From Natural Language to Executable Properties for Property-Based Testing of Mobile Apps (Experience Paper)

YIHENG XIONG, East China Normal University, China and Singapore Management University, Singapore
TING SU*, East China Normal University, China
JINGLING SUN, University of Electronic Science and Technology of China, China
JUE WANG, Nanjing University, China
QIN LI*, East China Normal University, China
GEGUANG PU, East China Normal University, China
ZHENDONG SU, ETH Zurich, Switzerland

Property-based testing (PBT) is a popular software testing methodology and is effective in validating the functionality of mobile applications (apps for short). However, its adoption in practice remains limited, largely due to the manual effort and technical expertise required to specify executable properties. In this experience paper, we propose a novel structured property synthesis approach that automatically translates property descriptions in natural language into executable properties, and implement it in a tool named ɪPBT. Our approach decomposes the problem into UI semantic grounding and executable property synthesis. It first builds an enriched widget context via multimodal LLMs to align visual elements with their functional semantics, and then uses an LLM with in-context learning to generate framework-specific executable properties. We evaluate ɪPBT with a closed-source LLM (GPT-4o) and an open-source LLM (DeepSeek-V3) on 160 diverse property descriptions across 20 apps (124 from an existing benchmark and 36 newly authored). ɪPBT achieves 95.0% (152/160) accuracy on both LLMs. Notably, an ablation study reveals that the enriched widget context contributes to an absolute improvement of up to 18.1% (from 76.9% to 95.0%). A user study with 10 participants demonstrates that ɪPBT reduces the time required to write executable properties by 56%, suggesting substantially lower manual effort. Furthermore, evaluations on 1,520 linguistically diverse paraphrases of the original property descriptions further confirm ɪPBT's robustness, achieving 88.2% accuracy on GPT-4o and 87.8% on DeepSeek-V3.

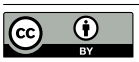
CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Property-Based Testing, Mobile App Testing, Property Synthesis, Large Language Models

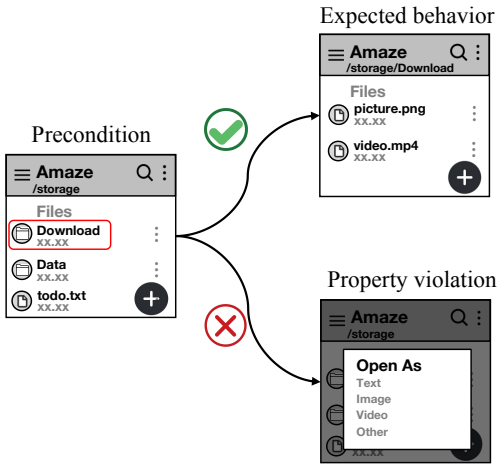
ACM Reference Format:
Yiheng Xiong, Ting Su, Jingling Sun, Jue Wang, Qin Li, Geguang Pu, and Zhendong Su. 2026. From Natural Language to Executable Properties for Property-Based Testing of Mobile Apps (Experience Paper). *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA017 (October 2026), 24 pages. https://doi.org/10.1145/3832108

*Ting Su and Qin Li are the corresponding authors.

Authors' Contact Information: Yiheng Xiong, East China Normal University, Shanghai, China and Singapore Management University, Singapore, Singapore, yihengx98@gmail.com; Ting Su, East China Normal University, Shanghai, China, tsu@sei.ecnu.edu.cn; Jingling Sun, University of Electronic Science and Technology of China, Chengdu, China, jingling.sun910@gmail.com; Jue Wang, Nanjing University, Nanjing, China, juewang591@gmail.com; Qin Li, East China Normal University, Shanghai, China, qli@sei.ecnu.edu.cn; Geguang Pu, East China Normal University, Shanghai, China, ggpu@sei.ecnu.edu.cn; Zhendong Su, ETH Zurich, Zurich, Switzerland, zhendong.su@inf.ethz.ch.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/10-ARTISSTA017
https://doi.org/10.1145/3832108



(a) The app's expected behavior

```
1 @precondition(
2   findWidget(id="line").exists() and
3   findWidget(id="Search").exists()
4 )
5 @property
6 def test_open_directory():
7   item_count = findWidget(id="line").count
8   for index in range(item_count):
9     directory_name = findWidget(id="line")[index].get("text")
10    if "." not in file_name:
11      findWidget(id="line")[index].click()
12      break
13  assert directory_name in findWidget(id="path").get("text")
```

(b) An example of executable properties in Kea

Precondition: The list item and search button exist
Function body:
1. Get the names of all items
2. Select an item name that does not contain "."
3. Click it
4. Assert the path contains the item name

(c) The property description in natural language

Fig. 1. An example of the property in app Amaze.

1 Introduction

Property-based testing (PBT) has emerged as a powerful testing methodology that validates software program correctness by checking properties. Unlike example-based testing [11] which relies on specific input-output pairs to determine test outcomes, PBT systematically generates a large number of inputs to verify whether the system under test satisfies the defined properties. The pioneering PBT framework QuickCheck [10] has inspired many other PBT frameworks that successfully uncover bugs difficult to detect with traditional testing techniques across various software domains. [5, 23, 24, 26, 42, 53, 73].

Recently, several research efforts have applied PBT in testing mobile apps [28, 32, 59, 60, 73] to address the oracle problem, determining whether an app's behavior aligns with expected outcomes. In these work, users specify expected behaviors as executable properties, and then the PBT framework automatically generates GUI event sequences to validate them. For example, consider Amaze [1], a popular file management app. A typical property is that when a user clicks on a directory (*e.g.*, "Download"), the app should open that directory and display its contents, rather than triggering a file-opening dialog (which is only expected for files). Fig. 1(a) illustrates the expected app behavior. Fig. 1(b) shows the corresponding executable property written in Kea [73], a recent effective PBT framework for finding functional bugs in mobile apps. The property consists of a precondition (lines 1-4) to check the presence of the file and search button; an interaction scenario (lines 7-12) simulates a click action to open the directory; and a postcondition (line 13) checks whether the path contains the directory name. Then, the PBT framework can generate a large number of GUI events to verify this property and report bugs when the property is violated.

However, these PBT frameworks see limited adoption because testers must translate high-level intents into framework-constrained, executable properties, which demands substantial manual effort and expertise. Testers must learn framework-specific DSLs, including specialized syntax, APIs, and conventions. This imposes a steep learning curve, particularly for those without a strong programming background. Moreover, specifying executable properties requires the tedious manual inspection of the app's view hierarchy to locate low-level UI widget identifiers (*e.g.*, `id="Search"`). It also requires correctly implementing complete executable properties under the constraints of

the testing framework, both of which are non-trivial and error-prone in practice. Together, these barriers create a significant gap that restricts the widespread application of PBT.

To bridge this gap, we enable testers to specify properties in natural language, which is more intuitive and lightweight. To reduce ambiguity while preserving accessibility, we structure property descriptions in a widely-used Hoare logic format, like Given-When-Then used in Gherkin [19] for Behavior-Driven Development [57]. Fig. 1(c) illustrates such a structured description. While translating natural language into executable code has been studied, classic rule-based approaches [12, 46, 61] are insufficient for mobile PBT due to two main limitations. First, rule-based approaches lack flexibility. Enumerating rules to cover diverse paraphrases of user intent (*e.g.*, "open the folder" vs. "navigate into the directory") does not scale and itself incurs substantial manual effort. Second, these approaches lack semantic grounding. They struggle to accurately map high-level widget descriptions (*e.g.*, "search bar") to the low-level widget identifiers.

More fundamentally, these limitations expose a deeper challenge: grounding the informal test intent into concrete executable properties. To address this challenge, we propose a structured property synthesis approach that automatically translates natural-language property descriptions into executable properties. We decompose this problem into two distinct phases: UI semantic grounding and executable property synthesis. We first tackle UI semantic grounding by extracting comprehensive GUI information (*e.g.*, view hierarchies and screenshots) from the app and constructing enriched widget contexts for each UI widget. We employ Multimodal Large Language Models (MLLMs) to generate semantic annotations for each widget by jointly reasoning over visual appearance and structural information. These annotations serve as grounding signals, enabling accurate mapping between user-described widgets (*e.g.*, "item name") and concrete widget identifiers (*e.g.*, "line"), even when the identifiers themselves are not descriptive. Building on this grounded widget context, we then perform executable property synthesis. We leverage LLMs as inference engines to synthesize executable properties that integrate (i) the property description, (ii) enriched widget context, (iii) framework APIs, and (iv) few-shot demonstrations [49]. This design eliminates the need for handcrafted rules while enabling the model to adapt to the specific constraints of property generation.

We implemented our approach as a tool named ıPBT. To evaluate its effectiveness, we conduct experiments with a closed-source LLM (GPT-4o) and an open-source LLM (DeepSeek-V3) on 160 properties across 20 apps: the 124 properties from the Kea benchmark [73] and 36 newly authored properties on 12 additional apps (2 open-source and 10 closed-source from the Google Play Store). ıPBT achieves 95.0% accuracy (152/160) on both LLMs.

Notably, our ablation study reveals that the enriched widget context contributed to a 18.1% absolute increase in accuracy, highlighting its critical role in grounding UI semantics. Beyond accuracy, a user study demonstrated the practical utility of ıPBT, reducing property authoring time by 56% compared to manual composition. We further evaluate robustness by using another LLM, Llama-3.1 [39], to generate 1,520 diverse paraphrased variants of the original property descriptions. Under these variations, ıPBT achieves 88.2% accuracy (1,340/1,520) with GPT-4o and 87.8% (1,335/1,520) with DeepSeek-V3. These results indicate that ıPBT is robust to the variability of natural language property descriptions.

