

Safeguarding Mobile GUI Agent via Logic-based Action Verification

Jungjae Lee*
School of Computing, KAIST
Republic of Korea
dlwjdwo00701@kaist.ac.kr

Dongjae Lee*
School of Computing, KAIST
Republic of Korea
dongjae.lee00@kaist.ac.kr

Chihun Choi
Korea University
Republic of Korea
clgns0102@korea.ac.kr

Youngmin Im
School of Computing, KAIST
Republic of Korea
ym.im@kaist.ac.kr

Jaeyoung Wi
School of Computing, KAIST
Republic of Korea
wijaeyoung@kaist.ac.kr

Kihong Heo
School of Computing, KAIST
Republic of Korea
kihong.heo@kaist.ac.kr

Sangeun Oh
Korea University
Republic of Korea
sangeunoh@korea.ac.kr

Sunjae Lee[†]
Sungkyunkwan University
Republic of Korea
sunjae.lee@skku.edu

Insik Shin[†]
School of Computing, KAIST
Republic of Korea
ishin@kaist.ac.kr

Abstract

Large Foundation Models (LFMs) have unlocked new possibilities in human-computer interaction, particularly with the rise of mobile Graphical User Interface (GUI) Agents capable of interpreting GUIs. These agents promise to revolutionize mobile computing by allowing users to automate complex mobile tasks through simple natural language instructions. However, the inherent probabilistic nature of LFMs, coupled with the ambiguity and context-dependence of mobile tasks, makes LFM-based automation unreliable and prone to errors. To address this critical challenge, we introduce VeriSafe Agent (VSA): a formal verification system that serves as a logically grounded safeguard for Mobile GUI Agents. VSA is designed to deterministically ensure that an agent’s actions strictly align with user intent before conducting an action. At its core, VSA introduces a novel *autoformalization* technique that translates natural language user instructions into a formally verifiable specification, expressed in our domain-specific language (DSL). This enables runtime, rule-based verification, allowing VSA to detect and prevent erroneous actions executing an action, either by providing corrective feedback or halting unsafe behavior. To the best of our knowledge, VSA is the first attempt to bring the rigor of formal verification to GUI agent, effectively bridging the gap between LFM-driven automation and formal software verification. We implement VSA using off-the-shelf LLM services (GPT-4o) and evaluate its performance on 300 user instructions across 18 widely used mobile apps. The results demonstrate that

VSA achieves 94.3%–98.33% accuracy in verifying agent actions, representing a significant 20.4%–25.6% improvement over existing LLM-based verification methods, and consequently increases the GUI agent’s task completion rate by 90%–130%.

1 Introduction

The advent of Large Foundation Models (LFMs) [1–3] has revolutionized human-computer interaction, paving the way for a new generation of agents capable of interacting with graphical user interfaces (GUIs) [4–13]. Among these, Mobile GUI Agents stand out for their ability to automate complex tasks within mobile applications, reducing manual effort and enhancing user convenience. By leveraging the reasoning and natural language understanding capabilities of LFMs, these agents can interpret user requests and translate them into sequences of UI interactions [4, 6–8].

However, despite significant advancements, existing Mobile GUI Agents still face fundamental limitations that hinder their reliability and safety in real-world applications. One major challenge stems from the probabilistic nature of LFMs, which can lead to unpredictable and erroneous actions. Additionally, mobile app interactions are often ambiguous and context-dependent, making it difficult even for state-of-the-art LFMs to generate consistently accurate actions [12, 14, 15]. These challenges are especially critical for tasks involving sensitive operations, such as financial transactions or private communications, where errors can have serious or irreversible consequences. Therefore, implementing robust safeguards and verification mechanisms is

*Co-first authors : Jungjae Lee, Dongjae Lee

[†]Co-corresponding authors : Sunjae Lee, Insik Shin

essential to ensuring the safe and reliable deployment of Mobile GUI Agents.

Recent approaches [8, 16] have attempted to mitigate these challenges by introducing reflection agents that use LFMs to review the actions of the primary GUI agent and provide feedback. However, these approaches come with significant drawbacks. Since reflection agents also rely heavily on LFMs, they remain susceptible to the same errors, leading to a compounding of inaccuracies. Furthermore, repeated LFM queries for reflection incur substantial computational costs and latency, further limiting their practicality. These limitations highlight three key challenges inherent in relying solely on LFMs for verification:

1) *Existence of Irreversible and Risky Action*: One way to improve verification accuracy is to analyze both the action and its resulting screen state. While the app screen after the action (*post-action verification*) provides useful context for verification, many critical mobile actions (e.g., making a payment, sending a message) are irreversible. In such cases, verification becomes meaningless, as the agent cannot filter or "undo" the erroneous actions. This fundamentally defeats the core purpose of verification, which prevents critical errors before they occur. To ensure genuine safeguard against critical errors, verification must be performed before executing the action (*pre-action*).

2) *The Forward Assessment Problem*: To Verify an agent's action before its execution, it requires a forward-looking assessment: predicting the effect of the proposed action before execution. However, this process can be as difficult – if not more difficult – than generating the action itself. Furthermore, the predicted outcome aligns with the user's overall goal adds another layer of complexity. Consequently, this approach is akin to solving a simpler problem with a more complex one, potentially turning verification into a performance bottleneck.

3) *Compounding Error Probability*: Mobile tasks typically comprise a sequence of actions, each requiring verification. Continuously relying on LFMs for validation at every step increases the likelihood of accumulated errors, which can ultimately degrade overall system accuracy, despite efforts to improve it.

To overcome these challenges, we introduce VeriSafe Agent (VSA), a deterministic, logic-based verification system for Mobile GUI Agents. To the best of our knowledge, this is the first attempt to provide a reliable pre-action verification mechanism grounded in logic-based reasoning rather than existing probabilistic methods, bridging the gap between probabilistic actions and deterministic safety. Specifically, VSA leverages the concept of *autoformalization* [17], which automatically translates natural language user instructions into a formal specification. Using the translated specification,

VSA performs runtime verification to ensure the correctness of the agent's actions in runtime.

However, unlike conventional software verification [18–21], which operates on predefined safety requirements, VSA faces the unique challenge of formalizing impromptu, user-defined instructions on-the-fly, across diverse and dynamic landscape of mobile applications. To accomplish this, VSA introduces three key innovations:

- (1) *Domain-Specific Language (DSL) and Developer Library*: A specialized language and accompanying tools tailored to dynamic nature of mobile environments. Collectively, they can encode both the natural-language user instructions and corresponding UI actions as logical formulas.
- (2) *Intent Encoder and Verification Engine*: A system that systematically translates user instructions into logical constraints (expressed as our DSL) and performs runtime, deterministic verification against these constraints.
- (3) *Structured Feedback Generation*: A proactive mechanism that provides actionable feedback to the GUI agent, *before* executing an action, identifying specific logical violations and unmet conditions to guide the agent toward correct task completion.

We implement a prototype of VSA using off-the-shelf online LLM service (GPT-4o) and integrate it with M3A (Multimodal Autonomous Agent for Android) GUI agent [12]. Our evaluation, conducted on 300 user instructions across 18 widely used mobile applications demonstrates that with an estimated 10 additional lines of code (LoC) per app, VSA successfully verifies GUI agent actions with up to 98.33% accuracy, achieving a false positive rate of 2.7% and a false negative rate of 0.7%. This significantly outperforms baseline reflection agents by 25.6%. Furthermore, VSA's structured feedback, derived from logical verification results, enables the GUI agent to enhance its task completion rate by 130%, a substantial improvement compared to reflection agents, which failed to correct any tasks.

