Model-Guided Property-Based Testing of WeChat Pay at Billion-User Scale

Xiangchen Shen
East China Normal University
Shanghai, China
XiangchenShen@stu.ecnu.edu.cn

Yiting Wang
East China Normal University
Shanghai, China
octenewang@stu.ecnu.edu.cn

Ting Su*
East China Normal University
Shanghai, China
tsu@sei.ecnu.edu.cn

Jingjing Liang*
East China Normal University
Shanghai, China
jjliang@sei.ecnu.edu.cn

Jingling Sun
University of Electronic Science and Technology of China
Chengdu, China
jingling.sun910@gmail.com

Xixian Liang
East China Normal University
Shanghai, China
xixian@stu.ecnu.edu.cn

Haiying Sun
East China Normal University
Shanghai, China
hysun@sei.ecnu.edu.cn

Xinjie Xu
Tencent
Shenzhen, China
xinjiexu@tencent.com

Haochuan Lu
Tencent
Shenzhen, China
hudsonhclu@tencent.com

Yuetang Deng
Tencent
Shenzhen, China
yuetangdeng@tencent.com

Pengcheng Wang
Tencent
Shenzhen, China
hadeswang@tencent.com

Geguang Pu
East China Normal University
Shanghai, China
ggpu@sei.ecnu.edu.cn

Zhendong Su
ETH Zurich
Zurich, Switzerland
zhendong.su@inf.ethz.ch

John Hughes
Chalmers University of Technology
Gothenburg, Sweden
rjmh@chalmers.se

Abstract

Ensuring the correctness of mobile payment applications is critical, as even subtle faults can lead to severe financial losses. To ensure reliability, the WeChat Pay (a popular payment app with over one billion active users worldwide) team developed a User Acceptance Testing (UAT) system based on model-based testing (MBT). It maintains use cases specifying user operation sequences and expected app behaviors under business rules. UAT converts these use cases into formal models and generates test cases to validate the app. Although UAT achieves 100% state coverage by covering all user operations and business rules, it still suffers from two major limitations: *insufficient test coverage* and *predefined test oracles*.

In this paper, we introduce a model-guided property-based testing approach to improving the existing UAT system. We have two key observations: (1) UAT test cases can effectively reach all app states; and (2) WeChat Pay exhibits some invariant behaviors, which can be used to synthesize generic properties for bug finding. Building on these observations, our approach performs random exploration from UAT-discovered states, and validates app behaviors against the automatically synthesized properties. We also propose a greedy trace shrinking algorithm to reduce failure-inducing traces. We implemented our approach as UAT++, an advanced testing tool built upon the UAT system. Across 20,000 machine hours, UAT++ found 59 previously unknown bugs, including 48 logic bugs and 11 crash bugs (55 confirmed, 45 fixed). These bugs could not be found by the UAT system and classic property-base testing (PBT). UAT++ improved state transition coverage by 82.6% and 600.0% over the UAT system and classic PBT, respectively. The shrinking algorithm attained an average trace reduction rate of 79.9%. To date, UAT++ has been deployed in the continuous testing pipeline of WeChat Pay across Android, iOS, and HarmonyOS platforms.

CCS Concepts

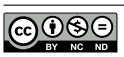
• **Software and its engineering → Software testing and debugging**.

Keywords

Property-based testing, Model-based testing, Mobile payment app testing, GUI Testing, Non-crashing functional bugs, Crash bugs

ACM Reference Format:
Xiangchen Shen, Yiting Wang, Ting Su, Jingjing Liang, Jingling Sun, Xixian Liang, Haiying Sun, Xinjie Xu, Haochuan Lu, Yuetang Deng, Pengcheng

*Ting Su and Jingjing Liang are the corresponding authors.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834537

Wang, Geguang Pu, Zhendong Su, and John Hughes. 2026. Model-Guided Property-Based Testing of WeChat Pay at Billion-User Scale. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3834537

1 Introduction

Digital transformation has shifted users from cash and in-person banking to mobile payment applications (*apps* for short). The adoption of mobile payment had increased over fivefold between 2015 and 2020, and grown from 0.4 to 2.3 billion users [54]. It is reported that the mobile payment market will reach USD 16.2 trillion by 2031 [17]. These trends have been driven by today's dominant mobile payment apps like WeChat Pay [46], Alipay [2], PayPal [33] and Google Pay [22]. For example, WeChat Pay, developed by Tencent, currently serves over 1 billion daily active users [53] and processes transactions exceeding USD 1 billion each day [19].

Like all software, payment apps can have bugs. Even subtle faults in these apps could lead to severe financial losses due to the huge user base and transaction volumes. For instance, in early 2025, a software fault in Alipay [7] erroneously applied an 80% discount to all transactions (miscategorized as government subsidy). This fault led to an estimated loss of USD 132 million in merely five minutes. Ensuring the correctness of payment apps is thus critical.

However, validating payment apps like WeChat Pay at billion-user scale is challenging. Existing app testing techniques [16, 30, 39, 42, 44] are ineffective because they struggle to exercise the complex business logic behind different payment scenarios. For example, WeChat Pay offers diverse payment scenarios, such as face-to-face, business, and social payments. Each scenario involves complex business logic (*e.g.*, authorization and discount rules). The challenge is further amplified by frequent code updates of WeChat Pay. Indeed, WeChat Pay encompasses about 120 core payment scenarios associated with over 10,000 business rules and undergoes bi-weekly updates.

To this end, the WeChat Pay team has developed a User Acceptance Testing (UAT) system [52] based on the idea of classic model-based testing (MBT) [15, 38]. Specifically, UAT maintains a number of well-structured use cases in a *restricted* form of natural language. Each use case comprehensively specifies the user operation sequences and the corresponding expected app behaviors of a payment scenario. These behaviors are defined under specific business rules and current app states. For instance, after five rounds of incorrect password attempts, the app should prompt the user to wait ten minutes.

In practice, the UAT testing process consists of three phases. First, UAT converts a use case into a formal model (similar to finite state machines). In this model, nodes represent app pages and edges capture their transitions triggered by user operations. Business rules are attached to some nodes to validate the correctness of that pages. Second, for each page, it generates a user operation sequence from the initial page to this page as a test case. Third, UAT executes the generated test cases and validates the reached page. A test case fails if the reached page fails the business rules (*e.g.*, the reached page is not intended or shows incorrect behaviors) or the app crashes, indicating a potential bug. UAT demands the generated test cases to cover *all* user operations and *all* business rules.

Although UAT has been deployed for effectively regression-testing WeChat Pay, it suffers from two major limitations. The *first* limitation is insufficient test coverage. To mitigate state-space explosion, UAT generates only one test case per page under each business rule, and ignores alternative operation sequences that could also reach the same page. As a result, UAT may miss exercising specific business logic which depends on other operation sequences. For example, the *password recovery* page can be reached by two different operation sequences corresponding to *fewer than five* or *more than five* rounds of incorrect password attempts. However, when UAT validates the *password recovery* page, it generates only one of these two sequences. It may fail to exercise specific business logic (*e.g.*, requiring a ten-minute wait to retry in the case of *more than five*).

The *second* limitation lies in the predefined test oracles. UAT relies on the oracles predefined for specific operation sequences. In other words, UAT test cases are example-based tests. For the sequences not covered by UAT test cases, UAT cannot infer how the operations would affect WeChat Pay, making validation infeasible. For instance, if a randomly generated sequence reaches the *incorrect password* page, UAT cannot know the current number of failed attempts. As a result, it cannot validate whether the page is correct (*e.g.*, whether a ten-minute warning should appear).

To address these limitations, this paper introduces a *model-guided property-based testing* approach to improve the existing UAT system. Property-based testing (PBT) [12] is a testing technique that validates programs against generic properties by automatically generating a large number of test inputs. It finds property violations without relying on a small number of manually crafted input-output examples. However, applying classic PBT to payment apps is challenging, as random input generation makes it difficult to exercise complex business logic without guidance.

Our *core* idea is to conduct property-based testing under the guidance of UAT's formal models (from the use cases). We have two key observations: (1) UAT test cases can effectively reach all app states due to the comprehensiveness of the model; and (2) WeChat Pay exhibits some invariant behaviors, which can be used to synthesize generic properties for bug finding. Building on these observations, our approach performs random exploration from UAT-discovered states while validating app behaviors against the synthesized properties. This design addresses the aforementioned two limitations. It achieves broader coverage guided by model-based testing while extending validation abilities beyond predefined oracles through property-based testing. Thus, our approach can uncover bugs hidden within complex business logic.

We realize our approach in three stages. First, we automatically synthesize one simple yet effective form of invariant, *page-transition invariants*, from use cases as properties. This type of invariant captures that performing some operation sequence on some page would always take the app to reach another page. Second, we use UAT test cases as guidance to systematically reach different states, then perform random exploration from these states to discover new transitions. We also use domain-specific input strategies to navigate through blocking pages (*e.g.*, authentication pages). During exploration, we validate app behaviors against synthesized properties for

bug detection. Third, we design a greedy trace shrinking algorithm to reduce failure-inducing traces to aid bug diagnosis.

