Reliable Execution of Natural Language Test Cases for GUI Applications using LLM Agents

Sébastien Salva[1,2*†]

[1*]IUT Clermont Auvergne, University Clermont Auvergne, 5 Av. Blaise Pascal, Aubière, 63170,France.
[2*], LIMOS - UMR CNRS 6158, 1 Rue de la Chebarde, Aubière, 63178,France.

Corresponding author(s). E-mail(s): sebastien.salva@uca.fr;
[†]These authors contributed equally to this work.

Abstract

Developing and maintaining automated tests for graphical user interface (GUI) applications remains costly, as test scripts are typically tightly coupled to low-level UI elements. Recent advances in large language models (LLMs) open new opportunities to execute test cases expressed in natural language (NL), potentially reducing the effort required to create and maintain test suites. However, delegating test execution to probabilistic LLM agents raises fundamental challenges for software testing, including ambiguity in NL test cases, unpredictable LLM agent behaviour and the lack of theoretical foundations for reasoning about NL test case execution reliability. This paper focuses on these problems and presents a NL test case execution algorithm that orchestrates specialised LLM agents to interpret and execute NL test steps in a controlled manner in order to limit false positives and false negatives. This algorithm includes guardrail mechanisms to mitigate uncertainty in language-driven execution. It combines grammar-based disambiguation of NL test steps, rule-based GUI readiness verification, and strict evaluation of assertions. To reason about the reliability of NL test case execution, we define the notions of weak unsoundness and weak laxness, which adapt classical conformance testing properties to the context of LLM agent-based testing. We implemented our approach in a prototype tool and constructed four NL test suites targeting six web applications to evaluate our approach in terms of weak unsoundness and weak laxness. Experiments with locally deployable open-weight LLMs (14B–70B parameters) show that, when paired with a high-capability model and under the conditions provided with our definitions, NL test cases can be executed reliably, they rarely reject conformant applications (false positives $<1\%$) and rarely accept non-conformant applications (false negatives $<4\%$). These results open the path toward treating natural language test descriptions as executable testing artifacts.

Keywords: GUI Application; Natural Language Test Case; Functional Testing; Unsoundness; Laxness; LLM

1 Introduction

Software testing is a cornerstone of software quality assurance, which often imposes substantial cost and effort, particularly in the development of automated test suites. This fact is especially pronounced for graphical user interface (GUI) applications, where functional test cases are costly to write and difficult to maintain. Indeed, implementing such tests typically requires low-level interaction scripts tightly coupled to specific UI elements, resulting in substantial maintenance overhead as the interface evolves. As a consequence, despite their importance, functional GUI test cases are often neglected or partially adopted in industrial practices [1, 2].

Artificial intelligence (AI) techniques offer promising opportunities to reduce the effort required for test development. Among the emerging applications in this space, two use cases merit particular attention.

First, natural language processing (NLP) techniques and more recently large language models (LLMs), have been employed to generate executable test cases from high-level scenarios or natural-language (NL) test descriptions. While this approach can substantially reduce manual development effort, assessing the functional correctness of the generated test code remains a significant challenge [3]. This difficulty arises from the inherent ambiguity of NL test descriptions, coupled with the constraint that the generated source code must be executable within a specific testing environment.

Second, recent work has investigated the use of agents powered by LLMs to directly interact with GUIs from natural language prompts [4–6], thereby enabling the potential execution of NL test cases without an intermediate code generation step. Although this approach would reduce test development costs and maintenance, delegating test execution to probabilistic LLM agents poses fundamental challenges to established software testing principles. In particular, it reintroduces the issue of ambiguity in NL test cases, and raises unresolved questions regarding the reproducibility of test outcomes and the conditions under which NL test case execution can be meaningfully related to conformance testing.

In this paper, we focus on the second use case, the direct NL test case execution on GUI applications by LLM agents. Our goal is to enable NL test cases to be treated as testing artifacts rather than informal documentation. To this end, we propose an algorithm that employs specialised LLM agents to interpret and execute NL test steps in a controlled and reproducible manner. To address the inherent uncertainty of language-driven execution, we introduce new NL test case execution property definitions and propositions, providing a rigorous theoretical foundation for reasoning about the reliability of NL test execution.

To position our contribution, we first review existing work on natural-language-based testing and LLM-driven test automation, with particular attention to their underlying assumptions and limitations. We then introduce the foundations of our approach and describe the main contributions.

1.1 Related work

AI has been applied to various software engineering activities, with steadily increasing adoption over the past decade. Some surveys covered the use of machine learning [7] or LLMs for software engineering [8]. These surveys classify AI-based approaches in program repair [9–13], source code generation [14–17], formal postcondition generation [18], or to audit the security of applications [19]. These approaches offer promising results but they still may generate code that contains bugs [20] or code with known and risky vulnerabilities [21].

Now, the problem of executing NL test cases on GUI applications using LLM agents intersects several research domains. These include recent work on LLM-based generation of formal software artifacts, NLP-based approaches for NL test case generation, AI-based automated test generation, and AI-driven GUI crawling techniques. Accordingly, we review the related literature along these four dimensions.

1.1.1 LLM Based Formal Artifact Generation

Recent work has investigated the capabilities of LLMs to translate NL into formal expressions, such as logical specifications or postconditions. These approaches primarily benchmark the semantic accuracy of LLMs in producing structured formal representations from textual inputs. Studies such as the one of Wu et al. [22] demonstrate that LLMs can convert mathematical problems stated in natural language into formal specifications and proofs in Isabelle/HOL, highlighting the potential of LLMs for formalisation tasks in formal verification and theorem proving contexts. Cosler et al. introduced nl2spec [23], an interactive framework for deriving formal temporal logic properties from unstructured NL descriptions via LLMs, enabling users to refine ambiguous translations through an iterative subformula mapping process. More recent work such as NL2ACSL [24] explores translating NL requirements into ANSI/ISO C specification language contracts using multi-agent LLM frameworks with retrieval-augmented generation and reinforcement learning components. Endres et al. [18] formulate the task of leveraging LLMs to transform natural language intent into formal method postconditions that can be programmatically checked. This work introduces metrics such as correctness and discriminative power to assess translation quality and shows that LLM-generated postconditions can catch real historical bugs in benchmarks like Defects4J [25].

Our work does not aim at generating formal expressions from descriptions or prompts. Instead, we study the interactive execution of NL test cases on GUI applications under test. This setting requires LLM agents not only to interpret instructions, but also to interact with the system, observe runtime states and evaluate assertions in a closed-loop manner. Moreover, we embed this execution process within conformance testing, which cover properties like soundness of the test cases. To our knowledge, existing NL-to-formal-expression do not address testing execution-level properties like soundness in the context of conformance testing.

1.1.2 NLP-based NL Test Case Generation

Mapping natural language instructions to executable actions is a long-standing research challenge spanning semantic parsing, program synthesis, and AI techniques. Prior work has explored the translation of NL requirements or use cases into NL test cases using NLP.

Text2Test [26] is among the earliest approaches to address this problem by proposing the automated generation of NL test cases from use cases. Text2Test analyses use cases and transforms them into abstract models capturing both linguistic and functional properties. These models enable automated validation of use-case style, completeness, structure, and control flow. NL test cases are then generated from the abstract use-case model by covering each of its valid execution path.

A more recent systematic literature review by Ayenew et al. [27] surveys research published between 2015 and 2023 on using NLP to automate test case generation from NL requirements. The authors identified 13 primary studies, encompassing seven NLP techniques, two tools, and one framework. Across these works, fundamental NLP tasks such as tokenization, part-of-speech (POS) tagging, syntactic parsing are commonly employed decompose requirements into structured representations that can be mapped to test case elements.

Adopting a different technique, the study presented in [3] investigates the generation of NL test cases from requirements by calling LLMs with specialised prompts. Their empirical evaluation, conducted on a single subject system, indicates that when those requirements are fully specified, LLMs can generate NL test cases that achieve high requirement coverage.

Overall, these approaches focus on producing NL test cases from textual requirements, which can be seen as complementary to our work. In contrast, our approach assumes NL test cases as input and focuses on their automated execution on GUI applications.

1.1.3 AI-driven GUI Crawling Techniques and Fuzz Testing

Closer to our work, several recent approaches extend traditional GUI crawling techniques with LLMs or LLM agent systems to improve bug detection, particularly for mobile applications. Unlike earlier approaches based on meta-heuristics or random exploration [28], these techniques leverage language-driven reasoning to guide exploration more effectively.

DroidAgent [5] is a multi-agent system for autonomous testing of Android applications. It employs three cooperating agents responsible for interaction, observation, and evaluation of task success. Experimental results show that DroidAgent outperforms state-of-the-art Android testing tools, achieving up to 10% higher code coverage than the baseline.

GPTDroid [4] is another LLM-based framework for Android application testing that aims to improve the application state coverage. The approach extracts the GUI context of the application, records past testing interactions and encodes this information into prompts for an LLM. The model's responses are then decoded into executable GUI actions, which are iteratively applied to the application. An evaluation on 93 popular Android applications demonstrates that GPTDroid achieves up to 32% higher coverage compared to the baseline.

Beyond mobile applications, PathFinder [6] proposes a multi-agent LLM framework for executing test scenarios on web applications. Given a high-level scenario, multiple agents coordinate to realise it by crawling the web application under test. The study also analyses the impact of agent diversity and LLM selection, showing that using the same LLM across all agents is more effective when targeting a single application, whereas employing heterogeneous LLMs yields better results when generalising across multiple websites.

UXAgent [29] focuses on usability testing of web applications using LLM-based agents. The system executes a predefined list of actions through a browser connector, while allowing testers to configure agent personas to simulate different user behaviours. The agent

reports page-level information, multimedia content, and simulated user interactions to support usability analysis.

