to ensure it is a valid path in the UTG, replays the shortened candidate on the device, and keeps the removal only if the violation is still triggered. Once no single removal succeeds, it attempts to remove adjacent pairs of events for a second reduction pass. This two-phase approach can eliminate event pairs that are individually necessary but jointly

redundant. After every pair removal round, another single-event pass is executed to check for any new reduction opportunities opened by the pair removal.

If a candidate event sequence fails to reproduce the violation, the attempt is discarded after re-execution, ensuring that only valid shrinking attempts are kept. If the shortest-path algorithm fails, several attempts are tested for execution, since there is no guarantee that the first valid attempt is minimal.

Lastly, independent of whether a smaller counterexample is found or not, a *post-check validation* is performed to test the violation for flakiness. It is set up to run the final event sequence ten times in total, where seven of these runs must reproduce the violation to be considered a stable identified bug.

The whole shrinking process is documented the same way as in the exploration phase, with corresponding artefacts to the different screen states and executed event sequences. Additional information about the number of shrinking attempts, flakiness statistics, initial and minimal event sequence lengths, and the final list of minimal events and their elements is stored for manual inspection.

5

Methods

This chapter presents the methodological approach for developing the Android GUI testing tool Shrinkdroid and evaluating it. The work follows a Design Science Research Methodology (DSRM) that seeks to increase knowledge by combining creation of innovative artefacts [73], [74].

For the implementation of Shrinkdroid, a series of steps has been followed, including iterations and reworking across different parts of the process. These steps include: analysing existing GUI testing strategies and conducting a literature review, designing a property-based testing tool, implementing it in Python, and evaluating it by conducting experiments on real applications used in the industry. Iterations are done across the design, implementation, and evaluation steps, which closely align with the design science research methodology.

5.1 Design Science Research Methodology

DSRM is a research methodology in which artefacts are created and evaluated to address identified real-world problems. Peffers et al. introduced the methodology and present six steps included in the process [73]. They are described as follows:

1. *Activity 1: Problem identification and motivation.* Identify a practical problem and justify the value of a solution. It is achieved by evaluating the existing field and performing a literature review to determine the need for a solution.

2. *Activity 2: Define the objectives for a solution.* Define the objectives and requirements for the solution to the problem. The objectives can be derived from the problem definition and be measurable, either quantitatively or qualitatively.

3. *Activity 3: Design and development.* Design the artefact (architecture, algorithms, interface), determine the artefact's desired functionality, and create it.

4. *Activity 4: Demonstration.* Show the artefact's use in its intended environment to solve one or more instances of the problem.

5. *Activity 5: Evaluation.* Assess the effectiveness of the artefact in supporting a solution to the identified problem.

6. *Activity 6: Communication.* All aspects of the problem and the designed artefact are communicated to the intended audience.

An iterative approach in this process means that, if evaluations show deficiencies, the design and implementation phases can be repeated to improve the final artefact. For example, requirements can be adjusted or algorithms optimised based on the evaluation result.

Every activity in the DSRM corresponds to concrete activities in this study, where Iteration I and II iterated over activities 3 and 4, while Iteration III iterated over activities 3, 4, and 5.

Activity 1 in this work included investigating and mapping existing problems and challenges with Android GUI testing, motivating the need for the new tool Shrinkdroid. This analysis has served as the basis for formulating relevant research questions to mitigate the identified problems. *Activity 2* included converting the findings from the problem identification into measurable objectives, leading to concrete research questions to answer in this work. The objectives of this study are to design and implement the tool Shrinkdroid to automatically generate event sequences during runtime, define properties to catch failures or errors in Android applications and systems, and minimise failing event sequences. The research questions are formulated in a way to make it possible to measure the effectiveness of the tool in contrast to existing PBT tools as well as current testing procedures in the industry. *Activity 3* initially included designing and developing the architecture of the tool and test setup, implementing it in Python. Main deliverables in this stage were the architecture diagram (for the components in Shrinkdroid), the UML class diagram for the exploration strategies, and a flowchart for the test setup where Shrinkdroid and PBT were combined. These were delivered along with the implemented code and required libraries.

In *Activity 4*, Shrinkdroid was used in its intended testing environment together with PBT and tested on an Android device and an industry application in an initial testing phase. Demonstration can be shown by the use of three different exploration strategies, where it can be steered by a scenario (GE), a previously executed event sequence (Replay), or randomly selected interactions (RWE). Data is collected by saving logs and graphics (i.e., generated sequence diagrams and UTG graphs).

For *Activity 5* in the process, the performance and effectiveness of the tool together with the PBT setup were measured by evaluating the results from testing sessions. The test result was evaluated in two aspects, depending on the used strategy. For the GE in the first iteration, the effectiveness was measured by evaluating how well the testing session worked in finding previously known bugs when using PBT, dynamic element detection, and defining a property. For the RWE in the second iteration, the performance was internally evaluated by how the application and system were

explored, looking at how many new states were discovered and how well they were discovered within the target application, in contrast to executing system events. In later iterations, the evaluation for the GE strategy also included measuring the effectiveness of applying shrinking compared to initial event sequences for already known bugs. The RWE was evaluated based on the number and types of identified bugs.

Activity 6 covered documentation of the developed tool collected in this report. Besides the text, UML diagrams, flow charts, tables and pseudocode are also included as part of the documentation of the tool and PBT setup.

Iterating over *activity 3, 4, and 5* is part of the DSRM process until the artefact is generated in a way to fulfil the findings in *activity 1 and 2*. Different evaluation types are considered depending on the iteration stage, where the first iteration initially focused on what Sonnenberg and vom Brocke [75] refer to as *ex ante* evaluation. The first evaluation is meant for evaluating the design of the solution by looking at aspects of clarity, simplicity, and consistency. This is tightly connected to the architecture, models and classes in the tool and test setup and how they function together. In the first iteration, the latter part focused on what they refer to as *ex post* evaluation. This included evaluating the solution instantiation, like ease of use, fidelity with real-world phenomena, and robustness. The second and third iterations are also *ex post* evaluations, while the second focused on the solution instantiation and the third evaluated the solution in use. To evaluate the solution in use, the criteria include evaluating the effectiveness, efficiency, and external consistency.

The work could be split into three iterations of development. Iteration I covers the development of Shrinkdroid and testing on a known bug using GE. Iteration II covers moving the Shrinkdroid exploration from completely guided to enable a random strategy. Iteration III involves implementation of shrinking, conducting several properties for testing, as well as fine-tuning the RWE, and proper bug reporting. After each iteration, the results were evaluated and taken into consideration when going forward in the development.

5.2 Iteration I: Initial Design and Development of Shrinkdroid

In this iteration, the focus lay on establishing a baseline prototype with a guided execution of a predefined event sequence. This enabled the basic infrastructure to be validated against a bug in the application that was already known, before introducing the more autonomous exploration. The goal was to validate the integration between PBT and Android GUI exploration in a controlled environment, catching a violation of a property by guiding the test through the application. The evaluation results showed that while basic failure detection was possible, the approach was considerably sensitive to changes in the user interface, which would be harmful to the event sequence if the UI evolved.

5.3 Iteration II: Extending Exploration Capabilities and Introducing Graph Modelling

In iteration II, the autonomous exploration was introduced to reduce the need for manually defining event sequences. The exploration was extended to support a state-aware exploration, where navigation through the application was less constrained, and explored other parts of the application that had not yet been covered in the guided exploration. To do so, a number of weighted factors could determine what action to take in the application, steering it to discover more unseen states rather than already-seen states. The weights were determined through trial-and-error, where the result showed whether the exploration was a success or not. To support this, the UI transition graph was developed, and explored screens and transitions were represented as a graph structure where states correspond to nodes and user interactions correspond to edges. This made it possible to track which parts of the application had been visited and to guide exploration towards unseen states. The strategy was run on the same application as the GE, and was evaluated based on the sequence it had taken. While this extended weighted exploration improved the exploration coverage, the evaluation also revealed challenges regarding accurately representing explored states such that the tool chooses to explore states it has not yet seen, which later motivated improvements in state representation and shrinking.

5.4 Iteration III: Extending Property Set, Implementing Shrinking, and Evaluation

In this last iteration, the aim was to improve the reliability of the exploration and interpretability of the shrinking result. To further improve coverage of common Android UI failure types, the property set was extended to be able to capture bugs of different types and categories. To be able to minimise the sequences leading to a property violation, shrinking was implemented as well. Regular shrinking on primitive data works efficiently, but in GUI testing with event sequences of an extensive set of steps, the number of subsequences that still violate a property becomes computationally expensive. Therefore, a customised shrinking technique had to be formed.

Based on the UTG developed in iteration II, the failing event sequences could be better interpreted to detect similar states and loops, instead of only on raw event sequences. This allows shrinking to prioritise meaningful paths through the application, where the shortest valid path is considered the strongest candidate for reproducing a failure. Visual information from screenshots was also included in the state representation, since UI structure alone was insufficient in cases where screens appeared identical but differed visually. The purpose of the visual representation was not only to detect differences and similarities between states, but they were also used to make detected failures clearer and more useful in an industrial context.

Lastly, this iteration resulted in a testing session period, where both the industrial

applications and mobile applications were tested, and during that same time, bugs were extracted from the company's internal bug reporting system, all resulting in metrics that are being presented in the chapter below.

Design decisions were driven by evaluation outcomes during all iterations. This follows the workflow of the DSRM, where the development of the artefact is guided by continuous feedback between the behaviour of the system and the research objectives.

5.5 Experimental Setup and Evaluation Metrics

The experimental evaluation consists of two parts: an autonomous exploration part, used to evaluate bug discovery and PBT efficiency (RQ1) and a shrinking part, used to evaluate the effectiveness of event sequence minimisation (RQ3). By separating this, it enables the evaluation of shrinking independently of bug discovery. To be able to evaluate how well it performs compared to the company's existing procedures, a comparable part is included as well (RQ2).