We implemented and integrated our approach into the existing UAT system, evolving it into an advanced testing tool, UAT++. To evaluate the practical effectiveness of UAT++, we conducted an extensive empirical study on WeChat Pay, accumulating over 20,000 machine hours across four core use cases: Business Payment (QR code payment collection), Social Payment (payment collection), Face-to-Face Payment (QR code payment) and WeChat Transfer. UAT++ automatically synthesized 208 properties from these use cases. The evaluation results demonstrate UAT++'s effectiveness in bug detection, state space exploration, and failure trace shrinking. Specifically, UAT++ discovered 59 previously unknown bugs that could not be found by the existing UAT system, including 48 logic bugs and 11 crash bugs. Among them, 55 have been confirmed and 45 have been fixed by developers. Furthermore, we compare UAT++ with the existing UAT system and classic PBT on state coverage and state transition coverage. While UAT++ achieves the same state coverage as the existing UAT system (100%), it improves state transition coverage by 82.6% and 600.0%, respectively. Moreover, to facilitate debugging, our greedy trace shrinking algorithm effectively reduced failure-inducing traces by an average of 79.9%. UAT++ has been deployed in continuous testing pipeline of WeChat Pay across Android, iOS, and HarmonyOS platforms.

In summary, this paper makes the following contributions:

- At the conceptual level, we introduce a model-guided property-based testing approach to effectively validate WeChat Pay.
- At the technical level, we design (1) an automated algorithm that synthesizes properties from page-transition invariants in UAT use cases; (2) a model-guided exploration strategy achieving better coverage by performing random exploration from states discovered by the model; and (3) a greedy trace shrinking algorithm to facilitate bug diagnosis.
- At the engineering level, we have implemented our approach as an automated testing tool UAT++ and deployed it in the daily continuous testing pipeline of WeChat Pay.
- At the empirical level, we have demonstrated the practicality of UAT++ through large-scale empirical evaluations on WeChat Pay. UAT++ discovers 59 previously unknown bugs, improves state transition discovery by 82.6% over the UAT system, and achieves an average failure-inducing trace reduction of 79.9%.

2 Background

2.1 Model-based Testing

Model-Based Testing (MBT) [15, 38] systematically generates functional test cases from abstract behavioral models of the system under test (SUT) [11, 14, 48]. A standard MBT workflow proceeds in three phases: (1) *model construction*, building a model to abstract representation of the SUT; (2) *test generation*, generating executable test cases from the model to maximize coverage; and (3) *execution and validation*, validating actual behavior against the test oracles defined by the model. MBT commonly evaluates test adequacy using two coverage metrics: *(1) state coverage*, which measures how many distinct states are visited, and *(2) state transition coverage*, which measures how many state transitions are executed by the generated test cases. In mobile app testing, many existing works apply MBT by building GUI state machines to validate the correctness of apps and uncover bugs [5, 8, 23, 39, 55].

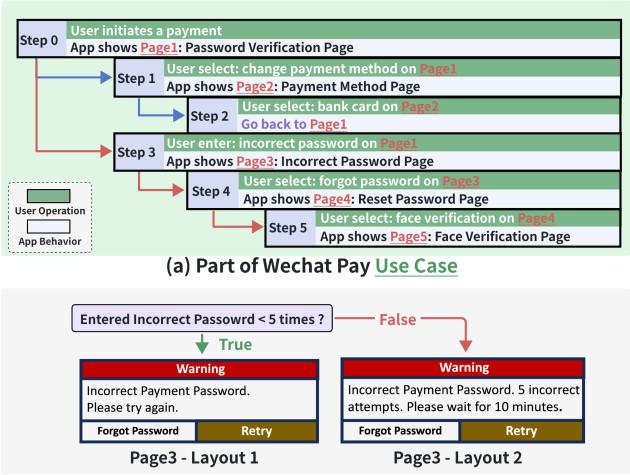


Figure 1: Example of Use Cases in Business Payment

2.2 Property-based Testing

Property-based testing (PBT) [12, 21] is a testing paradigm that evaluates system correctness against generic properties. It has been widely applied to uncover functional bugs across diverse software systems [4, 25, 29, 31]. The core workflow of PBT comprises three phases: (1) *property specification*, defining properties the SUT must satisfy; (2) *input generation*, automatically generating diverse inputs and executing them to check property violations; and (3) *test reduction*, minimizing failure-inducing inputs into their simplest reproducible forms. When adapting PBT to the functional testing of mobile apps [28, 42, 44, 56], a property is typically formalized as a triple $\phi = \langle Pre, I, Post \rangle$ [56]. For example, entering an incorrect password (I) on a validation page (Pre) must yield an error warning ($Post$). Any execution trace violating such defined properties is reported as a bug.

3 Motivation

This section introduces the UAT system and analyzes its limitations in testing WeChat Pay. It motivates our model-guided property-based testing approach.

UAT System. WeChat Pay is a popular payment app with over 1.4 billion users. To ensure reliability, it employs a User Acceptance Testing (UAT) system [52], a typical implementation of model-based testing in industrial practice. In this system, *use cases* serve as specifications to describe payment scenarios by characterizing the complete business workflow. Each use case is written in a *restricted form* of natural language. Specifically, a use case capture the interactions between user operations and expected app behaviors under specific *business rules*. *Models* are abstracted from these use cases to represent pages and their transitions triggered by user operations. *UAT test cases* are then generated from models with expected app behaviors serving as test oracles to validate business logic. Specifically, the UAT system generates one test case for *every* business rule attached to each node and for *every* user operation on each

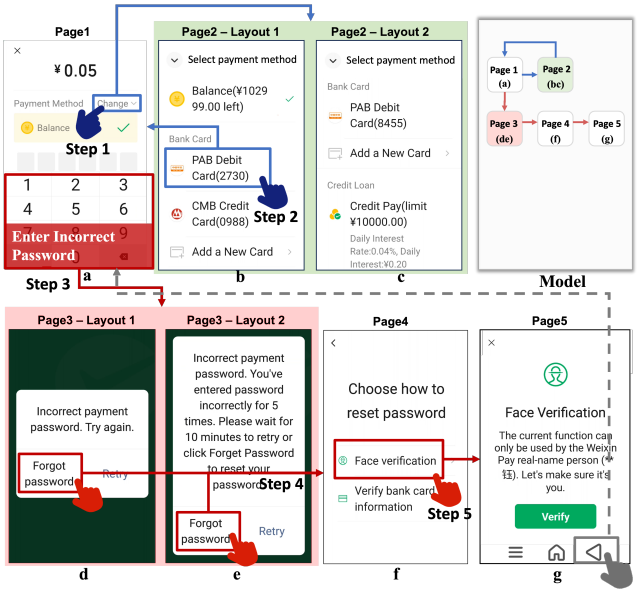


Figure 2: Example Trace of WeChat Pay

edge, thereby covering all business rules and all user operations in the model.

Figure 1a shows part of a use case for Business Payment containing six steps, with corresponding pages in Figure 2. Each step specifies a user operation and WeChat Pay's behavior. Step 0 shows the user entering the *password verification* page (Figure 2a). Step 1 navigates to Page2 when tapping "Change" (Figure 2b), and Step 2 returns to Page1 after selecting a bank card. Steps at the same indentation level in Figure 1 represent alternative paths: from Page1, users can either change the payment method (Steps 0→1→2, blue lines) or recover password (Steps 0→3→4→5, red lines). Most steps have associated business rules that determine the page rendering (*i.e.*, layout) based on the app state. As shown in Figure 1b, Step 3 (entering incorrect password) navigates to Page3 with a rule: if incorrect attempts fewer than 5, display Layout 1 (Figure 2d); otherwise display Layout 2 (Figure 2e). Similar to Page3, Page2 has two distinct layouts (Figure 2c and Figure 2d) based on the business rule of whether the user qualifies for credit loans.

The UAT system converts each use case into a model. As illustrated in Figure 2*Model*, nodes represent pages, and edges represent transitions triggered by user operations. Business rules associated with each page define its correct rendering under the current app state, against which the page's correctness is validated. Based on the model, UAT generates one test case per page under each business rule. For example, in Figure 2, eight test cases are generated: a→b, a→c, a→b→a, a→c→a, a→d, a→e, a→d→f, and a→d→f→g. This ensures coverage of all user operations and business rules, achieving 100% state coverage (*i.e.*, all layouts are covered). UAT then executes the generated test cases and validates the reached pages. A test case fails if the reached page violates business rules (e.g., the page is unintended or exhibits incorrect behavior) or if the app crashes, indicating a bug.

Although UAT has been deployed for effectively regression-testing WeChat Pay, it suffers from two major limitations.