In parallel, fuzz testing, which aims to expose software faults by providing unexpected or random inputs, has also been enhanced using AI techniques. LLMFuzz [30] leverages LLMs to generate high-quality inputs for fuzzing deep learning libraries. The approach first uses a code-generating LLM to synthesise valid seed inputs and then combines an infilling LLM with an evolutionary algorithm to guide input generation toward increased diversity, broader API coverage, and a higher number of unique program behaviours.

1.1.4 AI-based Automated Test Generation

Another substantial body of work focuses on generating executable test cases from NL requirements or specifications, aiming to reduce manual effort and improve traceability between requirements and tests.

Arruda et al. [31] proposed a practical test automation strategy that transforms NL test cases into executable test scripts without requiring programming expertise. The approach combines NLP with a capture-and-replay mechanism: existing automated "seed" test steps are stored in a database and reused by matching new natural language steps against them, while unmatched steps are automated through user interaction capture and subsequently stored for reuse. To address the ambiguity of free-form language, the authors also introduce a Controlled Natural Language for writing test cases, together with a formal semantic framework for consistency checking.

UMTG [32] is a toolset that automatically generates test cases from use case specifications written in natural language, with a particular focus on safety-critical systems. By combining NLP with constraint solving, UMTG extracts behavioural and control-flow information from restricted use case templates. The approach builds formal representations and OCL constraints from use cases and associated domain models, to support systematic test generation and improve traceability between requirements and generated tests.

NAT2TEST [33] also generates test scripts from NL requirements by transforming them into formal specifications. To mitigate ambiguity, requirements are again written using a controlled natural language.These representations are then translated into Software Cost Reduction (SCR) specifications, which serve as an intermediate model for test generation. The approach is evaluated on industrial systems from the aerospace and automotive domains, demonstrating that formalising requirements can effectively support high-quality test generation.

More recently, LLM-based approaches have been explored to automate test generation. ChatUniTest [34] is an LLM-driven framework for automated unit test generation that employs an adaptive focal context mechanism and a generation–validation–repair loop to improve test correctness. Experimental results show that ChatUniTest outperforms existing tools such as TestSpark [35] on a subset of evaluated projects, achieving higher line coverage. The study presented in [3] also investigated the use of LLM to generate concrete executable test cases. Their results indicate that the generation of executable test cases remains challenging, with at least 30% of generated tests requiring manual modification before execution.

Our work takes a fundamentally different perspective by assuming NL test cases as input and addressing the challenge of executing them directly on GUI applications.

1.2 Key Observations and Motivations

The reviewed literature shows a rapid expansion of AI- and LLM-based techniques across test generation. Prior work concentrates on transforming requirements into NL or executable test cases using NLP pipelines, controlled natural languages, or, more recently, LLM prompting. In parallel, LLM-augmented GUI crawlers and fuzzing frameworks improve coverage and bug discovery through autonomous, language-guided exploration, but typically operate without predefined test cases, which limits repeatability and consistency as execution paths are largely driven by stochastic LLM choices. Recent LLM-based test generation approaches further reveal that producing executable and correct tests remains challenging, with substantial rates of manual correction. Collectively, these works highlight three limitations: (i) the lack of approaches that take NL test cases as artifacts for direct execution on GUIs, (ii) limited attention to test execution consistency when LLM agents are involved, and (iii) insufficient theoretical grounding of NL test execution with respect to established test case properties, such as unsoundness, which characterises test cases that reject conformant implementations, and laxness, which characterises test cases that accept non-conformant implementations that they could otherwise reject.

Motivated by these observations, we initiated a first work [36] to investigate whether LLM agents can directly execute NL test cases. Unlike LLM-based GUI crawlers, we target conformance test execution with repeatability. To this end, we introduced a first algorithm and a multi-agent system for executing NL test cases on GUI applications. As LLM agents may behave unpredictably, or even hallucinate steps that are not specified, we introduced a first LLM-based guardrail mechanism, which checks whether the current GUI includes the expected UI elements before performing a navigation action. We also formulated a first definition of weak unsoundness, which characterises contexts under which NL test cases can be reliably executed by LLM agents on conformant GUI applications. Finally, we defined a measure to estimate the execution consistency of NL test cases, that is a measure assessing how stable the test outcomes would be if the tests were executed multiple times. An empirical evaluation of that approach revealed several limitations, notably the impact of ambiguity in NL test cases, which can lead to false test verdicts.

1.3 Contributions

We present in this paper a new execution algorithm for NL test cases on GUI applications that substantially extends our initial contribution through the following improvements:

1. Management of NL test case ambiguity: prior work has extensively studied ambiguity in NL requirements and test specifications, often mitigating it through constraint solving or the use of Controlled Natural Languages (CNLs) [31–33]. We also address ambiguity at execution time by analysing each NL test step against a restricted CNL defined as a formal grammar. When a step is detected as ambiguous, an LLM agent attempts to rewrite it into an equivalent sequence of unambiguous steps that conforms to the grammar. If this transformation fails, the execution yields an inconclusive verdict, explicitly capturing the inability to disambiguate the test intent;
2. Strict verification of GUI readiness: unlike our initial algorithm, which relied on LLM-based reasoning to assess GUI readiness, i.e. the fact to check whether the GUI is ready

and includes the required elements to perform an action, we introduce a strictly rule-based readiness verification mechanism. This change reduces reliance on probabilistic LLM judgments and aims to lower the rate of inconclusive verdicts;
3. Improved handling of quiescence: we refine the NL test case execution algorithm by explicitly incorporating quiescence, that is the detection that no further observable output is produced by the application within a bounded time window, through well-defined timeout mechanisms. This allows the executor to distinguish between delayed system responses and genuine deadlock or non-responsiveness, thereby improving execution robustness;
4. Formal analysis of test case properties: we study the unsoundness and laxness of NL test case execution with respect to the ioco conformance relation [37];
5. Extended tooling and empirical evaluation: we assess the effectiveness of NL test case execution with our algorithm in terms of unsoundness and laxness. The evaluation is supported by new tools and newly constructed test suites targeting six web applications.

In summary, the major contributions of this paper are:

- a new algorithm for executing NL test cases that incorporates grammar-based disambiguation and rule-based GUI readiness checks;
- an analysis of both the unsoundness and laxness of NL test case execution. To account for the inherent uncertainty introduced by LLM-based execution, we define two properties, weak unsoundness and weak laxness. We further provide propositions identifying the conditions under which these properties hold;
- two prototype tools and 4 test suites, which are made publicly available in [38];
- an experimental evaluation of the weak unsoundness and weak laxness of NL test case execution. The results show that the proposed algorithm performs reliable NL test case execution when paired with a high-capability LLM such as Llama3.3 70B. For NL test cases that satisfy the conditions specified in our propositions, conformant GUI applications are rarely rejected (false positives below 1%), and non-conformant GUI applications are rarely accepted (false negatives below 4%). In addition, these results are obtained using a locally deployable open-weight LLM that can run on local servers or workstations, demonstrating the practical applicability of our approach in industrial contexts where proprietary cloud-based models may not be suitable.

Paper organisation: the remainder of the paper is structured as follows. Section 2 presents the theoretical background and our NL test case execution algorithm. Section 3 investigates NL test case properties and presents the definitions of weak unsoundness and weak laxness. Section 4 reports and analyses the experimental results. Finally, Section 5 concludes the paper and outlines directions for future work.

2 NL Test Case Execution

2.1 Models and Assumptions

GUI Applications: we consider having a black-box GUI application under test, denoted AUT, for which either the Document Object Model (DOM) structure is accessible or screenshots can be obtained. The GUI further provides mechanisms for user interaction. Classically with this application type, we assume that interactions are reactive and yield observable effects, hence after each user interaction, the application produces observable GUIs or is quiescent. As a black box, we impose no structural constraints on AUT. In particular, we do not assume that AUT is deterministic. Although our implementation focuses on web applications, primarily due to the availability of mature browser automation tools, the proposed algorithms are not restricted to this domain. The approach can be generalised to other classes of GUI-based systems, such as mobile applications, provided that suitable adapter layers are implemented to expose comparable interaction primitives and state observation mechanisms.

Conformance Testing: in conformance testing theory, specifications, implementations, test cases and implementation relations are expressed using formal models. We consider in the paper that no (formal) specification is available. But we assume that this specification can be formulated to relate the execution of NL test cases to classical conformance testing theory and to reason about test case properties. We adopt the classical model of input output labelled transition systems (IOLTS) [37]. We assume that the specification is a deterministic IOLTS $S =< Q^S, L^S \cup \mathcal{T}, \rightarrow^S, q_0^S >$, where Q^S denotes the state set, $L^S = L_I^S \cup L_O^S$ the set of inputs starting by "?" and the outputs starting by "!" respectively, $\mathcal{T}$ the set of internal actions (not observable from the application environment), $\rightarrow^S$ the transition set, and q_0^S the initial state. We also adopt the standard IOLTS notations given in Annex, Section B. In the present work, inputs typically correspond to navigation actions, such as clicking on "Sign in", and outputs correspond to the resulting GUIs. The specified behaviours of the application can be extracted from the traces of S, denoted $traces(S)$, which gathers all the sequences of observable inputs and outputs obtained by covering the paths of S.