2 Background and Motivation

Graphical User Interface (GUI) Agents. GUI Agents [4–10] are software programs designed to interact autonomously with applications through graphical user interfaces. By interpreting visual and semantic information presented in GUIs and simulating human interactions (e.g., clicks, swipes, and text entry), GUI agents enable automation of digital tasks without requiring internal access to application code or APIs. Recent advancements in artificial intelligence, particularly driven by Large Foundation Models (LFMs), have significantly enhanced the capabilities of GUI agents, enabling

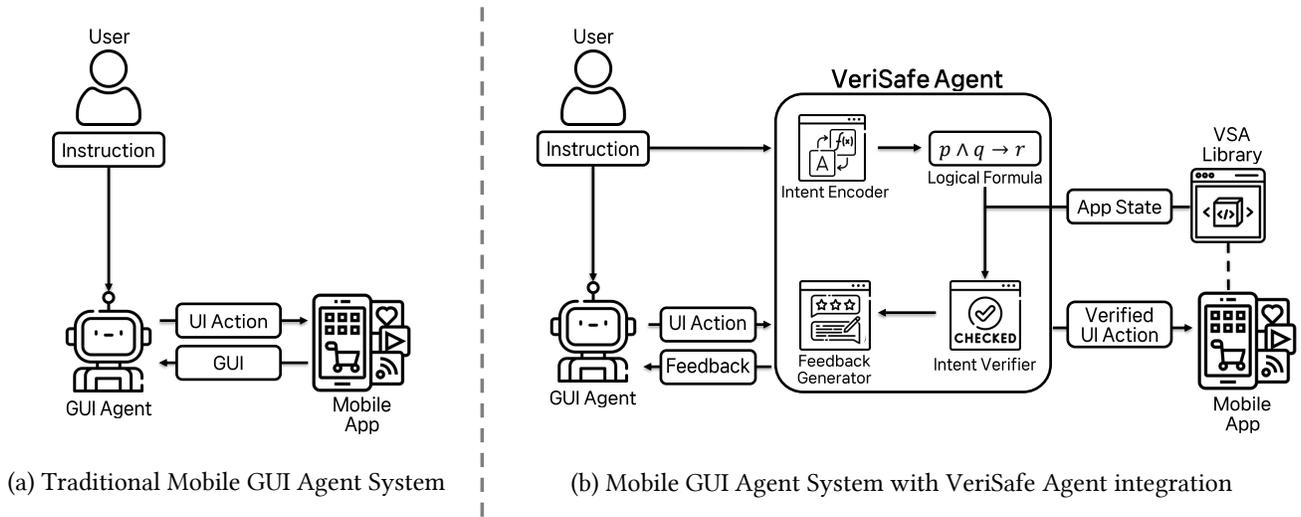


Figure 1: Comparison of (a) a traditional GUI agent system and (b) VSA integrated system.

them to understand complex user instructions and interpret sophisticated mobile application GUIs. However, due to the inherently probabilistic nature of LFM and often ambiguous and under-specified nature of GUI interactions, contemporary GUI agents are susceptible to incorrect actions [14, 15, 22–26], necessitating robust verification mechanisms to ensure reliability and safety in practical scenarios.

Safeguarding GUI Agents using Reflection. The current practice for safeguarding GUI agents often involves the use of *reflection agents*, which utilize LFM to review the actions generated by the primary GUI agent. Reflection-based approach can be broadly classified into two categories: pre-action verification and post-action verification.

Pre-action reflection [16] evaluates a proposed action *before* it is executed on the mobile app. This approach serves as an effective guardrail by providing an opportunity to abort or correct erroneous actions before they occur. However, it frequently suffers from low verification accuracy because it requires predicting the outcome of an action. Such inaccuracies lead to high false negatives and positives, paradoxically decreasing the overall system accuracy and undermining the intended safety benefits.

Conversely, *post-action reflection* [8] evaluates an action *after* its execution, leveraging the resulting application state to concretely assess the correctness of the action. While this approach significantly improves verification accuracy, it has a critical limitation: it cannot prevent irreversible actions. For irreversible actions such as financial transactions or sending messages, post-action verification becomes ineffective, as such actions cannot be undone once they are performed.

VSA is designed to address these limitations, providing a reliable *pre-action* verification that is based on a logic-based deduction instead of probabilistic reasoning.

Software Verification for Safety-Critical Systems. Software verification [27–36] encompasses a range of methodologies to ensure that software systems behave correctly according to specification. These formal verification techniques have been widely adopted to provide strong correctness guarantees in safety-critical domains, including hardware design [31], automotive systems [32], and cyber-physical systems [33, 34]. Among them, dynamic verification [35] aims to check the properties of the program or detect incorrect behaviors when the program is executed. The formal definition of malfunctions is written in mathematical expressions such as first-order logic, linear temporal logic, and finite-state automaton. Once the verification properties are defined, verification can be performed by checking whether an automaton reaches an accept state or by converting the problem into a satisfiability modulo theory (SMT) problem to find satisfying variable assignments using SMT solvers [37]. In particular, runtime verification is a type of dynamic verification that monitors the execution of a program in runtime [36]. It detects anomalous behavior through systematic observation of program execution.

3 VeriSafe Agent (VSA): Overview

Inspired by the success of formal verification in other domains, this work adapts techniques from software verification to solve the specific challenges of Mobile GUI Agent safety and reliability.

As illustrated in Figure 1, VSA is layered on top of an existing GUI agent, acting as a verification layer *before* the agent’s proposed UI actions are injected into a mobile app. Given a user instruction, VSA translates it into a logical formula representing the conditions for successful task completion.

Then, when the GUI agent generates a UI action, VSA verifies whether this action satisfies the pre-defined logical formula. If the action is verified as correct, it is passed on to the mobile application. If the action fails verification, VSA provides feedback to the GUI agent, explaining the reason for the failure and guiding the agent to generate a corrected action.

3.1 Challenges

In doing so, VSA addresses three key challenges:

- C1. How to *formally* express the user’s intent and the app’s execution flow in logically tractable representation?
- C2. How to *accurately* translate natural language user instructions into a formal representation and perform runtime verification against it?
- C3. How to *effectively* communicate verification results back to the GUI agent in an actionable, explainable manner?

C1. Conventional software verification techniques typically rely on precise, pre-defined formal specifications of program behaviors. However, mobile GUI agents operate on natural language instructions, which are inherently informal and ambiguous. Therefore, we need a representation system that can clearly express both the user’s intent and the corresponding logical flow of the application to fulfill that intent. To address this, VSA introduces a domain-specific language (DSL, § 4.1) and an associated developer library (§ 4.2). The DSL provides the syntax and semantics for expressing user intent as logical rules that must be satisfied. The developer library enables app developers to explicitly define the app states and transitions required to construct these rules. Together, they allow VSA to represent both the *desired* behavior (user intent) and the *actual* behavior (app execution) in a unified, logically verifiable manner.

C2. Natural language is often vague, underspecified, and prone to misinterpretation, making it difficult to accurately translate into a formal DSL suitable for logical verification. Even the most capable language models often fail to do this without proper external guidance. To address this challenge, VSA employs a *self-corrective encoding* approach (§ 5.1) that enables the language model to iteratively refine its translation based on syntactic and semantic checks. Furthermore, to achieve efficient yet robust verification, VSA provides a two-tiered verification strategy (§ 5.2), adjusting verification intensity according to the action’s significance.

C3. Simply flagging an action as incorrect is insufficient for improving the overall practicality of Mobile GUI Agents. To enhance their accuracy and reliability, we must provide actionable feedback that correctly guides agents toward the

goal. However, LLM-generated feedback is often vague, underspecified, or even incorrect, potentially leading to repeated errors or suboptimal actions. Therefore, VSA incorporates a structured feedback generation mechanism (Figure 6) that systematically generates precise, rule-based feedback by identifying unmet or violated conditions based on the logical verification results.

3.2 System Workflow

This section describes the high-level workflow of how VSA logically verifies GUI agent actions.

App Development Phase. To enable rigorous rule-based verification, VSA requires app developers to define the critical states and transitions relevant for verification using VSA developer library (§ 4.2). This includes declaring application state spaces and their update mechanisms, which are already common practice in modern app development workflows [38–40] and requires minimal additional effort, typically involving 5 to 10 lines of code per state.

Intent Encoding. Given a natural language user instruction, the *Intent Encoder* first translates it into structured logical rules capturing essential constraints and logical dependencies within the user instruction. For example, an instruction “Book tickets to the movie *M* at 7 pm” can be conceptually translated into the following logical rule:

$$\text{MovieInfo}(\text{title}=\text{“M”}, \text{time}=7\text{pm}) \rightarrow \text{Book}$$

where each constraint (e.g., title=“M”, time=7pm) is a condition must be satisfied before booking the ticket.

Verification. When the GUI agent generates a UI action, the VSA developer library (§ 4.2), embedded within the mobile application, intercepts the action and returns the *anticipated* state transition that would result from the execution of this action. The *Intent Verifier* (§ 5.2) then evaluates whether this state transition is valid according to the logical rules generated earlier. For example, if the state transition indicates that the name of the restaurant is ‘S’ instead of ‘R’, the `IsNameR` Boolean variable would be evaluated to `False`, indicating that the action is invalid and we can’t proceed to `Reserve`.