In summary, this paper has made the following contributions:

- At the conceptual level, we introduce a novel approach that reduces the manual effort and lowers the technical barrier for property-based testing of mobile apps.
- At the technical level, we have implemented our idea as a tool named ıPBT that (i) constructs enriched widget contexts via UI semantic grounding to align user-described widgets with concrete

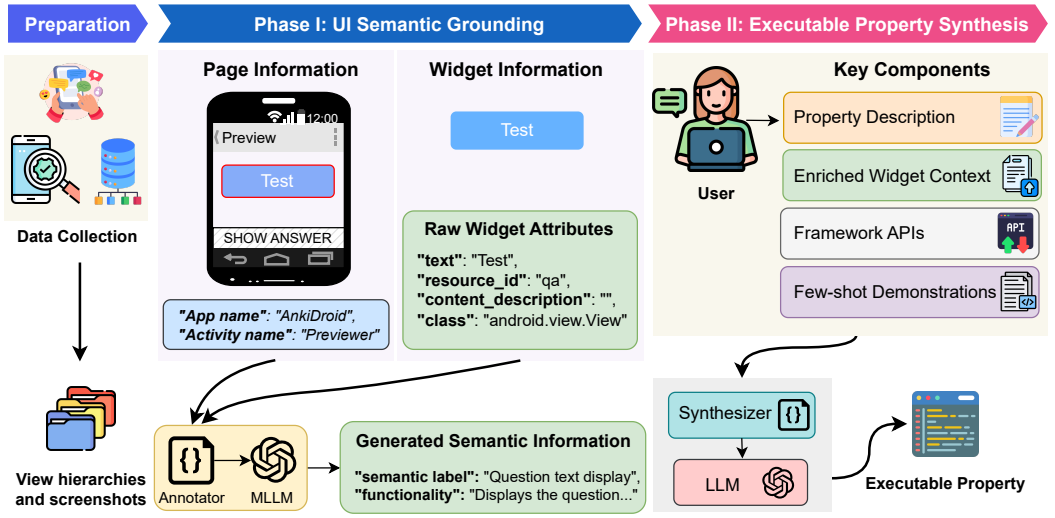


Fig. 2. Overview of iPBT.

UI elements, and (ii) leverages LLMs with in-context learning to synthesize framework-specific executable properties.

- At the empirical level, we construct a new evaluation dataset with 160 human-written natural language property descriptions. We further generate 1,520 linguistically diverse LLM-based variants for robustness evaluation. Based on this dataset, we conduct comprehensive evaluations and summarize practical lessons learned on applying LLMs to property-based testing. The dataset and tool are publicly released at *https://github.com/ecnusse/iPBT*.

2 Background

2.1 Large Language Models

Large language models (LLMs) have been shown to perform well on a wide range of tasks in natural language processing [41] and software engineering [21, 82]. LLMs, such as GPT-4o and DeepSeek-V3, are deep neural networks trained on massive amounts of text data and typically based on the Transformer architecture [64], enabling them to generate human-like text, understand complex queries, and perform diverse tasks. Recently, MLLMs have emerged, which extend the capabilities of traditional LLMs by incorporating additional modalities such as images, audio, or structured data alongside text [77]. This multimodal integration allows MLLMs not only to process natural language but also to understand and reason about visual or other non-textual inputs. In the context of GUI testing, MLLMs are particularly useful as they can jointly leverage textual descriptions and GUI screenshots to better interpret UI semantics.

2.2 In-Context Learning

LLMs are typically pre-trained on large corpora of text and code data. To adapt LLMs on customized tasks, fine-tuning [14], which requires training on a pre-trained model with additional massive data, and prompt engineering [49] (*e.g.*, chain-of-thought [67], in-context learning [49], and multi-step reasoning [84]) are two common approaches. In-context learning refers to the ability of LLMs to perform customized tasks with a few examples provided directly in the input prompt, without requiring model training, which is the strategy we adopt in our approach.

3 Approach

At a high level, iPBT operates as a structured synthesis approach designed to translate natural language property descriptions into executable properties. Fig. 2 presents the overall workflow of iPBT, which consists of two main phases: (i) the *UI Semantic Grounding* phase (§3.1), which extracts the GUI information of the app under test, leverages MLLMs to generate functionality annotations for widgets, and constructs the enriched widget context for each UI element; and (ii) the *Executable Property Synthesis* phase (§3.2) that generates executable properties by encoding user APIs, UI widget identifiers, property description and examples into a carefully designed prompt. Together, these two phases bridge the gap between informal user intent and concrete implementation details. We describe each phase below.

3.1 Phase I: UI Semantic Grounding

In Android app testing, UI widget identifiers (*e.g.*, resource_id, text, and content_description) are commonly used to locate and interact with target widgets displayed on the screen. Consequently, a critical prerequisite for synthesizing executable properties is grounding high-level natural language descriptions onto these concrete identifiers. For instance, an instruction like "click the Settings button" must be accurately mapped to a specific widget, such as `text="Settings"` or `id="action_settings"`. However, relying solely on raw identifiers is often insufficient. In practice, these identifiers are frequently opaque, poorly maintained, or generic (*e.g.*, `id="button1"`), failing to reflect the widget's true functionality. To bridge this semantic gap, we extract comprehensive GUI information and leverage MLLMs to generate semantic functionality annotations for each widget. The resulting enriched widget context consists of raw attributes (*e.g.*, `text`, `resource_id`) with MLLM-inferred semantics. This context serves as a robust bridge, enabling the subsequent executable property synthesis phase to accurately match user-described widgets to the correct identifiers.

3.1.1 Widget Context Extraction. In Android apps, the layout defines the structure of each page, while widgets (*e.g.*, `Button`) handle user-triggered events (*e.g.*, `onClick`) [3]. To extract the contextual information of these widgets, the first step is to collect the necessary GUI data (view hierarchies and screenshots). Notably, our approach supports GUI data acquisition via various methods, including: (1) manual interaction by testers; (2) automated exploration using testing tools; and (3) direct provision from app vendors. From the collected GUI data, we extract two complementary components that facilitate MLLM understanding and the generation of functionality annotations for widgets:

- **Page information** provides high-level context about the app page under test, including the *app name*, *activity name*, and *page screenshot*. Specifically, the *app name* typically reflects the domain or type of the app (*e.g.*, SimpleNote implies a note-taking app), offering prior knowledge about its overall functionality. The *app name* is obtained by statically analyzing the `AndroidManifest.xml` file. The *activity name*, parsed from the runtime layout file, specifies the functionality of the current page within the app (*e.g.*, `LoginActivity` for user authentication, `SettingsActivity` for configuration). The *page screenshot*, captured via the Android Debug Bridge (ADB), preserves the visual layout and arrangement of the UI widgets. It serves as a direct reference that complements textual and structural information. Because the MLLM annotates one widget at a time (Section 3.1.2), and a page typically contains many widgets, we highlight the widget currently being annotated, hereafter referred to as the target widget, with a red bounding box on the page screenshot. This allows the MLLM to unambiguously identify the widget that is the subject of the current annotation request.

Table 1. The prompt design for generating functionality annotation of the widget.

ID	Prompt Component	Instantiation
①	Role Assignment	You are a professional mobile app UI semantic annotation assistant.
②	Task	Please annotate the provided UI widget with the semantic label and functionality description based on the given context. - The full page screenshot, where the target widget is highlighted with a red box. - The cropped widget image and its attributes. - The provided app name and foreground activity name.
③	Few-shot Demonstrations	[example input] + [example output]
④	Input	[page information]+[widget information]
⑤	Constraints	[Strict rules]

- **Widget information** provides fine-grained details about the interactive widgets on each page. It consists of two parts: the cropped widget image and the widget attributes. The *widget image* is obtained by cropping the page screenshot according to the widget bounds. The *widget attributes* are extracted from the view hierarchy file. We focus on four commonly used fields: `"text"`, `"resource_id"`, `"content_description"`, and`"class"`, as they provide valuable signals about the intended behavior of the widget. Specifically, the `"text"` field corresponds to the string displayed on the widget (*e.g.*, "Login"), often directly indicating its functionality. The `"resource_id"` is a developer-assigned identifier that may encode semantic hints about the widget's role or logical grouping (*e.g.*, `id=btn_submit`). The `"content_description"` field, typically designed for accessibility [2]. Finally, the `"class"` specifies the Android class type of the widget (*e.g.*, `Button`, `EditText`, `ImageView`), which reflects the type of interaction it supports.

These two components together provide complementary structural and visual information, enabling the MLLM to better reason about widget semantics. Crucially, the extraction of these artifacts is fully automated. The extracted widget context serves as the foundation for the subsequent phase, where MLLMs are leveraged to generate accurate functionality annotations for each widget.

3.1.2 Widget Functionality Annotations Generation. Leveraging the extracted widget context, we employ an MLLM to synthesize functionality annotations. We construct a structured prompt to guide the model, as illustrated in Table 1. The prompt is organized into five components: Component ① defines the role of the model to establish a professional persona. Component ② specifies the Task, directing the MLLM to infer semantics based on the provided visual and structural context. Component ③ provides two demonstrations that serve as the concrete reference selected from the SimpleNote app [56]. They illustrate the expected mapping from the raw widget context to the target semantic label and functionality description. Empirically, we found that two representative demonstrations are sufficient for the MLLM to grasp the task requirements and generate high-quality annotations. Component ④ gives the input of context, including page information and widget information extracted in the previous step. Component ⑤ outlines the constraints to ensure output consistency.