Shrinkdroid is primarily evaluated on the application DigAssist, but is also evaluated on the application LoadAssist and the overall system setting. DigAssist and LoadAssist are used in construction machinery running on the Hydra platform. Both applications and platforms were developed by CPAC Systems. Hydra is an all-in-one electronic control unit (ECU) designed for automotive, marine, and industrial applications, providing computing capabilities, connectivity, and integration with various vehicle subsystems. The device, running on an Android OS, is integrated within the Hydra and uses Android Automotive version 12.1. Its hardware specifications contain a CPU Qualcomm Snapdragon SA8155P Android Processor, with RAM 6.0 GiB, and storage of 5.9 GB, with a screen resolution of 1080 x 1920 pixels. A test rig is set up for the Hydra, connected to a screen via USB and a laptop for test executions. Each of the applications has a separate designated test rig.

A Samsung Galaxy S8 smartphone running Android version 9 is used as a secondary device. To evaluate the Shrinkdroid automated testing, Themis [72] benchmarking applications *Amaze File Manager*, *Markor*, and *Omni Notes Alpha* are used. Unlike the CPAC Systems applications, Themis consists of applications with real-world, reproducible crash bugs drawn from open-source Android applications. The benchmark is constructed specifically to evaluate the effectiveness of automated GUI testing tools. It serves as a well-established reference point for evaluating Shrinkdroid's bug discovery, relating to RQ1. Themis contains 22 applications, but since the testing of these external applications was introduced later in the testing phase, which caused a limited time frame for testing, only three applications were drawn from the benchmark. They were chosen based on criteria such that they should not require an internet connection and no login, since these preconditions cannot be reliably automated in the current setup. They are treated as a scope boundary rather than a limitation to Shrinkdroid.

To enable the test sessions, each device needs to be connected to a laptop, which together constitutes a test rig. In this setting, there are three laptops available. Laptop 1 runs Ubuntu 24.04.4 LTS with CPU 13th Gen Intel® Core™ i7-1360P × 16 and RAM 32.0 GiB. Laptop 2 runs Ubuntu 24.04.4 LTS with CPU Intel® Core™ i7-10510U × 8, and RAM 16.0 GiB. Laptop 3 runs on Windows OS with CPU 11th Gen Intel(R) Core(TM) i7-1165G7 @2.80GHz, and RAM 16.0 GiB. Laptop 3 is solely used for the LoadAssist application, which runs on a separate test rig which was already configured and running within the company. The test rig for DigAssist alternates between laptops 1 and 2, since it can simply be connected through USB. The Samsung Galaxy S8 is mostly used on laptop 1 due to the settings and permissions granted for this computer.

For testing the DigAssist, two different versions were used: v3.5.121 and v3.5.442. The first one was primarily used as a version with various known recorded bugs and as a starting point during the development of Shrinkdroid. For testing the exploration on a version with fewer known bugs, a newer version was selected.

The rig for LoadAssist was already configured by other developers to update to the latest application version every night, so versions ranging from 3.5.244.0 to 3.5.294.0 were used for testing LoadAssist. No bugs were known in advance for this application and version by Shrinkdroid. This application was additionally selected to allow exploration of a different kind of application than the one targeted during implementation, as well as expanding the testing with new situations to find bugs and evaluate the tool.

For the mobile applications, version 3.7.0 was used for *Amaze File Manager*, 2.8.5 for *Markor*, and 6.1.0 were used for *Omni Notes Alpha*. The versions used were the latest versions available. They were installed at the beginning of each test run, from the application path stored on the laptop specified for the test execution, and uninstalled at the end of each test run.

Random testing sessions are conducted over a period of 4 weeks using Shrinkdroid's random weighted exploration strategy. After each testing session, the generated UTG, event sequences and reported property violations are manually reviewed to classify violations as either valid bugs or false positives. This validation process is performed by analysing the reproduced behaviour together with execution traces and generated artefacts. During the same time period, bug reports identified through the industrial partner's existing testing practices are collected to enable comparison between conventional GUI testing workflows and PBT with Shrinkdroid. The comparison focuses on the number, overlaps, and categories of discovered bugs.

The other evaluation part, event sequence minimisation, is conducted both by shrinking newly identified bugs found during the RWE sessions and by controlled shrinking using GE on previously known bugs. For bugs discovered during RWE, the detected violations are manually verified and analysed to identify the actual minimal event sequence required to reproduce the failure. This enables comparison between the minimal sequence identified manually and the minimised sequence

produced automatically during shrinking.

The event sequences for previously known bugs are stored in action lists as JSON objects, as required by the exploration strategy, but with redundant events inserted at different positions in the sequences. These action lists are used during GE, guiding the tool to reproduce known bugs and then run and evaluate the shrinking process. This allows the reduction ratio and shrinking effectiveness to be measured against a known ground truth, namely the minimal event sequence required to reproduce each bug.

For RQ1, the evaluation metrics are based on the number of discovered bugs, properties violated per application and the classification of each violation. Classification includes in which way the violation was confirmed or considered non-deterministic, connected to difficulties in reproducing the violation. Additionally, the question is evaluated based on how many violations were actual bugs and the number of false positives that were discarded as the last step during manual validation.

For RQ2, the evaluation metrics are based on the number and types of bugs identified both by Shrinkdroid and through the industrial partner's existing testing practices. The identified bugs are categorised according to their characteristics and compared to study similarities and differences in the types of failures detected by the respective approaches. Only GUI-related bugs that fall within the intended scope of Shrinkdroid are included in the analysis, meaning that the bug could theoretically be triggered through Shrinkdroid and evaluated through the defined properties.

For RQ3, the evaluation metrics are the reduction ratio compared to the original sequence, the ratio of the minimal number of events after shrinking and the known minimal sequence, and execution time. These will be evaluated in the form of A/B testing, where A represents the initial execution result (without shrinking), and B is the minimised event sequence result (after shrinking), and they are compared to each other, measuring the effects of applying shrinking.

6

Results

This chapter presents the results of evaluating Shrinkdroid across the applications. Shrinkdroid has successfully identified bugs in all applications except for Omni Notes Alpha. The results are organised into three sections. Section 6.2 presents the bugs discovered during RWE, addressing RQ1 and RQ2. Section 6.3 displays the GE test cases used in controlled testing for evaluating shrinking. Section 6.4 shows the shrinking result and its effectiveness, addressing RQ3.

6.1 Test Execution Overview

Shrinkdroid is being executed using the exploration strategies RWE or GE. Each test session is run independently on each of the industrial and mobile applications. The GE is only executed on the industrial application DigAssist. The runs in the test session are executed right after one has finished, with an application reset between each run. For the mobile test sessions, the targeted application is uninstalled and reinstalled between each run. This ensures that applications are restored to their original state when the application is reinstalled. The Hydra's configuration does not allow a total reinstall to reset the application to its initial state due to the application's development. Hence, as a setup step, Android Debug Bridge (ADB) commands are used to make sure that the starting screen is visually similar and that no previous run states influence the exploration. The mobile applications were significantly less evaluated than DigAssist and LoadAssist, but this was due to the primary testing focus lying in the industrial applications.

Table 6.1 summarises the test executions and violation counts for all the applications used in these testing sessions. Testing sessions were conducted with intentionally varied parameters to achieve variation in the setup and violations detected. Worst-case shrinking time also benefits from fewer steps in exploration. Violations that were discarded were detected by Shrinkdroid, but upon manual verification classified as false positives, meaning that the observed UI state was the correct behaviour and not an actual bug.

Table 6.1: Aggregated RWE test configurations and results per application.

Platform	Application	Runs	Steps per Run	Violations	Discarded
Android Automotive	DigAssist	40	200	5	3
Android Automotive	DigAssist	40	300	2	0
Android Automotive	LoadAssist	28	200	3	2
Android Automotive	LoadAssist	24	300	1	1
Android Smartphone	Amaze File Manager	15	300	1	1
Android Smartphone	Markor	15	300	7	2
Android Smartphone	Omni Notes Alpha	15	300	0	0

6.2 Random Weighted Exploration Results

This section displays the bugs discovered using RWE in both the industrial and mobile applications, addressing RQ1 and RQ2. Executions were conducted using 200 or 300 steps per run.

In tables 6.2 and 6.3, detected violations are categorised into three categories. *Confirmed violation* represents the property violation that was found during exploration, reproduced during the shrinking phase, and validated as an actual property violation. *Manual-only confirmed violation* refers to property violations raised during exploration, but were not reproduced during shrinking. It could, however, be reproduced by manual inspection and validated as a real property violation. Bugs that caused a violation during runtime but could not be reproduced in either a manual or automatic way are being categorised as *Non-deterministic violation*. By these classifications and further manual validation, the verdict on whether the violation is a real bug or should be discarded can be made.

Discarded verdicts give a short explanation of why a raised violation was discarded. *Non-reproducible* means that the violation could not be raised again, neither automatically by the tool nor manually when using reported artefacts as a guide. *False-positive* verdicts suggest that the violation was dependent on a fragile state or application state at the time of execution, originated from limitations in runtime UI hierarchy interpretation (e.g., layered or non-intractable elements exposed in the hierarchy). If the bug could be automatically or manually reproduced, and the violation persisted, the verdict would be not to discard the violation. In total, 9

violations were discarded following manual validation. Of these, four were genuine false positives, while the remaining five were non-reproducible violations that could not be confirmed.

6.2.1 Android Automotive Device

Table 6.2 shows the results of testing on the Android automotive devices. During 80 runs on DigAssist and 52 runs on LoadAssist, 11 property violations were detected. 5 bugs were validated as real bugs (4 in DigAssist and 1 in LoadAssist), and confirmed by manual or automatic verification. Since RW-DA3 and RW-DA6 were violations that could not be reproduced, they were discarded as non-reproducible and not considered valid bugs. Violation ID P3 are highly represented in this setting, but discarded due to being a false positive. This could be because the toggle property does not align with the application's state behaviour, leading to the violations being reported, although it is the correct application functionality. The toggle element that was not showing but could still be accessed was the main contributor to this issue. Otherwise, violation ID P1 are the property that is violated most often, and confirmed as well.

Table 6.2: Violated properties detected by Shrinkdroid on Android Automotive Device.