Feedback Generation. Based on the verification result, VSA provides structured feedback to the GUI agent. If verification succeeds, the action is passed on to the mobile app for execution, and *Feedback Generator* (Figure 6) guides subsequent actions. If verification fails, the action is discarded, and *Feedback Generator* explains precisely which predicates or constraints were unmet, guiding the agent toward generating a correct action.

This workflow continues until either the task reaches a final state or the GUI agent explicitly terminates the task despite feedback, resulting in task failure. Through this systematic approach, VSA ensures that Mobile GUI Agents adhere

$$\begin{aligned}
\textit{Specification} &::= \textit{Rule}^* \\
\textit{Rule} &::= (\textit{Pred} \wedge \dots \wedge \textit{Pred}) \rightarrow \textit{Objective} \\
&\quad | (\textit{Pred} \wedge \dots \wedge \textit{Pred}) \rightarrow \textit{Done} \\
\textit{Pred} &::= \textit{Objective} | \textit{StatePred}(\textit{Constraint}^*) \\
\textit{Constraint} &::= \textit{Variable} \textit{Operator} \textit{Constant} \\
\textit{Constant} &::= \textit{String} | \textit{Number} | \textit{Boolean} \\
&\quad | \textit{Date} | \textit{Time} | \textit{Enumeration} \\
\textit{Operator} &::= = | \neq | \approx | > | \geq | < | \leq | \subseteq | \not\subseteq
\end{aligned}$$

Figure 2: Simplified syntax of our DSL

to the logical constraints defined by user instructions while interacting with mobile applications.

4 Domain-Specific Language

This section details the domain-specific language (DSL) and the associated developer library. DSL and developer library are the basis for formally representing user instructions and GUI agent actions. Throughout the remainder of this paper, we illustrate VSA using the running example: “Reserve restaurant R before 7 PM. If the restaurant is not available at that time, do nothing.”

4.1 Design of Domain-Specific Language

We first clarify the formal syntax and semantics of our DSL, which enables the formal representation of the user intent and desired app execution flow. Our DSL is designed to flexibly express a wide range of constraints, execution flows, and conditional branches found in natural language user instructions.

4.1.1 Syntax and Semantics. Inspired by logic programming languages (e.g., Datalog [41] and Prolog [42]), our DSL is structured around Horn clauses. A Horn clause refers to a logical formula with the specific structure $p_1 \wedge p_2 \wedge \dots \wedge p_n \rightarrow o$, where p_1, p_2, \dots, p_n, o are predicates evaluating to boolean values. Horn clauses are optimized for representing objective attainment conditions and execution flow, commonly used in program verification [43–46]. Each Horn clause signifies that the preconditions for o to be satisfied are p_1, p_2, \dots, p_n . The formal syntax of our DSL is summarized in Figure 2.

Specification and Rule. In our DSL, a single user instruction is expressed as a set of Horn clauses called *specification*. Each Horn clause is called a *rule*, which represents a list of predicates (preconditions) and the objective to be achieved (conclusion). Rules break down a user instruction into multiple smaller steps, making it easier to verify complex instructions incrementally. Each rule consists of predicates

$$\begin{aligned}
R_1 &:\text{RestaurantInfo}(\text{name} = \text{“R”}) \\
&\quad \wedge \text{ReserveInfo}(\text{date} = \text{Today}, \text{time} < 19:00, \text{available} = \text{True}) \\
&\quad \rightarrow \text{Reserve} \\
R_2 &:\text{Reserve} \wedge \text{ReserveResult}(\text{success} = \text{True}) \rightarrow \text{Done} \\
R_3 &:\text{RestaurantInfo}(\text{name} = \text{“R”}) \\
&\quad \wedge \text{ReserveInfo}(\text{date} = \text{Today}, \text{time} < 19:00, \text{available} \neq \text{True}) \\
&\quad \rightarrow \text{Done}
\end{aligned}$$

Figure 3: Specification for the instruction “Reserve restaurant R before 7 PM. If the restaurant is not available at that time, do nothing.”

and an objective which represent the precondition and intermediate step, respectively. Predicates are composed of state predicates and objectives of the other rules. State predicates specify the conditions that must hold in the application for the intermediate step to be achieved. When the objective of another rule is used as a predicate, it establishes a precedence relationship between rules. A rule is satisfied if all predicates hold. If a rule ends with Done and such a rule is satisfied, the task is considered complete.

State Predicate. State predicates represent abstract states of applications. They ignore overly specific information (e.g., size of buttons) and represent only the information important for verification. A state predicate p takes the form $p(c_1, \dots, c_n)$, where each c_i is a constraint. Each constraint allows us to express a target state precisely. For example, a state predicate of RestaurantInfoResult can have constraints on restaurant_name, location, and cuisine_type.

Constraint. A constraint c has the form $\textit{var} \textit{op} \textit{const}$ (e.g., $x = 10$), where x is a variable representing a specific element within the application state (e.g., restaurant_name, location). op is a comparison operator, and $const$ represents a constant value. The expression $x; op; const$ returns true if the variable x meets the condition defined by the operator op and the constant $const$. If x has not been observed in the app, it defaults to *undefined*, making the constraint always false. The variable x can be one of the following six types: string, number, Boolean, enumeration, date, and time. For example, given a constraint $x \geq 10$, if the value of the variable x was observed to be 15 in the application, then the constraint is considered true.

4.1.2 Running Example. Consider our running example of reserving a restaurant. The instruction is translated into the specification shown in Figure 3. RestaurantInfo is a predicate that represents conditions on information about the restaurant you want to reserve, with the constraint that the name must be “R”. ReserveInfo is information about the reservation, with the selected date, time, and availability as constraints. The first rule, R_1 , means to make a reservation

if the restaurant name is “R” and the reservation is available before 19:00 today. The second rule, R_2 , indicates task completion if the reservation is confirmed to be successful. The last rule, R_3 , represents another task completion if the reservation is not available before 19:00 today.

Notice that the objective Reserve of rule R_1 appears as a predicate in rule R_2 . This signifies that R_1 must be satisfied before proceeding to R_2 .

Specification is dynamically evaluated during the agent’s interaction with the application. Initially, all variables x are considered *undefined*. The values of these variables are updated at runtime by the state transitions caused by the GUI agent’s actions. These updates are governed by pre-defined state update methods written in the developer library (see § 4.2). Whenever a variable’s value changes, the associated constraint is re-evaluated.

For example, consider the ReserveInfo predicate. If the GUI agent attempts to reserve restaurant “R” at 18:00 today, the update rule will set the values of the date and time variables to Today and 18:00, respectively. When we evaluate the constraints against the updated variables, $\text{date}=\text{Today}$ and $\text{time}<19:00$ will both be true. If availability is observed to be true in the app state after selection, all constraints of the ReserveInfo predicate are true, and the predicate itself becomes true. This runtime evaluation of the specification allows VSA to continuously monitor the agent’s progress and verify whether its actions are leading the application toward a state that satisfies the user’s intent.

4.2 Developer Library

To effectively generate *specification* from a user instruction and evaluate it against agent’s actions, VSA requires a definition of two key components: (i) the set of candidate state predicates (i.e., the application’s state space) that serve as building blocks for the specification, and (ii) the mechanisms through which each predicate’s constraint variables x are updated during app execution (i.e., state transitions).

To this end, VSA provides a *developer library* that empowers app developers to explicitly define application states (as state predicates) and their corresponding update mechanisms (as state transitions). This approach capitalizes on developers’ deep understanding of their application’s internal logic and intended behavior. Although it requires additional developer effort, it enables highly accurate state definitions suitable for rule-based logical verification—a level of precision that is unattainable with automated approaches like static code analysis or LLM-driven dynamic analysis.

Furthermore, this approach aligns with established practices in modern mobile app development. Frameworks like React Native (state management [38]), Android’s Jetpack Compose (state hoisting [39]), and iOS’s SwiftUI (@State [40])

all encourage developers to manage app state and transitions. By closely mirroring these familiar paradigms, VSA ensures seamless, intuitive, and minimally invasive integration into standard app development workflows.