Given these instructions, the MLLM generates two key fields for each widget: *semantic label* and *functionality*. The *semantic label* is a concise identification of the widget, *e.g.*, login button, or settings option. The *functionality* offers a description of its behavior, *e.g.*, allows the user to log into their account, or navigates to settings screen.

For example, consider the target widget shown in the green box in Fig. 2. Sole reliance on these raw attributes is insufficient to deduce its functionality. To resolve this ambiguity, we construct an enriched widget context by aggregating page information, such as the app name `AnkiDroid` and

Table 2. The prompt design for generating executable properties.

ID	Prompt Component	Instantiation
①	Role Assignment	You are an expert in Python programming and Android app testing, and your role is to write test snippets for Android apps.
②	Framework APIs	**The following APIs are available for writing property:** widget: findWidget(identifier); click: widget.click() long click: widget.long_click(); get text: widget.get("text"); exists: widget.exists(); back: press("back"); ...
③	Enriched Widget Context	**The app's UI widget identifiers are detailed below for reference, ensuring accurate element selection in tests**: { "text": "Download", "resource_id":"line", "description": "null", "class": "android.widget.TextView", "semantic label": "File name text", "functionality": "Display the name of the file" }, ...
④	Few-shot Demonstrations	Here are the two example test snippets that you might write, based on the given property descriptions: [Example property description and executable properties]
⑤	Property Description	**Your task: Using the available APIs, UI widget identifiers and following the example format, please write a test snippet for the following property:** **Precondition**: The list item and search button exist **Function body:** 1. Get the names of all items 2. Select an item name that does not contain "." 3. Click it 4. Assert the path contains the item name
⑥	Constraints	Respond only with the Python code, strictly adhering to the given property description. Do not include any explanations, comments, or text outside the code block.

activity `Previewer`, with the widget information. This combined information enables the MLLM to effectively anchor the widget's semantics. Consequently, iPBT produces functionality annotations, augmenting the original attributes with a semantic label ("Question text display") and a precise functionality description ("Displays the question text to the user for review"). These enriched widget contexts serve as the foundation for generating executable properties in the next phase.

3.2 Phase II: Executable Property Synthesis

3.2.1 Writing Property Descriptions. To ensure that property descriptions are both intuitive for testers and structured for synthesis, we adopt a Hoare logic-style representation. This style aligns with industry-standard practices, such as Gherkin [19] widely employed in Behavior-Driven Development (BDD) [57]. In the context of PBT in mobile apps, an executable property is formalized as a triple $\langle P, I, Q \rangle$, where (1) P is the precondition, which defines when the property can be checked, (2) I is the interaction scenario, which defines the sequence of user actions to execute the target functionality, and (3) Q is the postcondition, which specifies the expected UI state after the interaction. To facilitate this formalization, we structure the natural language property description into two segments:

- **Precondition:** This segment maps directly to P. It describes the initial visible state required to trigger the property (*e.g.*, "the file name exists").
- **Function Body:** This segment encapsulates both I and Q. It contains the execution steps of the target functionality (*e.g.*, "select a file name that does not contain '.' and click it") and explicitly states the expected effects (*e.g.*, "verify the path contains the file name").

3.2.2 Constructing Prompt and Generating Executable Properties. Table 2 illustrates the structured prompt employed by ıPBT to synthesize executable properties. The prompt comprises six distinct components, labeled ① through ⑥, designed to guide the LLM's synthesis process:

- **Role Assignment (①):** This component defines a specialized persona for the LLM, priming it to focus on the domain of mobile app testing and code generation.
- **Framework APIs (②):** To ensure the synthesized code is syntactically valid, we provide the complete list of user-facing APIs supported by the target framework (Kea in our implementation). We curated the set of APIs like `click()` and `exists()` by analyzing the framework's official documentation and source code. Crucially, these APIs are app-agnostic, allowing this component to be reused across different apps. Note that this component is modular and can be substituted with APIs from other PBT frameworks if desired.
- **Enriched Widget Context (③):** This component supplies the Enriched Widget Context constructed in Phase I. By integrating raw identifiers with MLLM-generated semantic annotations, this section enables the LLM to ground the natural language descriptions to the correct UI widget identifiers.
- **Few-shot Demonstrations (④):** To construct the few-shot demonstrations, we followed two principles to avoid bias and ensure that the prompt covers the essential components a model must learn. First, we selected examples from SimpleNote [56], an app that is not included among the twenty evaluation subjects, to prevent example-to-evaluation leakage. Second, we selected two representative properties that jointly cover (i) the use of key Kea APIs (e.g., findwidget(), exists()) and (ii) the mapping from a natural-language description to the Hoare-logic structure expected by the PBT framework. We empirically determined that two examples are sufficient through a pilot study reported in Section 4.
- **Property Description (⑤):** The component describes the task with the property description and highlights that the output should follow the format of the example executable properties.
- **Constraints (⑥):** To ensure the output is machine-readable, we enforce strict constraints. We explicitly instruct the LLM to generate *only* the code snippet without verbose explanations or markdown formatting, which facilitates the automated extraction of the generated properties.

Upon construction, the complete prompt is fed into the LLM, which then synthesizes the final executable properties.

4 Evaluation

We aim to answer the following research questions:

- **RQ1**: What is the correctness of executable properties generated by ıPBT based on property descriptions? How important are the widget functionality annotations in affecting the correctness?
- **RQ2**: To what extent can ıPBT reduce manual effort? What are the differences in the complexity of the natural language property descriptions and executable properties generated by ıPBT?
- **RQ3**: How robust is ıPBT in generating executable properties based on diverse property descriptions?

Large Language Models. We evaluate our approach using two representative LLMs for executable property generation: one closed-source model (GPT-4o) and one open-source model (DeepSeek-V3). This selection allows us to assess the effectiveness of our approach across different model ecosystems. In addition, we employ an MLLM (GPT-4o mini) to generate functionality annotations for UI widgets, as it supports both textual and visual inputs.

Writing property descriptions. To evaluate our approach, we first construct property descriptions based on the dataset from the prior work Kea [73], which provides 124 diverse executable properties across eight popular open-source Android apps covering diverse app categories, *e.g.*,

Table 3. Extended subject apps used in our evaluation. The "Downloads" column reports the install count reported on Google Play at the time of evaluation.

App Name	Category	Type	Version	Downloads
Firefox	Browsers	Open-source	150.0.3	100M+
WordPress	Art & Design	Open-source	26.7	100M+
Google Keep	Productivity	Closed-source	5.26.191	1B+
Pinterest	Lifestyle	Closed-source	14.17.0	1B+
Gallery	Photography	Closed-source	1.9.3.8	1B+
Zedge	Personalization	Closed-source	9.20.0	500M+
ESPN	Sports	Closed-source	8.16.0	100M+
Audiomack	Music & Audio	Closed-source	8.2.0	100M+
AllTrails	Health & Fitness	Closed-source	26.4.40	10M+
Scanner Radio	News & Magazines	Closed-source	8.25	10M+
Kwikpic	Events	Closed-source	15.8	1M+
Vizo Lite	Tools	Closed-source	1.29	1M+

tools, editor, education, and audio player. Importantly, each property is derived from a distinct real-world historical bug spanning different functional modules (*e.g.*, navigation, data management, and configuration), so the resulting properties capture a broad spectrum of app behaviors. As a benchmark originally curated for property-based testing of Android apps, it offers both diversity and practical relevance. The dataset also includes the corresponding bug reports to facilitate reproduction of each bug.

To construct the natural language property descriptions, we followed a three-step process: (1) Property collection. We collected 124 properties from Kea's dataset, including the executable properties, associated bug reports, and APK files. (2) Property understanding. Each line of an executable property typically represents a UI event, making it difficult to infer the corresponding widget based solely on its identifier. To gain a comprehensive understanding, we manually installed the associated APK files on mobile devices, interacted with the apps to reach the states satisfying the precondition, and executed the properties to observe each step. This helps us understand each component of the executable properties, including the precondition, interaction scenario, and postcondition. (3) Property description construction. One author wrote each property description in natural language based on the executable properties and observed app behavior. To ensure clarity and correctness, the remaining co-authors discussed together to resolve any inconsistencies or ambiguities through iterative revision. Each property description underwent at most two rounds of revision before reaching consensus among co-authors. Common revisions included clarifying implicit preconditions, removing ambiguous wording, and aligning terminology with the visible UI of the target app.

Extended subjects and property descriptions. To assess the generalizability of ɪPBT beyond the Kea benchmark, we additionally constructed an extended evaluation set of 12 new apps and 36 newly authored property descriptions. The new apps consist of two open-source apps and ten closed-source apps, summarized in Table 3.

Open-source apps. We selected Firefox and WordPress, from the open-source mobile app benchmark used in prior work [74]. The core functionalities of Firefox (mobile browsing) and WordPress (content management) are not covered by any of the original eight Kea apps.

