ProphetAgent: Automatically Synthesizing GUI Tests from Test Cases in Natural Language for Mobile Apps

Qichao Kong
East China Normal University / ByteDance
Shanghai, China
kongqichao@bytedance.com

Zhengwei Lv
ByteDance
Beijing, China
lvzhengwei.m@bytedance.com

Yiheng Xiong
East China Normal University
Shanghai, China
xyh@stu.ecnu.edu.cn

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

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

Dingchun Wang
ByteDance
Beijing, China
wangdingchun@bytedance.com

Letao Li
ByteDance
Beijing, China
liletao@bytedance.com

Xu Yang
ByteDance
Beijing, China
yangxu.swanoofl@bytedance.com

Gang Huo
ByteDance
Beijing, China
huogang@bytedance.com

Abstract

GUI tests is crucial for ensuring software quality and user satisfaction of mobile apps. In practice, companies often maintain extensive test cases written in natural language. Testers need to convert these test cases into executable scripts for regression and compatibility testing. Requirement changes or version updates often necessitate the addition and modification to these test cases. Thus, when faced with large volumes of test cases and regular updates, this process becomes costly, which is a common challenge across the industry. To address this issue, this paper proposes ProphetAgent that can automatically synthesize executable GUI tests from the test cases written in natural language. ProphetAgent first constructs a Clustered UI Transition Graph (CUTG) enriched with semantic information, then leverages large language models to generate the executable test case based on CUTG and test cases written in natural language. Experiment results show that ProphetAgent achieved a 78.1% success rate across 120 test cases in Douyin, Doubao, and six open-source apps, surpassing existing automated approaches (21.4% for AppAgent and 32.5% for AutoDroid). Additionally, statistical data from ByteDance's testing platform show that ProphetAgent increased testers' efficiency in synthesizing UI tests by 260%.

1 Introduction

Mobile apps are ubiquitous in people's daily life, and some software errors can lead to significant financial losses and user dissatisfaction [1, 6]. In practice, companies typically maintain numerous test cases written in natural language, and testers need to convert these test cases into executable GUI tests. These GUI tests can be run automatically to expedite regression and compatibility testing. However, writing and maintaining GUI tests still requires significant effort. For instance, ByteDance, the provider of Douyin (a popular social media app with over 800 million monthly active users worldwide[5]), maintains tens of thousands of test cases written in natural language. Each test case contains the start scene and test steps, such as: *In the main page (i.e., the start scene of this test case), click the Friends tab, click the add friends button, click the Follow button, click the Scan button and click the My QR code button.* ByteDance's testers are required to convert these natural language test cases into corresponding executable GUI tests and update them alongside requirement changes and version updates. This process typically involves: exploring the app until they find the start UI page, sequentially matching each test step to the appropriate UI widgets, writing GUI tests based on the identified UI widgets, and modifying the GUI tests if they fail during execution. A report from a testing team in ByteDance shows that each novice tester can only write (synthesizing GUI tests from natural language) and maintain (updating them with each app version change) 150 test cases per quarter, while even experienced testers can only write and maintain 700 test cases. Therefore, ByteDance needs significant human effort to write and maintain tens of thousands of test cases.

The rise of automated testing has gained significant attention, and more companies hope to automate GUI test synthesis to cut labor costs. Key challenges include accurately parsing natural language and aligning test cases with mobile app details (including matching the start scene with the corresponding UI page, and aligning each test step with the target UI widget). Some studies have explored converting natural language descriptions into executable GUI tests, which involves challenges similar to those faced in synthesizing GUI tests. Previous approaches [12, 22, 28] typically rely on Natural Language Processing (NLP) or Deep Learning (DL) to parse natural language descriptions and complete the automation conversion. However, traditional NLP and DL techniques have limited capabilities in natural language understanding, resulting in low accuracy of the conversion results. The rise of Large Language Models (LLM) [3, 18] has shown improved potential for addressing this issue. For example, some work[22, 25, 27] has leveraged LLMs

*Corresponding authors.

to automate the execution of simple natural language commands. However, these techniques face some challenges, making it difficult to synthesize GUI tests for complex, specialized functionalities. First, vision-based techniques often face difficulties as visual LLMs struggle to accurately capture all widgets on a page and interpret their semantic meanings. Second, these techniques entirely rely on the general app knowledge learned by LLMs for decision-making. However, LLMs often fail to predict the interaction outcomes of candidate widgets, leading to incorrect matches.