4.2.1 States Definition. A *state*, in the context of the VSA Library, is characterized by a set of typed variables (as defined in § 4.1.1) that capture essential details relevant to the application’s current context—each variable corresponds to the constraint of the state predicate. Each state definition includes i) *name*, ii) *description*, and iii) *variables*:

Developers can define these states using an external JSON file or provided APIs (i.e., `defineState()`):

```
{
  "name": "RestaurantInfo",
  "description": "Information about the
    restaurant you want to reserve.",
  "variables": [{"name": "String"}]
}
```

These defined states are then used as candidate state predicates when VSA’s Intent Encoder (§ 5.1) translates user instruction into a specification.

4.2.2 Pre-action State Update Triggers. VSA enables developers to define *triggers* specifying when and how state variables are updated. Triggers correspond directly to meaningful state transitions within the mobile application. Most triggers are associated with UI interaction handlers (e.g., `onClickListener`) or critical points in the application logic.

Triggers return the *expected* state update before executing the action. The returned update is used to verify the action. This ahead-of-time update provides two key benefits: i) It prevents irreversible actions (e.g., financial transaction, sending a message) from being executed if they violate the user’s intent; ii) It allows the GUI agent to easily retry with a different action without having to revert the effects of a previous, incorrect action (e.g., deleting incorrect text input).

To facilitate pre-action updates, the VSA developer library provides wrapper listeners for common UI input handlers. These wrappers intercept user interactions and update the relevant state variables according to the developer-defined logic *before* executing the original event handler’s code. The original ‘onClick’ code which performs the actual operation is executed only if the verification succeeds. For example, consider the ‘RestaurantInfo’ state predicate defined earlier. A developer could associate a trigger with the search button’s click listener as follows:

```
searchButton.VSAOnClickListener() -> {
  safeMATE.updateState("RestaurantInfo", {
    "name": searchTextField.getText(),
  });
  /* existing code for onClickListener */
});
```

Another key advantage of developer-defined state predicates is the control over *granularity*. In many mobile tasks, individual atomic actions are not inherently meaningful. For example, in an e-commerce application, entering individual fields of a shipping address (street, city, zip code) is a set of preparatory steps. The critical state change is the *submission* of the complete address. Verifying every single atomic action would be inefficient and could lead to false positives when verification occurs before completing all preparatory steps. By allowing developers to strategically define verification checkpoints that have a meaningful impact on the task’s execution, VSA can significantly improve the efficiency and accuracy of task verification.

5 Verification Engine

5.1 Intent Encoder

Natural language instructions cannot be directly used for formal verification; a formalization process is required. We optimize the LFM-based autoformalization method in the context of mobile agents to automatically convert instructions into specifications written in our DSL. Relying solely on LFM’s output poses risks due to their probabilistic nature and potential for hallucinations. To address this limitation, we applied *self-corrective encoding*, which corrects errors automatically, and *experience-driven encoding* techniques that gradually adapt to each app.

5.1.1 Self-Corrective Encoding. To ensure the encoded specification is correct and includes all relevant constraints, VSA employs two complementary self-corrective techniques: (i) rule-based syntax checking, and (ii) decoding-based semantics checking. These methods allow the LLM to incrementally identify, correct, and refine its translations, improving accuracy and reliability.

As an initial step, the *Intent Encoder* prompts the LLM with: i) the user’s natural language instruction and ii) the set of developer-defined states (from the developer library). From this information and user instruction, the Intent Encoder generates a draft specification representing the user’s intent using available states. The draft specification then undergoes two checking stages:

Rule-based Syntax Checking. Inspired by conventional programming languages, VSA checks that the generated Specification strictly adheres to the DSL’s syntax (§ 4.1.1) and that the types of constants c used within the Specification are consistent with the types of variables declared in the developer-defined states. For example, if the LLM generates a state predicate like `RestaurantInfo(name ≥ 100)` while the type of the name variable is defined as ‘String’ in the developer library, this inconsistency is immediately flagged as a syntax error. If an error is detected, a structured

error message is returned to the encoding LLM as feedback, guiding it toward a corrected translation.

Decoding-Based Semantics Checking. To further validate the semantic correctness of the translation, the generated Specification is *decoded* back into a natural language description using a separate *decoder LLM*. A *checker LLM* then compares this decoded description to the original user instruction, confirming that the encoding genuinely captures the user’s intent rather than merely manipulating symbols. If discrepancies are found, the checker LLM identifies the root cause and provides feedback to the *encoding LLM*.

These complementary checking mechanisms significantly improve the robustness and accuracy of the intent encoding process, effectively mitigating risks associated with solely relying on probabilistic LLM outputs.

5.1.2 Experience-driven Encoding. Although *Self-corrective Encoding* significantly enhances translation accuracy, it occasionally produces hallucinations when presented with an enormous number of candidate state predicates.

To address this challenge, VSA introduces an *Experience-driven Encoding*, which leverages previously successful encodings to guide subsequent translations. When an instruction is successfully translated and verified, its result specification is cached. Since a single specification consists of multiple rules representing an instruction’s intermediate objectives, numerous instructions for the same application may share similar rules. By retrieving and prioritizing predicates based on past encoding experience, we can significantly reduce the search space for the LLM encoder, improving both efficiency and consistency.

5.2 Intent Verifier

The verification process is straightforward. When the GUI agent generates an action, the associated state updates (defined via the developer library, § 4.2) are triggered. They update the values of the variables x of the corresponding constraints and state predicates in the Specification. After each update, the affected predicates are re-evaluated to a new truth value.

Based on this basic mechanism, VSA performs two levels of verification: *Predicate-level Verification* at the predicate level and *Rule-level Verification* at the rule level.

Predicate-level Verification is performed at *every* state update. Whenever a state predicate within the Specification is re-evaluated due to a variable update, its truth value is checked. If a predicate evaluates to false, this indicates a *potential* violation of the user’s intent. For example, if the Specification includes the predicate `RestaurantInfo(name = “R”)`, and the agent’s action causes the name variable to be updated to “S”, the predicate would evaluate to false, indicating a *potential* violation of user intent.

F_1 : "To perform Reserve, 1. RestaurantInfo that represents Information about the restaurant you want to reserve should have 'name' equal to "R"; 2. ReserveInfo that represents reservation details should have 'date' equal to "Today" and 'time' less than 19:00, and 'available' equal to True. So far, you have achieved step 1."

F_2 : "To complete the task, 1. perform Reserve 2. ReserveResult that represents reservation result should have 'success' equal to "True"."

F_3 : "To complete the task, 1. RestaurantInfo that represents Information about the restaurant you want to reserve should have 'name' equal to "R"; 2. ReserveInfo that represents reservation details should have 'date' equal to "Today" and 'time' less than 19:00, but 'available' not equal to True. So far, you have achieved step 1."

Figure 4: An example of roadmap feedback. Each F_i corresponds to Rule R_i in the Specification

It is important to note that an incorrect predicate at this stage does not necessarily mean a definitive error because the updated state may represent an intermediate step necessary to eventually reach the final, correct state. For example, when buying three apples, the number of apples might be incremented one at a time. Each intermediate update (1 apple, then 2 apples) would result in a false value for a predicate requiring three apples, until the final update.

Therefore, upon detecting a failed verification, VSA reverts the predicate update and provides the GUI agent with *soft feedback* (Figure 6). This feedback serves as a warning, indicating a potential deviation from the intended path, and encourages the agent to double-check its subsequent actions.

Rule-level Verification is performed at irreversible or critical checkpoints within the task execution. To identify these checkpoints, we modified the GUI agent's prompt so that it generates a *critical* flag along with the action if that action is intended to directly achieve one of the objectives o within the Specification (e.g., Reserve)

When the GUI agent generates a *critical* action, VSA checks for the complete satisfaction of the conditions required to achieve a specific Objective o . If any predicate in the corresponding rule is unsatisfied, the action is considered invalid and blocked. Hard feedback is provided to the agent, detailing the specific unmet conditions. This prevents the execution of potentially irreversible or incorrect actions that would definitively violate the user's intent.

These two-tiered verification approach guides the agent toward its objectives while ensuring the safe execution of high-risk actions.

6 Structured Feedback Generator

The effectiveness of a verification system hinges not only on error detection but also on its ability to guide the agent towards correct behavior. This section explains how VSA generates structured feedback based on verification results.

VSA provides three distinct types of feedback: i) Roadmap Feedback, ii) Predicate-level Soft Feedback, and iii) Rule-level Hard Feedback.