Closed-source apps. To obtain a representative set of widely used closed-source commercial apps, we adopted the following three-stage sampling protocol: (1) Category selection. We selected 10 representative app categories on Google Play that collectively span the major categories (e.g.,

productivity, lifestyle, and events). (2) Candidate pool construction. Within each category, we collected the top-20 apps from the official Google Play ranking as a candidate pool, ensuring that each candidate is widely used and representative of its category. (3) Random sampling. From each candidate pool, we uniformly sampled one app at random, yielding 10 closed-source apps in total.
Property construction. Since the new apps are not associated with a curated bug-derived benchmark like Kea, we authored property descriptions manually using the following protocol designed to reduce subjectivity. For each app, two co-authors independently read the app's official description on its store page and interacted with the app on a physical device for at least 10 minutes to familiarize themselves with its functionalities. They then jointly identified the three most representative core functionalities, and authored one natural-language property description per functionality, following the same Pre-I-Post Hoare-logic style introduced in Section 3. This process yielded 36 new property descriptions, bringing the total to 160 property descriptions across 20 apps used in our evaluation.
Constructing enriched widget context. The first step of iPBT aims to construct the enriched context for the widgets in each app. While our approach supports diverse data collection methods (as detailed in Section 3.1.1), we employed an automated strategy in our evaluation to minimize manual intervention. We utilized DroidBot [69], a popular open-source automated GUI testing framework, extended with a random exploration strategy, to perform a three-hour exploration on each app [58]. Achieving full coverage via automated exploration is a long-standing challenge, but advancing exploration algorithms is orthogonal to our primary contribution of property generation. To decouple our generation approach from the limitations of the exploration tool, we additionally executed the "main path" provided in the dataset, comprising events from the app entry to the state satisfying the property's precondition. This ensures that all relevant UI widgets involved in the target properties are captured, providing a fair basis for evaluating iPBT's generation capabilities. Finally, iPBT extracts the page and widget information and leverages GPT-4o mini to construct the enriched context.
Correctness of the generated executable properties. We use a uniform correctness evaluation protocol across all three research questions. Our evaluation includes 160 original property descriptions (124 from the Kea benchmark and 36 newly authored for the extended subjects), as well as 1,520 paraphrased descriptions used in RQ3. For each original description, we apply a tailored protocol according to whether ground-truth bug-triggering traces are available: bug reproduction for the Kea-derived properties and behavioral equivalence for the extended properties. For both the original and paraphrased descriptions, we further apply an intent-preservation check to ensure that the generated executable properties preserve the intended preconditions, interaction scenarios, and postconditions.
Bug reproduction (Kea properties only). The 124 executable properties in the Kea dataset are derived from 124 historical bugs. Thus, verifying whether the generated executable properties by iPBT can reproduce the associated bug serves as a primary indicator of its correctness. Specifically, for each executable property, we follow these steps: (1) Start from the app's entry point and navigate along the bug-triggering path. (2) Interact with the app until reaching the state that satisfies the preconditions described in the executable property. (3) Execute the generated executable properties to determine whether it successfully triggers the historical bug. The whole process can be automatically performed by running the scripts (from app entry to the state satisfying the precondition) and executable properties in Kea.
Behavioral equivalence (extended properties). The 36 extended properties are not associated with historical bugs, so bug reproduction is not directly applicable. Instead, for each of the 36 extended properties, two co-authors independently authored a corresponding ground-truth executable property in advance, based on the natural-language description and the actual UI of the target app. We then execute the iPBT-generated property against the app and verify that (1) it runs to

completion without runtime errors, and (2) its execution trace, including the satisfied precondition state, executed events, and resulting postcondition state, is equivalent to that of the ground-truth executable property.

Intent preservation (all 160 properties). For a few cases, even the generated executable properties are able to trigger the historical bug, they may not faithfully capture the intended logic of the original property. For example, the generated executable property may omit some conditions in the precondition. To address this, we additionally evaluate whether the generated properties preserve original intent from the following two dimensions:

- **Precondition and postcondition.** We verify whether the clauses and logical operators in the generated executable properties match those in the ground-truth. Missing or incorrect preconditions or postconditions can lead to inconsistencies in specific scenarios.
- **Interaction scenario.** First, we check whether the event sequence in the interaction scenario matches the ground-truth, where each event contains the event action and the target UI widget. Second, for executable properties containing conditional branches, we check whether these branches are consistent with the intended logic. Extra or missing branches may still allow bug reproduction but fail to reflect the precise execution flow of the original executable property.

In Android app testing, the same UI widget can be located through different identifiers (*e.g.*, `resource_id`, `text`). For example, `text="Settings"` and `resourceId="app.settings"` may both refer to the same button. When evaluating widget matching, we treat such cases as correct matches as long as the different attributes refer to the same underlying widget.

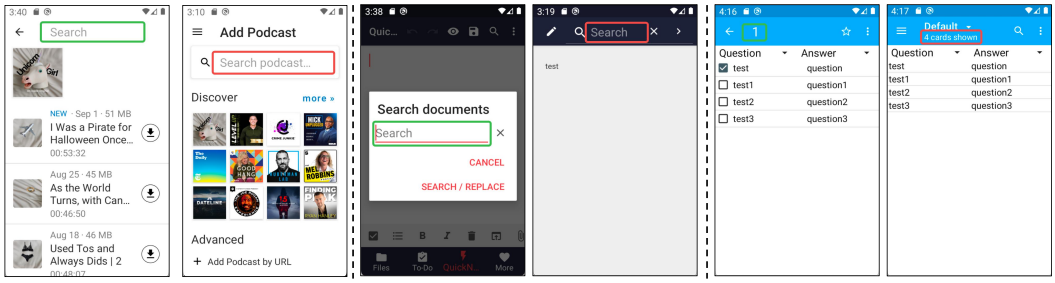
To ensure reliability, two co-authors of this paper independently evaluated each executable property. We measured inter-rater agreement using Cohen's Kappa, achieving a substantial agreement ($\kappa = 0.96$). Any disagreements were resolved through discussion.

Pilot study on the number of demonstrations. The number of few-shot demonstrations directly affects both prompt length and generation quality. To select an appropriate value, we conducted a pilot study by varying the number of demonstrations from 1 to 5 and measuring the accuracy of generated executable properties on 124 properties. The results showed that $k = 1$ yielded 102 of 124 correct executable properties (82.3%), while accuracy reached a plateau at $k = 2$ (118 of 124, 95.2%) with no further improvement for $k = 3$, 4, or 5. We additionally verified that demonstrations are essential at all: removing them entirely ($k = 0$) caused all 124 generated outputs to fail to preserve the required Pre-I-Post structure. Based on these results, we adopted k=2 as a practical tradeoff between accuracy and prompt conciseness. We use this configuration throughout all subsequent experiments.

4.1 RQ1: Correctness

Evaluation setup. To evaluate the correctness of iPBT in generating executable properties, we ran iPBT on the 160 property descriptions. For each description, iPBT automatically constructed the corresponding prompt using the template described in Section 3 (Table 2) and invoked the LLM to synthesize the executable property. To ensure deterministic and stable results, we set the temperature parameter of the LLM to 0, a setting commonly used in prior work [17, 44, 76]. For each prompt, we sequentially invoked the LLM and recorded the generated executable properties as the output. Moreover, we conduct an ablation study to evaluate the contribution of the widget functionality annotation module. Specifically, we removed the functionality annotation of each widget and then repeated the same experimental procedure to assess its impact. We also conduct the ablation study in RQ2 and RQ3 in the same process.

Evaluation results. Based on our experimental statistics, for all 160 property descriptions, both GPT-4o and DeepSeek-V3 successfully generated 152 correct executable properties, achieving an



(a) Page of Antennapod (Correct) (b) Page of Antennapod (Incorrect) (c) Page of Markor (Correct) (d) Page of Markor (Incorrect) (e) Page of AnkiDroid (Correct) (f) Page of AnkiDroid (Incorrect)

Fig. 3. Examples of failure cases. The green boxes annotate the correct widgets and red boxes annotate the incorrect widgets.

accuracy rate of 95.0%. These results demonstrate the effectiveness of iPBT and highlight its ability to reliably translate natural language descriptions into executable properties.

For the eight failure cases, all errors manifested as incorrect widget identifiers generated by iPBT. Upon further analysis, we found that in five cases, the failures were caused by the presence of other widgets within the same app that shared similar functionality with the target widget, which misled iPBT during the matching process. In the remaining case, the error originated from an ambiguous functionality annotation generated by iPBT, which subsequently caused the executable property to be matched to the wrong widget. Fig. 3 presents three examples from different apps. The first two cases (Fig. 3(a)–(d)) failed because different widgets in the same app shared similar functionality. The third case (Fig. 3(e)–(f)) failed due to ambiguity in the functionality annotation generated by iPBT. Specifically, iPBT produced the semantic label *"Current card number"* and the functionality description *"Indicates the number of the current card being viewed"*. In reality, the widget represents the number of the currently selected card, which differs subtly from the generated annotation and led to an incorrect mapping.

Ablation study. The result shows that after removing the functionality annotation of widgets, iPBT successfully generated 123 and 127 correct executable properties on GPT-4o and DeepSeek-V3, corresponding to accuracy rates of 76.9% and 79.4%, respectively. Compared with the full setting (152 correct executable properties, 95.0% accuracy), this represents a performance drop of 18.1 and 15.6 percentage points, respectively. This substantial decrease highlights the importance of functionality annotations in helping the LLM accurately understand widgets and generate correct executable properties.

4.2 RQ2: User study

Evaluation setup. In RQ2, we conduct a controlled human-subject experiment to assess how iPBT actually supports users in practice. Specifically, participants were asked to write executable properties manually, as well as to provide property descriptions for iPBT to generate executable properties. We then analyzed their time cost and correctness across various tasks to evaluate how iPBT can save manual effort. Our experimental design follows prior work [31, 59, 75].