To this end, we propose ProphetAgent, an efficient GUI test case synthesis technique. The key insight of ProphetAgent is to construct a knowledge graph that characterizes the app's behavior and scenarios. By leveraging both the knowledge graph and LLMs, we enable the synthesis of GUI tests from test cases written in natural language. Specifically, ProphetAgent starts by exploring the target app to collect all transition information. It then annotates the UI pages and events in the transition information with semantic meanings and clusters them, ultimately producing a semantically enriched Clustered UI Transition Graph (CUTG), which serves as the knowledge graph. Then, ProphetAgent matches natural language test steps with UI events in the CUTG and generates executable code based on the matched events. Our approach offers two advantages over existing techniques: (1) By pre-building a CUTG with semantically annotated nodes, our approach provides more precise semantic information to assist LLMs in matching natural language descriptions with UI widgets, (2) It does not require using LLM to process all the app information, which cuts down on training costs and increases accuracy.

To evaluate ProphetAgent, we applied it to 120 test cases written in natural language from two industrial apps (Douyin and Doubao), and six open-source apps. The results show that ProphetAgent achieved a 78.1% completion rate, significantly outperforming existing tools (21.4% for AppAgent [27] and 32.5% for AutoDroid [25]). Additionally, ProphetAgent exhibited a high action accuracy of 83.3% (compared to 28.2% for AppAgent and 36.5% for AutoDroid). Furthermore, the average time to synthesize a GUI test with ProphetAgent was 127 seconds, shorter than AppAgent's 138 seconds and AutoDroid's 339 seconds. We also conducted statistics on ByteDance's testing platform, revealing that before using ProphetAgent, testers managed an average of 25 test cases per day, whereas now they can manage 90 test cases daily, representing an astonishing 260% increase in tester efficiency. These findings highlight ProphetAgent 's efficiency and effectiveness in enhancing app quality assurance through advanced semantic understanding and precise action execution. ProphetAgent has been made publicly available at https://github.com/prophetagent/Home.

2 ProphetAgent

The primary goal of ProphetAgent is to enable developers and testers to automatically execute test steps in natural language. Figure 1 provides an overview of our approach, which includes a GUI Explorer, Graph Builder, and two key Agents: SemanticAgent and GenerationAgent. The GUI Explorer performs in-depth fuzzing on the given app and records all GUI transition events. SemanticAgent extracts essential information from these raw GUI operation sequences and annotates them with distinct semantics, setting the stage for informed decision-making by GenerationAgent. Subsequently, the Graph Builder applies clustering and graph construction rules to create a semantically enriched Clustered UI Transition Graph (CUTG). Then, GenerationAgent matches a path in this directed graph that conforms to the test steps.

2.1 GUI Explorer

We deploy the GUI Explorer to explore the target app and record the GUI transition information after each UI event is executed. Each piece of transition information is represented as a tuple $\langle p, e, p' \rangle$, where p denotes the UI page before the execution of the UI event, captured as the XML tree structure of the GUI layout, e describes the executed UI event, which consists of the action taken (such as clicking, long pressing, or editing) and the target widget, and p' represents the UI page after the execution of the UI event, also encapsulated as an XML tree.

2.2 SemanticAgent

SemanticAgent annotates each transition information with explicit semantic information on UI pages and UI events. Next, we will explain how to annotate pages and events separately.