Limitation-1: Insufficient Test Coverage. To mitigate state space explosion, UAT generates only one test case per page under each business rule, ignoring alternative operation sequences that could also reach the same page. This may miss business logic that depends on specific operation sequences, particularly in two aspects: (1) The UAT system cannot cover all combinations of business rules across different pages. As illustrated in Figure 2, Page2 and Page3 each contain a business rule. However, since UAT does not generate test cases that cover both Page2 and Page3, the system cannot evaluate their combined effects. For instance, test cases involving both Page2 and Page3, such as a→c→a→e, trigger both rules sequentially. In our evaluation, the absence of such test cases leads to an unexpected error ("reading 'BaseResponse' of undefined") on page3. (2) For cycle paths, the UAT system executes each cycle only once. For example, it tests a→b but never explores repeated iterations such as a→b→a→b. This may miss bugs that manifest only after multiple iterations. The UAT system also does not consider backward navigation. As shown in Figure 2, it generates forward paths (*e.g.*, a→d→f→g) but not backward ones (*e.g.*, a→d→f→g→a), missing common back-navigation cases. In our evaluation, clicking *back* button within Figure 2g results in an abnormal transaction termination.

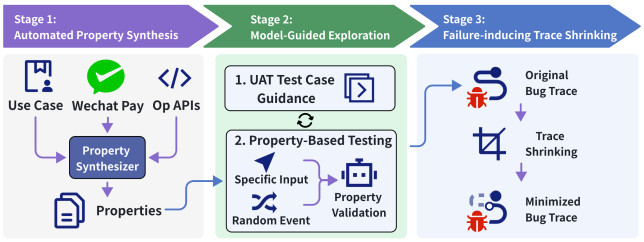


Figure 3: Overview of UAT++

Limitation-2: Limited to Predefined Test Oracles. UAT can only validate operation sequences predefined in UAT test cases, as it cannot infer how operations affect the app state. For example, given an initial state of five failed password attempts, a UAT test case a→e with an oracle expecting a ten-minute retry prompt on layout e. However, event if a new test case a→b→a→e is generated that is not covered by UAT test cases, no oracle exists. UAT cannot determine whether the additional operations b→a reset the failure counter, and thus cannot validate the reached layout e.

4 Approach

Our approach synthesizes properties from *use cases* of WeChat Pay and outputs failure-inducing event traces along with the violated properties or crash stack traces. To accomplish this, our approach works in three stages, as shown in Figure 3. *Stage 1* first extracts page-transition invariants from use cases, and then translates them into executable properties by leveraging UI layouts dynamically collected during WeChat Pay execution and operator APIs provided by the UAT system (Section 4.2). *Stage 2* iteratively executes each UAT test case to reach a target state, then performs property-based testing from that state to detect property violations and crashes (Section 4.3). *Stage 3* reduces failure-inducing traces via a greedy shrinking algorithm (Section 4.4).

4.1 Definitions

This section presents the basic definitions used in our approach.

Definition 1. **UI Layout.** A UI layout L represents a specific UI state of the app, formalized as a structural snapshot organized as a hierarchy of widgets. Each widget $w \in L$ is characterized by a set of attributes such as *resource-id*, *text*, *description*, and *clickable*. Let $\mathcal{L}$ denote the set of all possible layouts in the app.

Definition 2. **Page.** A page P represents a logical entity corresponding to a specific business function in the app. Each page is characterized by two types of widgets: *state-independent widgets* IW that remain consistent regardless of app state and serve as its unique identifier, and *state-dependent widgets* DW whose presence or attribute values vary with app state. We use $\mathbb{I}(P)$ to denote the instantiation set of page P, i.e., the set of all concrete layouts that instantiate P. Formally, a layout $L \in \mathbb{I}(P)$ if and only if it contains all widgets in IW and a subset $DW_L \subseteq DW$, whose attribute values are determined by the runtime state. Let $\mathcal{P}$ denote the set of all pages in the app.

Definition 3. **GUI Event.** The dynamic execution of the app is driven by GUI events. Formally, a GUI event e is defined as a triple: $e = \langle \alpha, w, d \rangle$, where $\alpha \in \mathcal{A} = \{click, input, swipe, back \dots\}$ represents the specific operation type. $w \in L \cup \{\emptyset\}$ denotes the target widget on which the operation α is performed, and d represents the optional data payload associated with the operation.

Example. In Figure 2, Page3 represents the *Incorrect password* Page P_3. Its state-independent widgets (IW) consist of $\{text : Forgot\ password, text : Retry\}$, while its state-dependent widgets (DW) include dynamic error messages such as *"Incorrect payment password. Try again."*. Layouts d and e are concrete instantiations of P_3, denoted as $L_d, L_e \in \mathbb{I}(P_3)$. Consider layout L_e, where w_{forgot} represents the clickable *"Forgot password"* button. The user operation of clicking this button is formalized as a GUI event $e = \langle \text{click}, w_{forgot}, \emptyset \rangle$.

Definition 4. **Event Trace.** An event trace E is defined as a trace of GUI events, denoted as $E = [e_1, e_2, \dots, e_n]$. The execution of E on an app triggers a series of UI transitions, producing a corresponding trace of UI layouts $[L_0, L_1, \dots, L_n]$. This dynamic process can be formally represented by the transition chain: $L_0 \xrightarrow{e_1} L_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} L_n$, where L_i represents the resulting UI layout after executing event e_i on the preceding layout L_{i-1} $(1 \le i \le n)$. For brevity, we use the notation $L_n = E(L_0)$ to signify the outcome of applying the entire event trace E to the initial layout L_0.

Definition 5. **Operator API.** In the engineering context of WeChat Pay, testing operations are often encapsulated into higher-level functions known as Operator APIs. An Operator API, denoted as op, is a programmatic wrapper that executes a specific business function. Depending on the complexity of the function, invoking an op generates a finite trace of one or more continuous GUI events, denoted as $E_{op} = [e_1, \dots, e_k]$ $(k \ge 1)$. The testing process can be driven either by single, independent GUI events or by invoking Operator APIs.

Example. Consider the password entry process a→d, shown in Figure 2. This process is driven by an event trace $E = [e_1, \dots, e_6]$, where each e_i represents a click event on the keypad to input a 6-digit password. Executing E from the layout L_a triggers a sequence of UI transitions, ultimately reaching layout L_d, denoted as $L_d = E(L_a)$. In the engineering context of WeChat Pay, developers encapsulate this entire sequence into a single Operator API, such as op_{input} = InputIncorrectPassword(). Invoking this op executes the exact same continuous event trace $E_{op} = [e_1, \dots, e_6]$.

Definition 6. **Use Case.** A use case u represents a specific payment scenario, which we formally define as a transition-based model [47] $u = (\mathcal{S}, \mathcal{O}, \delta, s_0, \mathcal{F})$, where $\mathcal{S} \subseteq \mathcal{P}$ is a finite set of pages, $\mathcal{O}$ is a finite set of Operator APIs representing the transitions, $\delta : \mathcal{S} \times \mathcal{O} \to \mathcal{S}$ is the transition function, $s_0 \in \mathcal{S}$ is the initial page, and $\mathcal{F} \subseteq \mathcal{S}$ is the set of terminal pages. Each use case captures all possible execution flows of an app scenario as transitions between pages driven by Operator APIs. Let $\mathcal{U}$ denote the set of use cases for the app.

Example. Figure 1 illustrates a part of the use case for Business Payment in the UAT system. To demonstrate the modeling process, we consider a simplified space only covered pages in Figure 2. The space $\mathcal{S} = \{P_1, P_2, \dots, P_5\}$. The initial page is $s_0 = P_1$, and the scenario terminates at either success or failure, making $\mathcal{F} = \{P_5\}$. The transitions are driven by a set of Operator APIs $\mathcal{O} = \{ op_{change}$ = ChangePaymentMethod(), op_{select} = SelectPaymentMethod(), op_{input} = InputIncorrectPassword(), …$\}$. The transition function δ formally maps these executions: for instance, $\delta(P_1, op_{change}) = P_2$, $\delta(P_1, op_{input}) = P_4$.

Definition 7. **App Property.** A property is defined as a triple $\phi = \langle Pre, I, Post \rangle$, where the precondition Pre and postcondition $Post$ are predicates over layouts, and the interaction scenario I denotes a sequence of Operator APIs. The property ϕ specifies:

$$L_0 \models Pre \wedge L_n = E(L_0) \Rightarrow L_n \models Post \tag{1}$$

That is, if the precondition holds in layout L_0 and executing event sequence E from L_0 yields layout L_n, the property ϕ holds if and only if the postcondition holds in L_n.

4.2 Automated Property Synthesis

Our approach synthesizes properties from UAT use cases by extracting *page-transition invariants*. Given an initial page P_{init} and an event trace E, a page-transition invariant, denoted $P_{target} = E(P_{init})$, holds if and only if:

$$\forall L \in \mathbb{I}(P_{init}), \exists! P_{target} \in \mathcal{P} : E(L) \in \mathbb{I}(P_{target}) \tag{2}$$

Here, $\exists!$ denotes uniqueness quantification [34]. That is, executing E from any layout instance of P_{init} always reaches the same target page P_{target}, regardless of the runtime state. We require the uniqueness of P_{target} because only a deterministic transition can serve as a reliable oracle: if executing E could reach different pages depending on hidden states, the transition would not qualify as an invariant.