Roadmap Feedback. VSA provides roadmap feedback that outlines the overall path toward task completion regardless of verification outcome. Because the logical Specification of VSA explicitly encodes the preconditions required for achieving the user's objective, it naturally serves as a comprehensive guideline for the GUI agent.

Specifically, the Feedback Generator converts each rule in the Specification into a natural language explanation, detailing which predicates must be satisfied for each objective. These explanations incorporate developer-defined descriptions for each state predicate, constraints, and currently satisfied predicates. For example, Figure 4 demonstrates the roadmap feedback for the restaurant reservation scenario after successfully searching for restaurant "R".

Unlike approaches relying on ambiguous action histories or abstract planning, this structured guidance clearly communicates both current progress and next objectives.

Predicate-level Soft Feedback. When predicate-level verification fails (i.e., a state predicate evaluates to false), VSA provides feedback indicating that the agent's action *may* be incorrect. This feedback includes a description of the desired state for the violated predicate, allowing the agent to reconsider its action. Because predicate-level errors could represent intermediate steps rather than genuine errors (as described in § 5.2), soft feedback is advisory rather than strictly prohibitive. If the GUI agent generates the same action even after receiving the feedback, VSA permits the action, acknowledging its non-critical nature.

Rule-level Hard Feedback. For critical actions subject to rule-level verification, VSA provides comprehensive feedback at the rule level, detailing all state predicates that remain unfulfilled. Unlike soft feedback, if the GUI agent generates the same action again after receiving hard feedback, VSA will not allow the action to proceed. Instead, it enforces that all required state predicates must be satisfied before permitting the critical action to be executed.

Through these structured, deterministic feedback mechanism, VSA effectively guide GUI agents toward correct task completion, significantly outperforming existing reflection-based methods. By eliminating ambiguity and providing clear, actionable guidance grounded in formal logic, VSA enhances the reliability, safety, and practicality of automated mobile task execution.

7 Implementation

Our VSA implementation is designed as a modular component that can be seamlessly integrated with existing GUI agents without requiring modifications to their underlying

architecture. VSA is implemented in using Python, and its Intent Encoder leverages off-the-shelf LFM GPT-4o.

Handling unpredictable state updates. While VSA focuses on pre-action verification, it also supports *post-action* updates where the outcome of an action is unpredictable or depends on external factors (e.g., the result of a network request). In these cases, developers can directly use the ‘updateState()’ API without a wrapper listener to update the state after the desired information is known. This allows for verification even in scenarios where pre-action prediction is impossible.

Developer Library Guidelines. Developers should carefully determine the granularity of State Predicates to suit their application context. Overly fine-grained State Predicates can make the encoded specification too detailed, increasing the false positive rate. Conversely, coarse-grained State Predicates may introduce loopholes in the encoded specification, leading to a higher false negative rate. Developers must find a sweet spot in this trade-off based on their application’s usage patterns. To assist with this, the VSA Library provides detailed documentation and guidelines on effectively defining and utilizing State Predicates to achieve robust verification. Given the potential severity of false negatives, it’s recommended to begin with a fine-grained verification first and then iteratively refine the predicates to minimize false positives.

String Similarity. From the perspective of structural equivalence, strings with different notations or synonymous words are treated as distinct entities (e.g., Jan 01 and January 1st). However, user instructions may not exactly match the values in the application, making it necessary to check for semantic equivalence rather than structural equivalence. To achieve this, we used OpenAI’s text-embedding-3-small model [1] to generate semantic embeddings and compared them using cosine similarity. Based on empirical studies, we determined an equivalence threshold of 0.7.

8 Evaluation

In this section, we demonstrate the performance of VSA in three key aspects: *i) verification accuracy*, *ii) feedback effect*, and *cost&latency*. To this end, we deployed *M3A* [12], a simple yet powerful mobile GUI agent powered by GPT-4o, on a Google Pixel 8 smartphone and integrated it with VSA. This setup allows VSA to validate the agent’s behavior in runtime while enabling autonomous app execution.

Baselines & VSA Variants. To evaluate VSA, we compare it with two LFM-based reflection schemes: Pre-action reflection [16] and Post-action reflection [8], both of which utilize GPT-4o as their base LFM. For VSA, we evaluate two variants: VSA-cold and VSA-warm. VSA-cold encodes user instructions solely using self-corrective encoding, without

experience-driven encoding. On the other hand, VSA-warm leverages both self-corrective encoding and experience-driven encoding, assuming that a cached list of candidate predicates is available for the given instruction (see § 5.1 for details).

8.1 Dataset

The primary contribution of VSA is its ability to proactively verify and provide feedback to a GUI agent when automating mobile tasks. To comprehensively evaluate VSA’s effectiveness, we constructed a dataset consisting of 300 user instructions. This dataset is divided into two complementary subsets: *Correct* and *Wrong* datasets, each containing 150 instructions.

The Correct dataset represents scenarios where the GUI agent’s execution path precisely matches the user’s instruction. This set contains instructions paired with their *ground-truth* execution paths (sequences of UI actions that correctly fulfill the instruction). It is designed to evaluate True Negatives (correctly identifying a valid execution) and False Positives (incorrectly flagging a valid execution as erroneous). Within this dataset, we included 125 instructions from the widely adopted LlamaTouch dataset [15], covering 18 mobile applications. However, since the LlamaTouch instructions are predominantly simple (average 5.67 steps), similar to other datasets [4, 12, 14, 47], we augmented our dataset with 25 additional complex instructions, referred to as the *Challenge dataset*. These instructions feature instructions with significantly greater complexity, including longer action sequences, multiple constraints, and conditional branches. These challenging instructions span 9 different applications, with an average of 19.16 steps and a maximum length of 40 steps. An example of such a challenging instruction in Google Maps is: “*I am planning to visit the Cattedrale di Santa Maria del Fiore soon. First, find the Cattedrale di Santa Maria del Fiore on the map and then look for nearby attractions. Next, identify one attraction with a rating of 4 stars or higher and at least 2000 reviews, then show me the wheelchair-accessible walking route from the Cattedrale di Santa Maria del Fiore to that attraction.*”

To thoroughly evaluate error-detection performance, we constructed a complementary Wrong dataset by deliberately introducing mismatches between instructions and their corresponding execution paths from the Correct dataset. Specifically, we altered key details in the original instructions to create discrepancies. For instance, we modified the instruction: “*On Play Books, turn to the search page and check for top-selling books.*” to: “*On Play Books, turn to the search page and check for newly released books.*” Because the execution path no longer aligns with the modified instruction, these scenarios effectively test the verification system’s ability to detect errors— True Positive (correctly identified errors) and False Negative (missed errors).

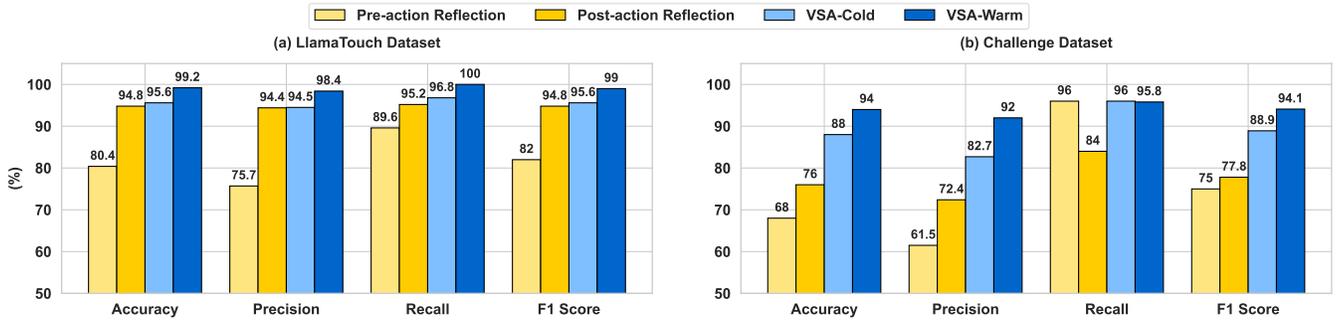


Figure 5: Verification accuracy

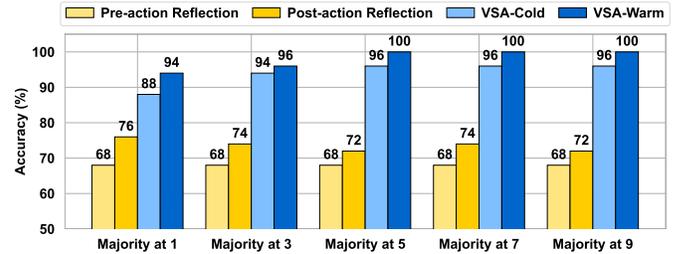
Finally, as described in § 4.2, VSA relies on developer-defined state definitions and state-update triggers. However, since the LlamaTouch dataset uses commercial applications without available source code, we manually annotated each application’s GUI screens with the required predicate information and state transitions using a GUI annotation tool. This manual annotation achieves the same effect as direct integration via the developer library, effectively replicating the developer library’s intended functionality.