Semantic annotation of UI pages. To extract semantic information from the UI page, we first group and optimize the UI page's XML tree to focus exclusively on widgets within the target app, excluding system widgets such as the device's battery level. We categorize these XML entries into two groups: layouts and UI components. We remove the layout entries and keep only the UI components. Specifically, we recursively process all nodes, merge non-functional, invisible, or non-interactive nodes into their parent nodes, and combine their descriptive information into the parent node. In our experiments, it can reduce the average number of nodes per page from 148 to 70, effectively decreasing the number of tokens needed for LLM input. We also compare the similarity of consecutive pages: if the difference in nodes is less than five, we provide detailed explanations for these nodes. This reduced the token count for a page information request from about 40,000 to 1,500. Additionally, WebView pages often lack explicit information within the XML tree. To address this, we use OCR[21] to extract textual information from the page as supplementary data for further processing by the LLM.

Semantic annotation of UI events. To extract event information, we focus on the target widget of the event and the changes between previous and subsequent UI pages. For each target widget, we inspect the presence of "text" or "content-desc" attributes, as these often indicate the widget's function or semantic meaning. If such attributes are available, they are stored as semantic information. In their absence, we use "resource-id" and "class" attributes as substitutes. To get more information, we also extract information from the target widget's child and parent widgets. For child widgets, we focus on the "bounds" attribute to evaluate their spatial relationship with the target widget. If a child widget occupies more than 75% of a target widget's area, we extract its semantic information using the same rules as for the target widget. Similarly, if a widget's area exceeds 75% of its parent widget's area, we also extract the parent widget's key information. We obtain a regional screenshot of the event area during operation, which serves as supplementary information to enhance the GPT-4o[16] model's understanding. Additionally, changes between previous and subsequent UI pages

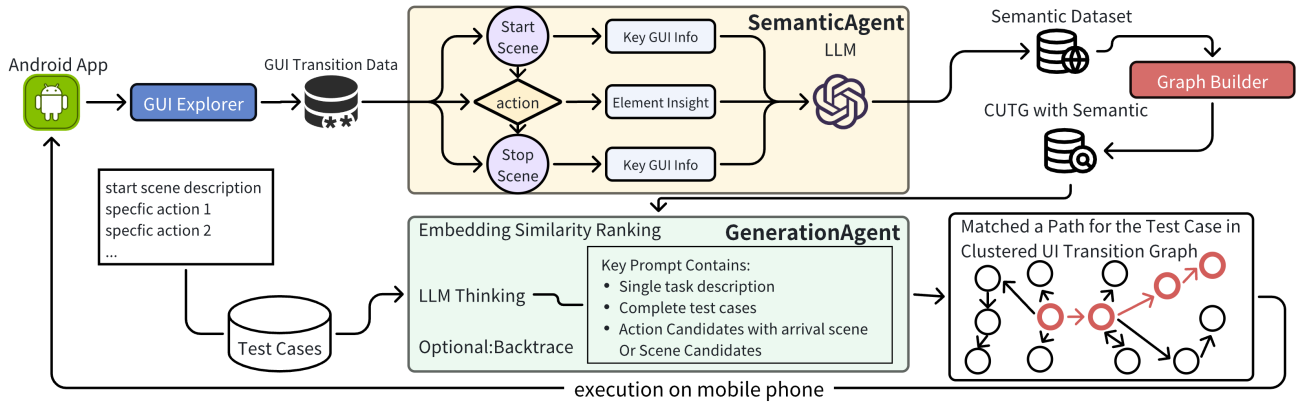

Figure 1: Overview of our ProphetAgent

can reveal critical event information. To assist the LLM in understanding the event, we extract key UI page details and provide the model with node changes from both UI pages.