(1) Page-Transition Invariants Extraction. This stage extracts page-transition invariants by exploring transition paths within use cases. In a use case, each page carries a unique name assigned by UAT engineers (*e.g.*, *Page 1* and *Page 2* in Figure 2), and its transitions are encoded by δ. For each use case $u \in \mathcal{U}$, Algorithm 1 explores all execution paths up to $\mathcal{N}$ steps (*i.e.*, $\mathcal{N}$ Operator API calls) and outputs a set of transition invariants Δ. Specifically, the algorithm maintains a mapping $\mathcal{R}$ that records, for each pair of initial page P_{init} and event trace E, all target pages reached during exploration. The `Main` function first extracts all pages from the use case u (Line 4) and then initiates traversal from each page to obtain all reachable target pages up to $\mathcal{N}$ steps (Lines 5-7). Finally, for

Algorithm 1: Page-Transition Invariants Extraction

Input: $\mathcal{N}$: Maximum transition steps
u: a use case
Output: Δ: Set of extracted page-transition invariants

1 $\mathcal{R} \leftarrow \{\}$; // Map from (P_{init}, E) to all reachable pages $\mathcal{P}_{reached}$
2 $\Delta \leftarrow \emptyset$;
3 **Function** `Main()`:
4 $\mathcal{P}_{all} \leftarrow$ ExtractAllPages(u);
5 **foreach** $P \in \mathcal{P}_{all}$ **do**
6 `TraverseTransitions`$(P, P, 0, \langle\rangle, u)$;
7 **end**
8 **foreach** $((P_{init}, E), \mathcal{P}_{reached}) \in \mathcal{R}$ **do**
9 **if** $|\mathcal{P}_{reached}| == 1$ **then**
10 $P_{target} \leftarrow$ the unique element in $\mathcal{P}_{reached}$;
11 $\Delta \leftarrow \Delta \cup \{P_{target} = E(P_{init})\}$;
12 **end**
13 **end**
14 **return** Δ;
15 **return**
16 **Function** `TraverseTransitions`$(P_{init}, P_{curr}, depth, E, u)$:
17 **if** $depth == \mathcal{N}$ **then**
18 $\mathcal{R}[(P_{init}, E)] \leftarrow \mathcal{R}[(P_{init}, E)] \cup \{P_{curr}\}$;
19 **return**;
20 **end**
21 **if not** IsExitPage(P_{curr}) **then**
22 $\mathcal{T} \leftarrow$ GetTransitions(P_{curr}, u);
23 **foreach** $(op, P_{next}) \in \mathcal{T}$ **do**
24 $E = E.append(op)$;
25 `TraverseTransitions`$(P_{init}, P_{next}, depth + 1, E, u)$;
26 **end**
27 **end**
28 **return**

each pair of initial page and event trace, if it consistently reaches exactly one target page, the corresponding transition is extracted as an invariant transition (Lines 8-13). The `TraverseTransitions` function recursively explores paths from each page. At each step, it invokes `GetTransitions`(P_{curr}, u), which returns the set of outgoing transitions $\{(op, P_{next})\}$ of P_{curr} directly from the use case u, and recursively explores each subsequent page P_{next}, building the event trace E incrementally (Lines 21-27). The exploration is bounded by $\mathcal{N}$, when reaching $\mathcal{N}$ steps, traversal terminates and records the reached page in $\mathcal{R}$ (Lines 17-20).

(2) Property Synthesis. This phase transforms page-transition invariants $P_{target} = E(P_{init})$ into executable properties. To determine whether a runtime layout L matches a logical page P (*i.e.*, $L \in \mathbb{I}(P)$), we extract *state-independent widgets* (IW) as unique identifiers for each page. We achieve this through two strategies:

- *Static Derivation of Basic Widget.* In WeChat Pay, developers use the `waitUI(target_widgets)` function in Operator APIs to ensure prerequisite UI widgets are rendered. For each transition $P \xrightarrow{op} P'$ in the use cases, we statically parse the `waitUI` arguments within the source code of op. These arguments represent the UI elements that must be present for the API to execute, thereby forming the statically derived state-independent widget set IW_s. Since developers typically specify only minimal widgets necessary for their test cases, this static set is incomplete.
- *Dynamic Augmentation via Widget Intersection.* We dynamically capture UI hierarchies immediately before each Operator API executes across all UAT test cases. Since the UAT test cases achieve 100% state coverage, executing all test cases ensures that we collect IW for each page. We then compute the intersection of all captured hierarchies for a given page, where consistently appearing widgets constitute the dynamic set IW_d.

Combining both strategies yields the final state-independent widget set for each page: $IW = IW_s \cup IW_d$.

Example. Consider the example in Figure 2. Regardless of the preceding execution context (*e.g.*, paying with a wallet balance or a bank card), inputting an incorrect password (op_{input}) on the *password verification* page P_1 consistently navigates to the *incorrect password* page P_3, This constitutes a single-step page-transition invariant $P_1 \xrightarrow{op_{input}} P_3$. Through our extraction strategies, we identify the state-independent widgets for both pages: $IW_1 = \{$*text: Payment Method*, *text: Change*, *desc: Password Box*, *desc: Close*$\}$ for P_1, and $IW_3 = \{$*text: Forgot password*, *text: Retry*$\}$ for P_3. From this transition, we synthesize property $\phi = \langle Pre, op_{input}, Post\rangle$, where Pre requires layout $L \in \mathbb{I}(P_1)$ to contain IW_1, and $Post$ requires layout $L' \in \mathbb{I}(P_3)$ to contain IW_3. The property ϕ holds if and only if for all layouts L satisfying Pre, executing op_{input} yields a layout L' satisfying $Post$. Any violation of this condition indicates a potential bug in the app.

4.3 Model-Guided Exploration

To effectively explore the complex state space of WeChat Pay, we propose a model-guided exploration approach that alternates iteratively between two phases. The first phase leverages UAT test cases to systematically reach different app states, while the second phase performs property-based testing by generating random event traces from these states and validating synthesized properties. These two phases iterate continuously until all UAT test cases have been executed.

(1) Model-Guided State Initialization. Each iteration begins by executing a predefined UAT test case as an initial guiding sequence. This phase systematically traverses complex business logic paths, advancing the app into all reachable states that would be difficult to reach through random exploration alone.

(2) Property-Based Testing. Once the UAT test case execution completes and establishes an initial state, the testing engine transitions into a continuous exploration loop that interleaves two steps (shown in Figure 4):

- *Event Generation.* The engine evaluates the current UI layout (L_{curr}) to determine the next event. If L_{curr} matches a predefined blocking layout (e.g., password prompts), the engine injects a domain-specific input event extracted from the payment model to bypass the barrier. Otherwise, the engine generates a random GUI event to explore deeper states.
- *Property Validation.* If L_{curr} satisfies the precondition (Pre) of any property ϕ, the engine intercepts exploration with a 50% probability. It executes the corresponding interaction scenario (I), i.e., invoking the associated operator APIs, and asserts the postcondition ($Post$) to validate the property. Here, we following

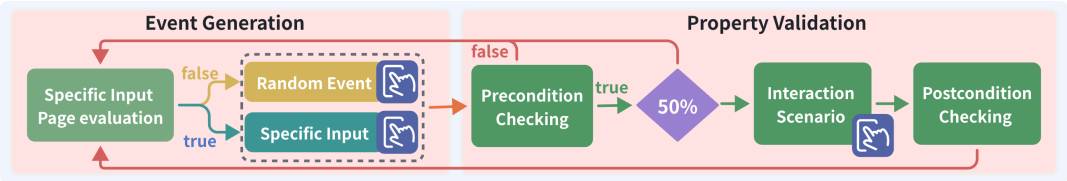


Figure 4: Property-based Testing Loop

prior work [56] to set the probabilistic, this is explicitly designed to balance validating specific properties with exploring new app states.

The exploration continues until reaching a terminal state (e.g., payment finished) or exhausting the allocated time budget, at which point the engine returns to first phase to initialize from the next UAT test case. Throughout this exploration process, a runtime monitor continuously records the concatenated event trace E. When a crash or property violation is detected, the complete failure-inducing trace is immediately captured and saved.

4.4 Failure-inducing Trace Shrinking.

We propose a greedy trace shrinking algorithm to reduce failure-inducing execution traces. Formally, given an event trace E that triggers a bug (*e.g.*, a crash or property violation), the objective is to identify a reduced subsequence $E' \subseteq E$ that reproduces the original bug. Before formalizing our shrinking operators, we note that mobile GUI event replay is non-deterministic: due to timing and asynchronous updates, a widget may be present in one execution but absent in another, even for the same event sequence. We formalize two trace manipulation operators accordingly.