NL Test Case: a NL test case is defined as a sequence of actions $a_1 \dots a_k \mathcal{A}_{k+1} \dots \mathcal{A}_l$, where $a_1 \dots a_k \in L_I$ are inputs of the specification S and $\mathcal{A}_{k+1} \dots \mathcal{A}_l \in \mathcal{T}$ are assertions treated as internal actions. An assertion $\mathcal{A}$ may be a simple predicate or a composite expression formed through conjunctions and disjunctions of simpler assertions. Under this formulation, navigation actions and assertions cannot be interleaved; assertions are therefore interpreted as postconditions. Nevertheless, interleaved navigation actions and assertions can be represented in our framework by decomposing them into a sequence of smaller NL test cases, each consisting of a prefix of actions followed by assertions. This decomposition provides a practical strategy that improves the readability of our test case execution algorithm. NL test cases for GUI applications are typically written as linear scenarios; we thus restrict NL test cases to sequential structures. To relate NL test cases with conformance testing, we say that a test case $tc = a_1 \dots a_k, \mathcal{A}_{k+1} \dots \mathcal{A}_l$ is consistent with a specification S if there exists a trace $?a_1!g_1 \dots ?a_k!g_k \in traces(S)$ and each assertion $\mathcal{A}_i (k+1 \leq i \leq l)$ evaluates to true on the GUI g_k. This condition ensures that the actions and assertions encoded in an NL test case correspond to behaviours permitted by the specification and thus constitutes a faithful abstraction

of the intended system behaviour. This trace structure directly reflects our modelling assumption for GUI applications, i.e. each user interaction is reactive and followed by an observable effect (quiescence included).

NL Test Case Example

1. Go to the website `https://google-gruyere.appspot.com/start`
2. Click on the button “Agree & Start”
3. Click on “Sign up”
4. Type in “toto” in the field “User name”
5. Type in “toto” in the field “Password”
6. Click on “Create account”
7. Assert that the “Account created” is displayed on this page

Fig. 1: NL test case ”Sign Up to Google Gruyere” for verifying the user registration functionality of the web application google-gruyere

Figure 1 illustrates a NL test case example whose objective is to verify that the sign-up functionality of the Google Gruyere website operates correctly. The first six lines describe user interactions with the GUI, while the final line expresses an assertion that checks whether the account has been successfully created.

2.2 Conformance Testing with NL Test Cases

Conformance relations characterise the extent to which an implementation AUT conforms to its formal specification S. To enable their definition, it is assumed that AUT can likewise be represented as an (unknown) IOLTS. In this paper, we adopt the ioco conformance relation [37] whose definition is recalled below. Intuitively, under *ioco*, AUT conforms to S if every action sequence $\sigma \in \text{Traces}(S)$, produces, when executed on AUT, only outputs anticipated by S. The operator $after$ returns the state set reached after the execution of σ and *out* the set of outputs observed from those states (definitions are given in the Annex).

Definition 1 (*ioco* implementation relation) Let AUT and S be two IOLTS.
$AUT\ ioco\ S \Leftrightarrow_{def} \forall \sigma \in traces(S) : out(AUT \text{ after } \sigma) \subseteq out(S \text{ after } \sigma)$

Conformance testing is typically enabled through the synthesis of test cases from a specification. In this paper, however, we consider having a set of NL test cases. The correspondence between the conformance testing of AUT and the execution of a NL test case tc is here established through the construction of the IOLTS $tc|AUT$, which represents the synchronous execution of tc on AUT. This IOLTS must meet the same constraints as the synchronous executions defined in conformance testing approaches, e.g. [37]. It must be deterministic, have no cycles, and have terminal verdict states. An execution path hence ends in a terminal state *pass*, $fail$ or *inconclusive*. The latter state serves as a protective verdict that prevents AUT

from being incorrectly rejected by the test case when the execution context is unreliable on account of LLM agents or of environment issues. Furthermore, $tc|AUT$ incorporates a special symbol θ to represent quiescence, that is, the absence of output produced by AUT.

Definition 2 (Synchronous Execution) Let AUT and S be two IOLTS, and tc a NL test case consistent with S.

- $tc|AUT$ is the IOLTS $< Q, L \cup \mathcal{T} \cup \{\theta\}, \rightarrow, q_0 >$ such that:
  - $tc|AUT$ is finite, deterministic, and acyclic,
  - Q contains three distinguished terminal states: *pass*, $fail$ and *inconclusive*
- AUT passes tc iff $\forall \sigma \in (L^S \cup \{\theta\})^*, tc|AUT \stackrel{\sigma}{\Rightarrow} pass$

2.3 NL Test Case Execution

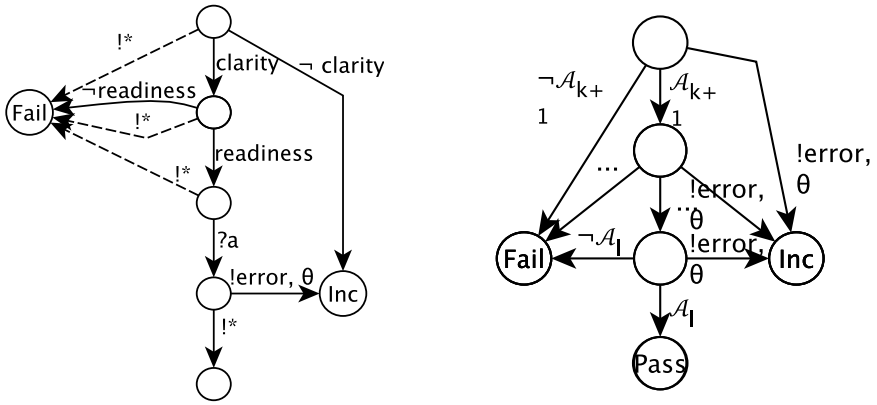


Fig. 2: Illustration of the test case execution algorithm with two IOLTS. Navigation actions start by "?", outputs (GUIs, errors) start by "!", θ expresses no observation from AUT. (a) Navigation actions are injected with internal actions, readiness and observe. For readability, "!*" represents any other output. (b) Assertions are incrementally evaluated to determine the final verdict.

Given a NL test case $tc = a_1 \dots a_k \mathcal{A}_{k+1} \dots \mathcal{A}_l$, the synthesis of the synchronous execution $tc|AUT$ is performed by Algorithm 1. This algorithm calls LLM agents to execute navigation actions and evaluate assertions while constructing the transitions of $tc|AUT$.

Algorithm 1 starts by covering every navigation action a_i in $a_1 \dots a_k$ (lines 2-23). Intuitively, it builds one of the branches of the tree illustrated in Figure 2a. The newly added transitions may be labelled by: inputs representing navigation actions applied to AUT; outputs representing GUIs or errors produced by AUT; θ expressing no observation from AUT. Algorithm 1 also inserts internal actions that serve as guardrails to supervise agent execution, ensuring that each navigation action is feasible and correctly executed. Specifically, two

actions, *clarity* (lines 5–7) and *readiness* (lines 10–12), are systematically introduced for every navigation action, as illustrated in Figure 2a. *clarity* encapsulates a substep whose purpose is to ensure that each navigation action a_i is interpretable when executed by LLM agents. This substep may require the translation of a_i into one or more additional navigation actions when a_i is ambiguous. If the resulting navigation actions cannot be interpreted again, the test execution terminates with an inconclusive verdict. *readiness* checks whether the next navigation action is executable, i.e., whether the necessary UI elements are present on the interface to perform the navigation action. As illustrated in Figure 2a, a fail verdict is returned if either $readiness(a_i)$ evaluates to false or if an unexpected output is observed. Additionally, if an error is observed, or if there is no observation, an inconclusive verdict is returned, as such reactions may originate either from the behaviour of the LLM agents or from AUT itself. Otherwise, the algorithm proceeds to the next navigation action in the sequence. When the algorithm reaches the assertions $\mathcal{A}_{k+1} \dots \mathcal{A}_l$ (lines 24-31), it evaluates them incrementally and completes $tc|AUT$ with some transitions as illustrated in Figure 2b. Assertions are evaluated using a two-step strategy to limit the use of LLM. An assertion is first evaluated using strict predicate-logic formulas derived from the assertion and the GUI. If the strict evaluation fails, the assertion is subsequently evaluated by an LLM agent. If an assertion finally evaluates to false, the execution terminates with a fail verdict. The inconclusive state is reached if an error is triggered by an LLM agent during evaluation or if a quiescent state is observed. Otherwise, the pass verdict is given.

We now detail how every action (navigation action, assertion, readiness, observe) are performed.

2.4 Navigation Action Execution

Navigation actions are performed by the agent $agent_{nav}$, which combines an LLM-based reasoning and decision-making component, a persistent memory module for tracking pending and completed tasks, and a function-invocation interface.

The memory module allows to store the current execution state (e.g., completed or erroneous), step queues and intermediate results, while also maintaining a historical context. This historical context enables the agent to revisit previously accessed pages and make informed decisions based on past interactions. The function invocation interface allows the agent to programmatically interact with web content via the Document Object Model (DOM) or to interpret visual information extracted from screenshots.

This agent is invoked using a prompt that we designed and structured following the patterns proposed in [39]. we considered the following patterns: (1) the “Fact Checklist” pattern, which prompts the agent to output a list of facts included in the final answer, thereby enhancing transparency of the results; (2) the “Template” pattern, which enforces an output format in JSON and includes fields such as extracted facts, results, and a boolean indicator of task completion status; and (3) the “Recipe pattern”, which generates a complete sequence of steps to accomplish the task. The full prompt is provided in the Annex in Figure A1.