2.3 Graph Builder

The Graph Builder leverages GUI transition data collected by the GUI Explorer to cluster UI pages and events, constructing the CUTG enriched with semantic information. The clustering of page nodes is conducted in real-time during the GUI exploration process, based on the current UI page's activity and qualified widget attributes. Specifically, widgets with a predefined depth range are extracted, and their Class and Resource-ID attributes are used to characterize them. The predefined depth is dynamically determined based on the complexity of the app's XML structure. All qualified widgets and activities form a unique identifier for the UI page, which is then assigned to a unique hash value. The clustering of event nodes is derived from data recorded during the Fuzzing process. Each event is uniquely identified based on the hash values of the preceding and subsequent UI pages, the type of event, and the widget's Resource-ID, and is assigned a unique hash value. During graph construction, both UI pages and events are designated as nodes, which enhances the representation and storage of event information while clarifying relationships within the clustered graph.

2.4 GenerationAgent

GenerationAgent is responsible for matching a path in the CUTG that aligns with the test case written in natural language and generating executable code for execution. We propose an approach that combines embedding[17] similarity for initial filtering with an LLM for fine-grained selection, enabling accurate matching of UI pages and event nodes in the CUTG. To address potential failures and hallucinations that can arise from embedding similarity and LLM matching in some extreme cases, a dynamic rollback mechanism is employed. Specifically, the start scene of the test case is matched by calculating the cosine similarity of embeddings for all UI page nodes in the graph database, from which the top 50 candidates are selected. The LLM then chooses the UI page node that best matches the start scene. For the sequence of event nodes, we first locate the starting UI page node in the graph database and retrieve all outgoing nodes, which represent the candidate set of event nodes. For each event description, the resulting UI page information is obtained from the graph database, predicting the effects of executing all candidate events. The matching process for the best event node is the same as for UI page nodes. If the LLM determines that no node in the candidate set satisfies the current operation, a dynamic recursive rollback is triggered to rematch the previous node. This process continues until all operation sequences are matched, forming a complete executable code path.

3 Tool Implementation

We have developed a prototype tool, ProphetAgent, to support our approach. Given an application and a set of prepared test cases. ProphetAgent traverses the application to generate a CUTG, selects a path in the CUTG that meets the test-case requirements, and generates an executable script, which is subsequently run. For the traversal task, we employ an improved version of Droidbot [13] to perform thorough exploration. We use the graph database Neo4j [15] to store and index the CUTG map data we have constructed. We utilize Doubao's embedding model [24] and a cosine similarity algorithm for text similarity matching. We rely on the GPT4o model as the foundation for our Agent, enabling it to accomplish various tasks. Finally, we adopt the uiautomator2 testing framework [4] to assist in executing the generated scripts. A demonstration of ProphetAgent is available at https://youtu.be/iCsGis__5gg.

4 EVALUATION

Our experiment aims to answer these research questions:

- **RQ1** : How effective and efficient is ProphetAgent in synthesizing GUI tests from the test steps in natural language, and how does it compare with state-of-the-art techniques?
- **RQ2** : What is the impact of different LLM agents on the method we propose?

4.1 Experiment Setup

App subjects. First, we selected two industrial apps from ByteDance, Douyin (version 29.7.0) and Doubao (version 6.3.0), for the following reasons: (1) their popularity (over 800 million monthly active users) and complexity (with hundreds of app states), which provide a more realistic evaluation of the method's effectiveness; (2) our collaboration with ByteDance, which allowed us to access real-world test cases as inputs. Additionally, to assess the method's