Definition 8. **Trace Truncation.** Given an event sequence $E = [e_1, \ldots, e_k, \ldots, e_n]$, $\text{trunc}(E)$ returns the longest executable prefix of E from the initial state *with respect to a given execution*. Specifically, $\text{trunc}(E) = [e_1, \ldots, e_k]$, where k is the largest index such that every event in this prefix applies to a widget present in that execution. Since replay is non-deterministic, k may vary across executions;

Definition 9. **Event Elimination.** We define the elimination operator $\ominus$ to remove a contiguous subsequence from an event trace. Given a trace $E_a = [e_1, \ldots, e_n]$ and a subsequence $E_b = [e_i, \ldots, e_j]$ $(1 \le i \le j \le n)$, the elimination $E_a \ominus E_b$ yields a new trace comprising the prefix $[e_1, \ldots, e_{i-1}]$ followed by executable events from the suffix $[e_{j+1}, \ldots, e_n]$. The resulting trace is formally defined as $E_a \ominus E_b = \text{trunc}([e_1, \ldots, e_{i-1}, e_{j+1}, \ldots, e_n])$.

(1) Operator API Cycle Shrinking. The execution of an Operator API (op) is strictly conditional, requiring a specific prerequisite page. Therefore, if an event trace $E = [e_1, \ldots, op_i, \ldots, e_{j-1}, op_j, \ldots e_n]$ contains two invocations of the same API (op_i and op_j, where $i < j$), their preceding prerequisite layouts must be equivalent. This equivalence dictates that the intermediate trace $[op_i, \ldots, e_{j-1}]$ constitutes a redundant cycle that merely restores the layout. To maximize the excised length, we identify the smallest index i satisfying this condition. Thus, E is reduced by excising the cycle:

$$E' = E \ominus [op_i, \ldots, e_{j-1}] \quad \text{where} \quad op_i = op_j \tag{3}$$

(2) Widget Cycle Shrinking. Formally, let e_i be an event interacting with a target widget w_{target} on the current layout L_{i-1}. If w_{target} is already present in an earlier layout L_{k-1} $(1 \le k < i)$, it indicates that the intermediate event sequence $[e_k, \ldots, e_{i-1}]$ is functionally redundant with respect to the availability of w_{target}. To maximize the excised length, we identify the smallest index k satisfying this condition. Consequently, the original trace E can be optimized by directly applying e_i to the earlier layout L_{k-1}, yielding the reduced trace E':

$$E' = E \ominus [e_k, \ldots, e_{i-1}] \quad \text{where} \quad w_{target} \in L_{k-1} \cap L_{i-1} \tag{4}$$

We apply these strategies iteratively to reduce failure-inducing traces. During each iteration, the algorithm prioritizes the *Operator API Cycle Shrinking*, proceeding to the *Widget Cycle Shrinking* only when no further Operator API cycle can be identified. Since excising events may disrupt implicit dependencies, each reduced trace E' is validated via dynamic replay. If E' fails to reproduce any bug, it is discarded; otherwise, it is retained for further iterative shrinking. This process terminates when no further reductions exist, yielding the reduced trace. Two event traces are considered to expose the same bug if they violate the same property.

Example. We illustrate these strategies using a practical payment scenario in WeChat Pay: adding a bank card. The initial failure-inducing trace is depicted below, where the test initiates at the *password verification* layout (L_0). An Operator API (op_{change}) navigates to the *change payment method* layout (L_1), but a subsequent Back event (e_{back}) reverts the app to the *password verification* layout (L_2). The Operator is invoked again (op_{change}), followed by an add bank card event (e_{add}) reaching the *add bank card* layout (L_4). A failed submission (e_{submit}) fails to trigger a page transition, yielding layout L_5, and ultimately, inputting the card number (e_{card}) triggers a bug. The initial failure-inducing trace is: $L_0 \xrightarrow{op_{change}} L_1 \xrightarrow{e_{back}} L_2 \xrightarrow{op_{change}} L_3 \xrightarrow{e_{add}} L_4 \xrightarrow{e_{submit}} L_5 \xrightarrow{e_{card}}$ **BUG**.

In the provided trace, the Operator API op_{change} is invoked twice, indicating that their respective prerequisite layouts are equivalent. The subtrace $L_0 \xrightarrow{op_{change}} L_1 \xrightarrow{e_{back}} L_2$ forms a redundant cycle. Eliminating this cycle yields a shorter candidate trace: $L_0 \xrightarrow{op_{change}} L_3 \xrightarrow{e_{add}} L_4 \xrightarrow{e_{submit}} L_5 \xrightarrow{e_{card}}$ **BUG**. Upon validating that this reduced candidate trace successfully reproduces the original bug, the algorithm detects that there is no Operator API cycle present; then it proceeds to apply *widget cycle shrinking*. Within the reduced trace, the bug is triggered by e_{card} on layout L_5. However, the target widget w_{card} required by e_{card} is already present in the preceding layout L_4. So, we remove it and yield the reduced failure-inducing trace: $L_0 \xrightarrow{op_{change}} L_3 \xrightarrow{e_{add}} L_4 \xrightarrow{e_{card}}$ **BUG**. After validating that the reduced candidate trace can still reproduce the target failure, the algorithm finds that there is no widget cycle present. It then terminates and outputs the reduced failure-inducing trace.

5 Evaluation

Our evaluation aims to answer these research questions:

- **RQ1 (Bug Discovery):** How effective is UAT++ in bug finding?
- **RQ2 (Exploration Capability):** Does UAT++ outperform UAT and classic PBT in state and state transition coverage?
- **RQ3 (Trace Shrinking):** How do the proposed greedy shrinking strategies perform in reducing failure-inducing traces?

5.1 Tool Implementation

We implemented and integrated our approach into the existing UAT system, evolving it into an advanced testing tool UAT++ across Android, iOS and HarmonyOS platforms. UAT++ comprises three modules: property synthesis, model-guided exploration, and trace shrinking. The implementation consists of approximately 15,000 lines of Go code and 6,600 lines of Python code. The empirical evaluation targets four core use cases (*i.e.*, Social Payment, Business Payment, Face-to-Face Transfer, and WeChat Transfer) within WeChat Pay[1]. UAT++ synthesized properties based on single-step page-transition invariants, generating a total of 208 properties (see the first two columns in Table 1). The experiments were conducted on a cluster of 2,000 Android 12.0 devices, continuously testing the internal daily builds of WeChat Pay. A total of 4,610 existing UAT test cases were used as exploration guidance, with 0.5 machine hours allocated to each for property-based testing.

5.2 Experiment Setup

Baselines. We compared UAT++ against two established baselines: *(1) UAT*, which executes only existing test cases; and *(2) Classic PBT* randomly selects an event at each step and performs property validation with 50% probability, representing a purely random exploration strategy without guidance.

Setup for RQ1. To answer RQ1, we define one testing round as executing all 4,610 existing UAT test cases. We conducted 10 such rounds, consuming approximately 20,000 machine hours in total. All identified bugs were reported to the WeChat Pay team for confirmation and remediation.

Setup for RQ2. To evaluate the exploration capability of UAT++, we compared it against two baselines. Each approach was executed continuously for 120 hours. Because executing the existing UAT test cases requires 60 hours, we allocated an equivalent 60 hours for property-based testing. We measured exploration adequacy using two metrics: *(1) State Coverage*, the number of distinct UI layouts covered; and *(2) State Transition Coverage*, the number of unique transitions between UI layouts covered.

Setup for RQ3. To assess the effectiveness of our trace shrinking algorithm, we evaluated it on 20 randomly selected bugs identified by UAT++ during RQ1. We measured two aspects: *(1) Reduction Effectiveness*, the length difference between original and reduced traces; and *(2) Algorithmic Efficiency*, the total number of shrinking iterations required. We use iteration count rather than wall-clock time because GUI event replay is inherently non-deterministic and subject to device-specific overhead.

[1]WeChat Pay's use cases are under continuous construction; each use case will be tested after it has been built.

Table 1: Bugs Found by UAT++ in four Use Cases

Use Cases	Properties	#Bugs
Business Payment *(QR code payment collection)*	64	35
Social Payment *(payment collection)*	79	15
Face-to-Face Payment *(QR code payment)*	21	6
WeChat Transfer	44	3
Total	**208**	**59**

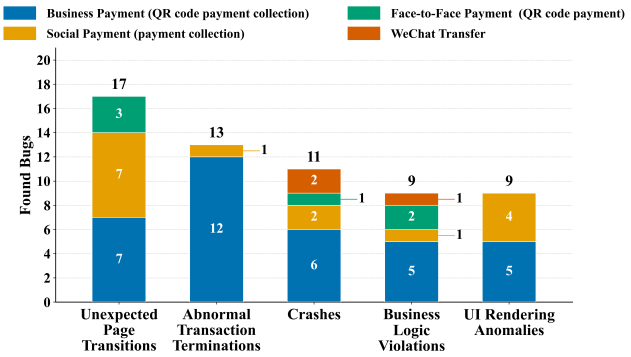


Figure 5: Different Types of bugs found in four Use Cases

5.3 Experiment Results

Results for RQ1. Table 1 shows the number of synthesized properties and the previously unknown bugs found by UAT++ across four distinct use cases. Overall, UAT++ found a total of 59 unique and previously unknown bugs (55 confirmed, 45 fixed). Specifically, the numbers of bugs discovered in these scenarios are 35, 15, 6, and 3, respectively. Neither UAT nor classic PBT is able to detect any of these bugs. Notably, the number of discovered bugs correlates with the functional complexity of each scenario. Business Payment involves substantially more business functions and payment subflows than WeChat Transfer, resulting in a larger and more intricate state space; consequently, it exposes far more bugs.