2.5 Action Ambiguity Evaluation and Disambiguation

Controlling ambiguity in NL test case actions is crucial, as ambiguous formulations may lead to multiple, non-equivalent interpretations and therefore to inconsistent test executions. The

Algorithm 1: NL test case execution

input : Test case $tc = a_1 \dots a_k \mathcal{A}_{k+1} \dots \mathcal{A}_l$
output: Test verdict, IOLTS $tc|AUT = < Q, L \cup \mathcal{T} \cup \{\theta\}, \rightarrow, q_0 >$

1 $i := 1$;
2 **while** $i \leq k$ *AND* $\neg$ *stop* **do**
3 //**Observe** AUT;
4 Get observation from the environment Add $q_{i-1} \xrightarrow{!observation} fail$; Return fail, stop;
5 //**Check Clarity**;
6 Add $q_{i-1} \xrightarrow{clarity} q_{i,1}$ if $clarity(a_i, tc)$ true;
7 Add $q_{i-1} \xrightarrow{\neg clarity} Inc$ if $clarity(a_i, tc)$ false, Return Inconclusive, stop;
8 //**Observe** AUT;
9 Get observation from the environment Add $q_{i,1} \xrightarrow{!observation} fail$; Return fail, stop;
10 //**Check Readiness**;
11 Add $q_{i,1} \xrightarrow{readiness} q_{i,2}$ if $readiness(a_i, !g)$ true;
12 Add $q_{i,1} \xrightarrow{\neg readiness} fail$ if $readiness(a_i, !g)$ false, Return fail, stop;
13 //**Observe** AUT;
14 Get observation from the environment Add $q_{i,2} \xrightarrow{!observation} fail$; Return fail, stop;
15 //**Give** a_i **to** AUT;
16 Execute a_i;
17 Add $q_{i,2} \xrightarrow{?a_i} q_{i,3}$;
18 //**Observe** AUT;
19 Get observation from the environment Add $q_{i,3} \xrightarrow{!g} q_{i,4}$ if observation is GUI g;
20 Add $q_{i,3} \xrightarrow{!error} inc$ if observation is $error$, Return inc verdict, stop;
21 Add $q_{i,3} \xrightarrow{\theta} inc$ if quiescence is observed, Return inc verdict, stop;
22 Add $q_{i,3} \xrightarrow{!*} fail$ if anything else observed; Return fail; stop;
23 i++;
24 //**Assertions on last GUI**;
25 **while** $k+1 \leq i \leq l$ *AND* $\neg$ *stop* **do**
26 Add $q_{i-1} \xrightarrow{\mathcal{A}_i} q_i$ if $eval(\mathcal{A}_i, !g)$ true AND $i \neq l$;
27 Add $q_{i-1} \xrightarrow{\mathcal{A}_i} pass$ if $eval(\mathcal{A}_i, !g)$ true AND $i = l$, return pass;
28 Add $q_{i-1} \xrightarrow{\neg \mathcal{A}_i} fail$ if $eval(\mathcal{A}_i, !g)$ false, Return fail, stop;
29 Add $q_{i-1} \xrightarrow{!error} inc$ if $eval(\mathcal{A}_i, !g) = error$, Return inconclusive, stop;
30 Add $q_{i-1} \xrightarrow{\theta} inc$ if quiescence is observed, Return inconclusive, stop;
31 i++;

literature includes several works that address the ambiguity of scenarios and test descriptions [31, 40]. In this paper, we address ambiguity by evaluating actions against a controlled natural language defined by a grammar that restricts the set of syntactically valid actions. Let $\mathcal{G}$ be this grammar, and $\mathcal{L}(\mathcal{G})$ the language it generates. We say that an action of an NL test case is ambiguous if it does not conform to $\mathcal{L}(\mathcal{G})$:

Definition 3 (NL Test Case Ambiguity) Let $tc = a_1 \dots a_k \dots \mathcal{A}_{k+1} \dots \mathcal{A}_l$ be a NL test case, $\mathcal{G}$ be a grammar and a be an action of tc.

- $ambiguous(a)$ iff $a \notin \mathcal{L}(\mathcal{G})$,
- $ambiguous(tc)$ iff $\exists a$ in $a_1 \dots a_k \mathcal{A}_{k+1} \dots \mathcal{A}_l$, $ambiguous(a)$

Algorithm 1 invokes the boolean procedure *clarity*, given in Algorithm 2 to ensure that every navigation action is unambiguous before its execution. If a navigation action a does not

Algorithm 2: Procedure $clarity(a, tc)$

```
input  : action a, NL test case tc
output: Boolean
if ambiguous(a) then
    a'_1 ... a'_n := rewrite(a);
    Replace a with a'_1 ... a'_n in tc;
    if ambiguous(a'_1) then
        return false;
return true;
```

conform to $\mathcal{L}(\mathcal{G})$, *clarity* attempts to rewrite a into one or more actions that satisfy the grammatical constraints. The synthesised steps are then inserted into the NL test case, replacing or expanding the original action. The first resulting action is subsequently revalidated against the grammar. If it conforms to $\mathcal{L}(\mathcal{G})$, *clarity* returns true and the test execution proceeds. Otherwise, *clarity* returns false, and Algorithm 1 terminates the test execution with an inconclusive verdict.

The translation of a navigation action is performed by the LLM agent $agent_{clarity}$, which is requested using a structured prompt that combines the "Fact Check List" and "Template" patterns [39], similar to the prompts used for navigation execution. We additionally employ the "Chain of Thought" prompting technique [41] to encourage intermediate reasoning and to obtain stepwise transformations. The full prompt is provided in Annex (Figure A2), together with a minimal example of the grammar used for validation.

Test Case A (Ambiguous)

1. Open the website `http://localhost/opencart`
2. Click twice with the mouse on the link "My Account"
3. Click on "Register"
4. Fill the form with "toto" in the field *firstname* and "Tata" in the field *lastname* and "toto@example.com" in the field *email* and "new_password" in the field *password*
5. Click on agree
6. Click on "Continue"
7. Assert that "Continue" is displayed on the page

Test Case B (Disambiguated)

1. Open the website `http://localhost/opencart`
2. Click on "My Account"
3. Click on "Register"
4. Fill the field *firstname* with "toto"
5. Fill the field *lastname* with "Tata"
6. Fill the field *email* with "toto@example.com"
7. Fill the field *password* with "new_password"
8. Click on agree
9. Click on "Continue"
10. Assert that "Continue" is displayed on the page

Fig. 3: Comparison of an ambiguous NL test case (left) and its disambiguated version (right).

Figure 3 illustrates the transformation of an ambiguous NL test case into its disambiguated counterpart, with respect to the grammar given in Annex in Figure A2. In the ambiguous

version, several steps combine multiple operations, for example, issuing a double click or populating several form fields within a single instruction. The disambiguated version resolves these issues by decomposing each compound instruction into a sequence of explicit, atomic actions, ensuring a unique and deterministic interpretation. Each form field is filled individually, and navigation steps such as selecting "My Account" or "Register" are expressed as single, well-defined operations.

2.6 Readiness Evaluation

Following *clarity*, which ensures that a navigation action a is well-formed, the procedure *readiness* of Algorithm 1 checks whether the current GUI is suitable for executing a next navigation action, i.e., whether all required UI elements are present.

Given a navigation action a that has to be executed on the GUI $!g$, $readiness(a, !g)$ translates an action a into a predicate $p(a)$ and evaluates the formula $\bigwedge_1^k \phi_i \wedge p(a)$, where each sub-formula ϕ_i specifies a concrete GUI condition or constraint that must hold for a to be executable on $!g$. Rather than querying an agent, this formulation offers a precise and deterministic guarantee that the action is doable on the GUI. Table 1 lists examples of sub-formulae designed for evaluating readiness.

For example, consider the navigation action "Click the link 'Sign in'". The action is translated into the predicate "Click('Sign in')". A representative sub-formula (ϕ) used to assess whether this action is executable on the GUI $!g$ may be: $\forall x, \text{Click}(x) \implies \text{In}(x, content)$ with the predicates $\text{Click}(x) \implies$ "Element x is clicked" and $\text{In}(x) \implies$ "Element x is in the GUI content".

Table 1: Examples of formulas for evaluating GUI readiness.

Action	**Formula**
click(x)	$\forall x, \text{Click}(x) \implies \text{In}(x, \text{GUIcontent}) \wedge (\text{Type}(x, \text{"link"}) \vee (\text{Type}(x, \text{"button"}))$
select(x, v)	$\forall x, \text{Select}(x, v) \implies \text{In}(x, \text{GUIcontent}) \wedge \text{Type}(x, \text{"list"}) \wedge \text{In}(v, \text{option(x)})$
check(x)	$\forall x, \text{Check}(x) \implies \text{In}(x, \text{GUIcontent}) \wedge \text{Type}(x, \text{"checkbox"}) \wedge \neg\text{Checked}(x)$
uncheck(x)	$\forall x, \text{Uncheck}(x) \implies \text{In}(x, \text{GUIcontent}) \wedge \text{Type}(x, \text{"checkbox"}) \wedge \text{Checked}(x)$
fill(x, v)	$\forall x, \text{Fill}(x, v) \implies In(x, \text{GUIcontent}) \wedge \text{Type}(x, \text{"input"})$

2.7 Assertion Evaluation

Before evaluating assertions, we chose to not assess their ambiguity using the clarity procedure, unlike navigation actions, for two main reasons. First, assertions serve directly as test oracles; consequently, test engineers must explicitly and precisely specify the properties they intend to verify on GUIs. Modifying assertions would risk altering the intended test objective. Second, the semantics of assertions are substantially broader and more diverse than those of navigation actions. Assertions may range from simple predicates (e.g., "Assert that the 'Send' button is present") to more complex properties (e.g., "Assert that the total price displayed is 100$"). Designing a grammar that is sufficiently expressive to capture this wide semantic spectrum while remaining tractable is considerably more challenging than for navigation actions.