Random Weighted ID	Application	Violation ID	Number of Steps	Classification	Discarded Verdict
RW-DA1	DigAssist	P1	249	Manual-only confirmed violation	Not discarded
RW-DA2	DigAssist	P2	64	Manual-only confirmed violation	Not discarded
RW-DA3	DigAssist	P1	18	Non-deterministic violation	Yes - Non-reproducible
RW-DA4	DigAssist	P1	81	Manual-only confirmed violation	Not discarded
RW-DA5	DigAssist	P5	22	Confirmed violation	Not discarded
RW-DA6	DigAssist	P3	92	Non-deterministic violation	Yes - Non-reproducible
RW-DA7	DigAssist	P5	118	Manual-only confirmed violation	Yes - False positive

Random Weighted ID	Application	Violation ID	Number of Steps	Classification	Discarded Verdict
RW-LA1	LoadAssist	P3	39	Confirmed violation	Yes - False positive
RW-LA2	LoadAssist	P3	17	Manual-only Confirmed violation	Yes - False positive
RW-LA3	LoadAssist	P4	6	Confirmed violation	Not discarded
RW-LA4	LoadAssist	P3	164	Manual-only Confirmed violation	Yes - False positive

6.2.2 Android Mobile Device

For the mobile testing sessions, the Themis benchmark applications were explored for five test executions, containing three runs with 300 steps per run. Omni Notes Alpha didn't trigger any proper violation during all runs, suggesting that the application is robust for the properties specified in Shrinkdroid, the exploration never reached any violating states, or the properties didn't cover the flaws of the application. Still, 8 violations occurred, where 3 violations were discarded due to not being able to reproduce them.

Table 6.3: Violated properties detected by Shrinkdroid on Android Mobile Device.

Random Weighted ID	Application	Violated ID	Number of Steps	Classification	Discarded Verdict
RW-AFM1	Amaze File Manager	P1	169	Non-deterministic violation	Yes - Non-reproducible
RW-M1	Markor	P1	59	Confirmed violation	Not discarded
RW-M2	Markor	P1	77	Confirmed violation	Not discarded
RW-M3	Markor	P1	139	Manual-only confirmed violation	Not discarded
RW-M4	Markor	P1	106	Confirmed violation	Not discarded
RW-M5	Markor	P1	37	Manual-only confirmed violation	Not discarded

Random Weighted ID	Application	Violated ID	Number of Steps	Classification	Discarded Verdict
RW-M6	Markor	P4	13	Non-deterministic violation	Yes - Non-reproducible
RW-M7	Markor	P1	80	Non-deterministic violation	Yes - Non-reproducible

As seen in table 6.3, Markor produced the most violations in a single application, where more than half were confirmed bugs of violating the crash property, implying that crashes are a recurring behaviour under different navigation paths. Apart from RW-M6, no other property was violated during the testing sessions of these mobile applications. This could be due to the property coverage being insufficient, or that the exploration depth was too shallow, not reaching enough deep states in which a violation could be triggered.

The confirmed bug failures were previously not reported in the Themis GitHub repository, and therefore not reported in Themis's set of known bugs at the time of the testing. This is a promising indicator that Shrinkdroid is able to detect unreported bugs in applications where there are already known bugs, strengthening the case for RQ1 that PBT contributes to detecting bugs beyond existing knowledge of the application.

6.3 Guided Exploration Test Cases

The guided test session targets one application, DigAssist, which contains previously known bugs used to derive the defined properties. Each test case consists of 12-18 events, where the minimal event sequence required to reproduce a bug is extended to a longer event trace containing redundant and non-essential events. These additional events are intentionally introduced to simulate realistic but inefficient user interaction patterns and to create a baseline for evaluating Shrinkdroid's ability to minimise event sequences. All test executions in this test session were run with the embedding model activated for state signature similarity validation, allowing a shared evaluation baseline for all test cases.

Table 6.4: Guided Exploration scenarios and violated properties for previously known bugs in DigAssist.

Guided Exploration ID	Test Case Description	Violated Property ID	Property Summary
G1	In System UI settings, it is possible to change settings for Assistant & Voice. When pressing the option to select a digital assistant application, the system crashes	P1	Application should not crash
G2	When a dialogue with radio buttons is present, and the camera is switched on, the application crashes when selecting any of the buttons	P1	Application should not crash
G3	When a dialogue with radio buttons is present, and the day/night mode is switched, the application crashes when selecting any of the buttons	P1	Application should not crash
G4	When double-clicking on an object, the user is navigated to an overview screen and clicking the same object in that view gives a NullReferenceException	P2	Application should not throw NullReferenceException
G5	When adding a new target weight to a truck, leaving the application and returning, the add weight dialogue disappears	P5	Target application screen should be preserved after external detour
G6	When adding new material to a site, leaving the application and returning, the application is not in the same state as upon leaving	P5	Target application screen should be preserved after external detour
G7	When changing boundary limits and switching an activation toggle, a confirmation dialogue pops up with ”Yes” and ”No” options. If the user leaves the application with the dialogue open, returns, and selects ”Yes”, the toggle is still in the same state as before when it should have switched	P3	Toggle actions must follow expected checked-state behaviour

6.4 Shrinking Results

In this section, the results for the final evaluation of Shrinkdroid and PBT for Android application testing are presented regarding test case minimisation and shrinking performance. The result aims to answer RQ3 by measuring the efficiency and quality effects of property-based testing suites when applying shrinking methods.

For table 6.5 and 6.8, all the violations have been extracted and are presented with what the *minimal event number* was found during the shrinking phase, and what the actual *known minimal event number* of steps are, based on manual interaction. The event step reduction is presented in the *reduction* section, stating the minimised event number found during shrinking compared to the initial event number. For reference, the reduction ratio of the known minimal event number compared to the initial is also presented in parentheses. If the violation can be reproduced, it is stated in the last column whether it's done automatically, manually or not at all.

Table 6.7 and 6.10 show the execution times of the violations, with the *initial time* during exploration and *execution time after shrinking* the event sequences, together with the time it took for the full shrinking phase to complete. In this full *shrinking time*, it includes a re-execution of the initial event sequence that produced the violation, the execution time of finding the minimal event sequence, and a post-check to validate that the shrunken event sequence violation is preserved. Depending on the resulting shrinking technique, the total shrinking time can vary significantly.

The tables 6.6 and 6.9 present the total number of shrinking attempts, number of attempts that successfully reproduce the violation, skipped attempts where the candidate event sequence was not a valid path in the UTG, and lastly, which shrinking techniques were used.

6.4.1 Shrinking of Randomly Discovered Violations (Random Weighted Exploration)

Table 6.5: Event sequence reduction result for Random Weighted Exploration bugs.

Random Weighted ID	Initial Event Number	Minimised Event Number	Known Minimal Event Number	Reduction	Reproduced
RW-DA1	249	-	5	-	Yes, manually
RW-DA2	64	-	13	-	Yes, manually
RW-DA3	18	-	-	-	No
RW-DA4	81	-	5	-	Yes, manually

Random Weighted ID	Initial Event Number	Minimised Event Number	Known Minimal Event Number	Reduction	Reproduced
RW-DA5	22	18	4	18% (81%)	Yes
RW-DA6	92	-	9	-	Yes, manually
RW-DA7	118	-	7	-	Yes, manually
RW-LA1	39	39	2	0% (94%)	Yes
RW-LA2	17	-	3	-	Yes, manually
RW-LA3	6	2	2	66%	Yes
RW-LA4	164	-	3	-	Yes, manually
RW-AFM1	169	-	-	-	No
RW-M1	59	38	11	36% (81%)	Yes
RW-M2	77	46	12	40% (84%)	Yes
RW-M3	139	98	12	30% (91%)	Yes
RW-M4	105	64	12	39% (88%)	Yes
RW-M5	37	-	11	-	Yes, manually
RW-M6	13	-	-	-	No
RW-M7	79	-	-	-	No

Table 6.6: Shrinking attempts and techniques for Random Weighted Exploration bugs.

Random Weighted ID	Total Attempts	Repr. Violation Attempts	Skipped Attempts (filtered by UTG check)	Shrinking Technique
RW-DA5	2	1	0	Yen k Shortest Path
RW-LA1	19	1	66	Greedy
RW-LA3	1	1	0	Shortest path
RW-M1	60	20	298	Greedy
RW-M2	70	27	543	Greedy
RW-M3	104	42	Not known, terminated early	Greedy
RW-M4	80	34	703	Greedy

Table 6.7: Execution times before/after applied shrinking, and shrinking time for Random Weighted Exploration bugs.

Random Weighted ID	Initial Execution Time	Execution Time After Shrinking	Shrinking Time
RW-DA1	41min 38s	-	-
RW-DA2	4min 16s	-	-
RW-DA3	2min 49s	-	-
RW-DA4	3min 52s	-	-
RW-DA5	2min 55s	57s	2min 54s
RW-DA6	4min 18s	-	-
RW-DA7	16min 1s	-	-
RW-LA1	9min 36s	9min 30s	1h 56min 42s
RW-LA2	4min 15s	-	-
RW-LA3	1min 58s	12s	4min 6s
RW-LA4	11min 23s	-	-
RW-AFM1	8min 55s	-	-
RW-M1	2min 24s	59s	49min 13s
RW-M2	6min 20s	2min 06s	58min 37s
RW-M3	8min 19s	4min 38s	5h 1min 23s
RW-M4	5min 1s	2min 42s	4h 52min 23s
RW-M5	2min 54s	-	-
RW-M6	6min 45s	-	-
RW-M7	4min 15s	-	-

When shrinking the randomly discovered violations using RWE, it was found that seven violations were able to be reproduced correctly, and six of them were minimised. The minimised violations also show a reduction ratio of at least 18%, showing that the violations can be preserved by smaller event sequences. Table 6.5 shows that violations RW-LA1 were able to be reproduced, but did not actually shrink the event sequence with any event. This suggests that the exploration had already produced a near-minimal sequence, or that the greedy algorithm could not find a valid shorter path while preserving the violation. The violations RW-DA1, RW-DA2, RW-DA4, RW-DA6, RW-DA7, RW-LA2, RW-LA4, and RW-M5 were detected during exploration but could not be reproduced automatically due to not being able to perform all the events in order at the re-execution. They could be verified manually, which validated the bug; however, they could not be minimised due to not being successful in the shrinking phase. The remaining violations, RW-DA3, RW-AFM1, RW-M6, and RW-M7, detected during the exploration, have been discarded as non-reproducible violations since they could not be reproduced.