The 59 bugs were classified into five distinct categories based on their failure manifestations, shown in Figure 5. Representative examples are detailed below: *(1) Unexpected Page Transitions (17/59):* 17 instances of unexpected UI page-transitions were identified. For instance, pressing *back* on the *Profile Page* returns to the previous page; however, doing so after a specific sequence of profile edits yields no response, violating page-transition invariants. *(2) Abnormal Transaction Terminations (13/59):* 13 instances of unexpected payment closures were identified, often caused by unhandled cyclic interactions. For instance, navigating to the *Switch Phone Number Page* via the "Unable to receive code" option, then press *Back* to return to the *Phone Number Verification Page*, which forms a loop; subsequently pressing "Next" abnormally terminates the entire payment transaction. *(3) Crashes (11/59):* 11 instances of fatal crashes were identified. For example, tapping the WeChat Pay assistant

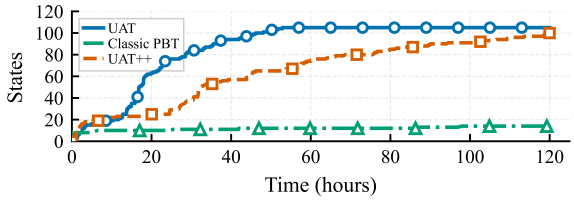


Figure 6: Growth of cumulative states

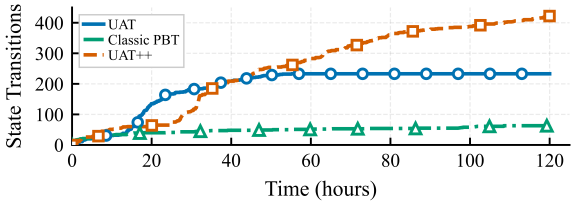


Figure 7: Growth of cumulative state transitions

resulted in an immediate app crash instead of successfully launching the intended mini-program. *(4) Business Logic Violations (9/59):* 9 instances of logic errors were identified, primarily triggered by complex rule combinations and state inconsistencies. For example, canceling a selected discount coupon before confirming a face-to-face Payment still triggers an unauthorized, silent deduction of that coupon. *(5) UI Rendering Anomalies (9/59):* 9 instances of erroneous state presentations were identified. For example, under specific combinations of business rules, inputting an incorrect password causes the app to display unhandled internal exception information (e.g., "reading 'BaseResponse'") rather than a standard error message.

Answer to RQ1 (Bug Discovery): UAT++ found 59 previously unknown bugs (55 confirmed, 45 fixed) across four use cases, which can be classified into five types (Unexpected Page Transitions: 17, Abnormal Transaction Terminations: 13, Crashes: 11, Business Logic Violation: 9, UI Rendering Anomalies: 9).

Results for RQ2 . We compare the exploration capability of the three approaches using two metrics:

(1) State Coverage. Figure 6 presents the cumulative distinct states discovered over the 120 machine hour evaluation period. The results show that both UAT++ and UAT achieve complete state coverage, discovering all 105 states of WeChat Pay.

The classic PBT baseline exhibits the weakest performance, plateauing almost immediately at approximately 15 states. It shows that without an understanding of the business logic, unguided random exploration can easily be trapped by some blocking pages. Conversely, the UAT baseline shows a rapid initial discovery rate, peaking at 105 states around the 50-hour mark before hitting a plateau. In contrast, UAT++ exhibits a more gradual initial growth curve compared to the UAT baseline. This slower pace occurs because upon reaching a target state via model-based guidance, UAT++ extensively explores local transitions before moving on to explore new states. However, effectively guided by the model, UAT++ steadily progresses and ultimately reaches the exact same level as the UAT baseline (105 states) by the end of the period, successfully achieving full state coverage. As a result, both UAT++ and UAT eventually cover the same finite set of WeChat Pay states.

(2) State Transition Coverage. Figure 7 illustrates the cumulative state transitions discovered by the three approaches over the 120 machine hours evaluation period. The empirical results demonstrate that UAT++ significantly outperforms both individual baselines, ultimately discovering over 420 state transitions.

Constrained by its inability to reach deep UI states, the classic PBT exhibits the weakest performance here, plateauing early with only approximately 60 transitions. Conversely, the UAT demonstrates a rapid initial discovery rate, reaching approximately 230 transitions within the first 50 hours. However, it subsequently hits a plateau. This stagnation aligns with its design limitation: predefined UAT test cases are strictly fixed, so they never dynamically explore execution spaces (e.g., complex combinations of business rules, rollbacks, or cyclic interactions). Once all predefined test cases have been executed, the state space ceases to expand. In contrast, our model-guided property-based approach performs well in this metric. Building upon the model guidance, UAT++ dynamically explores the execution spaces around each state, successfully uncovering those unforeseen combinations and complex rollbacks that the UAT misses. Furthermore, the domain-specific inputs in UAT++ ensure that this deep transition exploration is not prematurely halted by specialized blocking pages (e.g., password verification page). As a result, UAT++ successfully breaks through the 50-hour plateau, continuously discovering new transitions to reach a total of 420 after 120 hours[2]—an approximate 82.6% ((420-230)/230 ≈ 82.6%) improvement over the UAT and 600% ((420-60)/60 = 600.0%) improvement over classic PBT. This demonstrates the capability of UAT++ for state transition coverage.

Answer to RQ2 (Exploration Capability): UAT++ achieves full state coverage (105 states) on par with the UAT, while outperforming both baselines in state transition coverage by discovering 420 transitions (an 82.6% and 600.0% improvement over UAT and classic PBT, respectively).

Results for RQ3 . Table 2 presents the trace shrinking results for 20 random selected bugs. In Table 2, *#Input* denotes the length of the original failure-inducing event trace for each bug, *#Out* denotes the length of the reduced reproducible trace automatically produced by UAT++, and *#Iters* denotes the number of iterations during the shrinking process. In addition, *%Red* denotes the ratio of the reduced trace length to the original trace length.

Overall, UAT++ achieved a reduction rate exceeding 85% for the majority of the test cases, with a maximum reduction rate of 95.2% and a minimum of 50%, yielding an average reduction rate of 79.9% across the dataset. This substantial reduction demonstrates that the proposed shrinking strategies effectively eliminate redundant cycles and irrelevant GUI events. Specifically, the lengths of the reduced traces (*#Out*) were successfully compressed to an average of 5.1 steps (with a median of 4.5). Regarding algorithmic efficiency, as shown in Table 2, the algorithm converges to an optimized reduced

[2]The curves show no plateau, indicating that running for longer would likely find more state transitions.

Table 2: Statistical Results of UAT++ Trace Shrinking

Bug ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Avg.	Med.	Std.
#Input	16	12	10	25	28	23	54	10	36	36	219	37	48	52	68	18	48	12	63	15	41.5	32.0	45.6
#Out	2	3	4	7	9	3	8	5	7	2	11	4	4	5	5	7	4	5	3	4	5.1	4.5	2.4
#Iter	6	10	6	14	17	4	18	9	16	6	50	7	14	14	15	11	6	6	10	7	12.3	10.0	9.9
%Red	87.5%	75.0%	60.0%	72.0%	67.9%	87.0%	85.2%	**50.0%↓**	80.6%	94.4%	95.0%	89.2%	91.7%	90.4%	92.6%	61.1%	91.7%	58.3%	**95.2%↑**	73.3%	**79.9%**	**86.1%**	**14.2%**

Note: ↑ and ↓ indicate the maximum and minimum reduction rates, respectively.

trace by running only 12.3 shrinking iterations (*#Iters*) on average, highlighting its capability to rapidly prune the search space.

> ***Answer to RQ3 (Trace Shrinking):*** UAT++'s trace shrinking algorithm is highly effective and efficient, achieving an average trace reduction of 79.9% (reducing traces to an average of 5.1 steps) by evaluating only 12.3 shrinking iterations per bug.