Dataset of the user study. We selected 124 executable properties from RQ1 and categorized them into three groups (low, medium, and high complexity) using the metric defined in Kea. This metric accounts for the number of logical clauses and operators in the pre- and postconditions, as well as the number of events in the interaction scenario. From each complexity group, we randomly selected 2 executable properties from different apps, resulting in a final dataset of 6 executable properties

Table 4. Groups in user study

Group (Participant ID)	Writing property descriptions	Writing executable properties
Group A (P1-P5)	Property 1, 3, 5	Property 2, 4, 6
Group B (P6-P10)	Property 2, 4, 6	Property 1, 3, 5

(Property 1–6, two per complexity level). This captures diverse levels of difficulty while keeping the number of tasks manageable for participants. For each property, we installed the corresponding app on an Android emulator and recorded a video walkthrough. Each video demonstrated how to: (1) reach the state that satisfies the precondition, (2) perform the functionality in the interaction scenario, and (3) verify the expected behavior in the postcondition.

Participants. We recruited 10 participants for our user study, a sample size similar to previous related work [9, 59, 83] (which recruited 10, 8, and 12 participants, respectively). All participants are graduate students majoring in software engineering, with at least four years of programming experience and familiarity with Python programming. This ensured that they had the necessary technical background to understand and write the executable properties without using ɪPBT. A prior study has shown that graduate students can serve as professionals in software engineering tasks [52]. In addition, none of the participants was from the authors of this paper.

Procedure. We designed the study as a conventional within-subject controlled experiment, in which each participant was required to write both property descriptions and executable properties. To avoid learning bias caused by increased familiarity, no participant wrote both the description and executable version of the same property. To achieve this, the 10 participants were evenly divided into two groups (Group A and Group B) with comparable programming expertise, and each group was assigned different tasks for the same property (as shown in Table 4). In total, this resulted in 60 property-writing tasks.

At the beginning, participants attended a dedicated tutorial introducing the study's background, the concept of properties in Android apps, and the procedures for writing them. Each participant was then given an example property video and asked to write both the property description and the corresponding executable properties. This warm-up exercise lasted approximately 45 minutes.

Afterwards, participants independently completed six distinct tasks, each accompanied by recorded videos containing the necessary information. The study was conducted in a preconfigured desktop environment, where participants used Visual Studio Code with code auto-completion disabled to ensure fairness. We recorded the time each participant spent per task (including writing, reading the documentation when needed, and self-checking) and collected all written property descriptions and executable properties. For each collected property description, we leverage ɪPBT to generate the corresponding executable properties, and assessed their correctness using the evaluation metrics described in our experimental setup.

Complexity of property description and code. We measured the complexity of the property descriptions and their corresponding executable properties using character count. For instance, the natural language step "click the undo button" has a complexity of 21 characters. This measurement allows us to quantitatively compare the writing effort required for natural language descriptions versus executable properties.

Evaluation results. We conducted a detailed analysis of the time efficiency and correctness of executable property generation under two settings: (1) writing property descriptions in natural language and generating executable via ɪPBT, and (2) manually writing executable properties.

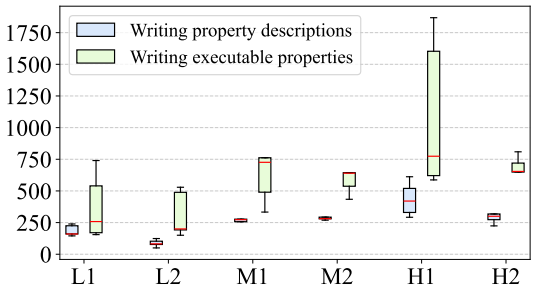


Fig. 4. Time consumption of participants in writing property descriptions and executable properties. The x-axis represents different properties, and the y-axis represents the time spent (in seconds).

Table 5. The correctness of the executable properties generated in different ways.

Task	LLM	Setting	L1	L2	M1	M2	H1	H2	Total
Writing property descriptions	GPT-4o	with annotations	5	5	5	4	5	5	**29**
		without annotations	4	5	5	1	4	5	24
	DeepSeek-V3	with annotations	5	5	5	4	5	5	**29**
		without annotations	4	5	0	1	4	2	16
Writing executable properties			5	5	4	5	3	4	26

- **Time efficiency**: Fig. 4 shows the time spent per task in both approaches. We can see that writing natural language property descriptions significantly reduces the time required to produce executable properties. On average, our approach achieved a 56% reduction in time, compared to manual writing property descriptions (272.7s vs 625.7s). For low-complexity executable properties (*e.g.*, L2), time savings reached up to 72%, highlighting the efficiency gains of leveraging LLMs for code generation.
- **Correctness**: Table 5 presents the number of correctly generated executable properties in different approaches. Writing property descriptions with ıPBT produced 29 correct executable properties, outperforming the manual approach, which yielded 26 correct executable properties. One incorrect executable property generated by ıPBT approach stemmed from a user typo (a long-click was mistakenly written as a click). In contrast, most errors in the manually written executable properties were caused by participants' unfamiliarity with the Kea framework, such as incorrect API usage in postconditions.

To better understand these differences, we identified two main contributing factors. First, when manually writing executable properties, participants needed additional time to inspect UI attributes for widget identification, while ıPBT automated this process based on matching widgets from property descriptions. Second, even though all participants received training on Kea, we observed frequent consultation of the documentation during manual writing, especially for framework-specific APIs. By contrast, ıPBT allows users to focus on specifying intended behaviors in natural language, while delegating low-level implementation details to the LLM. This not only improves efficiency but also reduces errors caused by limited familiarity with the framework.

Ablation study. In Table 5, we can see that without functionality annotations, ıPBT only generated 24 and 16 correct executable properties, respectively. This result indicates the necessity of the designed functionality annotations in our approach.

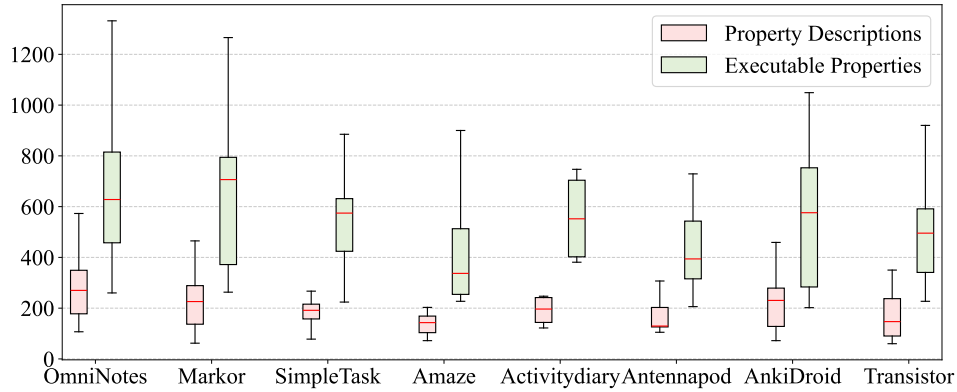


Fig. 5. Comparison of the complexity (measured in character count) between natural language property descriptions and their corresponding executable properties across eight Android apps.

Complexity comparison. Fig. 5 presents the complexity comparison between the 124 natural language property descriptions and their corresponding executable properties, across eight popular Android apps. The x-axis denotes the app names, while the y-axis represents the complexity, quantified by the number of characters. On average, the property descriptions contain 211.3 characters, whereas the corresponding executable properties contain 555.0 characters. This significant difference in complexity highlights the efficiency of the natural language-based approach. Property descriptions are inherently easier to write than their corresponding executable properties, further emphasizing the practicality of ıPBT in executable properties generation.

4.3 RQ3: Robustness

Evaluation setup. Building on the user study in RQ2, which evaluates how ıPBT supports users in practice, RQ3 serves as a complementary experiment that examines robustness. While RQ2 considers the property descriptions actually written by participants, RQ3 evaluates whether ıPBT can generate correct executable properties when faced with semantically equivalent but differently worded property descriptions, simulating the natural variation in how different users might express the same intent.

Prior work has shown that LLMs can generate high-quality paraphrases with greater lexical and syntactic diversity than those produced by crowd workers [7, 8]. Following this line of work, we use an LLM to automatically generate paraphrased versions of each original property description.

To avoid bias from using the same model for both paraphrasing and code generation, we employ a different LLM, Llama-3.1-405B, for paraphrase generation. For each property description, we invoke Llama-3.1 multiple times to generate a pool of candidate paraphrases. Specifically, for each property description, we invoke the Llama-3.1 10 times using the designed prompt (the prompt is provided in the artifact due to space limitations), generating 10 paraphrased descriptions per call, resulting in a total of 100 paraphrased variations per property description.

However, the LLM-generated variations may not be mutually diverse between them. To select a diverse subset of 10 paraphrases per property description, we employed the BLEU score [48] to quantify lexical similarity and guide the selection process, where a lower BLEU score indicates higher diversity. Specifically, we adopted a greedy selection strategy. We initialized the selection set S with the two most mutually diverse paraphrases (*i.e.*, the pair with the lowest BLEU score between them). Then, in each iteration, we computed the Self-BLEU score [85] between each

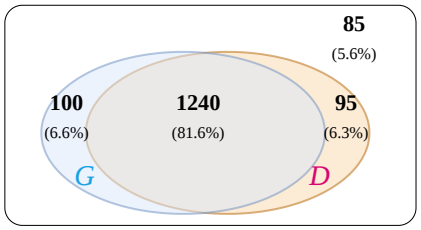


Fig. 6. Venn diagrams of executable property generation. *G*: GPT-4o; *D*: DeepSeek-V3.