RW-LA3 were able to minimise its event sequence to the minimal known sequence using the Shortest Path shrinking technique in a single attempt, displayed in Table 6.6. It also shows its effectiveness in both time and the number of attempts to find the minimal event sequence. RW-DA5 also resulted in an efficient shrinking case,

where Yen k Shortest Path were used to shrink in only two attempts. The violations RW-LA1, RW-M1, RW-M2, RW-M3, and RW-M4 all used the Greedy algorithm as the shrinking technique, which was shown to be a slower approach. Apart from RW-LA1, which was not able to reduce any event while preserving the violation, none of the shrinking reached the known minimal event sequence. Since RW-M3 was terminated early, the skipped attempts could not be determined, but during the time it did shrink, it was able to reduce the initial event step from 139 steps to 98. Its shrinking time was the longest recorded during the test session, tightly followed by RW-M4, suggesting that the greedy algorithm becomes time-consuming as the sequence length grows.

As seen in table 6.7, the execution time after shrinking has been reduced by a significant amount of time for the violations that could be minimised. RW-LA3 demonstrates the best time reduction, both regarding shrinking time and execution time after shrinking. While some violations produced good results regarding the shrinking times, RW-M3 and RW-M4 show a significantly higher total time of the shrinking phase.

6.4.2 Shrinking of Known Violations (Guided Exploration)

For the GE evaluation, previously identified bug-triggering event sequences are used as a baseline to assess shrinking performance. Since the minimal event sequence required to reproduce each bug is already known, the effectiveness of Shrinkdroid can be measured by comparing the reduced sequences against this reference length. This setup allows a controlled A/B comparison between PBT executions with and without integrated shrinking, highlighting the impact of the shrinking mechanism on sequence minimisation.

Table 6.8: Event sequence reduction result for Guided Exploration bugs.

Guided Exploration ID	**Initial Event Number**	**Minimised Event Number**	**Known Minimal Event Number**	**Reduction** (Min Reduction)	**Reproduced**
G1	12	4	4	66%	Yes
G2	13	7	7	46%	Yes
G3	15	10	9	33% (40%)	Yes
G4	14	2	2	83%	Yes
G5	13	9	5	31% (62%)	Yes
G6	17	6	6	67% (67%)	Yes
G7	16	9	6	44% (63%)	Yes

Across all test cases, the shrinking process successfully reduced the original event sequences while still reproducing the corresponding property violation. In table 6.8, the resulting sequence lengths vary between cases, with some matching the known minimal sequence length exactly, while others remain slightly above the minimal

Table 6.9: Shrinking attempts and techniques for Guided Exploration bugs.

Guided Exploration ID	Total Attempts	Repr. Violation Attempts	Skipped Attempts (filtered by UTG check)	Shrinking Technique
G1	1	1	0	Shortest path
G2	1	1	0	Shortest path
G3	1	1	0	Shortest path
G4	19	8	31	Greedy
G5	14	3	51	Greedy
G6	1	1	0	Shortest path
G7	14	5	44	Greedy

Table 6.10: Execution times before/after applied shrinking, and shrinking time for Guided Exploration bugs.

Guided Exploration ID	Initial Execution Time	Execution Time After Shrinking	Shrinking Time
G1	58s	41s	2min 53s
G2	2min 16s	1min 34s	6min 4s
G3	2min 31s	1min 55s	7min 1s
G4	2min 48s	59s	34min 13s
G5	2min 28s	1min 31s	23min 47s
G6	1min 11s	47s	2min 45s
G7	2min 29s	1min 45s	21min 46s

reference.

Table 6.10 shows that the execution time after shrinking is reduced for all test cases when executing the minimised event sequences, while the result for the shrinking phase itself shows noticeable variation in runtime across the test cases.

The number of attempts required during shrinking and the technique that successfully produced the minimal event sequence for each test case can be seen in Table 6.9. The Shortest Path algorithm proved to be the most efficient shrinking technique when a violation was successfully reproduced. It showed in both the number of attempts and the shrinking time for G1, G2, G3, and G6. The Greedy algorithm was revealed to be the least efficient algorithm, with G4, G5 and G7 requiring a long time to shrink and a lot of attempts.

6.5 Existing Testing Practices Findings from Industrial Partner

To evaluate the behaviour and results, bug reports have been extracted from the company's bug reporting system during the same evaluation period as the Shrinkdroid evaluation. The current testing practices mainly contain manual testing of the application.

During the same testing period as for the RWE and GE, CPAC Systems has reported 51 bugs over the two industrial applications following their practices. Out of the 51 bugs, 24 were considered actual GUI bugs. This extraction ensures a fair comparison that only bugs related to GUI behaviour and interaction sequences are included, as these fall within the detection scope of PBT. The reporting of bugs in the system is described with the detected error, and at times, steps to reproduce it and a video showing the error.

By analysing the relevant reported bugs regarding GUI, 3 bugs were considered detectable by the categories of the properties. The remaining bugs were reported for cases such as updating or correcting the application to the marketed design, too application-specific or changing behaviour between different application versions, which are outside the scope of this work. Since these bugs are not in the scope of Shrinkdroid's properties, which were designed to apply to different applications, they were discarded and excluded from the comparison.

Table 6.11: Comparable metrics of bugs found by PBT and testers.

Testing Practices	**Shrinkdroid (PBT)**	**Industrial Existing Practice**
Total Bugs Reported	11	51 (24 GUI-relevant)
Relevant Property Violation	5	3
Relevant Bug Categories	Crashes, null ref, Navigation, Layout Bounds	Crashes
Overlap	None	None

The comparison in Table 6.11 shows that although the industrial testing reported a larger total number of defects, a small subset overlapped with the behaviours that have been expressed as general properties. Shrinkdroid, despite not identifying as many total defects, detected more bugs within the property specifications defined. Still, no overlap was observed between the bug sets, indicating that the approaches detected different bugs at different places in the application.

7

Discussion

In alignment with previous studies, this work shows that it is possible to apply PBT for GUI testing and identify new bugs in applications and systems. In addition to previous studies, this work also confirms that shrinking is possible and successfully executed both on guided and random weighted application exploration and is possible to achieve in dynamically changing Android environments. In this chapter, the research questions will be discussed based on the results derived from the previous chapter.

7.1 RQ1: PBT's Effect on Bug Discovery

How do property-based testing tools affect the discovery of bugs in Android applications?

Shrinkdroid was able to detect bugs across five Android applications, confirming that PBT can effectively support bug detection through both guided and autonomous exploration. This also confirms that it is not tightly coupled to a single application or device, and can be applied to different Android environments. The results indicate that PBT captures diverse types of bugs and inconsistencies in an application as well, without needing to write a specified unit test for each distinct setting. Since the properties are defined as specifications that should always hold regardless of what state the application is in, Shrinkdroid evaluates for each interaction that the application follows the intended behaviour, strengthening the case that PBT as an appropriate approach to Android GUI testing.

Across 80 runs of DigAssist, 52 runs of LoadAssist, and 15 runs on each of the mobile applications, a total of 26 violations were caught, of which 17 were validated as real bugs. The violations confirmed as real bugs spanned a variety of properties defined in this tool, including crashes, null reference exceptions, toggle state inconsistencies, layout bound violations, and navigation preservation failures, showing that many of the properties in the defined property set capture the behaviour of real Android failures. Some applications, such as Markor and DigAssist, violated multiple properties during exploration, while Omni Notes Alpha did not report any violation at all. This suggests that, depending on the interaction between the application and the defined properties, the discovery of bugs can vary.

Out of the 26 violations, 9 were discarded as false positives or non-reproducible after manual inspection. This represents a rate of approximately 35%, with three in DigAssist, three in LoadAssist, one in Amaze File Manager, and two in Markor. The primary reason for the false positive results was the layered UI. While UIAutomator2 could expose elements at runtime, detecting all elements on screen, it could also detect elements in the background that are not visible or interactable for the user. This particularly affected the violations of property 3, toggle consistency, where the state changes were evaluated against elements that could not be reached from a normal user interaction, triggering violations that did not correspond to real application failures. The non-reproducible violations, on the other hand, could not be triggered again. Those violations do not necessarily mean that the detection was incorrect, but rather a failure in that specific execution, caused by a runtime exception. While these violations cannot be confirmed as stable bugs, they may still indicate fragile application states worth investigating.

Not all properties were violated during the testing period. Properties P6 and P7 were never triggered, since these properties cover less common scenarios that happen in a random weighted exploration, and would require more guided exploration to violate them. It shows that the bug discovery in PBT is also influenced by both the formal specification of the property and the exploration strategy used in the testing. Whether this reflects correct application behaviour or insufficient exploration coverage cannot be determined from the current results. Guided test cases specifically designed to target these conditions would be needed to distinguish between the two explanations.

The results also show that the increasing number of executions does not necessarily lead to a higher number of violations. For example, DigAssist produced 2 real bugs when having 200 steps and 2 bugs when having 300 steps, which indicates that deeper exploration does not guarantee an increased number of bugs. However, Markor had a significantly lower number of runs, at 300 steps, but was still able to produce 5 real bugs, showing that PBT does not solely depend on the amount of executions or depth, but how effective it is when reaching relevant application states. Applications used in the mobile environment contain a simpler system, which may induce more violations than the industrial automotive applications, where the application states are more robust.

Still, with other PBT-testing tools for GUI, like KEA [2], Shrinkdroid adds to the evidence that PBT can be used as an effective approach to test Android GUI, and extends the approach by integrating event sequence shrinking, which has not been supported by the tools available today. It is an effective way to debug the failing event sequence, transforming long and complex interaction traces into concise, human-readable counterexamples that display the minimal steps essential to reproducing the failure.

The observations made provide an insight into how PBT influences bug discovery in practice, but an important note is that while Shrinkdroid identifies violations by the property, the final determination regarding the severity of the bug remains with the

developer or tester to decide. The tool reports when a violation has been detected, but it is not able to assess the impact that the violation had on the end-user or on the affected functionality. Severity assessment requires domain knowledge that cannot be automated reliably, and therefore, Shrinkdroid can be used as a tool that candidates bugs for the human reviewer.

Overall, these findings suggest that PBT shifts how GUI-testing can be performed. Instead of manually defining interaction sequences scenarios, testers can describe properties that should always hold during execution. This allows for bugs to appear naturally during exploration, resulting in property-based testing complementing traditional GUI testing by identifying behavioural issues that may not be covered by predefined test cases.