6 Discussions

Effectiveness of Our Approach. The combination of Model-Based Testing (MBT) and Property-Based Testing (PBT) is a promising direction for testing complex software systems with well-defined state transitions. The core insight is that MBT provides structured state transitions through explicit models, while PBT explores state spaces beyond predefined test cases. This complementary relationship addresses a fundamental challenge in high-assurance systems where exhaustive manual testing becomes prohibitively expensive. Random testing alone cannot guarantee critical state coverage. Our tool UAT++ at WeChat Pay demonstrates this effectiveness in practice. Test case development for UAT required 2000 person-hours to manually deliver 1,000 test cases, yet yielded only 10 bugs monthly on average. In contrast, UAT++ automatically discovered 59 previously unknown bugs over approximately 20,000 machine hours without introducing additional manual scripting overhead. As a senior QA architect validated: "*Some bugs can only be triggered after repeatedly entering and exiting states; they cannot be exposed by following the normal path. This is exactly where the value of combining PBT with MBT lies.*" The approach requires only a reference model capturing system behaviors and state transition points, which has stable semantics and infrequent updates. This makes it adaptable to domains such as financial systems, industrial control software, and safety-critical apps where state complexity exceeds manual testing capacity.

Tackling False Positives in Practice. In theory, UAT++ is expected to produce zero false positives because the properties (ϕ) are synthesized from page-transition invariants in manually validated UAT use cases. Any deviation from these transitions indicates a real bug. However, industrial deployment reveals three sources of engineering-level false positives. *First, asynchronous UI rendering latency may cause premature postcondition checks before pages are fully loaded.* We mitigate this by dynamically detecting common loading widgets through their `classname` (e.g., `android.widget.ProgressBar`) or by injecting wait intervals between UI events only when necessary. *Second, execution flakiness in UAT Operator APIs introduces interruptions in page transitions.* During property validation, API failures interrupt state transitions and trigger incorrect violation reports. During property synthesis, such interruptions capture unintended UI layouts, which corrupt the dynamic extraction of state-independent widgets[37]. UAT++ addresses both by monitoring API execution status in real time. If an execution failure is detected, UAT++ gracefully falls back to random exploration to avoid false positives. Meanwhile, the extraction module discards the invalid layout snapshot, ensuring that only intended layouts are recorded. *Third, ambiguous page identification during property synthesis.* In rare cases, distinct pages share identical state-independent widgets (*IW*), causing ambiguity in page identification. This may lead UAT++ to validate properties on an incorrect page, resulting in false positives. We mitigate this by adding the identified pages to the whitelist of the affected property.

Extension of Property Synthesis. Our evaluation focuses on properties synthesized from page-transition invariants, but the approach naturally extends along two orthogonal dimensions. First, the transition step count can be increased beyond the currently implemented single-step transitions. We have already implemented 2-step transition invariants, which generated 809 properties and successfully detected one bug within a single testing round. Critically, this bug has not been found by single-step transitions. Multi-step properties enable the extraction of more invariants by capturing longer event traces within the model. The optimal step count should balance testing comprehensiveness against computational cost based on specific deployment contexts. Second, the property types can extend beyond the current focus on UI navigational transitions to encompass data-level invariants, such as mathematical correctness of transaction computations or consistency constraints across business entities (akin to the invariants detected by Daikon [20]). Together, these two dimensions offer a systematic pathway to enrich the synthesized property set, enabling more comprehensive coverage of both structural and semantic app behaviors.

Threats to Validity. The empirical evaluation was conducted on WeChat Pay. While WeChat Pay is one of the most complex and widely used payment apps, its specific architectural patterns and implementation details may differ from other payment apps. However, despite these potential differences, the core approach of UAT++ is fundamentally agnostic to app-specific logic and can be generalized to other payment apps, which only need to provide an equivalent model as input. Specifically, the automatic synthesis of properties from UAT use cases extracts page-transition invariants, while the model-guided exploration strategy and the failure-inducing trace-shrinking algorithms operate on event traces. None of them rely on app-specific logic.

7 Related Work

Model-based GUI Testing. Model-Based Testing (MBT) [15, 38] is a widely used testing approach. Several studies [3, 39, 55, 57] automatically extract GUI finite state machines via dynamic traversal or by combining static and dynamic analysis. Then they measure test adequacy with code coverage. However, these approaches lack oracles for functional validation. DemGen [8] generates models from UI/UX design documents, but its oracle is built by merging visually similar pages into the same state without defining the system contexts under which transitions should occur. However, in complex payment scenarios, visually identical pages can represent entirely different system states. In contrast, our approach constructs oracles directly from manually validated use cases that explicitly define the business conditions governing each page transition, thereby forming a ground-truth behavioral specification.

Functional Testing of Android Apps. Most existing techniques are limited to finding crashing bugs [39, 45, 50] due to a lack of test oracles [6]. Thus, they fail to find page-transition bugs in our scenarios. To detect functional bugs, numerous automated oracles have been proposed [36, 40, 43, 49]. Notably, Odin [49] detects non-crashing functional bugs in Android apps by mining majority behaviors from execution traces as oracles. Genie [40] exploits the independent view property through metamorphic testing. SetDroid [43] applies metamorphic testing by injecting and restoring system setting changes. However, all of these approaches suffer from limited practicality due to high false-positive rates, and they struggle to explore different states in a complex app to validate its functional correctness. In contrast, UAT++'s page-transition invariants theoretically yield zero false positives, and its model-guided strategy can guide it to different states of WeChat Pay for functional testing.

Property-based Testing. Property-based testing [21] is a powerful testing approach that gained widespread recognition through the work of Claessen and Hughes [12]. It has been applied to find functional bugs in various software systems [4, 25, 27, 29, 31, 32, 35]. Furthermore, PBT has been increasingly adopted in mobile app testing. ChimCheck [28] directly reuses the assertions in existing test cases as test oracles. However, these test oracles cannot be directly applied to our scenario because they are tightly coupled with the app state and preceding operational steps. PBFDroid [42], PDTDroid [44] and Kea [18, 56] share a similar limitation: they require testers to manually write properties before testing and manually inspect executions afterward to identify key failure-inducing traces. In contrast, our approach automatically synthesizes page-transition invariants from the predefined use case instead of defining them manually or using the test cases' oracle directly. Moreover, we propose a shrinking algorithm to effectively reduce failure-inducing traces, thereby aiding bug diagnosis.

To the best of our knowledge, few approaches combine model-based testing with property-based testing [1, 9, 24]. Cabrera Castillos et al. [9] define coverage criteria and mutation operators over property automata derived from a pattern language, providing a systematic framework for MBT with temporal properties. Huang et al. [24] integrate PBT and MBT using symbolic finite state machines with LTL safety properties, formally proving that the generated test suite is exhaustive for any SUT violating a property satisfied by the model. Aichernig and Schumi [1] automatically derive extended finite state machine based generators from XML business-rule models to test web-service applications. However, these approaches require manually specified properties and were evaluated on relatively small systems, thus failing to address the challenges of testing large-scale applications with complex business logic. In contrast, our approach automatically synthesizes properties directly from existing use cases. Furthermore, we employ our approach in a highly complex industrial app WeChat Pay, which comprises 120 core scenarios associated with over 10,000 business rules.

Input Shrinking for GUI Testing. Test input shrinking aims to identify a reduced subsequence of a test input trace that retains the ability to trigger a specific failure. This problem was first formalized by Zeller et al. [58], who proposed delta debugging (DD), a widely recognized foundational technique for optimizing failure-inducing test inputs. In the context of Android apps, existing input shrinking solutions remain limited [10, 13, 26, 41, 51]. ND3MIN [13] addresses non-determinism in the execution of GUI event traces by retrying each candidate trace multiple times. However, this approach still requires a substantial number of test runs, thereby limiting its efficiency in practice. SimplyDroid [26] employs hierarchical delta debugging (HDD), an extension of traditional DD. It constructs a GUI transition hierarchy tree based on abstracted GUI layouts and performs DD at each layer of the tree. Similarly, DetReduce [10] shrinks GUI test traces through state abstraction by eliminating execution cycles that begin and end at the same state. However, both SimplyDroid and DetReduce rely heavily on state abstraction, which can be inaccurate in complex real-world scenarios. This inaccuracy results in additional time overhead due to multiple invalid execution attempts. In contrast, our approach applies GUI-specific strategies to effectively identify and eliminate cycles from GUI test traces, thereby requiring fewer shrinking iterations and efficiently pruning the search space.

8 Conclusion

We propose a model-guided property-based testing approach to effectively validate the correctness of WeChat Pay. Our approach expands test coverage and enables validation of event traces without relying on predefined oracles. The implementation of our idea, UAT++, has been deployed in the continuous testing pipeline of WeChat Pay. It has successfully discovered 59 previously unknown bugs that evaded the existing UAT system. So far, 55 bugs have been confirmed by developers, and 45 have already been fixed. Furthermore, UAT++ triggers over 420 state transitions, achieving an 82.6% improvement over the existing UAT system and overcoming its exploration plateau. Finally, our shrinking algorithm reduced failure traces by an average of 79.9%.

Acknowledgments

We thank the anonymous ASE reviewers for their valuable feedback. This work was partially supported by NSFC Project (No.62572192, No.62072178, No.62502077, No.92582203), Tencent Research Fund and the Fundamental Research Funds for the Central Universities (ZYGX2024XJ036).