8.2 Verification Accuracy

Figure 5 presents the verification accuracy of different verification methods, evaluated using standard metrics: Accuracy, Precision, Recall, and F1 Score. Notably, VSA consistently outperforms reflection-based methods across all metrics and datasets, achieving near-perfect accuracy on the LlamaTouch dataset. One interesting observation is the high recall performance of Pre-action Reflection on the Challenge Dataset. However, this comes at the cost of a high false positive rate (60%), leading to excessive error flagging when instructions become complex. As a result, while it has a higher likelihood of detecting *any* existing error (high Recall), the probability that a flagged error is *actually* an error remains low (low Precision).

Analysis. For the relatively simpler LlamaTouch dataset, Post-action Reflection achieves performance comparable to VSA. This is because it leverages the action’s resulting screen state as additional context for verification, boosting accuracy compared to Pre-action Reflection. However, it has critical limitations: it cannot correct irreversible actions. Moreover, our results indicate that this advantage diminishes significantly, as instructions become more complex and involve longer action sequences (as seen in the Challenge Dataset). This trend highlights the compounding effect of errors stemming from over-reliance on LFM for verification.

In contrast, VSA enables *pre-action* verification while maintaining high performance across both datasets. This demonstrates the effectiveness and scalability of VSA’s design,

Figure 6: Verification accuracy with *Majority-at-N*

which encodes the specification once and performs subsequent verification in a rule-based manner, regardless of task complexity or length. Notably, VSA-Warm achieves nearly perfect results on the LlamaTouch dataset and significantly surpassing all other methods on the Challenging dataset. This demonstrates the potential of VSA’s design when equipped with a well-constructed memory of candidate state predicates. While the construction of such a memory (e.g., offline exploration, user demonstrations, runtime analysis) is beyond the scope of this paper (see § 10), the results clearly underscore the promise of VSA.

Randomness Mitigation. Although VSA employs only one LFM query per task, when repeatedly verifies the same instruction, inconsistent results may still arise due to the probabilistic nature of LFM. The same inconsistency applies to all methods that includes LFM in its process, including reflection methods. To mitigate this randomness, a majority voting approach can be employed, wherein the verification process is repeated N times, and the final outcome is determined by majority voting (referred to as *Majority at N*).

Figure 6 illustrates the each method’s verification accuracy for 50 instructions in the Challenge dataset as N increases. We observed that while reflection-based schemes consistently shows lower accuracy, both VSA-Cold and VSA-Warm show significant improvements, reaching 96% and 100%, respectively. This disparity stems from differences in their verification mechanisms: reflection schemes produce direct true/false responses with lower randomness, whereas both VSA-Cold and VSA-Warm generate logical formulas with

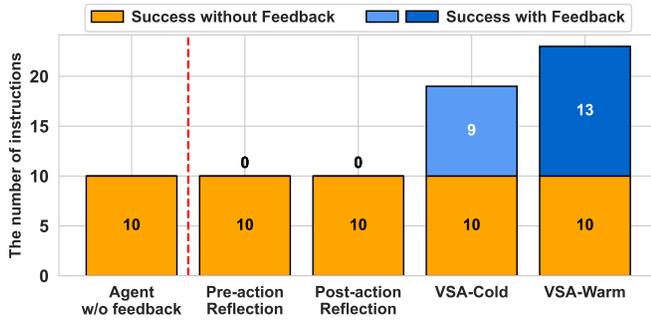


Figure 7: Effect of feedback on task completion

multiple predicates through the LFM, making them more susceptible to stochastic variations. These findings suggest that VSA has significant room for performance enhancement through majority voting for verification. However, it is worth noting that excessively increasing the number of verification iterations can introduce considerable overhead in terms of latency and cost. A detailed analysis of this overhead issue is provided in § 8.4.

8.3 Effectiveness of Feedback Generation

The usefulness of a verification system lies not just in identifying errors, but in improving the GUI agent’s performance through feedback generation. This section compares the effectiveness of VSA’s logic-based feedback generation with that of the LFM-based feedback used in reflection methods.

To evaluate this, we instructed the M3A GUI agent to execute 25 challenging instructions from our Challenge dataset. Initially, without external feedback (Agent w/o feedback), M3A successfully completed only 10 out of 25 tasks. Next, we integrated each verification system to M3A and re-executed each instruction, observing whether previously failed tasks could now be correctly performed. Note that we evaluated using our own dataset over LlamaTouch dataset because LlamaTouch’s ground truth fails to account the multiple paths often present in real-world mobile tasks. On the other hand, our Challenge dataset includes hand-crafted ground truth encompassing multiple valid execution paths, providing more accurate evaluation than LlamaTouch’s single-path approach which showed poor correlation with human evaluation (<60% fidelity).

Figure 7 summarizes these results. Notably, neither pre-action nor post-action reflection successfully corrected any previously failed tasks. In contrast, VSA-Cold feedback corrected 9 out of 15 tasks (60%), and VSA-Warm successfully corrected 13 tasks (86.7%). It is worth noting that VSA-Warm’s two failures stemmed from unconventional application designs, not from the inability to understand the execution context. For instance, a clock application required time input as a single numerical sequence (hour, minute, second combined)

rather than separate components. This unconventional interaction hinders VSA’s ability to provide meaningful feedback. Nevertheless, given that M3A is a state-of-the-art GUI agents powered by one of the most powerful LFM (gpt-4o), achieving this significant enhancement underscores VSA’s effectiveness.

This superior performance stems from VSA’s ability to generate precise, deterministic feedback, clearly indicating what aspects of the agent’s execution need to be corrected. Furthermore, VSA’s *roadmap feedback* proactively guides the GUI agent toward the correct execution path, preventing errors from occurring at the first place. For example, with the instruction: “Go to the *r/technology* subreddit. In the most controversial post in your field over the past month, go to the comments on that post. Find the most controversial comment in the comments, and upvote that comment,” the M3A agent w/o feedback repeatedly toggled the upvote action due to unclear state tracking, not knowing that it has already upvoted. Yet, when given a clear guideline saying that it has fulfilled the precondition to set ‘upvote’ to ‘true’, M3A to successfully complete the task without unnecessary repetition.

In contrast, reflection methods, which rely on an LFM for feedback generation, frequently produce misleading guidance and fails to correct any of the previously failed instructions. Furthermore, we observed cases where reflection initially flagged an action as incorrect but reversed its decision after the GUI agent repeated the same action, illustrating inherent indecisiveness and unreliability in its feedback generation.

One potential limitation of VSA, though not observed in our evaluation, is that false positive errors could prematurely terminate tasks that the GUI agent might have otherwise completed successfully. Unlike reflection-based methods, where an agent can typically recover quickly—within 1-3 actions—after a misleading feedback, a single incorrect predicate in the Specification could cause VSA to halt task execution entirely. Nevertheless, our evaluation clearly demonstrates that the practical benefits of VSA’s decisive feedback significantly outweigh the potential risks of rare false positives. Particularly in safety-critical systems, such as automated mobile tasks, strictly preventing erroneous actions—even at the expense of occasional false positives—is significantly more beneficial than allowing potentially harmful actions to proceed, as traditional reflection methods currently do.