7.2 RQ2: Comparing PBT to Existing Testing Strategies in the Industry

How does property-based testing compare to existing testing strategies in the industry?

The lack of overlapping bugs in the two bug sets indicates that Shrinkdroid explores different areas of the application compared to existing industrial testing practices, strengthening the case for Shrinkdroid as a complementary tool rather than a redundant one. During the same testing period of the DigAssist and LoadAssist, Shrinkdroid was able to find 5 reproducible unknown bugs out of 11 bugs in total using RWE. CPAC Systems' existing testing practices revealed 24 relevant bugs out of 51 total reported bugs during the same time. Of the 24 relevant bugs, only 3 fell within the detection scope of Shrinkdroid's defined properties. The remaining 87% required domain-specific visual inspection, application context knowledge, or interaction types not expressible through the current property set. This underscores that PBT as implemented in this work is a complementary rather than a stand-alone testing solution, and that the manual effort required to define properties and interpret results must be weighted against the types of bugs the approach is capable of detecting.

The absence of overlap highlights the differences in the two approaches. The company's testers possess deep domain knowledge about the application's behaviour and visual design, which enables them to recognise incorrect UI states, specification deviations, and inconsistencies in the layout that are difficult to specify in a formal property. Several reported bugs regarding overlapping elements or changes not being applied in the GUI fall into categories that require a visual inspection or application-specific context to identify. Shrinkdroid, on the other hand, reaches application states through autonomous exploration and covers parts that are not an obvious event sequence, which have been missed by scripted or manual tests. The bugs it identified were triggered through state-dependent paths that had not been

anticipated in the existing test suite. Together, these approaches provide broader testing coverage than either method alone.

At the same time, the distinct set of bugs discovered by Shrinkdroid and those found by the industrial testers raises questions about how representative its exploration paths are of real user behaviour. Since RWE relies on weighted random exploration, it may navigate the application and create event sequences that a real user would be unlikely to follow. This is simultaneously a strength, enabling the discovery of unexpected states and corner cases, and a potential limitation in terms of representativeness. Determining the balance between broad state-space exploration and realistic user behaviour remains an important consideration for automated exploration-based GUI testing.

While PBT shows the ability to find unique defects, its efficiency must also consider adoption effort. Defining properties requires a different testing mindset than with traditional or manual testing [8], and developers and testers might not experience the approach as effective since their existing unit testing is well established. Although PBT can reduce manual test design effort during testing, the initial setup and adoption of this method might reduce efficiency in testing. Since this study does not measure development time, no quantitative conclusions can be drawn, but still considered a factor with important consideration for industrial deployment.

Although integrating this kind of automated testing into a well-established industry workflow requires additional effort, the efficiency of debugging could be improved with the artefacts produced by Shrinkdroid. Current bug reporting in the system consists of testers having to reproduce failures, document interaction steps, and describe observed behaviour. Shrinkdroid, on the other hand, creates the event sequence diagram and relevant documentation at runtime. By capturing the sequence leading to an error, the tool minimises the risk of incomplete or insufficient bug reports and facilitates debugging by automatically retrieving the event sequence that caused a failure.

Shrinkdroid detected fewer total defects than the existing testing practices, but helped expand the overall coverage and reduced manual effort. Manual and scripted testing are more efficient in detecting visual and domain-specific correctness issues, whereas PBT is efficient at uncovering unexpected runtime failures through its exploration. Rather than replacing existing strategies, PBT increases overall testing efficiency by expanding coverage of the application.

7.3 RQ3: Efficiency and Quality of Shrinking

How does the application of shrinking methods affect the efficiency and quality of property-based testing suites?

The shrinking results presented in section 6.4 demonstrate that event-sequence minimisation methods through shrinking can effectively reduce failing exploration traces while still preserving the original property violation. Successful shrinking

was achieved for both RWE and GE, showing that the approach applies to both randomly discovered and previously known bug-triggering event sequences.

Shrinking was completed for 14 out of 26 evaluated test cases. The remaining test cases were discarded as part of the initial validation step in the shrinking phase because the detected violation could not be reproduced. This validation stage is necessary to ensure that the violation can be deterministically reproduced before attempting computationally expensive shrinking procedures. If the original event sequence cannot reliably reproduce the same behaviour, the resulting shrinking process becomes unstable and risks not giving any result.

Although several discarded cases could later be manually reproduced and verified as valid bugs, the difficulties of automatically reproducing them prevented reliable automated shrinking. This highlights the importance of deterministic initialisation and stable application state management when applying shrinking techniques to dynamic systems. If the reported violations could be reproduced by manually executing the same event sequence, they were further analysed to find the minimal event sequence which still preserved the same property violation. It was an essential step in the process to validate the bug discovery capabilities of the tool, while also reporting and documenting details for further bug analysis.

For the successful shrinking attempts, the shrinking process consistently reduced the original event sequence length while preserving the corresponding property violation in 13 out of 14 cases. In several guided exploration cases, the resulting sequence matched the known minimal sequence exactly, while other cases remained slightly above the optimal reference length. This indicates that the shrinking approach is generally capable of producing near-minimal reductions even when the globally shortest bug-triggering sequence is not reached.

All of the minimised event sequences passed the last reproducibility check during post-validation when testing them for flakiness. Passing that check confirms that the minimal found event sequence is possible to reproduce several times while still preserving the same property violation, showing stability and accuracy. The execution time for all successful minimised event sequences was shorter compared to the original event sequence execution times, showing increased efficiency in time.

The guided exploration results further demonstrate that graph-based shrinking is particularly effective when the UTG contains deterministic transition paths between the relevant screen states. Cases such as G1, G2, G4, and G6 show that the shortest-path-based technique can identify minimal reproducing sequences efficiently, requiring very few attempts and relatively short shrinking times.

While shrinking reduced the execution time of the final minimised event sequences, the shrinking process itself introduced significant computational overhead in several cases. The results show a clear trade-off between reduction quality and shrinking cost. For shorter and more deterministic event sequences, shrinking was completed efficiently and produced significantly smaller and reproducible traces. However, for

longer exploration sequences where shortest-path techniques could not reproduce the violation directly, the fallback greedy-based shrinking became much more time-consuming.

This behaviour is particularly visible in cases such as RW-M3 and RW-M4, where shrinking durations exceeded several hours, although the resulting event sequence was only partially reduced. The results suggest that the practical usefulness of shrinking depends strongly on whether efficient graph-based reduction techniques succeed early in the process. As the original event sequence length increases, the computational cost of repeated greedy reductions grows rapidly, while the relative improvement in sequence length becomes smaller. Nevertheless, even partially reduced event sequences improved execution efficiency and simplified bug traceability compared to the original exploration traces.

7.4 Limitations and Delimitations

Although the results demonstrate that Shrinkdroid can successfully discover, reproduce, and minimise event sequences violating properties in Android applications, several limitations affected both the implementation and evaluation of the tool. These limitations originate from the dynamic and stateful nature of Android systems, practical constraints in the experimental setup, and the scope boundaries defined for the study. This section discusses the primary delimitations of the work together with the technical and experimental limitations that have the most impact on the interpretation and generalisability of the results.

7.4.1 Scope Delimitations

The scope of the system to evaluate was primarily set to applications within the company's Android Automotive platform, but later extended to include additional validation of applications on an Android mobile system. It was extended to give initial evidence on the generalisability of the developed tool, and measure its performance and compatibility on different systems and versions.

The application evaluation on the mobile system was further delimited to open-source applications from the benchmark, requiring neither an internet connection nor user authentication. This was an intentional decision based on security considerations around connecting unknown third-party applications to the industrial partner's internal network, rather than a technical constraint of the tool itself. Consequently, application behaviours dependent on third-party service interactions were not evaluated as part of this study. This applied to all applications, both on the company's platform and the mobile system.

The focus of this study has primarily emphasised the shrinking methods and possibilities within Android systems, while also evaluating the bug discovery capabilities when using PBT for testing such systems. Different exploration strategies have not been a major part of this study, as well as screen state similarity determination with

both the use of semantic embedding models and structural evaluation approaches.

7.4.2 Experimental and Implementation Limitations

Android applications are inherently stateful and affected by persistent system and application-level configuration. Changes applied during one execution may therefore influence subsequent runs, although efforts such as resetting the application between runs and application re-installation have been implemented. Although applications were uninstalled between runs for all test executions on the Android mobile device, the results showed that there were inconsistencies connected to permission requests, making shrinking and re-execution of defined event sequences difficult to reproduce due to the dynamic nature of such permission dialogues, dependent on OS and runtime state.

A limitation connected to the test configuration, but not within the scope of the study, was that the maximum execution steps for the test sessions were limited to 300, which might have been a limitation of bug discovery possibilities. Deeper states within the applications may remain unreachable due to different application sizes, and complex workflows were potentially under-explored. While the same number of bugs were discovered at both 200 and 300 steps, it may not be sufficient for deeply nested interaction flows, but increasing the steps would have impacted the shrinking and reproducibility possibilities. Executing test runs beyond 300 steps was not evaluated during the testing period, and the effect of deeper exploration on bug discovery and shrinking reliability remains an open question.

Some test executions were discarded following manual inspection due to environment-specific interface artefacts. In certain cases, UI elements violating layout-related properties were associated with debugging or developer-specific features not accessible to end users during normal application execution. Although these elements appeared in the runtime hierarchy as expected and triggered relevant property violations, they were considered non-representative of real user behaviour and were therefore discarded during manual validation.

Some limitations connected to hardware and infrastructure during test executions included instability and unexpected shutdown of the laptops as a result of having laptop-based test rigs that were not isolated solely for testing. Additional hardware limitations included connectivity-related difficulties, were connection was lost between the laptop and the device during test executions, which is not a limitation solely encountered in this study but a general limitation. Differences were seen between the different laptops used during test execution for different applications, where LoadAssist (running on an isolated test rig) was more stable than DigAssist and the mobile applications (running on laptops 1 and 2).

The use of multimodal embeddings for semantic state matching was not part of the scope for this study, but rather seen as a possibility for more accurate state matching. It increased confidence in graph construction, but its use was limited due to computational and financial cost constraints, which prevented large-scale usage

and evaluation across implementation and testing. Therefore, structural signature matching approaches were primarily used.