Data Availability

Since this research was conducted in cooperation with the WeChat Pay team in Tencent, the experimental datasets and source code cannot be made publicly available due to commercial usage.

References

[1] Bernhard K. Aichernig and Richard Schumi. 2019. Property-based testing of web services by deriving properties from business-rule models. *Softw. Syst. Model.* 18, 2 (April 2019), 889–911. doi:10.1007/s10270-017-0647-0

[2] Alibaba. 2003. *AliPay.* Alibaba. https://www.alipayplus.com/

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE '12)*. Association for Computing Machinery, New York, NY, USA, 258–261. doi:10.1145/2351676.2351717

[4] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–4. doi:10.1109/ICSTW.2015.7107466

[5] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 238–249. doi:10.1145/2970276.2970313

[6] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[7] Cai. 2025. *Many netizens have reported that there is a bug in Alipay, with all orders receiving a 20 percent discount. What is happening? Who will be held responsible for this mistake?* https://www.zhihu.com/en/answer/82788896904

[8] Shaoheng Cao, Renyi Chen, Minxue Pan, Wenhua Yang, and Xuandong Li. 2024. Beyond Manual Modeling: Automating GUI Model Generation Using Design Documents. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 91–103. doi:10.1145/3691620.3695032

[9] Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Julliand. 2014. Coverage Criteria for Model-Based Testing using Property Patterns. *Electronic Proceedings in Theoretical Computer Science* 141 (March 2014), 29–43. doi:10.4204/eptcs.141.3

[10] Wontae Choi, Koushik Sen, George Necula, and Wenyu Wang. 2018. DetReduce: minimizing Android GUI test suites for regression testing. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 445–455. doi:10.1145/3180155.3180173

[11] Tsun S. Chow. 1978. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering* 3 (1978), 178–187. doi:10.1109/TSE.1978.231496

[12] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. doi:10.1145/357766.351266

[13] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. 2016. Minimizing GUI event traces. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 422–434. doi:10.1145/2950290.2950342

[14] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. 1999. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, California, USA) *(ICSE '99)*. Association for Computing Machinery, New York, NY, USA, 285–294. doi:10.1145/302405.302640

[15] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. 2007. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007* (Atlanta, Georgia) *(WEASELTech '07)*. Association for Computing Machinery, New York, NY, USA, 31–36. doi:10.1145/1353673.1353681

[16] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 481–492. doi:10.1145/3377811.3380402

[17] Jekaterina Drozdovica. 2024. *Mobile Payment Trends: What Merchants Need to Know.* https://noda.live/articles/mobile-payment-trends

[18] ECNUSSE. 2026. *Kea2.* ECNUSSE. https://github.com/ecnusse/Kea2

[19] Barry Elad. 2025. *Alipay vs. WeChat Pay Statistics 2026: Must-See Data Now.* CoinLaw. https://coinlaw.io/alipay-vs-wechat-pay-statistics/

[20] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1–3 (Dec. 2007), 35–45. doi:10.1016/j.scico.2007.01.015

[21] George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes* 22, 4 (July 1997), 74–80. doi:10.1145/263244.263267

[22] GooglePay. 2015. *GooglePay.* GooglePay. https://pay.google.com/intl/en_us/about/

[23] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI Testing of Android Applications Via Model Abstraction and Refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 269–280. doi:10.1109/ICSE.2019.00042

[24] Wen-ling Huang, Niklas Krafczyk, and Jan Peleska. 2024. Exhaustive property oriented model-based testing with symbolic finite state machines. *Sci. Comput. Program.* 231, C (Jan. 2024), 28 pages. doi:10.1016/j.scico.2023.103005

[25] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 135–145. doi:10.1109/ICST.2016.37

[26] Bo Jiang, Yuxuan Wu, Teng Li, and W. K. Chan. 2017. SimplyDroid: Efficient event sequence simplification for android application. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 297–307. doi:10.1109/ASE.2017.8115643

[27] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 131–141. doi:10.1109/ICST46399.2020.00023

[28] Edmund S. L. Lam, Peilun Zhang, and Bor-Yuh Evan Chang. 2017. 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) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 58–77. doi:10.1145/3133850.3133853

[29] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 181 (Oct. 2019), 29 pages. doi:10.1145/3360607

[30] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2023. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 135, 5 pages. doi:10.1145/3551349.3559505

[31] Liam O'Connor and Oskar Wickström. 2022. Quickstrom: property-based acceptance testing with LTL specifications. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 1025–1038. doi:10.1145/3519939.3523728

[32] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. doi:10.1145/3293882.3339002

[33] PayPal. 2001. *PayPal.* PayPal. https://www.paypal.com/us/home

[34] Uniqueness Quantification. 2003. *wikipedia.* https://en.wikipedia.org/wiki/Uniqueness_quantification

[35] Manuel J. C. S. Reis. 2025. Property-Based Testing for Cybersecurity: Towards Automated Validation of Security Protocols. *Computers* 14, 5 (2025). doi:10.3390/computers14050179

[36] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data loss detector: automatically revealing data loss bugs in Android apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 141–152. doi:10.1145/3395363.3397379

[37] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An Empirical Analysis of UI-based Flaky Tests. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1585–1597. doi:10.1109/ICSE43902.2021.00141

[38] Muhammad Shafique and Yvan Labiche. 2010. A Systematic Review of Model Based Testing Tool Support. https://api.semanticscholar.org/CorpusID:16566357

[39] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 245–256. doi:10.1145/3106237.

3106298

[40] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 156 (Oct. 2021), 31 pages. doi:10.1145/3485533

[41] Yulei Sui, Yifei Zhang, Wei Zheng, Manqing Zhang, and Jingling Xue. 2019. Event trace reduction for effective bug replay of Android apps via differential GUI state analysis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1095–1099. doi:10.1145/3338906.3341183

[42] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-Based Fuzzing for Finding Data Manipulation Errors in Android Apps *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1088–1100. doi:10.1145/3611643.3616286

[43] Jingling Sun, Ting Su, Kai Liu, Chao Peng, Zhao Zhang, Geguang Pu, Tao Xie, and Zhendong Su. 2023. Characterizing and Finding System Setting-Related Defects in Android Apps. *IEEE Trans. Softw. Eng.* 49, 4 (April 2023), 2941–2963. doi:10.1109/TSE.2023.3236449

[44] Jingling Sun, Ting Su, Jun Sun, Jianwen Li, Mengfei Wang, and Geguang Pu. 2024. Property-Based Testing for Validating User Privacy-Related Functionalities in Social Media Apps. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 440–451. doi:10.1145/3663529.3663863

[45] Monkey Team. 2022. *Android Monkey*. Google. https://developer.android.com/studio/test/other-testing-tools/monkey

[46] Tencent. 2013. *WeChat Pay*. Tencent. https://pay.weixin.qq.com/index.php/public/wechatpay_en/

[47] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. 22, 5 (Aug. 2012), 297–312. doi:10.1002/stvr.456

[48] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. 2015. Automatic generation of system test cases from use case specifications. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 385–396. doi:10.1145/2771783.2771812

[49] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 434–446. doi:10.1145/3540250.3549170

[50] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: generating high-quality test inputs for Android apps via use case combinations. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 469–480. doi:10.1145/3377811.3380382

[51] Ping Wang, Jiwei Yan, Xi Deng, Jun Yan, and Jian Zhang. 2019. Understanding Ineffective Events and Reducing Test Sequences for Android Applications. In *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 264–272. doi:10.1109/TASE.2019.00012

[52] Zhitao Wang, Wei Wang, Zirao Li, Long Wang, Can Yi, Xinjie Xu, Luyang Cao, Hanjing Su, Shouzhi Chen, and Jun Zhou. 2024. XUAT-Copilot: Multi-Agent Collaborative System for Automated User Acceptance Testing with Large Language Model. arXiv:2401.02705 [cs.AI] doi:10.48550/arXiv.2401.02705

[53] WIKIPEDIA. 2026. *WeChat Pay*. WIKIPEDIA. https://en.wikipedia.org/wiki/WeChat_Pay

[54] Ian Wright. 2025. *Rise of Mobile Wallets In 2022*. https://merchantmachine.co.uk/rise-of-mobile-wallets-2022

[55] Shuohan Wu, Jianfeng Li, Hao Zhou, Yongsheng Fang, Kaifa Zhao, Haoyu Wang, Chenxiong Qian, and Xiapu Luo. 2023. CydiOS: A Model-Based Testing Framework for iOS Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3597926.3598033

[56] Yiheng Xiong, Ting Su, Jue Wang, Jingling Sun, Geguang Pu, and Zhendong Su. 2024. General and Practical Property-based Testing for Android Apps. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 53–64. doi:10.1145/3691620.3694986

[57] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering* (Rome, Italy) *(FASE'13)*. Springer-Verlag, Berlin, Heidelberg, 250–265. doi:10.1007/978-3-642-37057-1_19

[58] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. doi:10.1109/32.988498

Received 2026-05-01; accepted 2026-07-01