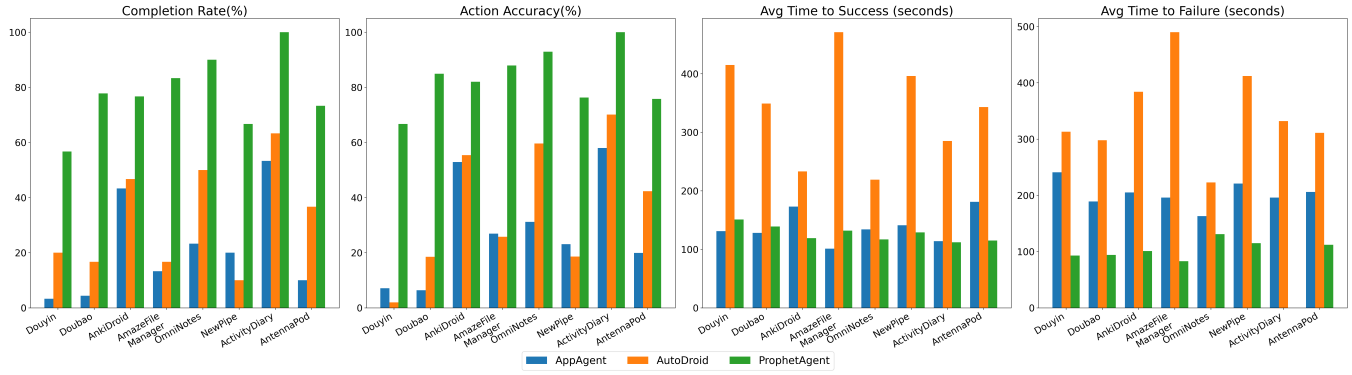

Figure 2: Comparative Performance Analysis of AppAgent, AutoDroid and ProphetAgent Across Various Apps

generalizability, we selected six additional representative open-source Android apps from prior research on functional testing of Android apps [26]. Other apps with similar features or that have been archived were excluded. The selected apps include AnkiDroid (version 2.18.1, 9k+ stars in Github), AmazeFileManager (version 3.10, 5.4k+ stars), OmniNotes (version 6.3.1, 2.7k+ stars), NewPipe (version 0.27.4, 32.3k+ stars), ActivityDiary (version 1.4.2, 73 stars), and AntennaPod (version 3.2.0, 6.6k+ stars).

Evaluation method of RQ1. To answer RQ1, we evaluated the effectiveness of ProphetAgent on the app subjects we selected. Specifically, ByteDance provided us with 30 real-world test cases each for Douyin and Doubao as test inputs. For the other six open-source apps, we developed 10 test cases per app, focusing on core activity pages and functionalities. Each test case consisted of 4–15 steps. We compared our method with two key techniques for converting natural language into executable GUI tests: a most established technique (AppAgent [27]) and a recent state-of-the-art technique (AutoDroid [25]), both of which utilize LLMs to generate executable GUI tests for Android apps from sentences describing GUI tasks. We excluded other research that was either closed-source or too costly to include, but we still discuss the differences with them in section 6. AppAgent and AutoDroid were configured with the latest GPT-4o [16] across the same 120 test cases for a fair comparison. To effectively compare the performance of different agents, we executed each test case three times and adopted the average values from these runs for four key metrics.

- **Completion Rate (CR)**: This metric measures the overall completion rate ($\frac{\text{number of correctly synthesized test cases}}{\text{the total number of test cases}}$) in consistently and successfully synthesizing the test case within the app. Note that the "correctly synthesized test cases" refer to the cases synthesized by the tool that were manually judged as correct, with false positives (cases where the agent incorrectly assumes success) being manually filtered out.
- **Action Accuracy (AAC)**: This metric measures the accuracy of the steps ($\frac{\text{number of correct steps}}{\text{the total number of steps generated by ProphetAgent}}$) performed. Despite the tool not always achieving success in fully generating GUI tests, each successfully synthesized step reduces the time required for testing.
- **Average Time to Success (ATS)**: We recorded the average time required to successfully execute test cases, reflecting the agent's response speed and processing capabilities in actual operations. Notably, this metric is rarely recorded in other real-time executing agents.
- **Average Time to Failure (ATF)**: We also recorded the average time taken when test cases fail. Ideally, the agent should immediately terminate operations and report errors when the fault occurs, thereby saving resources and enhancing the overall efficiency of the test.

Among the four metrics mentioned above, higher CR and AAC indicate greater effectiveness of the tool, while lower ATS and ATF (measured in seconds) represent higher efficiency of the tool.

Furthermore, to assess the efficacy of ProphetAgent in reducing testing overhead, we measured the efficiency of test case conversions by testers on ByteDance's testing platform (used by over 1000 testers), both before and after using ProphetAgent.