Algorithm 3: Procedure $eval(\mathcal{A}, !g)$

input : Assertion $\mathcal{A}$
output: Boolean
1 $\mathcal{A}_1 op_1 \dots op_{k-1} \mathcal{A}_k$:= Split($\mathcal{A}$);
2 $bool_i := assert\_strict(\mathcal{A}_i) \vee (\neg assert\_strict(\mathcal{A}_i) \implies assert\_w\_agent(\mathcal{A}_i))$;
3 Evaluate $bool_1 op_1 \dots op_{k-1} bool_k$;

Instead, we adopt a two-step evaluation strategy. An assertion $\mathcal{A}$ is evaluated on a GUI $!g$ by the procedure $eval(\mathcal{A}, !g)$ implemented by Algorithm 3. This procedure takes an assertion $\mathcal{A}$ and splits it into a set of elementary assertions separated by logical operators. Each elementary assertion $\mathcal{A}_i$ is then evaluated using the following logical formula: $assert\_strict(split(\mathcal{A}_i)) \vee (\neg assert\_strict(split(\mathcal{A}_i)) \implies assert\_w\_agent(\ split(\mathcal{A}_i)))$. Finally, the evaluation of the original assertion is computed by combining the results of the elementary assertions according to the previously identified operators. The formula includes:

1. $assert\_strict$, which refers to a boolean procedure, firstly used to strictly evaluate an elementary assertion $\mathcal{A}_i$, in the same spirit as the readiness check for navigation actions. $assert\_strict$ translates the assertion $\mathcal{A}_i$ into a predicate $p(\mathcal{A}_i)$ and evaluates the predicate by means of the formula $\bigwedge_{i=1}^{k} \phi_i \wedge p(\mathcal{A}_i)$. The sub-formula ϕ_i expresses simple but usual generic assertions considered in GUI testing. For example, "Assert that 'x' is present" is formulated by $\forall x, Present(x) \implies In(x, page)$. Table 2 lists several sub-formula examples;
2. $assert\_w\_agent$, which is another procedure called to evaluate the assertion $\mathcal{A}_i$ when $assert\_strict$ initially evaluates $\mathcal{A}_i$ to false. The procedure invokes the LLM agent $agent_{assert}$ with another prompt also constructed with the patterns "Fact Check List", "Template" and "Recipe". The latter is specifically used to force the agent to focus on the UI elements relevant to the assertion, before evaluating them and giving a final verdict. It also uses "Chain of Thought" to explicitly require the generation of intermediate steps before generating the final response. The prompt is given in Annex in Figure A3.

Table 2: Examples of assertions expressed with formulas.

Assertion	Formula
'x' is present	$\forall x, \text{IsPresent}(x) \implies \text{In}(x, \text{GUIcontent})$
'x' is not present	$\forall x, \neg\text{IsPresent}(x) \implies \neg\text{In}(x, \text{GUIcontent})$
'x' is checked	$\forall x, \text{IsChecked}(x) \implies \text{In}(x, \text{GUIcontent}) \wedge \text{Checked}(x)$
'x' is visible	$\forall x, \text{IsVisible}(x) \implies \text{In}(x, \text{GUIcontent}) \wedge \text{Visible}(x)$

3 NL Test Case Properties

This section examines the impact of executing NL test cases with LLM agents on classical test-case properties. In particular, we focus on unsoundness and laxness [42]. Both properties characterise the absence of false positives and false negatives respectively imputed by

test cases, under a deterministic specification. In classical conformance testing, these properties are defined under the assumption that test execution is deterministic, and that any non-determinism originates solely from AUT. Flakiness [43] complements these notions by empirically accounting for execution-time nondeterminism arising from environmental factors, which may cause the same test case to yield inconsistent verdicts. From the perspective of verdict reliability, flakiness can be interpreted as an operational form of unsoundness, as it undermines the guarantee that observed failures correspond to real faults.

With NL test case execution, the situation fundamentally differs because the execution process itself becomes probabilistic due to the use of LLM agents. While AUT may still be non-deterministic, the test executor (LLM agents) also introduces stochastic behaviour, including variability in the interpretation of NL steps, occasional hallucinations, and execution errors. As a result, even a correct NL test case executed on a conformant AUT may occasionally yield an incorrect verdict due to the probabilistic nature of the LLM agents rather than the behaviour of AUT itself. In this section, we introduce the notions of weak unsoundness and weak laxness to explicitly account for this probabilistic execution context by relaxing the corresponding classical properties.

Another key property is the exhaustiveness of a test suite [37], which characterises that if all relevant test cases have been generated and if they all pass, then the implementation relation necessarily holds. However, because we assume the absence of a formal specification, we cannot presume the existence of a complete test suite composed of NL test cases. Consequently, exhaustiveness is not considered further in this paper.

Test case properties are typically defined with respect to formal models: their definitions require a specification, model-based test cases, and an implementation relation. NL test cases, however, do not satisfy these prerequisites. To formally relate the execution of NL test cases to classical conformance-testing theory, we assumed the existence of a specification S, and that a NL test case $tc = a_1 \dots a_k \mathcal{A}_{k+1} \dots \mathcal{A}_l$ should be consistent with S. Furthermore, the execution of a NL test case on AUT is modelled by an IOLTS dynamically constructed with Algorithm 1. This model incorporates inputs, outputs, and internal actions that lead to terminal states labelled with verdicts. It thus provides a formal basis for analysing the properties of NL test cases.

3.1 NL Test Case Unsoundness

A test case is unsound if there exists at least one conformant AUT that can be rejected by the test case. NL test cases executed by LLM agents can be unsound for a variety of reasons. The actions of a NL test case may be ambiguous and lead to unpredictable verdicts, assertions may be incorrect, or incorrectly evaluated, the LLM agents may hallucinate or return errors during the execution of tc.

In some practical contexts, we observed that the probability of obtaining fail verdicts for a conformant AUT when executing a NL test cases with Algorithm 1 remains within acceptable bounds, especially when compared to other sources of uncertainty, such as environmental disturbances, that can affect test execution. From this observation, we introduce the notion of weak unsoundness, defined with respect to rare contexts under which NL test case execution may be unsound. In other terms, we say that a NL test case is weakly unsound if there exists a conformant AUT that is rejected by the test case only under rare circumstances. We

make that formal through the following definition. Let Ω denote the set of all possible contexts of the form $\langle \lambda_{env}, \lambda_{agent} \rangle$, with λ_{env} modelling external runtime conditions and λ_{agent} expressing stochastic and configurable aspects of the LLM-based components. Ω_r denote a low-probability subset of Ω representing exceptional conditions under which a conformant AUT is rejected by a test: $\Omega_r = \{c \in \Omega \mid p(c) \leq \alpha\}$, with α a user-chosen error tolerance.

Definition 4 (Weak Unsoundness) Let $S =< Q^S, L^S \cup \mathcal{T}, \rightarrow^S, q_0^S >$ be a deterministic IOLTS, $tc = a_1 \ldots a_k \mathcal{A}_{k+1} \ldots \mathcal{A}_l$ be a NL test case such that $\exists ?a_1 !g_1 \ldots ?a_k !g_k \in traces(S)$ and $\forall (k+1 \leq j \leq l), eval(\mathcal{A}_j, !g_k)$ true, $tc|AUT$ be a deterministic IOLTS given by Algorithm 1, and $\Omega_r = \{c \in \Omega \mid p(c) \leq \alpha\}$.

tc is weakly unsound w.r.t. S for $ioco$ iff $\forall AUT, AUT$ ioco S, $\forall c \in \Omega \setminus \Omega_r, AUT$ passes $tc \wedge \exists AUT, AUT$ ioco $S, \exists c \in \Omega_r, \neg(AUT$ passes $tc)$

This definition relaxes the classical unsoundness property by tolerating incorrect verdicts only in rare contexts with low probability. This probability is influenced by a context of the form $\langle \lambda_{env}, \lambda_{agent} \rangle$ and more precisely by environment issues captured by λ_{env} and the variability induced by LLM agents captured by λ_{agent}. We abstract this component by considering only the probabilities of correct behaviour (accuracies) of our agents $\lambda_{agent} = \langle p(agent_{clarity}), p(agent_{nav}), p(agent_{assert}) \rangle$. This proposition provides a sufficient condition for weak unsoundness by relating the average reliability of the agents to this tolerance threshold.

Proposition 1 $disambiguable(a) =_{def} \neg ambiguous(a) \vee clarity(a, tc)$

Let $c = \langle \lambda_{env}, \lambda_{agent} \rangle \in \Omega_r$. tc is weakly unsound iff

1. $p(\lambda_{env}) < \alpha$,
2. $\lambda_{agent} = \langle p(agent_{clariry}) \geq 1 - \alpha, p(agent_{nav}) \geq 1 - \alpha, p(agent_{assert}) \geq 1 - \alpha \rangle$,
3. $disambiguable(a_i)_{1 \leq i \leq k}$

To minimise the impact of the residual uncertainty introduced by the LLM agents, we can restrict our attention to unambiguous NL test cases only. Under this restriction, navigation actions no longer require reformulation by $agent_{clarity}$, and assertions can be evaluated solely by $assert_{strict}$, without invoking $agent_{assert}$. This is captured by the following proposition:

Proposition 2 *tc is weakly unsound iff*

1. $p(\lambda_{env}) < \alpha$,
2. $\lambda_{agent} = \langle p(agent_{clariry}), p(agent_{nav}) \geq 1 - \alpha, p(agent_{assert}) \rangle$,
3. $\neg ambiguous(tc)$

Proofs of these propositions are given in Annex, Section B.1.

We propose a quantitative characterisation of the user-chosen error tolerance by leveraging the Six-Sigma methodology [44]. This approach provide a standardised way to express acceptable error rates by relating them to the probability that a process operates within predefined tolerance bounds. In this framework, higher Sigma levels correspond to lower defect

rates and therefore to stricter reliability requirements. Specifically, we define the lower bound of acceptable agent performance at the 3-Sigma threshold, which corresponds to a probability of $p_3 \geq 0.9332$. In many industrial contexts, 3-Sigma represents the critical inflection point between experimental instability and operational performance. By adopting the 3-Sigma threshold, we ensure that LLM agent behaviour remains within statistically predictable margins, providing a robust and acceptable baseline for NL test case execution.