7.5 Validity Threats

Since this work involves the design and empirical evaluation of a property-based Android GUI testing tool, several factors may affect the reliability and generalisability of the results. This section discusses the primary threats to internal and external validity, together with the measures taken to mitigate them throughout the study.

7.5.1 Internal Validity

Since several properties were derived from previously identified bugs, there is a risk that the evaluation favours behaviour already known to violate the system requirements. This risk is mitigated by highlighting that the test execution result shows that it was possible to discover new bugs of different kinds during RWE in four different applications.

Another threat to the composition of the tests is the knowledge about the system needed to be able to define general and effective properties. When writing the specification, the developer must know how the system is used and built, for when writing properties, there are limited flaws that can threaten the results of the tests. Writing incorrect properties can harm the validity of the tool. Thus, the knowledge needs to be adequate to ensure that the properties are written specifically and with confidence. This threat was mitigated through continuous discussions with application developers with certain domain knowledge, and information collection covering Android system workflows and general UI semantics.

Limitations in the runtime UI hierarchy inspection are a reason for another internal validity threat. UIAutomator2 may expose elements in the hierarchy that are not currently visible or interactive to the user due to layered views or inactive interface states. As a consequence, certain actions were interpreted as interactions with target UI elements even though the interaction was effectively blocked by another foreground layer, creating misleading transition results and tracing difficulties. This particularly affected toggle-related properties, in which state changes were incorrectly evaluated, even though no user-visible interaction occurred. Such cases resulted in false-positive violations and led to several test executions being discarded during manual validation.

Additionally, non-deterministic UI behaviour is a reason for another internal validity threat, where reproducibility of event sequences is challenging due to the dynamic nature of stateful systems with permission dialogues, timing issues and dynamic UI transitions. This affects the reproducibility and consistency of shrinking validation. To mitigate the threat, experimental consistency is important between test executions and PBT phases by having proper setup and tear-down processes ensuring a standard initial state for applications and systems under test.

A potential threat arises from the manual classifications of the results. Developer-reported issues were seen as comparable or not comparable in the scope of Shrinkdroid, which introduces selection bias as it depends on the subjective interpretation of whether a defect could be detected through property-based testing.

7.5.2 External Validity

The study was narrowed to explore and test two of the applications in the system developed at the company, making this study limited in test result collection and testing the generalisability of the tool. This limitation is intentional, based on the resources available and the scale of the system to be tested. To further strengthen the generalisability of the tool, the testing on mobile applications shows that it is not limited to the application at the company, but can be used on any Android application. Generalisability remains limited by the number of evaluated applications, although experiments across both Android Automotive and mobile Android environments show that the approach is not tightly coupled to a single application domain.

However, while selecting several applications from the Themis benchmark provides evidence that Shrinkdroid is not tightly coupled with the industrial applications, the scope of this mobile evaluation is limited in two ways. First, there were only three of the 22 Themis applications that were tested, each selected because they contained many known failures and required neither an internet connection nor a login, since these preconditions could not be automated in the application setup at that point. The industrial applications were evaluated with an active internet connection, demonstrating that the tool is not inherently restricted to offline applications, although testing such services was outside the scope of this study. Additionally, several other candidate applications from the benchmark were attempted but could not be reliably executed and further analysed within the available time frame for the testing period. Second, each mobile application was running for 15 test executions compared to over 50 or more runs in the automotive applications, resulting in a substantially lower exploration depth. The results of mobile discoveries should therefore be interpreted as an initial indicator of generalisability between platforms.

The study was also narrowed to an Android-specific implementation with dependencies on external libraries, which means the results may not directly transfer to other operating systems and are limited by the maintenance degrees in the libraries used.

7.5.3 Use of Generative AI

Throughout this thesis, generative artificial intelligence (AI) has been used for grammar, spelling and text cohesion, using ChatGPT and Claude AI. All text has been written by the authors, but revised using AI, which can induce a risk of incorrectness. To mitigate this, all processed content has been reviewed and verified by the authors to ensure the correctness of the content.

7.6 Future Work

Although Shrinkdroid demonstrates effective exploration, bug detection, and event-sequence shrinking capabilities, several areas for further improvement and investigation were identified during the implementation and evaluation phases. The following directions outline potential extensions related to state representation, UI interaction modelling, exploration strategies, and shrinking optimisation.

7.6.1 Improved State Representation and UI Understanding

Future work could investigate more advanced state signature representations to improve screen distinction and state equivalence detection. Although the structural hashing approach used in this work proved effective in many cases, visually or semantically similar screens may still be incorrectly merged or separated. More robust state abstraction techniques, potentially combining structural, visual, and semantic information in better ways, could improve both UTG accuracy and shrinking reliability.

Another important extension covers improved handling of layered UI hierarchies and overlapping interface states. Runtime hierarchy inspection may expose elements that are technically present in the hierarchy but not currently visible or possible interactions to the user. Future work could investigate methods for identifying effective foreground interaction layers to reduce false-positive interactions and improve property validation reliability.

7.6.2 Improved Exploration and Interaction Possibilities

The current implementation explores both the target application and system-level interfaces as part of the complete system under test. Future extensions could provide configurable separation between AUT-focused and full system exploration modes, allowing developers to tailor exploration scope depending on the testing objective.

Future work could also focus on improving exploration efficiency and reducing repeated navigation patterns. Although implemented weighting heuristics reduce local cycles and repeated state traversal, long-running sessions may still become trapped in limited interaction states. More adaptive exploration strategies or learning-based navigation approaches could improve state-space coverage. Integrating applications' usage frequency or interaction data, which creates an exploration bias toward common user paths, could be a direction for future work if the intention is to simulate real user navigation.

The current implementation limits action generation primarily to explicitly actionable UI elements. Future work could investigate techniques for dynamically deriving interactions for semantically relevant but non-actionable elements, such as gesture-based interactions or clickable regions, with corresponding property definitions for

expected behaviour. Extending the interaction model in this way may improve exploration depth and increase bug discovery capabilities.

Extending the evaluation to applications requiring authentication or network access to third-party communication, for example, through controlled network environments or pre-configured login and authentication setup, would further strengthen the generalisability of the results across a broader set of real-world applications.

7.6.3 Shrinking Strategy Optimisation

The shrinking process could also be further standardised and optimised. The current implementation applies multiple shrinking strategies in a prioritised order, but the most effective approach may vary depending on application structure, transition graph complexity, or test execution configurations such as the number of exploration steps. Future work could investigate the effectiveness of different shrinking methods to determine optimal shrinking strategies for different classes of event sequences, UI transition graphs and test configurations.

8

Conclusion

This research aimed to address key limitations of existing Android GUI testing approaches by designing, implementing, and evaluating Shrinkdroid. This property-based testing tool combines dynamic UI element detection, automated event sequence generation, and integrated test case minimisation through shrinking.

The three research questions guiding this work have been answered through empirical evaluation across industrial and open-source Android applications. Regarding RQ1, Shrinkdroid successfully identified 17 validated bugs across five Android applications, covering several different failures such as crashes, null reference exceptions, toggle state inconsistencies, layout bound violations, and navigation preservation. Ten of these bugs were previously unknown and unreported, demonstrating that property-based testing can detect faults that are not anticipated by existing test suites. The property-driven approach enabled exploration of application states that manual testing and scripted test cases do not typically reach, particularly for state-dependent interaction sequences.

Regarding RQ2, the comparison with the industrial partner's existing testing practices revealed that no overlap was observed between the bugs identified by Shrinkdroid and those reported by company testers over the same period. This complementarity suggests that Shrinkdroid explores different parts of the application state space, making it a valuable addition to rather than a replacement for established testing workflows. However, the tool was less effective at detecting visually apparent layout bugs that human testers can identify intuitively, highlighting the continued importance of human inspection for certain bug categories.

Regarding RQ3, the shrinking pipeline was completed in 14 out of 26 evaluated cases, and successfully minimised event sequences in 13 out of these 14, consistently reducing execution time for the final minimised sequences compared to the original exploration traces. Graph-based shortest-path reduction proved most efficient for deterministic traces, while greedy reduction remained viable for longer sequences at the cost of substantially increased shrinking time. In several guided exploration cases, the minimised sequences matched the known minimal bug-triggering sequences exactly. Together, these results show that shrinking can effectively be applied to reduce failing Android GUI event sequences, producing concise and reproducible counterexamples that facilitate debugging.

This work demonstrates that property-based testing can be effectively applied to Android GUI systems despite the challenges posed by dynamic interfaces, state-dependent behaviour, and long event sequences. By combining automated exploration with test case minimisation through shrinking, Shrinkdroid was able to detect previously unknown bugs while also producing concise and reproducible failing traces that support debugging and analysis. The results further suggest that property-based GUI testing is best positioned as a complementary testing strategy that broadens behavioural exploration beyond conventional scripted and manual testing approaches.

Several directions remain open for future work. The property set could be extended to cover additional bug categories, such as visual overlap detection and data persistence errors. Shrinking strategy selection could be made adaptive based on sequence length and UTG structure to reduce computational overhead. More robust state representation techniques, combining structural, visual, and semantic signals, could improve UTG accuracy and shrinking reliability. Additionally, an evaluation on a larger scale across a broader set of applications would further strengthen the generalisability of the results.

Overall, this thesis demonstrates that integrating property-based testing, dynamic UI element detection, and event sequence reduction through shrinking provides a practically applicable approach to automated Android GUI testing.

Bibliography

[1] H. Elahi et al., "A qualitative study of app acquisition and management," *IEEE Transactions on Computational Social Systems*, vol. 11, no. 2, pp. 1907–1925, 2024.

[2] Y. Xiong et al., "General and practical property-based testing for android apps," *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE), Automated Software Engineering (ASE), 2024 39th IEEE/ACM International Conference on, ASE*, pp. 53–64, Oct. 27, 2024, Publisher: ACM, ISSN: 979-8-4007-1248-7. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=7557ec66-8da6-3977-a080-ec17df59d309.

[3] A. Löscher and K. Sagonas, "Targeted property-based testing," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Santa Barbara CA USA: ACM, Jul. 10, 2017, pp. 46–56, ISBN: 978-1-4503-5076-1. DOI: 10.1145/3092703.3092711. Accessed: Nov. 28, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3092703.3092711.