Evaluation method of RQ2. To answer RQ2, we conducted tests using a different LLM, Doubao-pro-128k, under the same experimental conditions as in RQ1. Specifically, we replaced the GPT-4o model in both SemanticAgent and GenerationAgent with the Doubao model. We manually analyzed and recorded the evaluation results, focusing on four same metrics and identifying the causes of any discrepancies.

4.2 Results for RQ1.

Figure 2 shows the detailed performance metrics. ProphetAgent still exhibits a clear advantage in mobile app automation testing, reaching an average completion rate (CR) of 78.1%—markedly surpassing both AppAgent (21.4%) and AutoDroid (32.5%). This reflects ProphetAgent's superior capability to complete test cases. Such an improved CR largely stems from the use of CUTG, which not only provides semantic insights into UI widgets and their subsequent effects but also employs a dynamic rollback mechanism when tracing reachable paths. In terms of action accuracy (AAC), ProphetAgent posts 83.3%, while AppAgent and AutoDroid achieve 28.2% and 36.5%, respectively, indicating that ProphetAgent tends to execute a larger portion of the necessary steps, even if it does not fully complete certain test scenes. We explored why these two tools fell short of expectations, and found that the primary reason lies in the underlying LLMs. They lack sufficient knowledge of GUI pages and are thus unable to fully grasp the semantics of the widgets, which in turn prevents them from accurately mapping the test steps to the appropriate UI widgets. Meanwhile, ProphetAgent 's average time to

success (ATS) is 127 seconds, falling below AppAgent's 138 seconds and considerably under AutoDroid's 339 seconds, thereby highlighting more efficient test execution. Lastly, ProphetAgent 's average time to failure (ATF) of 104 seconds also proves notably shorter compared to 202 seconds for AppAgent and 345 seconds for AutoDroid, as it rapidly terminates when path matching fails in the CUTG and does not execute the GUI tests on a real device, thereby enhancing overall testing efficiency.

In terms of reducing testing overhead, ByteDance's testing platform statistics show that ProphetAgent significantly reduced testing overhead, boosting tester efficiency from 25+ to 90+ test cases per day. Additionally, testers at ByteDance mentioned that ProphetAgent can help them handle over 70% of the test case synthesis, leaving them with less than 30% of the original workload, which significantly improves their work efficiency.

4.3 Results for RQ2.

Our evaluation of 120 test cases shows only a modest decline in performance after replacing the GPT-4o model with the Doubao model. Specifically, the completion rate decreased from 78.1% to 69.8%, while the action accuracy increased from 83.3% to 77.5% . Notably, the average execution time decreased by 13%, indicating a more efficient runtime when using Doubao. These observations suggest that our tool does not heavily depend on GPT-series models; substituting GPT-4o with other open-source LLMs results in only a moderate reduction in effectiveness. In practice, this flexibility allows developers to balance performance, cost, and other factors by choosing among different LLM providers. Furthermore, Doubao's cost per million tokens is approximately one-seventh that of GPT-4o [2, 19], offering a significant economic advantage for large-scale or continuous deployments. Consequently, our approach provides both adaptability and cost-effectiveness, making it more accessible to a broader range of users and apps.

5 Discussion

Limitations of ProphetAgent. Despite our evaluation results demonstrating that ProphetAgent is highly effective and accurate in synthesizing textual test cases, ProphetAgent does have some limitations. First, ProphetAgent is designed to handle specific test cases rather than high-level intents. This means that each sentence in a test case description must denote a single action. In contrast to research focused on mobile task automation, which targets more general scenarios, our work concentrates on app-specific and intricate testing steps and scenarios in mobile apps. Second, ProphetAgent can only generate executable test scripts for scenarios that are included in our constructed knowledge graph. In our experiments, we employ Droidbot [13] for exploration, utilizing basic exploration algorithms, and have achieved a high generation rate of over 70%. We believe that as exploration technologies continue to advance—a field that is receiving considerable attention—the effectiveness of our algorithm will further improve, allowing ProphetAgent to handle a broader range of scenarios.