8.4 Latency and Cost

VSA not only enhances verification reliability but also significantly reduces latency and cost overhead. Figure 8 plots the additional latency and cost incurred by verification as a function of task length (number of steps). As shown by the

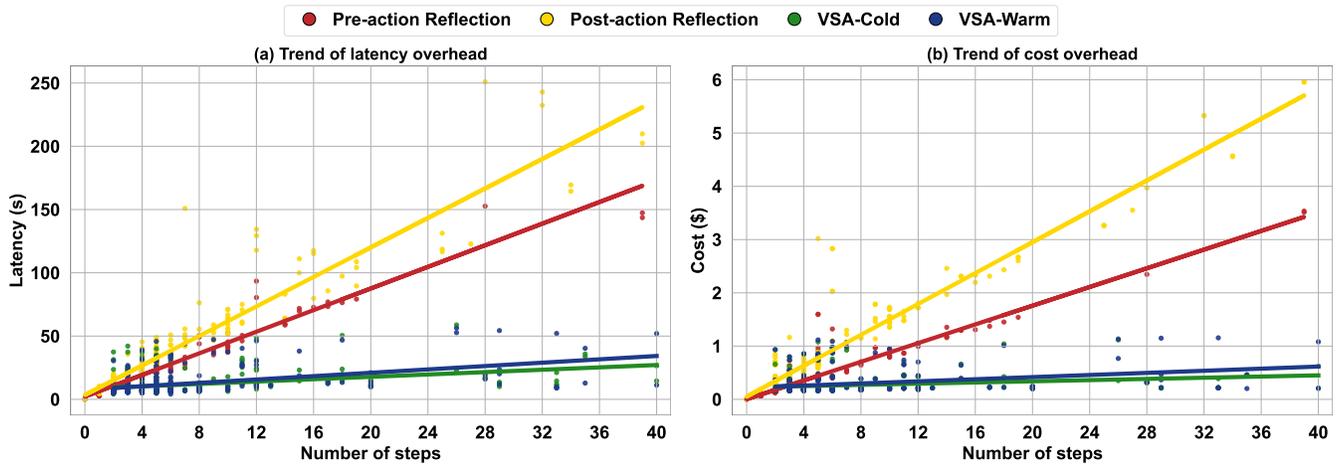


Figure 8: Latency and cost overhead of verification methods with respect to the number of action steps.

fitted lines, VSA consistently maintains lower overhead than reflection-based methods across all task lengths. Notably, because reflection-based approach requires an additional LFM query for each action, its overhead scales proportionally with task length. While not depicted in the graph for clarity, its overhead is comparable to that of the GUI agent itself, effectively doubling the total expense.

In contrast, VSA shows near constant overhead, independent of task length. This is because VSA uses LFM only once – when encoding user instructions into formal specifications. All subsequent action verifications are performed using rule-based logic. Although Figure 8 shows a slight upward trend in VSA’s overhead for longer tasks, this increase is attributed to more complex instructions naturally producing longer specifications with additional constraints. However, this increase remains trivial compared to the $O(N)$ complexity of reflection-based methods.

GUI Agents inherently incur substantial latency and cost due to their reliance on complex cognitive reasoning. As shown in Figure 8, tasks exceeding 10 steps can cost over \$1 and take nearly a minute to complete. Therefore, minimizing this overhead is as critical for practical deployment as achieving high accuracy. VSA thus proves to be more suitable and practical for real-world deployment across all dimensions compared to existing approaches.

9 Related Work

Validating LFM responses. The probabilistic nature of LFM often leads to hallucinated or inaccurate content. Numerous methods have been proposed to mitigate these issues, including Chain-of-Verification (CoVe) [48], concept-level validation and rectification (EVER) [49], self-refinement [50], and retrieval-augmented knowledge grounding (Re-KGR) [51].

These approaches typically leverage additional LLM invocations for self-reflections [8, 16] or rely on external validation against ground-truth data or knowledge bases [52–56]. However, in mobile task automation, actions of a mobile GUI agent lack a ground truth and its semantic ambiguity makes verification exceptionally difficult. VSA effectively handles this challenge by designing a domain-specific language (DSL, § 4.1) and developer library (§ 4.2) that effectively capture the underlying semantics of these ambiguous GUI actions.

Leveraging Formal Logic for AI Safety. Recent research has explored the use of formal logic to ensure the safety and reliability of AI agents, particularly in areas like planning and control [18–21, 57, 58]. Notably, SELP [20] ensures safe and efficient robot task planning by integrating constrained decoding and domain-specific fine-tuning. Ziyi et al. [19] proposed a safety module for LLM-based robot agents by converting natural language safety constraints into Linear Temporal Logic. However, while previous works typically focus on agents with a pre-defined safety concerns such as map navigation or indoor task planning [18–21], our system targets agents with a enormous number of environment, i.e., mobile apps. To handle different environments of mobile apps, our optimized autoformalization algorithm effectively translates safety requirements for diverse set of mobile tasks.

10 Discussion & Future Work

Piggybacking on App Development Frameworks. The current VSA implementation relies on developer-defined state and transition definition. However, given that modern mobile development frameworks like React Native, Jetpack Compose, and SwiftUI [38–40] already incorporate built-in state managements, there is significant potential to "piggyback" on these existing mechanisms. Integrating VSA in this

way could streamline the development process and reduce the overhead associated with adopting VSA.

Automated State Definition via LFM. Another approach for simplifying VSA’s integration is automating the state definition process using LFMs. Existing GUI agents [4, 6, 7] often employ LLM-powered screen analysis to automatically annotate GUI elements and infer application state. A similar approach could be adapted to identify and define candidate state predicates for VSA. However, while this automation could reduce developer effort, it introduces inherent risks due to the probabilistic nature of LLMs. Relying on potentially inaccurate LLM-derived state definitions could compromise the system’s reliability. Therefore, while LLM-based automation offers a potential convenience, the current approach of developer-defined states ensures the precision necessary for a robust verification system.

Automated Predicate Memory Construction. As demonstrated in the evaluation, VSA-Warm’s performance significantly benefits from the pre-populated predicate memory. However, the current implementation relies on VSA-Cold to successfully complete tasks before they can be added to the memory, creating a performance bottleneck. One promising approach involves leveraging direct user feedback. By simply asking the user whether a task was completed successfully or not, we can obtain a ground-truth verification result. This user-provided ground truth can then be used to selectively populate the predicate memory. This approach would bypass the VSA-Cold bottleneck and accelerate the creation of a robust and accurate predicate memory, leading to improved overall performance.

11 Conclusion

This paper introduced VeriSafe Agent (VSA), a novel logic-based verification system designed to enhance the safety and reliability of Mobile GUI Agents. Through VSA, we have demonstrated that logic-driven verification can effectively safeguard GUI agents against the inherent uncertainties of LFM-based task automation. As AI-driven tasks becomes increasingly integrated into our daily life, VSA presents a crucial step toward safer use of AIs.

References

- [1] openai. Creating safe agi that benefits all of humanity, 2023. URL <https://openai.com/>.
- [2] anthropic. Talk to claude, 2023. URL <https://claude.ai/>.
- [3] meta. Introducing llama 2, 2023. URL <https://ai.meta.com/llama/>.
- [4] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Empowering llm to use smartphone for intelligent task automation. *arXiv preprint arXiv:2308.15272*, 2023.
- [5] Hao Wen, Shizuo Tian, Borislav Pavlov, Wenjie Du, Yixuan Li, Ge Chang, Shanhui Zhao, Jiacheng Liu, Yunxin Liu, Ya-Qin Zhang, and Yuanchun Li. Autodroid-v2: Boosting slm-based gui agents via code generation, 2024. URL <https://arxiv.org/abs/2412.18116>.
- [6] Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steve Ko, Sangeun Oh, and Insik Shin. Mobilegpt: Augmenting llm with human-like app memory for mobile task automation. New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704895. doi: 10.1145/3636534.3690682. URL <https://doi.org/10.1145/3636534.3690682>.
- [7] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. Appagent: Multimodal agents as smartphone users, 2023. URL <https://arxiv.org/abs/2312.13771>.
- [8] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration, 2024. URL <https://arxiv.org/abs/2406.01014>.
- [9] openai. Introducing operator, 2025. URL <https://openai.com/index/introducing-operator/>.
- [10] anthropic. Computer use (beta), 2025. URL <https://docs.anthropic.com/en/docs/agents-and-tools/computer-use>.
- [11] Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. Gpt-4v(ision) is a generalist web agent, if grounded. In *Proceedings of the 41st International Conference on Machine Learning, ICML’24*. JMLR.org, 2024.
- [12] Christopher Rawles, Sarah Clinckemallie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. Androidworld: A dynamic benchmarking environment for autonomous agents, 2024. URL <https://arxiv.org/abs/2405.14573>.
- [13] Wenyi Hong, Weihang Wang, Qingsong Lv, Jiazhen Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxuan Zhang, Juanzi Li, Bin Xu, Yuxiao Dong, Ming Ding, and Jie Tang. Cogagent: A visual language model for gui agents, 2024. URL <https://arxiv.org/abs/2312.08914>.
- [14] Mingzhe Xing, Rongkai Zhang, Hui Xue, Qi Chen, Fan Yang, and Zhen Xiao. Understanding the weakness of large language model agents within a complex android environment, 2024. URL <https://arxiv.org/abs/2402.06596>.
- [15] Li Zhang, Shihe Wang, Xianqing Jia, Zhihan Zheng, Yunhe Yan, Longxi Gao, Yuanchun Li, and Mengwei Xu. Llamatouch: A faithful and scalable testbed for mobile ui task automation. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2024.
- [16] Yanda Li, Chi Zhang, Wanqi Yang, Bin Fu, Pei Cheng, Xin Chen, Ling Chen, and Yunchao Wei. Appagent v2: Advanced agent for flexible mobile interactions, 2024. URL <https://arxiv.org/abs/2408.11824>.
- [17] Yuhuai Wu, Albert Qiaocho Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 2022.
- [18] Jason Xinyu Liu, Ziyi Yang, Ifrah Idrees, Sam Liang, Benjamin Schornstein, Stefanie Tellex, and Ankit Shah. Grounding complex natural language commands for temporal tasks in unseen environments. In *Conference on Robot Learning, CoRL 2023, 6-9 November 2023, Atlanta, GA, USA*, volume 229 of *Proceedings of Machine Learning Research*, pages 1084–1110. PMLR, 2023.
- [19] Ziyi Yang, Shreyas Sundara Raman, Ankit Shah, and Stefanie Tellex. Plug in the safety chip: Enforcing constraints for llm-driven robot agents. In *IEEE International Conference on Robotics and Automation, ICRA 2024, Yokohama, Japan, May 13-17, 2024*, pages 14435–14442. IEEE, 2024.