[4] 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*, Vancouver BC Canada: ACM, Oct. 25, 2017, pp. 58–77, ISBN: 978-1-4503-5530-8. DOI: 10.1145/3133850.3133853. Accessed: Nov. 28, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3133850.3133853.

[5] J. Hughes, "QuickCheck Testing for Fun and Profit," in *Practical Aspects of Declarative Languages*, M. Hanus, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–32, ISBN: 978-3-540-69611-7. DOI: https://doi.org/10.1007/978-3-540-69611-7_1. Accessed: Dec. 16, 2025. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-69611-7_1#citeas.

[6] D. MacIver, Z. Hatfield-Dodds, and M. Contributors, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, Nov. 21, 2019, ISSN: 2475-9066. DOI: 10.21105/joss.01891. Accessed: Dec. 5, 2025. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.01891.

[7] J. Wang and J. Wu, "Research on mobile application automation testing technology based on appium," in *2019 International Conference on Virtual Reality*

and Intelligent Systems (ICVRIS), Jishou, China: IEEE, Sep. 2019, pp. 247–250, ISBN: 978-1-7281-5050-5. DOI: 10.1109/ICVRIS.2019.00068. Accessed: Dec. 1, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8920799/.

[8] H. Goldstein et al., "Property-based testing in practice," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon Portugal: ACM, Apr. 12, 2024, pp. 1–13, ISBN: 979-8-4007-0217-4. DOI: 10.1145/3597503.3639581. Accessed: Nov. 28, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639581.

[9] S. Ravi and M. Coblenz, "An Empirical Evaluation of Property-Based Testing in Python," en, *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA2, pp. 3897–3923, Oct. 2025, ISSN: 2475-1421. DOI: 10.1145/3764068. Accessed: Feb. 26, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3764068.

[10] T. E. Schmidt, *Software testing*, https://www.ebsco.com/research-starters/computer-science/software-testing, Accessed: 2026-02-13, EBSCO Research Starters, 2018.

[11] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge: Cambridge University Press, 2016, ISBN: 978-1107172012.

[12] J. S. Collofello, "Introduction to software verification and validation," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, SEI Curriculum Module SEI-CM-13-1.1, 1988. [Online]. Available: https://www.sei.cmu.edu/documents/1541/1988_007_001_15695.pdf.

[13] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*, 4th. Boca Raton, FL: Auerbach Publications / CRC Press, 2013, ISBN: 9781466560680.

[14] E. T. Barr et al., "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015, ISSN: 0098-5589, 1939-3520. DOI: 10.1109/TSE.2014.2372785. Accessed: Mar. 6, 2026. [Online]. Available: http://ieeexplore.ieee.org/document/6963470/.

[15] 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)*, Lincoln, NE, USA: IEEE, Nov. 2015, pp. 429–440, ISBN: 978-1-5090-0025-8. DOI: 10.1109/ASE.2015.89. Accessed: Feb. 17, 2026. [Online]. Available: http://ieeexplore.ieee.org/document/7372031/.

[16] I. S. T. Q. Board, *Testing principles: Testing shows the presence, not the absence of defects*, https://istqb.org, Certified Tester Foundation Level (CTFL) Syllabus, 2024.

[17] G. J. Myers, *The art of software testing*, eng, 3rd ed. Hoboken, N.J: J. Wiley & Sons, 2012, ISBN: 978-1-118-03196-4 978-1-118-13315-6.

[18] J. Gao, H. S. Tsai, and Y. Wu, *Testing and Quality Assurance for Component-Based Software*. Norwood, MA, USA: Artech House, 2003, ISBN: 9781580535484.

[19] I. Dobles, A. Martinez, and C. Quesada-Lopez, "Comparing the effort and effectiveness of automated and manual tests," in *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*, Coimbra, Portugal: IEEE,

Jun. 2019, pp. 1–6, ISBN: 978-989-98434-9-3. DOI: 10.23919/CISTI.2019.8760848. Accessed: Feb. 24, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/8760848/.

[20] S. Anand et al., "An orchestrated survey of methodologies for automated software test case generation," en, *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013, ISSN: 01641212. DOI: 10.1016/j.jss.2013.02.061. Accessed: Feb. 18, 2026. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121213000563.

[21] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ACM, Sep. 2000, pp. 268–279, ISBN: 978-1-58113-202-1. DOI: 10.1145/351240.351266. Accessed: Dec. 5, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/351240.351266.

[22] R. Padhye, C. Lemieux, and K. Sen, "JQF: Coverage-guided property-based testing in Java," en, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Beijing China: ACM, Jul. 2019, pp. 398–401, ISBN: 978-1-4503-6224-5. DOI: 10.1145/3293882.3339002. Accessed: Mar. 9, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3293882.3339002.

[23] B. K. Aichernig and R. Schumi, "Property-based testing with FsCheck by deriving properties from business rule models," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Chicago, IL, USA: IEEE, Apr. 2016, pp. 219–228, ISBN: 978-1-5090-3674-5. DOI: 10.1109/ICSTW.2016.24. Accessed: Dec. 5, 2025. [Online]. Available: http://ieeexplore.ieee.org/document/7528965/.

[24] F.-Y. Lo, C.-H. Chen, and Y.-p. Chen, "Shrinking Counterexamples in Property-Based Testing with Genetic Algorithms," in *2020 IEEE Congress on Evolutionary Computation (CEC)*, Glasgow, United Kingdom: IEEE, Jul. 2020, pp. 1–8, ISBN: 978-1-7281-6929-3. DOI: 10.1109/CEC48606.2020.9185807. Accessed: Feb. 6, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/9185807/.

[25] L. Lampropoulos et al., "Beginner's luck: A language for property-based generators," en, in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, Paris France: ACM, Jan. 2017, pp. 114–129, ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009868. Accessed: Mar. 3, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3009837.3009868.

[26] L. Lampropoulos, M. Hicks, and B. C. Pierce, "Coverage guided, property based testing," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–29, OOPSLA Oct. 10, 2019, ISSN: 2475-1421. DOI: 10.1145/3360607. Accessed: Nov. 28, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3360607.

[27] E. De Angelis et al., "Property-Based Test Case Generators for Free," en, in *Tests and Proofs*, D. Beyer and C. Keller, Eds., vol. 11823, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019,

pp. 186–206, ISBN: 978-3-030-31156-8 978-3-030-31157-5. DOI: 10.1007/978-3-030-31157-5_12. Accessed: Mar. 3, 2026. [Online]. Available: http://link.springer.com/10.1007/978-3-030-31157-5_12.

[28] Y. Wei et al., "Stateful testing: Finding more errors in code and contracts," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Lawrence, KS, USA: IEEE, Nov. 2011, pp. 440–443, ISBN: 978-1-4577-1639-3 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100094. Accessed: Mar. 8, 2026. [Online]. Available: http://ieeexplore.ieee.org/document/6100094/.

[29] T. Arts et al., "Testing telecoms software with quviq QuickCheck," en, in *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, Portland Oregon USA: ACM, Sep. 2006, pp. 2–10, ISBN: 978-1-59593-490-1. DOI: 10.1145/1159789.1159792. Accessed: Mar. 8, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/1159789.1159792.

[30] L. E. Bueso De Barrio et al., "Makina: A new QuickCheck state machine library," en, in *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*, Virtual Republic of Korea: ACM, Aug. 2021, pp. 41–53, ISBN: 978-1-4503-8612-8. DOI: 10.1145/3471871.3472964. Accessed: Mar. 9, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3471871.3472964.

[31] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," en, *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, Aug. 2012, ISSN: 0960-0833, 1099-1689. DOI: 10.1002/stvr.456. Accessed: Mar. 9, 2026. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/stvr.456.

[32] J. Hughes, "Experiences with quickcheck: Testing the hard stuff and staying sane," in *A List of Successes That Can Change the World*, ser. Lecture Notes in Computer Science, S. Lindley et al., Eds., vol. 9600, Cham: Springer, 2016, pp. 169–183. DOI: 10.1007/978-3-319-30936-1_9.

[33] L. Nie et al., "A systematic mapping study for graphical user interface testing on mobile apps," en, *IET Software*, vol. 17, no. 3, pp. 249–267, Jun. 2023, ISSN: 1751-8806. DOI: 10.1049/sfw2.12123. Accessed: Feb. 13, 2026. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12123.

[34] S. Di Martino et al., "GUI testing of Android applications: Investigating the impact of the number of testers on different exploratory testing strategies," en, *Journal of Software: Evolution and Process*, vol. 36, no. 7, e2640, Jul. 2024, ISSN: 2047-7473, 2047-7481. DOI: 10.1002/smr.2640. Accessed: Feb. 16, 2026. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/smr.2640.

[35] S. Di Meglio et al., "Investigating the adoption and maintenance of web GUI testing: Insights from GitHub repositories," en, *Information and Software Technology*, vol. 189, p. 107 928, Jan. 2026, ISSN: 09505849. DOI: 10.1016/j.infsof.2025.107928. Accessed: Feb. 16, 2026. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584925002678.

[36] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving GUI-directed test scripts," in *2009 IEEE 31st International Conference on Software Engineering*,

Vancouver, BC, Canada: IEEE, 2009, pp. 408–418, ISBN: 978-1-4244-3453-4. DOI: 10.1109/ICSE.2009.5070540. Accessed: Feb. 16, 2026. [Online]. Available: http://ieeexplore.ieee.org/document/5070540/.

[37] D. Amalfitano et al., "AI in GUI-based testing: A survey of techniques, tools, and perceived advantages and limitations," en, *Journal of Systems and Software*, vol. 235, p. 112751, May 2026, ISSN: 01641212. DOI: 10.1016/j.jss.2025.112751. Accessed: Feb. 16, 2026. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121225004200.

[38] A. Adamoli et al., "Automated GUI performance testing," en, *Software Quality Journal*, vol. 19, no. 4, pp. 801–839, Dec. 2011, ISSN: 0963-9314, 1573-1367. DOI: 10.1007/s11219-011-9135-x. Accessed: Feb. 16, 2026. [Online]. Available: http://link.springer.com/10.1007/s11219-011-9135-x.