Threats to Validity. The external validity of our results could be compromised by the representativeness of the selected apps and the precision of the input textual test cases. We addressed these threats by choosing widely used industrial apps and emphasizing the need for meticulous formatting of test cases to ensure accurate parsing and execution, aiming to make our findings more generalizable to typical industry settings. When it comes to internal threats, the semantic annotation accuracy is a potential threat to the method's effectiveness. To evaluate the accuracy of SemanticAgent, we recruited five professionals to verify 3,500 instances of GUI transition information in our Douyin test dataset. The results showed that SemanticAgent achieved an accuracy rate of 88%.

6 Related Work

Automating test automation. Research on automating test automation is limited. Early studies[20, 23] focused on converting natural language test cases into executable GUI tests, addressing ambiguities through backtracking. With AI advancements, recent approaches[10, 11] focused on utilizing pretrained models to translate test intents into GUI tests. Our method diverges by integrating knowledge graph data and leveraging advanced model judgment, marking the first initiative to separate test generation from execution. Although we cannot compare our tool directly with those from previous studies (because these works have not made their tools public), our approach appears to be more efficient and effective based on their published results.

Mobile task automation. Recent years have seen a surge in research [12, 22, 25, 27] on mobile task automation (i.e., enabling hands-free user interaction with smartphones). For instance, META-GUI [22] introduces a GUI-based dialogue system that directly manipulates app interfaces to execute tasks, AppAgent [27] builds a knowledge base from autonomous exploration or human demonstrations to execute tasks, and AutoDroid [25] integrates large language models with dynamic app analysis to automate arbitrary tasks on Android apps. Unlike these techniques, which focus on simple, general tasks, ProphetAgent shows better performance in experiments (as shown in Section 4), highlighting the limitations of these approaches in complex scenarios.

Bug Report Reproduction. Some work [7–9, 14] aims to automatically reproduce user-written bug reports. These studies also involve the automated execution of interaction steps written in natural language. These work primarily attempt to reproduce bugs by trying all possible paths until the same exception is triggered. Since test case synthesis tasks lack a clear termination condition (i.e., the specific erroneous behavior of the app), these approaches cannot be directly applied to test case synthesis tasks.

7 Conclusion

This paper presents ProphetAgent, a novel agent that automatically synthesizes executable GUI tests from the test cases written in natural language. By leveraging large-scale GUI fuzzing testing, semantic annotations, and constructing a Clustered User Transition Graph (CUTG), ProphetAgent accurately matches test cases and anticipates GUI transitions . This approach achieves an execution completion rate of 78.1% across 120 test cases and maintains a low false positive rate of 6.8%, which surpasses existing tools in experiments. Furthermore, ProphetAgent significantly reduces testing overhead, increasing tester efficiency by 260% (from 25+ to 90+ test cases per day). The ability to pre-generate and review executable code before execution enhances the reliability and speed of test case automation, making ProphetAgent a valuable tool for handling complex test cases while reducing operational costs.

Acknowledgments

We thank the anonymous FSE reviewers for their valuable feedback. This work was partially supported by NSFC Project (No.62072178), ByteDance Research Fund, Shanghai Collaborative Innovation Center of Trusted Industry Internet Software, "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (No.22510750100).

References

[1] 9to5Mac. 2024. App Store experiences downtime, affecting users worldwide. https://9to5mac.com/2024/04/03/app-store-down/

[2] ByteDance. 2024. Pricing Doubao. https://console.volcengine.com/ark/region:ark+cn-beijing/model/detail?Id=doubao-pro-128k.

[3] Claude. 2023. The Claude 3 Model Family: Opus, Sonnet, Haiku. https://api.semanticscholar.org/CorpusID:268232499

[4] Appium Contributors. 2023. Appium UIAutomator2 Driver. https://github.com/appium/appium-uiautomator2-driver.