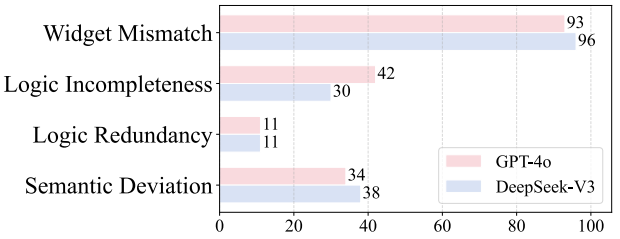


Fig. 7. Failure symptom distribution.

remaining candidate and the current set S, and added the candidate with the lowest score to S, until S contained 10 paraphrased descriptions. Formally, the objective of our selection process is to identify a subset $S \subseteq C$ of size k, where C is the set of all candidate paraphrases and $k = 10$) that minimizes the Self-BLEU score:

$$\underset{S \subseteq C,\ |S|=k}{\arg\min} \frac{1}{|S|(|S|-1)} \sum_{\substack{x,y \in S \\ x \neq y}} \mathrm{BLEU}(x, y)$$

This ensures that the chosen paraphrases are diverse both relative to the original text and mutually within the set.

In RQ3, we focus on 152 property descriptions from RQ1 that were successfully translated into correct executable properties. Using the paraphrasing procedure described above, we generate 10 paraphrases for each property description, resulting in a total of 1,520 paraphrased property descriptions. We then evaluate whether ɪPBT can still generate executable properties from these paraphrases.

Evaluation results. As shown in Fig. 6, GPT-4o and DeepSeek-V3 generated 1,340 and 1,335 correct executable properties, respectively, achieving accuracy rates of 88.2% and 87.8%. Specifically, both models successfully generated correct executable properties for 81.6% (1240/1,520) of the descriptions, demonstrating that ɪPBT is robust to variations in natural language expressions. Among the remaining cases, 6.6% (100/1,520) of the descriptions were handled correctly by GPT-4o but not by DeepSeek-V3, while 6.3% (95/1,520) were correctly processed by DeepSeek-V3 but not by GPT-4o. In 5.6% (85/1,520) of the cases, both models failed to generate correct executable properties.

To further understand the effectiveness of ɪPBT in executable property generation, we analyzed the symptoms of the failure cases. Finally, we identified four main categories of failure symptoms.

Fig. 7 presents the distribution of failure symptoms in the incorrect executable properties generated by ɪPBT with two LLMs. The results reveal that widget mismatch is the most frequent error type, accounting for 51.7%(93/180) and 51.9%(96/185) of the total failures in executable properties generated by GPT-4o and DeepSeek-V3, respectively. This indicates that while LLMs demonstrate a general understanding of UI semantics, they still struggle to precisely align UI descriptions with their corresponding identifiers in some cases. Logic incompleteness is the second most common symptom, responsible for 23.3%(42/180) and 21.6%(40/185) of the errors. This is followed by semantic deviation (18.9%(34/180) and 20.5%(38/185), which typically involve incorrect API usage or faulty postcondition assertions. Logic redundancy ranks fourth, contributing 6.1%(11/180) and 5.9%(11/185) of the failures. The similar distribution of errors across GPT-4o and DeepSeek-V3 demonstrates that ɪPBT performs robustly across different LLMs, while also revealing typical challenges in executable property generation.

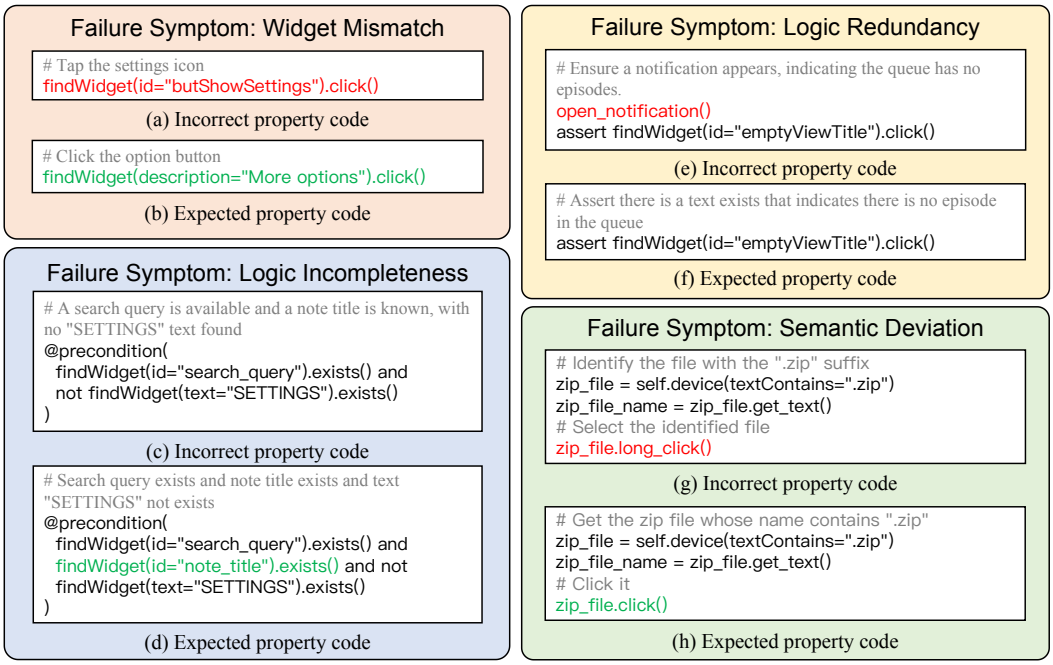


Fig. 8. Illustrative examples of failure symptoms. Incorrect and correct implementations are marked in red and green, respectively, with property descriptions in gray (code snippets are simplified for clarity).

- **Widget Mismatch**. Widget mismatch occurs when the generated executable properties fail to match the intended UI widget. As shown in Fig. 8(a), an example of incorrect executable properties generated by GPT-4o for the app AntennaPod [4]. The expected UI event is to click the option button, but the UI widget identifier highlighted in red fails to match the target widget. In contrast, Fig. 8(b) shows the expected correct executable properties, where the UI widget identifier accurately matches the target UI widget.
- **Logic Incompleteness**. Logic incompleteness refers to generated executable properties that omit statements needed. This manifests primarily as (1) incomplete preconditions/postconditions, (2) missing conditional branches in interaction logic. Such incompleteness may lead to two consequences during execution: (1) execution failures during the property checking, (2) false positives in test results. Fig. 8(c) shows an incomplete code example, which was generated by DeepSeek-V3 for the app OmniNotes [43]. The precondition fails to verify the existence of the "note title" UI widget, causing the testing framework to raise an exception while executing the interaction scenario when the widget is absent. Fig. 8(d) shows correct implementation, which includes the necessary existence check for the UI widget (highlighted in green).
- **Logic Redundancy**. Logic Redundancy refers to the generated code containing unnecessary statements, primarily manifested as redundant UI events (*e.g.*, `click`) in interaction logic. Such errors typically lead to execution failures during property checking. Fig. 8(e) shows an example, which was generated by GPT-4o for the app AntennaPod. The last line (highlighted in red) introduces an unnecessary event to open the notification page. Then, since the GUI state changes to the notification page, the assertion statement will fail during the property checking. Fig. 8(f) shows the expected executable properties.

- **Semantic Deviation**. This refers to other semantic errors, such as incorrect API usage or assertions in postconditions. These types of errors often result in property execution failures or inaccurate test results, including false positives and false negatives. Fig. 8(g) shows an example, which was generated by DeepSeek-V3 for the app Amaze [1]. The implementation (highlighted in red) tries to long-click a file. However, the property description indicates that it should click the file. Fig. 8(h) shows the expected executable properties.

Ablation study. We remove the widget functionality annotations from the prompt. The results show that iPBT generated 1,090 and 1,093 correct executable properties on GPT-4o and DeepSeek-V3, corresponding to accuracy rates of 71.7% and 71.9%, respectively. In comparison, the full setting achieved 88.2% and 87.8% accuracy. This represents a performance drop of 16.5 and 15.9 percentage points, respectively, highlighting the critical role of functionality annotations in guiding the LLM to generate correct executable properties.

5 Discussion and Lessons

Cost Analysis. We measured the API token usage and monetary cost of iPBT across all subjects, separating the two LLM-driven stages. Constructing the enriched widget context with GPT-4o mini costs 13.4M tokens (USD 2.0) per app on average, a one-time cost reused across all properties of that app. Property generation then costs 17,196 tokens (USD 0.044) per property with GPT-4o and 18,428 tokens (USD 0.010) with DeepSeek-V3. For large industrial apps, the context-construction cost may grow with the number of distinct screens; iPBT mitigates this by deduplicating redundant states and widgets during grounding, and can reuse existing widget metadata or UI specifications in place of grounding when available.