Our experimental results indicate that, at present, Propositions 1 and 2 hold with a few LLMs, which can be deployed on local servers, with $\alpha = 1 - p_3$. We believe that ongoing advancements in both LLM and agent technologies will soon render Propositions 1 and 2 attainable with more small language models.

3.2 NL Test Case Laxness

A test case is classically lax if there exists one execution that incorrectly accepts a non conformant AUT, whenever the test case is capable of executing an action sequence that leads to an unexpected output in AUT. We propose to relax this property with the introduction of weak laxness. The execution of NL test cases by LLM agents can be lax, again for several reasons, including ambiguity in the test steps and stochatic behaviours of the LLM agents. Analogous to weak unsoundness, the defintinion captures the rare contexts under which an NL test case fails to reject a non-conformant AUT, while keeping such occurrences within acceptable bounds:

Definition 5 (Weak Laxness) Let $S =< Q^S, L^S \cup \mathcal{T}, \rightarrow^S, q_0^S >$ be a deterministic IOLTS, $tc = a_1 \dots a_k \mathcal{A}_{k+1} \dots \mathcal{A}_l$ be a NL test case such that $\exists ?a_1 !g_1 \dots ?a_k !g_k \in traces(S)$ and $\forall (k+1 \leq j \leq l), eval(\mathcal{A}_j, !g_k)$ true, and $tc|AUT$ be a deterministic IOLTS given by Algorithm 1.

tc is weakly lax w.r.t. S for $ioco$ iff $\forall AUT, \neg(AUT \text{ ioco } S), \forall c \in \Omega \setminus \Omega_r, \forall \sigma \in traces(S) \cap traces(AUT) \cap traces(tc|AUT), \forall !o \in L_O^S$ such that $\sigma.!o \in traces(tc|AUT), !o \in out(AUT$ after $\sigma), !o \notin out(S \text{ after } \sigma), tc|AUT$ after $\sigma!o = fail, \quad \wedge \exists AUT, \neg(AUT \text{ ioco } S), \exists c \in \Omega_r, \exists \sigma \in traces(S) \cap traces(AUT) \cap traces(tc|AUT), \exists !o \in L_O^S$ such that $\sigma.!o \in traces(tc|AUT), !o \in out(AUT \text{ after } \sigma), !o \notin out(S \text{ after } \sigma), tc|AUT$ after $\sigma!o \neq fail$

The following proposition establishes a first set of constraints under which a NL test case is weakly lax. As previously, weak laxness depends on the context $c = \langle \lambda_{env}, \lambda_{agent} \rangle \in \Omega_r$; it is also related to the ambiguity of the NL test cases and on the fact that an action should be successfully performed on a single GUI only, otherwise, the same action could also be valid on an unexpected or faulty GUI. This last restriction implies that navigation actions are restricted to be performed on a single GUI only and that assertions must hold on a single GUI, and can thus be regarded as forming a signature that characterises that GUI:

Proposition 3 *Let $c = \langle \lambda_{env}, \lambda_{agent} \rangle \in \Omega_r$. tc is weakly lax iff*

1. $p(\lambda_{env}) < \alpha$,
2. $\lambda_{agent} = \langle p(agent_{clariry}) \geq 1 - \alpha, p(agent_{nav}) \geq 1 - \alpha, p(agent_{assert}) \geq 1 - \alpha \rangle$
3. $disambiguable(a_i)_{1 \leq i \leq k}$,
4. *the navigation action can only be enabled on one GUI:* $\forall a_i (1 \leq i \leq k), \exists !g \in L_O^S, readiness(a_i, !g)$ *and* $\forall !g2 \neq !g \in L_O^S, \neg readiness(a_i, !g2)$,

5. *the assertions* $\mathcal{A}_{k+1} \ldots \mathcal{A}_l$ *hold only on a unique GUI:* $\exists !g \in L_O^S, \forall \mathcal{A}_j (k+1 \leq j \leq l), eval(\mathcal{A}_j, !g)$ *true, and* $\forall !g2 \neq !g \in L_O^S, \exists \mathcal{A}_j (k+1 \leq j \leq l), eval(\mathcal{A}_j, !g2)$ *false.*

As in the case of weak unsoundness, the residual uncertainty introduced by the LLM agents can be mitigated by enforcing that NL test case actions are unambiguous, thereby eliminating the need to invoke $agent_{clarity}$ or $agent_{assert}$:

Proposition 4 *tc is weakly lax iff :*

1. $p(\lambda_{env}) < \alpha$,
2. $\lambda_{agent} = \langle p(agent_{clariry}), p(agent_{nav}) \geq 1 - \alpha, p(agent_{assert}) \rangle$,
3. $\neg ambiguous(tc)$,
4. $\forall a_i (1 \leq i \leq k)$, $\exists !g : readiness(a_i, !g)$ *and* $\forall !g2 \neq !g : \neg readiness(a_i, !g2)$,
5. $\exists !g \in L_O^S, \forall k+1 \leq j \leq l, eval(\mathcal{A}_j, !g)$ *true, and* $\forall !g2 \neq !g \in L_O^S, \exists k+1 \leq j \leq l, eval(\mathcal{A}_j, !g2)$ *false,*
6. $\forall k+1 \leq j \leq l, \forall !g \in L_O^S, eval(\mathcal{A}_j, !g)$ *is performed by assert_strict only,*

Proofs of these propositions are given in Annex, Section C.1.

4 Experimental Results

In our previous work [36], we conducted an initial empirical evaluation to assess the feasibility of executing NL test cases using LLM agents. In particular, we investigated two research questions. The first evaluated the capability of eight LLM agents to correctly perform navigation actions and evaluate assertions across repeated executions. This analysis enabled us to identify three groups of models with different capability levels, characterised in terms of mean accuracy of execution. The first group includes models achieving mean accuracies greater than the level 3σ ($> 93\%$). It only includes Llama3.3 70B with a mean accuracy of 98%. The second group gathers mixed performance LLMs having mean accuracies greater than 80%. It includes Qwen2.5 7B, DeepSeek R1, and Devstral 24B. The last group comprises lower-capability models, with mean accuracies below 80%, including Qwen2.5 3B, Mistral 7B, Mistral Nemo 12B, and Qwen3 14B. The second question examined how our estimated measure of test case execution consistency aligns with the empirically observed consistency obtained from repeated test executions. The results provided initial evidence that our measure is reliable when the underlying LLM exhibits moderate to strong capabilities, while highlighting limitations for weaker models. Overall, the experiments showed that high-capability LLMs can execute NL test cases with both high accuracy and strong consistency, whereas smaller or less capable models exhibit significant variability and higher rates of execution errors.

In this paper, we conduct a new empirical evaluation to analyse the capabilities of Algorithm 1 with respect to weak unsoundness and weak laxness. This evaluation addresses two research questions:

- RQ1: To what extent do NL test cases executed by LLM agents incorrectly reject correct AUT behaviour? This question investigates weak unsoundness and the occurrence of

false positives during NL test case execution. We quantify this phenomenon using the unsound test ratio, and the Failure Exposure Rate (FER), which measures the frequency of false positives across repeated executions;
- RQ2: How robust is NL test case execution by LLM agents to incorrect expected behaviours? This question examines weak laxness and the rate of false negatives. We quantify this aspect using the lax test ratio and the Pass Exposure Rate (PER), which captures how frequently false negatives occur across repeated executions.
- RQ3: What is the average execution time of NL test cases executed by LLM agents compared to traditional test scripts? This question investigates the runtime overhead introduced by LLM agents during NL test case execution. We quantify this aspect by measuring the average execution time per NL test case and per execution step, and compare these results with equivalent test scripts manually implemented using the Playwright framework [1] to assess the practical cost of adopting LLM-driven execution.

This study was conducted with two prototype tools composed of three agents. Two agents, $agent_{clarity}$ and $agent_{assertions}$, are implemented as described in Section 2. The navigation agent, $agent_{nav}$, is built on top of the Stagehand framework [2], which automates browser interactions using natural language converted into low level method calls that perform interactions via LLMs. The system prompts used to interact with these agents are given in the figures A2, A1 and A3. These prompts along with parameters and implementation details are available this public repository [38]. Due to the novelty and specificity of our problem, namely the execution of NL test cases, there are no direct state-of-the-art baselines. Existing approaches either focus on NL test generation or GUI exploration using LLM agents, rather than deterministic execution with verdict correctness. This makes direct comparisons challenging. To address this, we developed a simplified version of our algorithm, a naive baseline that can yet execute NL test cases. As a consequence, the prototype tools are:

- **NLTestRunner**, which implements Algorithm 1. Given a test suite and a number of runs N, it executes NL test cases, returns verdicts, computes estimated consistency measures, and reports observed execution consistencies when $N > 0$. This tool incorporates the full architecture, including guardrail mechanisms.
- **SimpleNLTestRunner**, a lightweight baseline designed to approximate a minimal agent-based execution pipeline. It executes NL test cases with a navigation agent based on Stagehand and a separate agent for assertion evaluation, without guardrails by sequentially prompting LLM agents to perform each action. It relies on the same prompts and parameters as Algorithm 1, but removes coordination and validation mechanisms.

We conducted our experiments using three representative models selected based on the experiments conducted in our previous work [36]: Llama3.3 70B (highest observed capability), Qwen3 14B (mixed performance), and Mistral Nemo 12B (lowest observed capability). Those LLMs are publicly available and can be deployed on local infrastructure. This choice of

[1] https://playwright.dev/
[2] https://www.browserbase.com/stagehand

considering open-source rather than commercial models ensures reproducibility and strengthens the practical relevance of our approach for industrial contexts, where LLMs are often hosted on servers or workstations equipped with sufficient VRAM. The evaluated models range from 12B to 70B parameters and support tool-calling, which is a required feature for enabling GUI interaction. The LLM inference was carried out using Ollama[3], installed on a local server equipped with an NVIDIA L40 GPU (48 GB memory).