- [20] Yi Wu, Zikang Xiong, Yiran Hu, Shreyash S. Iyengar, Nan Jiang, Aniket Bera, Lin Tan, and Suresh Jagannathan. SELP: generating safe and efficient task plans for robot agents with large language models. *CoRR*, abs/2409.19471, 2024.
- [21] Ike Obi, Vishnunandan LN Venkatesh, Weizheng Wang, Ruiqi Wang, Dayoon Suh, Temitope I Amosa, Wonse Jo, and Byung-Cheol Min. Safeplan: Leveraging formal logic and chain-of-thought reasoning for enhanced safety in llm-based robotic task planning. *arXiv preprint arXiv:2503.06892*, 2025.
- [22] William Liu, Liang Liu, Yaxuan Guo, Han Xiao, Weifeng Lin, Yuxiang Chai, Shuai Ren, Xiaoyu Liang, Linghao Li, Wenhao Wang, et al. Llm-powered gui agents in phone automation: Surveying progress and prospects. 2025.
- [23] Chaoyun Zhang, Shilin He, Jiayu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Large language model-brained gui agents: A survey, 2025. URL <https://arxiv.org/abs/2411.18279>.
- [24] Shuai Wang, Weiwen Liu, Jingxuan Chen, Yuqi Zhou, Weinan Gan, Xingshan Zeng, Yuhan Che, Shuai Yu, Xinlong Hao, Kun Shao, Bin Wang, Chuhan Wu, Yasheng Wang, Ruiming Tang, and Jianye Hao. Gui agents with foundation models: A comprehensive survey, 2025. URL <https://arxiv.org/abs/2411.04890>.
- [25] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, Rui Kong, Yile Wang, Hanfei Geng, Jian Luan, Xuefeng Jin, Zilong Ye, Guanqing Xiong, Fan Zhang, Xiang Li, Mengwei Xu, Zhijun Li, Peng Li, Yang Liu, Ya-Qin Zhang, and Yunxin Liu. Personal llm agents: Insights and survey about the capability, efficiency and security, 2024. URL <https://arxiv.org/abs/2401.05459>.
- [26] Junlin Xie, Zhihong Chen, Ruifei Zhang, Xiang Wan, and Guanbin Li. Large multimodal agents: A survey, 2024. URL <https://arxiv.org/abs/2402.15116>.
- [27] B. Berard, P. McKenzie, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Berlin Heidelberg, 2014. ISBN 9783662045596.
- [28] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 026202649X.
- [29] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd Edition*. MIT Press, 2018.
- [30] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. *CoRR*, abs/cs/0701193, 2007.
- [31] Thomas Kropf. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems*. Springer-Verlag, 1st edition, 1999. ISBN 3540654453.
- [32] Nijat Rajabli, Francesco Flammini, Roberto Nardone, and Valeria Vitorini. Software verification and validation of safe autonomous cars: A systematic literature review. *IEEE Access*, 9:4797–4819, 2020.
- [33] Rose Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. Veriphy: verified controller executables from verified cyber-physical system models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 617–630, 2018.
- [34] Stefan Mitsch and André Platzer. Modelplex: Verified runtime validation of verified cyber-physical system models. *Formal Methods in System Design*, 2016.
- [35] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.
- [36] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. *Lectures on Runtime Verification: Introductory and Advanced Topics*, pages 1–33, 2018.
- [37] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.
- [38] Meta. State, 2025. URL <https://reactnative.dev/docs/state>.
- [39] Google. State and jetpack compose, 2025. URL <https://developer.android.com/develop/ui/compose/state>.
- [40] Apple. State, 2025. URL <https://developer.apple.com/documentation/swiftui/state>.
- [41] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1995. ISBN 0201537710.
- [42] Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0262193388.
- [43] Temesghen Kahsay, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. Jayhorn: A framework for verifying java programs. In *Computer Aided Verification*. Springer International Publishing, 2016.
- [44] Arie Gurfinkel, Temesghen Kahsay, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In *Computer Aided Verification*, pages 343–361. Springer International Publishing, 2015.
- [45] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. Rusthorn: Chc-based verification for rust programs. *ACM Trans. Program. Lang. Syst.*, 43(4), 2021.
- [46] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. *Horn Clause Solvers for Program Verification*, pages 24–51. Springer International Publishing, 2015.
- [47] Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Android in the wild: a large-scale dataset for android device control. In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23*, Red Hook, NY, USA, 2023. Curran Associates Inc.
- [48] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models, 2023. URL <https://arxiv.org/abs/2309.11495>.
- [49] Haoqiang Kang, Juntong Ni, and Huaxiu Yao. Ever: Mitigating hallucination in large language models through real-time verification and rectification, 2023. URL <https://arxiv.org/abs/2311.09114>.
- [50] Aman Madaan, Niket Tandon, Prakhar Gupta, et al. Self-refine: Iterative refinement with self-feedback, 2023. URL <https://arxiv.org/abs/2303.17651>.
- [51] Mengjia Niu, Hao Li, Jie Shi, Hamed Haddadi, and Fan Mo. Mitigating Hallucinations in Large Language Models via Self-Refinement-Enhanced Knowledge Retrieval, 2024. URL <https://arxiv.org/abs/2405.06545>.
- [52] Sewon Min, Kalpesh Krishna, Ximing Lyu, Mike Lewis, Wen-tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. FactScore: Fine-grained atomic evaluation of factual precision in long form text generation, 2023.
- [53] Ian Chern, Shiyang Chern, Shushan Chen, Weizhi Yuan, Kai Feng, Chunting Zhou, Junxian He, Graham Neubig, Pengfei Liu, et al. FacTool: Factuality detection in generative AI—A tool augmented framework for multi-task and multi-domain scenarios, 2023.
- [54] Feng Nan, Ramesh Nallapati, Zhiling Wang, Cicero Nogueira dos Santos, Huan Zhu, Dong Zhang, Kathy McKeown, and Bing Xiang. Entity-level factual consistency of abstractive text summarization, 2021.
- [55] Ben Goodrich, Vrindavan Rao, Peter J. Liu, and Mina Saleh. Assessing the factual accuracy of generated text. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 166–175, 2019.

- [56] Kurt Shuster, Spencer Poff, M. Chen, Douwe Kiela, and Jason Weston. Retrieval augmentation reduces hallucination in conversation, 2021.
- [57] Francesco Fuggitti and Tathagata Chakraborti. NL2LTL - a python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI*, pages 16428–16430. AAAI Press, 2023.
- [58] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 383–396. Springer, 2023.