Generality of our work. iPBT generalizes along three dimensions. First, it generalizes across diverse apps: we extended the original 124 Kea-benchmark properties on 8 open-source apps with 36 newly authored properties on 12 additional apps (2 open-source and 10 closed-source from Google Play), yielding 160 properties across 20 apps, on which iPBT achieves a consistent ∼95% accuracy. Second, it generalizes beyond Kea. We adapted iPBT to *ChimpCheck* [28], a Scala-based Android PBT tool, by replacing only the *Framework APIs* component of the prompt (Table 2) with ChimpCheck's API list and two ChimpCheck-style demonstrations. Among the 160 properties, 45 fall within ChimpCheck's built-in expressiveness; iPBT correctly generated all 45 corresponding ChimpCheck properties (45/45, 100%). The remaining 115 require unsupported DSL features such as widget-text reading or value capture. Third, the core idea of translating informal specifications into formal properties can naturally extend to related domains, such as web-application testing and formal specification generation for program verification, where tools such as Dafny [51] also require pre/postconditions or invariants.

Applying LLMs to PBT of mobile apps. Although existing PBT frameworks are effective, specifying properties requires significant manual effort and deep familiarity with UI structures and framework-specific APIs. Allowing testers to write properties in structured natural language shifts this burden. In practice, this change enables testers to focus on what the app should do, rather than how to encode it, substantially lowering the barrier to using PBT.

Importance of UI semantic grounding. A central lesson is that UI semantic grounding is essential. Without widget functionality annotations, LLMs frequently select incorrect UI widgets, even when the generated code logic is correct. Our ablation results confirm that raw widget identifiers alone are insufficient in real-world apps, where identifiers are often ambiguous or poorly named.

Designing natural language as an effective specification interface. We found that natural language property descriptions should not be treated as completely free-form input. In practice, adopting a lightweight structure (precondition–interaction–postcondition) significantly reduces

ambiguity and improves generation quality. This indicates that natural language specifications function as an interface between humans and LLMs, and even minimal structural constraints can greatly enhance reliability without harming usability.

What the failures reveal. Most failures arise from UI widget mismatch, not from incorrect control flow or API usage. This indicates that current LLMs can generally synthesize reasonable test logic, but still struggle with fine-grained UI disambiguation when multiple widgets have similar semantics. In addition, paraphrased descriptions sometimes omit implicit constraints, leading to incomplete preconditions or redundant actions. This suggests that LLM-based property synthesis is robust to linguistic variation, but sensitive to semantic underspecification.

Complementing rather than replacing existing PBT frameworks. Rather than replacing existing property-based testing frameworks, iPBT complements them by addressing one of their most labor-intensive stages: executable property authoring. Since iPBT does not modify the execution or input generation mechanisms of PBT, it can be integrated into existing workflows with minimal disruption.

Threats to Validity. ***First***, the evaluated properties may not cover all real-world app behaviors. To mitigate this, we expanded the evaluation from 124 Kea bug-derived properties to 160 properties across 20 apps, adding 36 newly authored properties on 12 open- and closed-source apps; future work can validate iPBT on larger industrial suites. Also, our dynamic exploration to collect UI widget identifiers may not capture all widgets, a common challenge of input generation in app testing [6, 58]. ***Second***, the property descriptions may differ from how practitioners describe properties. To mitigate this, the initial descriptions were authored by a PBT expert and reviewed by all co-authors; moreover, RQ2 uses descriptions from 10 real users and RQ3 assesses robustness under paraphrased variants. ***Third***, we evaluate only two LLMs (GPT-4o and DeepSeek-V3), which may limit generalizability; we chose them as representative state-of-the-art closed- and open-source models, and future LLMs should further improve performance. ***Fourth***, our user study recruited 10 graduate students rather than non-technical testers, since the manual baseline (writing Python/Kea code) requires a technical background; the reported 56% time reduction thus likely understates the benefit for less-technical users. ***Finally***, the models may have seen the evaluated properties during training. This is unlikely, as both were trained before the Kea dataset's release and all descriptions were manually created and never publicly available; we also used a different LLM for paraphrasing.

6 Related Work

LLM for SE tasks. The rapid advancement of LLMs has spurred significant research across software engineering domains, including code completion [15, 80], program repair [71, 81], and comprehensive studies of their capabilities, effects, and limitations across tasks [21, 82]. Our work specifically addresses executable property generation for PBT.

Automated test generation. The research on automated test generation can be categorized into two types: traditional and deep learning-based approaches. Traditional approaches include techniques such as fuzzing [40], symbolic execution [20, 54, 62], search-based [18, 45]. These approaches primarily aim to achieve high test coverage. However, these approaches often struggle to generate assertions [47, 55].

Deep learning-based approaches leverage the capabilities of pre-trained language models to generate tests from code snippets [27, 63, 76, 78]. For example, AthenaTest [63] leverages a transformer model, BART [29], to generate unit test cases based on the given method input. In recent years, LLM-based approaches have shown promising results in test generation [13, 25, 27, 78]. TiCoder [27] leverages LLMs to formalize user intent into tests, and ChatTester [78] can generate unit tests through interactive conversations with LLMs. While these approaches focus on generating unit test cases, some recent work has moved closer to property-based testing. Endres *et al.* [16]

conduct a study to evaluate the LLM's capability of generating postconditions for individual functions based on the function comments. Vikram *et al.* [65] propose an approach to leverage LLM for generating property-based tests from specifications for Python libraries. Liu *et al.* [34] target properties for smart contracts. In contrast, our approach focuses on mobile apps, generating executable properties (including preconditions, interaction scenarios, and postconditions) from natural language descriptions to capture real app behavior while reducing manual effort.

In mobile app testing, recent work like Kea [32, 73], PBFDroid [59], and PDTDroid [60] has demonstrated the effectiveness of PBT in detecting functional and privacy bugs. However, these frameworks primarily focus on the execution and input generation phases, assuming the existence of high-quality executable properties. Consequently, they still require significant manual effort and domain expertise to write executable properties. Our work complements these approaches by automating the executable property generation.

Some work leverages LLM to analyze GUI pages during the dynamic exploration to find data inconsistency bugs [22], functional bugs [36], or inconsistencies between app design and implementation [33]. Our work differs from these approaches in its focus: rather than detecting bugs directly, we generate executable properties that can be used by existing PBT frameworks. Recently, different agents have been proposed to automatically perform tasks on mobile apps [50, 66, 68, 69, 79]. These works focus on executing user-specified tasks, whereas our work centers on generating executable properties to guide property-based testing.

UI widget understanding. Various approaches try to understand the widget from different perspective [30, 35, 37, 38, 70, 72]. For example, IconIntent [72] leverages program analysis and computer vision techniques to identify sensitive UI widgets in Android apps. DroidGem [38] leverages deep neural networks to predict the permissions behind the UI widgets. HintDroid [35] aims to generate hint-text of the UI widget to improve the accessibilty for low-vision users. In contrast, our work focuses on constructing UI widget context for matching the property description with widget identifiers in PBT.

7 Conclusion

We presented iPBT, an approach that generates executable properties from natural-language property descriptions to reduce the manual effort and expertise required for property-based testing of mobile apps. iPBT first constructs enriched widget contexts by extracting GUI information and annotating widget functionality, then uses in-context learning to guide LLMs in synthesizing executable properties. Our evaluation shows that iPBT achieves 95.0% accuracy on original property descriptions, maintains over 87% accuracy on paraphrased variations, and substantially reduces manual authoring effort in a user study, demonstrating its practicality for mobile-app PBT.

Data Availability

We have made all the artifacts (including iPBT and its source code, the designed prompts, and the dataset of property descriptions used in evaluation) publicly available at: *https://github.com/ecnusse/iPBT*.

Acknowledgments

We thank the anonymous ISSTA reviewers for their valuable feedback. This work was partially supported by NSFC Project (No. 62572192, No.62072178, No.92582203, and No.62502077), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, Shanghai Special Program for Promoting High-Quality Industrial Development (Project No. 250401), and the Fundamental Research Funds for the Central Universities (ZYGX2024XJ036).

References

[1] AmazeFileManager Team. 2024. *AmazeFileManager*. https://github.com/TeamAmaze/AmazeFileManager

[2] Android. 2025. *Describe each UI element*. https://developer.android.com/guide/topics/ui/accessibility/apps#describe-ui-element

[3] Android. 2025. *Layouts in views*. https://developer.android.com/develop/ui/views/layout/declaring-layout

[4] AntennaPod Team. 2024. *AntennaPod*. https://github.com/AntennaPod/AntennaPod

[5] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. 2–10. doi:10.1145/1159789.1159792

[6] Farnaz Behrang and Alessandro Orso. 2020. Seven reasons why: an in-depth study of the limitations of random test input generation for Android. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1066–1077. doi:10.1145/3324884.3416567

[7] Auday Berro, Vitor Gaboardi dos Santos, Boualem Benatallah, and Khalid Benabdeslem. 2025. LLMs to Replace Crowdsourcing in Generating Syntactically Diverse Paraphrases for Task-Oriented Chatbots. In *International Conference on Advanced Information Systems Engineering*. Springer, 145–162. doi:10.1007/978-3-031-94569-4_9

[8] Jan Cegin, Jakub Simko, and Peter Brusilovsky. 2023. ChatGPT to Replace Crowdsourcing of Paraphrases for Intent Classification: Higher Diversity and Comparable Model Robustness. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 1889–1905. doi:10.18653/v1/2023.emnlp-main.117

[9] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676. doi:10.1145/3180155.3180240

[10] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP)*. 268–279. doi:10.1145/351240.351266

[11] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 201–211. doi:10.1109/ISSRE.2014.11

[12] Alaka Das and Rakesh Chandra Balabantaray. 2019. MyNLIDB: a natural language interface to database. In *2019 International conference on information technology (ICIT)*. IEEE, 234–238. doi:10.1109/ICIT48102.2019.00048

[13] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435. doi:10.1145/3597926.3598067

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2019). doi:10.48550/arXiv.1810.04805

[15] Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems* 36 (2023), 46701–46723.