We selected six Web applications in the objective to evaluate our approach on heterogeneous and realistic GUI-based systems: Google Gruyere, UCA, ARTEMIS, a personal academic website, a Water Management System, and OpenCart. These applications were chosen to reflect a diversity of usage contexts, ranging from educational platforms and institutional websites to e-commerce and domain-specific systems. They offer the possibility to experiment different functionalities such as user authentication (e.g., sign-up/login workflows), navigation across multiple pages, and form-based interactions (e.g., data entry and validation). Table 3 describes some technical specifications and the approximate number of GUIs for each application.

Table 3: Technical Details and Domains of Evaluated Web Applications

Application	Domain/Description	Technologies	GUI NB
Google Gruyere	Educational platform specifically developed for security research.	Python, AJAX	11
OpenCart	Open-source and FREE eCommerce platform.	PHP 8, JavaScript, MySQL	∼45
Personal Website	Management of homepage, blog, and projects via Markdown.	HTML, JavaScript	22
Water Supply	Management and stock tracking with WhatsApp billing integration.	PHP, AJAX	15
UCA	Main website of the University Clermont Auvergne.	CMS KSUP, Java, JavaScript	>200
ARTEMIS	Secondary website of the University Clermont Auvergne.	CMS KSUP, Java, JavaScript	∼40

We also designed four evaluation test suites, whose descriptions are summarised in Table 4 for these six web sites. In the research questions, we studied two complementary aspects of NL test case execution: navigation action execution and assertion evaluation as different LLM agents are involved. To analyse these aspects, we designed two categories of test suites: TestG (navigation-intensive test cases), TestA (assertion-focused test cases). For the study of RQ2, we constructed two other test suitesTestG-lax and TestA-lax, which include incorrect behaviours to focus on laxness. These test suites are not tied to a single application. Instead, each suite contains test cases that are distributed across multiple web applications, depending on the functionality being exercised. From an experimental standpoint, this corresponds to 24 effective test suite instances (4 suite types x 6 applications). However, to improve readability and avoid redundancy, we grouped test cases by testing objective rather than by application. Table 4 provides the number of test cases involved by application.

[3]https://ollama.com/

Table 4: Summary and details of the evaluation test suites.

Test Suite	# test cases	# steps total	# assertions total	# web sites covered
TestG	36	257	42	6
TestA	29	58	29	4
TestG-lax	16	110	17	3
TestA-lax	29	58	29	4

It is worth noting that no formal specification is constructed or used during the empirical evaluation. In practice, we encoded the expected behaviour implicitly in the NL test cases of TestG and TestA, which serve as executable test oracles designed to reflect the intended behaviour of AUT. Therefore, when we previously state that a test case is consistent with a specification S, this must be understood as the assumption meaning that the NL test cases correctly capture intended behaviours of AUT, rather than as a reference to an explicitly available formal specification.

These test suites, the tools along with the web site source codes are available in [38].

4.1 RQ1: To What Extent Do NL Test Cases Executed by LLM Agents Incorrectly Reject Correct AUT Behaviour?

Setup: To address this question, we analyse two complementary aspects. First, we compute the *unsound test ratio*, defined as the proportion of NL test cases that incorrectly reject a conformant AUT at least once over 20 repeated executions per test case. Second, to characterise the severity of unsoundness, we measure the *Failure Exposure Rate (FER)* for each test case, defined as the proportion of executions that yield either a fail or an inconclusive verdict.

Our evaluation relies on two complementary test suites. TestG includes 36 NL test cases, each containing between 4 and 15 actions. TestA consists of 29 NL test cases that do not include navigation actions, allowing us to isolate and evaluate the capability of Algorithm 1 to evaluate assertions independently of the success of executing navigation actions. To remain within the scope of weak unsoundness analysis, we constructed the test cases so that they satisfy the constraints stated in Proposition 1, namely that they are disambiguable under our grammar-based clarity procedure. Test cases that cannot be successfully disambiguated systematically yield an inconclusive verdict with our algorithm.

NL test case execution is performed using NLTestRunner, configured to run its three agents with the same LLM. Each NL test case is executed 20 times for each LLM, and verdicts are recorded for analysis. To evaluate the benefits of Algorithm 1 implemented in NLTestRunner, we repeated the same experimental protocol using SimpleNLTestRunner, a naïve NL test case executor, and compare the resulting observations.

Results:

Unsound test ratio:

Figure 4 reports the measured unsound test ratios per LLM and per tool, distinguishing between ratios computed by considering both fail and inconclusive verdicts and those considering only fail verdicts. With Llama3.3 70B, NLTestRunner exhibits an unsound test ratio of 9.2% (6/65) (fail+inc in the figure) when both fail and inconclusive verdicts are taken into account, and 0% (fail only) when only fail verdicts are considered. This difference is entirely due to inconclusive verdicts, which arise from the guardrails managing timeouts or agent-level errors. Under this configuration, no NL test case incorrectly rejects a conformant AUT with a fail verdict.

For Qwen3 14B, which demonstrated mixed capability, the unsound test ratio increases substantially to 39% when considering fail and inconclusive verdicts, and to 9.4% when considering fail verdicts only. With MistralNemo 12B, the lowest-capability model, the unsound test ratio further increases to 42% and 26%, respectively. Overall, these results indicate progressively higher degrees of unsoundness as model capability decreases, suggesting that NL test execution becomes unreliable when the underlying LLM lacks sufficient capabilities.

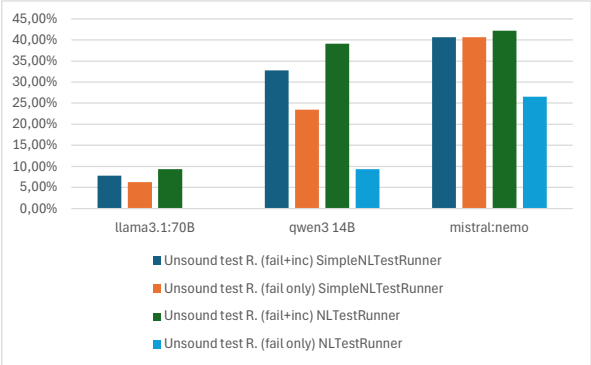
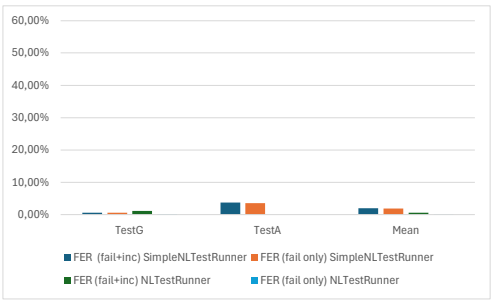
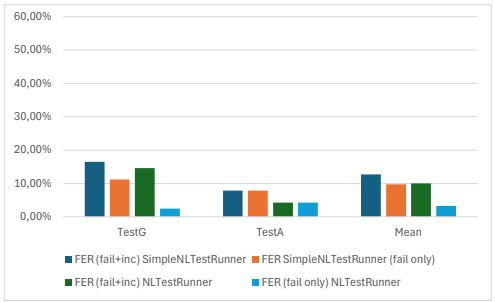


Fig. 4: Unsound test ratio for NLTestRunner

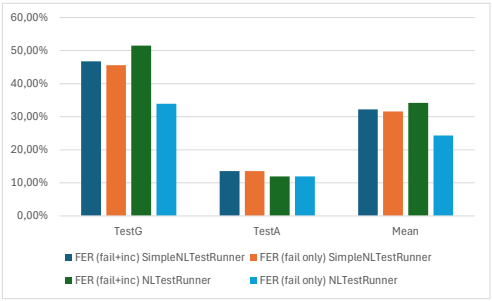
Failure Exposure Rate (FER):
Figures 5a), b), c) now report the FER, which characterises the severity of unsoundness across repeated executions. With Llama3.3 70B and NLTestRunner, the mean FER is extremely low: 0,6% (fail+inc) and 0% (fail only). In practice, this corresponds to approximately one incorrect rejection out of 1280 executions, indicating that NL test cases executed with our algorithm and a high-capability LLM rarely reject a conformant AUT. With Qwen3 14B, the FER increases to 10% (fail+inc) and 3.3% (fail only). The proportion of false fail verdicts increases but remains limited. With Mistral Nemo 12B, the mean FER further increases to 34% (fail+inc) and 24% (fail only). At this level of model capability, the guardrail mechanisms are no longer sufficient to prevent frequent false positives, indicating that our guardrails cannot compensate for an ineffective LLM. Across all models, we also observe consistently lower FER values with the test suite TestA, suggesting that it is easier for LLM agents to correctly evaluate assertions than to reliably perform navigation actions.
Comparison with SimpleNLTestRunner:
When comparing NLTestRunner with the baseline SimpleNLTestRunner, we observe that NLTestRunner yields a slightly higher unsound test ratio (Figure 4). A closer inspection shows that this increase is primarily driven by inconclusive verdicts rather than fail verdicts. This behaviour is a direct consequence of the guardrails introduced by NLTestRunner, which explicitly detect timeouts or agent failures and prevent from returning incorrect fail verdicts. Consistently with this observation, Figures 5a), b), and c) show that the FER (fail only) obtained with NLTestRunner is always strongly lower than that obtained with SimpleNLTestRunner. This result confirms that the guardrails implemented in Algorithm 1 are effective at mitigating false positives, even when the underlying LLM exhibits limited reliability.
Variability of execution of the LLM agents:
Figure 6 presents box-plots of the distributions of FER across the NL test cases, providing a visual assessment of the variability in the execution behaviour of the LLM agents. For Llama3.3 70B, most NL test case executions exhibit a FER close to 0%, indicating highly stable behaviour with very limited variability across executions. In contrast, Qwen3 14B shows FER values primarily concentrated between 0% and 10%, suggesting a moderate level of variability. The highest variability is observed with Mistral Nemo, for which FER values span a