[5] Digital Crew. 2023. Douyin Trends Every Marketer Should Know. https://www.digitalcrew.agency/douyin-trends-every-marketer-should-know/

[6] TechNode Feed. 2024. Alipay bugs allow users to get 20% discount on orders, no reimbursement to follow. https://technode.com/2025/01/17/alipay-bugs-allow-users-to-get-20-discount-on-orders-no-reimbursement-to-follow/

[7] Sidong Feng and Chunyang Chen. 2022. GIFdroid: automated replay of visual bug reports for Android apps. In *Proceedings of the 44th International Conference on Software Engineering* (New York, NY, USA) *(ICSE '22)*. Association for Computing Machinery, 1045–1057. https://doi.org/10.1145/3510003.3510048

[8] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA) *(ICSE '24)*. Association for Computing Machinery, Article 67, 13 pages. https://doi.org/10.1145/3597503.3608137

[9] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction *(ICSE '23)*. IEEE Press, 2312–2323. https://doi.org/10.1109/ICSE48619.2023.00194

[10] Chun Li. 2022. Mobile GUI test script generation from natural language descriptions using pre-trained model. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems* (New York, NY, USA) *(MOBILESoft '22)*. Association for Computing Machinery, 112–113. https://doi.org/10.1145/3524613.3527809

[11] Chun Li, Yifan Xiong, Zhong Li, Wenhua Yang, and Minxue Pan. 2023. Mobile Test Script Generation from Natural Language Descriptions. In *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 348–359. https://doi.org/10.1109/QRS60937.2023.00042

[12] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping Natural Language Instructions to Mobile UI Action Sequences. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, 8198–8210. https://doi.org/10.18653/v1/2020.acl-main.729

[13] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-Guided test input generator for android. In *ICSE-C '17*. 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[14] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic matching of GUI events for test reuse: are we there yet?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA) *(ISSTA 2021)*. Association for Computing Machinery, 177–190. https://doi.org/10.1145/3460319.3464827

[15] Neo4j. 2023. Neo4j: The Leader in Graph Databases. https://neo4j.com/.

[16] OpenAI. 2023. Hello, GPT-4o. https://openai.com/index/hello-gpt-4o/.

[17] OpenAI. 2023. New Embedding Models and API Updates. https://openai.com/index/new-embedding-models-and-api-updates/.

[18] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[19] openAI. 2024. Pricing GPT. https://openai.com/api/pricing/.

[20] Pablo Pedemonte, Jalal Mahmud, and Tessa Lau. 2012. Towards automatic functional test execution. In *Proceedings of the 2012 ACM International Conference on Intelligent User Interfaces* (New York, NY, USA) *(IUI '12)*. Association for Computing Machinery, 227–236. https://doi.org/10.1145/2166966.2167005

[21] pytesseract. 2023. pytesseract. https://pypi.org/project/pytesseract/.

[22] Liangtai Sun, Xingyu Chen, Lu Chen, Tianle Dai, Zichen Zhu, and Kai Yu. 2022. META-GUI: Towards Multi-modal Conversational Agents on Mobile GUI. In *Conference on Empirical Methods in Natural Language Processing*. https://api.semanticscholar.org/CorpusID:248986378

[23] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. 2012. Automating test automation. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 881–891. https://doi.org/10.5555/2337223.2337327

[24] volcengine. 2025. volcengine Embeddings. Retrieved 2025-1 from https://www.volcengine.com/docs/82379/1302003/

[25] 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 *MobiCom '24* (New York, NY, USA) *(ACM MobiCom '24)*. Association for Computing Machinery, 543–557. https://doi.org/10.1145/3636534.3649379

[26] 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* (New York, NY, USA) *(ASE '24)*. Association for Computing Machinery, 53–64. https://doi.org/10.1145/3691620.3694986

[27] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023. AppAgent: Multimodal Agents as Smartphone Users. arXiv:2312.13771 [cs.CV] https://arxiv.org/abs/2312.13771

[28] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ 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.