[16] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1889–1912. doi:10.1145/3660791

[17] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[18] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419. doi:10.1145/2025113.2025179

[19] Gherkin Team. 2025. *Gherkin Reference*. https://cucumber.io/docs/gherkin/reference/

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223. doi:10.1145/1064978.1065036

[21] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79. doi:10.1145/3695988

[22] Yongxiang Hu, Hailiang Jin, Xuan Wang, Jiazhen Gu, Shiyu Guo, Chaoyi Chen, Xin Wang, and Yangfan Zhou. 2024. Autoconsis: Automatic gui-driven data inconsistency detection of mobile apps. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 137–146. doi:10.1145/3639477.3639748

[23] John Hughes. 2016. Experiences with QuickCheck: testing the hard stuff and staying sane. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer, 169–186. doi:10.1007/978-3-319-30936-1_9

[24] 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)*. IEEE, 135–145. doi:10.1109/ICST.2016.37

[25] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1408–1420. doi:10.1145/3691620.3695513

[26] 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)*. IEEE, 131–141. doi:10.1109/ICST46399.2020.00023

[27] Shuvendu K Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, et al. 2022. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950* (2022). doi:10.48550/arXiv.2208.05950

[28] Edmund SL 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*. 58–77. doi:10.1145/3133850.3133853

[29] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019). doi:10.48550/arXiv.1910.13461

[30] Linlin Li, Ruifeng Wang, Xian Zhan, Ying Wang, Cuiyun Gao, Sinan Wang, and Yepang Liu. 2023. What you see is what you get? it is not the case! detecting misleading icons for mobile applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 538–550. doi:10.1145/3597926.3598076

[31] Jingjing Liang, Ruyi Ji, Jiajun Jiang, Shurui Zhou, Yiling Lou, Yingfei Xiong, and Gang Huang. 2021. Interactive patch filtering as debugging aid. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 239–250. doi:10.1109/ICSME52107.2021.00028

[32] Xixian Liang, Cheng Peng, Bo Ma, Xiangchen Shen, Yiheng Xiong, and Ting Su. 2026. Kea2: Practical Property-based Testing for Mobile Apps. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*. 182–186. doi:10.1145/3803437.3806416

[33] Ruofan Liu, Xiwen Teoh, Yun Lin, Guanjie Chen, Ruofei Ren, Denys Poshyvanyk, and Jin Song Dong. 2025. GUIPilot: A Consistency-Based Mobile GUI Testing Approach for Detecting Application-Specific Bugs. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 753–776. doi:10.1145/3728909

[34] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2024. PropertyGPT: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. *arXiv preprint arXiv:2405.02580* (2024). doi:10.14722/ndss.2025.241357

[35] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Yuekai Huang, Jun Hu, and Qing Wang. 2024. Unblind text inputs: predicting hint-text of text input in mobile apps via LLM. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–20. doi:10.1145/3613904.3642939

[36] Zhe Liu, Cheng Li, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Yawen Wang, Jun Hu, and Qing Wang. 2024. Seeing is Believing: Vision-driven Non-crash Functional Bug Detection for Mobile Apps. *arXiv preprint arXiv:2407.03037* (2024). doi:10.48550/arXiv.2407.03037

[37] Junayed Mahmud, Antu Saha, Oscar Chaparro, Kevin Moran, and Andrian Marcus. 2025. Combining language and app ui analysis for the automated assessment of bug reproduction steps. *arXiv preprint arXiv:2502.04251* (2025). doi:10.48550/arXiv.2502.04251

[38] Vikas K Malviya, Yan Naing Tun, Chee Wei Leow, Ailys Tee Xynyn, Lwin Khin Shar, and Lingxiao Jiang. 2023. Fine-Grained In-Context Permission Classification for Android Apps Using Control-Flow Graph Embedding. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1225–1237. doi:10.1109/ASE56229.2023.00056

[39] Meta. 2024. *Meet Llama 3.1* . https://www.llama.com/

[40] Michał Zalewski. 2016. *American Fuzzy Lop - Whitepaper*. https://lcamtuf.coredump.cx/afl/technical_details.txt

[41] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2023. Recent advances in natural language processing via large pre-trained language models: A survey. *Comput. Surveys* 56, 2 (2023), 1–40. doi:10.1145/3605943

[42] 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 (PLDI)*. 1025–1038. doi:10.1145/3519939.3523728

[43] Omni-Notes Team. 2024. *Omni-Notes*. https://github.com/federicoiosue/Omni-Notes

[44] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2025. An empirical study of the non-determinism of chatgpt in code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–28. doi:10.

1145/3697010

[45] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816. doi:10.1145/1297846.1297902

[46] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 815–825. doi:10.1109/ICSE.2012.6227137

[47] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. 2020. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In *2020 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 523–533. doi:10.1109/ICSME46990.2020.00056

[48] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318. doi:10.3115/1073083.1073135

[49] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with GPT-4. *arXiv preprint arXiv:2304.03277* (2023). doi:10.48550/arXiv.2304.03277

[50] Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. 2025. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326* (2025). doi:10.48550/arXiv.2501.12326

[51] Rustan Leino. 2009. *The Dafny Programming and Verification Language*. https://dafny.org/

[52] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *2015 IEEE/ACM 37th IEEE international conference on software engineering*, Vol. 1. IEEE, 666–676. doi:10.1109/ICSE.2015.82

[53] André Santos, Alcino Cunha, and Nuno Macedo. 2018. Property-based testing for the robot operating system. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 56–62. doi:10.1145/3278186.3278195

[54] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT software engineering notes* 30, 5 (2005), 263–272. doi:10.1145/1095430.1081750

[55] Sina Shamshiri. 2015. Automated unit test generation for evolving software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 1038–1041. doi:10.1145/2786805.2803196

[56] Simplenote Team. 2022. *Simplenote*. Retrieved 2025-5 from https://github.com/Automattic/simplenote-android

[57] John Ferguson Smart and Jan Molak. 2023. *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster.

[58] 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 (FSE)*. 245–256. doi:10.1145/3106237.3106298

[59] 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. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1088–1100. doi:10.1145/3611643.3616286

[60] 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*. 440–451. https://doi.org/10.1145/3663529.3663863

[61] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. 2012. Automating test automation. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 881–891. doi:10.1109/ICSE.2012.6227131

[62] Nikolai Tillmann, Jonathan De Halleux, and Tao Xie. 2014. Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 385–396. doi:10.1145/2642937.2642941

[63] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020). doi:10.48550/arXiv.2009.05617

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017). doi:10.48550/arXiv.1706.03762

[65] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. 2023. Can large language models write good property-based tests? *arXiv preprint arXiv:2307.04346* (2023). doi:10.48550/arXiv.2307.04346

[66] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *arXiv preprint arXiv:2406.01014* (2024). doi:10.48550/arXiv.2406.01014

[67] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837. doi:10.48550/arXiv.2201.11903

[68] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in Android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 543–557. doi:10.1145/3636534.3649379

[69] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. Droidbot-GPT: GPT-powered ui automation for Android. *arXiv preprint arXiv:2304.07061* (2023). doi:10.48550/arXiv.2304.07061

[70] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, et al. 2019. Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2436. doi:10.1145/3319535.3363193

[71] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023). doi:10.48550/arXiv.2304.00385

[72] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. Iconintent: automatic identification of sensitive ui widgets based on icon classification for Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 257–268. doi:10.1109/ICSE.2019.00041

[73] 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*. 53–64. doi:10.1145/3691620.3694986

[74] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An empirical study of functional bugs in Android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'23)*. 1319–1331. doi:10.1145/3597926.3598138

[75] Chenyang Yang, Shurui Zhou, Jin LC Guo, and Christian Kästner. 2021. Subtle bugs everywhere: Generating documentation for data wrangling code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 304–316. doi:10.1109/ASE51524.2021.9678520

[76] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. 2024. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1607–1619. doi:10.1145/3691620.3695529

[77] Shukang Yin, Chaoyou Fu, Sirui Zhao, Ke Li, Xing Sun, Tong Xu, and Enhong Chen. 2024. A survey on multimodal large language models. *National Science Review* 11, 12 (2024), nwae403. doi:10.1093/nsr/nwae403

[78] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving ChatGPT for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726. doi:10.1145/3660783

[79] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2025. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–20. doi:10.1145/3706598.3713600

[80] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023). doi:10.48550/arXiv.2303.12570

[81] Quanjun Zhang, Chunrong Fang, Yang Xie, Yuxiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A Systematic Literature Review on Large Language Models for Automated Program Repair. *arXiv preprint arXiv:2405.01466* (2024). doi:10.48550/arXiv.2405.01466

[82] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A Survey on Large Language Models for Software Engineering. *arXiv preprint arXiv:2312.15223* (2023). doi:10.48550/arXiv.2312.15223

[83] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. Recdroid: automatically reproducing Android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 128–139. doi:10.1109/ICSE.2019.00030

[84] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (2022). doi:10.48550/arXiv.2205.10625

[85] Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, Jun Wang, and Yong Yu. 2018. Texygen: A benchmarking platform for text generation models. In *The 41st international ACM SIGIR conference on research & development in information retrieval*. 1097–1100. doi:10.1145/3209978.3210080

Received 2026-01-30; accepted 2026-06-25