[39] M. Nass, E. Alégroth, and R. Feldt, "Why many challenges with GUI test automation (will) remain," en, *Information and Software Technology*, vol. 138, p. 106625, Oct. 2021, ISSN: 09505849. DOI: 10.1016/j.infsof.2021.106625. Accessed: Feb. 16, 2026. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584921000963.

[40] Android Developers, *Layouts in Views*, en, https://developer.android.com/develop/ui/views/layout/declaring-layout, Accessed: 2026-03-12.

[41] Android Developers, *Activity*, https://developer.android.com/reference/android/app/Activity, Accessed: 2026-03-13.

[42] J. Yan et al., "Widget-Sensitive and Back-Stack-Aware GUI Exploration for Testing Android Apps," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Prague, Czech Republic: IEEE, Jul. 2017, pp. 42–53, ISBN: 978-1-5386-0592-9. DOI: 10.1109/QRS.2017.14. Accessed: Dec. 9, 2025. [Online]. Available: http://ieeexplore.ieee.org/document/8009907/.

[43] R. Coppola, M. Morisio, and M. Torchiano, "Mobile GUI Testing Fragility: A Study on Open-Source Android Applications," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 67–90, Mar. 2019, ISSN: 0018-9529, 1558-1721. DOI: 10.1109/TR.2018.2869227. Accessed: Feb. 18, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/8477182/.

[44] P. Kong et al., "Automated Testing of Android Apps: A Systematic Literature Review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, Mar. 2019, ISSN: 0018-9529, 1558-1721. DOI: 10.1109/TR.2018.2865733. Accessed: Feb. 6, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/8453877/.

[45] I. A. Qureshi and A. Nadeem, "GUI Testing Techniques: A Survey," *International Journal of Future Computer and Communication*, pp. 142–146, 2013, ISSN: 20103751. DOI: 10.7763/IJFCC.2013.V2.139. Accessed: Feb. 18, 2026. [Online]. Available: http://www.ijfcc.org/index.php?m=content&c=index&a=show&catid=38&id=369.

[46] H. Singh et al., "GUI Testing Android Application," in *2022 10th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, Noida, India: IEEE, Oct. 2022, pp. 1–6, ISBN: 978-1-6654-7433-7. DOI: 10.1109/ICRITO56286.2022.9965072. Ac-

cessed: Dec. 10, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9965072/.

[47] Y. L. Arnatovich et al., "Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, Hamilton: IEEE, 2016, pp. 193–200, ISBN: 978-1-5090-5575-3. DOI: 10.1109/APSEC.2016.036. Accessed: Dec. 2, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/7890588/.

[48] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," en, in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Saarbrücken Germany: ACM, Jul. 2016, pp. 94–105, ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931054. Accessed: Jan. 27, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/2931037.2931054.

[49] Y. Lan et al., "Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning," en, in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon Portugal: ACM, Feb. 2024, pp. 1–13, ISBN: 979-8-4007-0217-4. DOI: 10.1145/3597503.3623344. Accessed: Feb. 18, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3623344.

[50] E. Collins et al., "Deep Reinforcement Learning based Android Application GUI Testing," en, in *Brazilian Symposium on Software Engineering*, Joinville Brazil: ACM, Sep. 2021, pp. 186–194, ISBN: 978-1-4503-9061-3. DOI: 10.1145/3474624.3474634. Accessed: Feb. 18, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3474624.3474634.

[51] M. Pan et al., "Reinforcement learning based curiosity-driven testing of Android applications," en, in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Event USA: ACM, Jul. 2020, pp. 153–164, ISBN: 978-1-4503-8008-9. DOI: 10.1145/3395363.3397354. Accessed: Feb. 18, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397354.

[52] L. V. Haoyin, "Automatic android application GUI testing—A random walk approach," *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), Wireless Communications, Signal Processing and Networking (WiSPNET), 2017 International Conference on*, pp. 72–76, Mar. 2017, Publisher: IEEE, ISSN: 978-1-5090-4442-9. DOI: 10.1109/WiSPNET.2017.8299722. [Online]. Available: https://research.ebsco.com/linkprocessor/plink?id=62655311-40a9-3a36-b18e-4e31323327bf.

[53] T. Gereziher, S. Gebrekrstos, and G. Gay, "Search-based test generation targeting non-functional quality attributes of android apps," in *Proceedings of the Genetic and Evolutionary Computation Conference*, Lisbon Portugal: ACM, Jul. 15, 2023, pp. 1518–1526, ISBN: 979-8-4007-0119-1. DOI: 10.1145/3583131.3590449. Accessed: Nov. 28, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3583131.3590449.

[54] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200,

Feb. 2002, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/32.988498. Accessed: Feb. 27, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/988498/.

[55] A. Zeller, *Why programs fail: a guide to systematic debugging*, 2nd ed. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2009, pp. 117–139, ISBN: 978-0-12-374515-6.

[56] M. Hammoudi et al., "On the use of delta debugging to reduce recordings and facilitate debugging of web applications," en, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo Italy: ACM, Aug. 2015, pp. 333–344, ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786846. Accessed: Feb. 27, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/2786805.2786846.

[57] L. Clapp et al., "Minimizing GUI event traces," en, in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seattle WA USA: ACM, Nov. 2016, pp. 422–434, ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950342. Accessed: Feb. 27, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/2950290.2950342.

[58] B. Jiang et al., "SimplyDroid: Efficient event sequence simplification for android application," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana, IL: IEEE, Oct. 2017, pp. 297–307, ISBN: 978-1-5386-2684-9. DOI: 10.1109/ASE.2017.8115643. Accessed: Feb. 19, 2026. [Online]. Available: http://ieeexplore.ieee.org/document/8115643/.

[59] Y. Sui et al., "Event trace reduction for effective bug replay of Android apps via differential GUI state analysis," en, in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn Estonia: ACM, Aug. 2019, pp. 1095–1099, ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3341183. Accessed: Feb. 27, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3341183.

[60] J. Yan et al., "Efficient testing of GUI applications by event sequence reduction," en, *Science of Computer Programming*, vol. 201, p. 102 522, Jan. 2021, ISSN: 01676423. DOI: 10.1016/j.scico.2020.102522. Accessed: Feb. 2, 2026. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167642320301301.

[61] Hypothesis Documentation, *Welcome to Hypothesis!* https://hypothesis.readthedocs.io/, Accessed: 2026-03-13.

[62] FsCheck Documentation, *FsCheck: Random Testing for .NET*, https://fscheck.github.io/FsCheck/, Accessed: 2026-03-13.

[63] M. Papadakis and K. Sagonas, "A PropEr integration of types and function specifications with property-based testing," en, in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, Tokyo Japan: ACM, Sep. 2011, pp. 39–50, ISBN: 978-1-4503-0859-5. DOI: 10.1145/2034654.2034663. Accessed: Mar. 3, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/2034654.2034663.

[64] J. Sun et al., "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*, San Francisco CA USA: ACM, Nov. 30, 2023, pp. 1088–1100, ISBN: 979-8-4007-0327-0. DOI: 10.1145/3611643.3616286. Accessed: Nov. 28, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3611643.3616286.

[65] T. Su et al., "Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs," *Proceedings of the ACM on Programming Languages*, vol. 5, pp. 1–31, OOPSLA Oct. 20, 2021, ISSN: 2475-1421. DOI: 10.1145/3485533. Accessed: Nov. 28, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3485533.

[66] Android Developers, *UI/Application Exerciser Monkey*, https://developer.android.com/studio/test/other-testing-tools/monkey, Accessed: 2026-03-13.

[67] RobotiumTech, *Robotium: Android UI Testing*, https://github.com/RobotiumTech/robotium, Accessed: 2026-03-13.

[68] Appium Documentation, *Welcome - Appium Documentation*, https://appium.io/, Accessed: 2026-03-13.

[69] Android Developers, *Write automated tests with UI Automator*, https://developer.android.com/training/testing/other-components/ui-automator, Accessed: 2026-03-13.

[70] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," en, in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, Saint Petersburg Russia: ACM, Aug. 2013, pp. 224–234, ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491450. Accessed: Mar. 13, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/2491411.2491450.

[71] Yuanchun Li et al., "DroidBot: A lightweight UI-Guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, Buenos Aires: IEEE, May 2017, pp. 23–26, ISBN: 978-1-5386-1589-8. DOI: 10.1109/ICSE-C.2017.8. Accessed: Mar. 13, 2026. [Online]. Available: http://ieeexplore.ieee.org/document/7965248/.

[72] T. Su, J. Wang, and Z. Su, "Benchmarking automated GUI testing for Android against real-world bugs," en, in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens Greece: ACM, Aug. 2021, pp. 119–130, ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3468620. Accessed: Mar. 8, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3468264.3468620.

[73] K. Peffers et al., "A Design Science Research Methodology for Information Systems Research," en, *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, Dec. 2007, ISSN: 0742-1222, 1557-928X. DOI: 10.2753/MIS0742-1222240302. Accessed: Apr. 29, 2026. [Online]. Available: https://www.tandfonline.com/doi/full/10.2753/MIS0742-1222240302.

[74] J. Vom Brocke, A. Hevner, and A. Maedche, "Introduction to Design Science Research," en, in *Design Science Research. Cases*, J. Vom Brocke, A. Hevner, and A. Maedche, Eds., Series Title: Progress in IS, Cham: Springer International Publishing, 2020, pp. 1–13, ISBN: 978-3-030-46780-7 978-3-030-46781-4. DOI: 10.1007/978-3-030-46781-4_1. Accessed: Apr. 29, 2026. [Online]. Available: http://link.springer.com/10.1007/978-3-030-46781-4_1.

[75] C. Sonnenberg and J. Vom Brocke, "Evaluations in the Science of the Artificial – Reconsidering the Build-Evaluate Pattern in Design Science Research," in *Design Science Research in Information Systems. Advances in Theory and Practice*, D. Hutchison et al., Eds., vol. 7286, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 381–397, ISBN: 978-3-642-29862-2 978-3-642-29863-9. DOI: 10.1007/978-3-642-29863-9_28. Accessed: May 6, 2026. [Online]. Available: http://link.springer.com/10.1007/978-3-642-29863-9_28.