(a) Llama3.3 70B



(b) Qwen3 14B



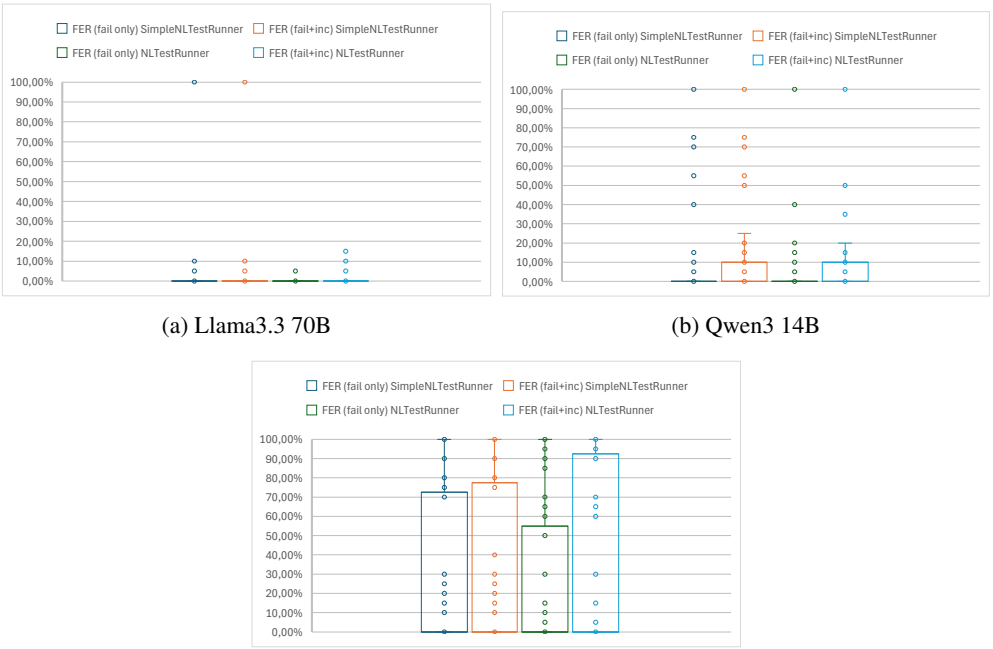
(c) Mistral Nemo 12B

Fig. 5: FER

broad range from 0% to nearly 90%, revealing substantial instability across executions. All three LLMs exhibit several extreme values reaching up to 100% FER. These outliers indicate that certain test cases systematically result in failure or inconclusive verdicts for some executions, highlighting a significant heterogeneity among test scenarios. Moreover, the FER (fail+inc) distributions generally display a larger dispersion than the FER (fail only) distributions. This behaviour is expected, as the inclusion of inconclusive verdicts increases the proportion of unexpected execution outcomes.

Analysis of inconclusive verdicts:

A manual inspection of the inconclusive verdicts revealed that they were predominantly caused by execution-level issues rather than by failures of the grammar-based disambiguation procedure. With Llama3.3 70B, all inconclusive verdicts originated from agent timeouts occurring during navigation actions, due to network instability or model-runner execution errors. For both Qwen3 14B and Mistral Nemo 12B, the frequency of inconclusive verdicts strongly correlates with the capability of the underlying LLM, with lower-capability models exhibiting substantially more timeout failures. With these models, inconclusive verdicts also rarely happened after disambiguation failures during the interpretation of navigation actions. As discussed previously, we observe that NLTestrunner produces more inconclusive verdicts than SimpleNLTest because it incorporates quiescence management and ambiguity checks into the execution process. When these conditions are not satisfied within a predefined timeout, NLTestRunner returns an inconclusive verdict rather than risking an incorrect fail verdict.



(a) Llama3.3 70B

(b) Qwen3 14B

(c) Mistral Nemo 12B

Fig. 6: Box-plots illustrating the FER variability accross 20 NL test case executions

In contrast, SimpleNLTestRunner continues execution thereby increasing the likelihood of false fail verdicts.

Initial conclusions: 1) With a high-capability LLM, NL test cases that respect the constraints of Proposition 1 executed using our algorithm rarely reject a conformant AUT. 2) The guardrails implemented in NLTestRunner effectively reduce false fail verdicts by shifting execution uncertainty toward inconclusive outcomes rather than incorrect failures. 3) The choice of the underlying LLM is critical: insufficient model capability leads to high unsound test ratios and FER, regardless of our guardrails.

4.2 RQ2: How Robust Is NL Test Execution by LLM Agents to Incorrect Expected Behaviours?

Setup:

This question, which addresses laxness, requires the availability of non-conformant implementations. However, beyond practical limitations (e.g., lack of source code or control), evaluating laxness requires faults that correspond to precisely defined deviations from expected behaviour, for which test cases are expected to detect the deviation. This requirement is difficult to satisfy with implementation-level fault injection, where faults may have indirect, delayed, or masked effects, making their GUI-level impact unclear. Our evaluation instead targets fine-grained GUI-level deviations, such as subtle changes in navigation paths, UI element properties (e.g., labels or visibility), and small perturbations in assertions, which

are essential to assess whether LLM agents may accept incorrect behaviours under minimal discrepancies. In contrast, implementation-level faults often lead to coarse-grained failures (e.g., crashes or error pages), which do not align with this objective and may bias the evaluation. Therefore, we adopt a controlled fault-simulation strategy by mutating NL test cases to encode incorrect expected behaviours. From the perspective of test execution, such incorrect expected behaviours are observationally equivalent to faults in AUT: in both cases, the execution must detect and reject behaviour that violates the specification. We considered both original test suites TestA and TestG, and constructed corresponding mutated NL test suites as follows:

- **Navigation action mutations (*TestG-lax*):** We replace one navigation action with an alternative action that is valid on a different GUI of AUT, thereby deviating from the correct execution sequence;
- **Assertion mutations (*TestA-lax*):** We replace one assertion in each test case with an alternative predicate that is either (i) valid on a different GUI but not on the intended execution path, or (ii) a slight variant of the original assertion (e.g., word inversion, altered phrasing, or changes in letter case) that does not hold on AUT.

Similarly to RQ1, to remain within the scope of weak laxness analysis, we constructed the mutated test cases so that they satisfy the constraints stated in Proposition 3. In particular, they are disambiguable; each navigation action is enabled on at most one GUI; and each valid assertion holds on a unique GUI.

Following the methodology of RQ1, we measure the *lax test ratio*, defined as the proportion of mutated NL test cases that yield a pass verdict in at least one out of 20 repeated executions against a conformant AUT. Second, to characterise the severity of laxness, we compute the *Pass Exposure Rate (PER)* for each test case, defined as the proportion of executions that result in a pass verdict. The evaluation is conducted using the same three representative LLMs selected in RQ1. Finally, to assess the benefits of Algorithm 1, we again repeated the same experimental protocol using SimpleNLTestRunner and compare the resulting lax test ratios and PER values with those obtained using NLTestRunner.

Results:

Lax test ratio:

Figure 7 shows that across all evaluated LLMs, NLTestRunner achieves a lax test ratio of 0% on TestG-lax, indicating that incorrect navigation behaviours are consistently detected. This result comes from the use of the readiness action, which prevents the execution of navigation actions on unintended GUIs. In contrast, for TestA-lax, NLTestRunner returns lax test ratios of 4/29, 2/29, and 12/29 when using Llama3.3 70B, Qwen3 14B, and Mistral Nemo 12B, respectively. In particular, Qwen3 14B produces fewer false negatives than Llama3.3 70B on mutated assertions. Manual inspection revealed that Llama3.3 70B tends to accept assertions containing minor textual perturbations (e.g., word inversions or character substitutions), whereas Qwen3 14B more consistently rejects them. Test case execution with Mistral Nemo 12B exhibits the highest degree of laxness.

Pass Exposure Rate (PER):

Using NLTestRunner with Llama3.3 70B, the mean PER is equal to 3.7%, corresponding to 32 false negatives out of 860 executions. Qwen3 14B achieves a lower mean PER of

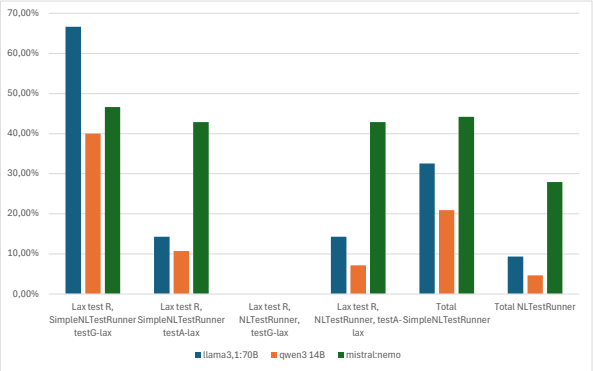


Fig. 7: Lax test ratio

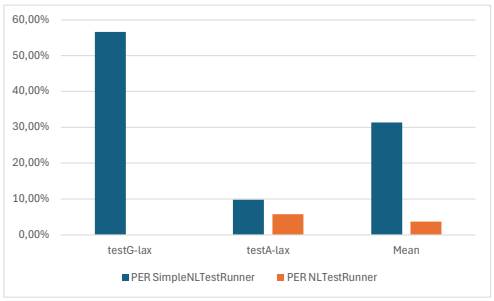
2.3%, suggesting stricter or more reliable assertion evaluation when confronted with incorrect expected behaviours. In contrast, Mistral Nemo 12B exhibits a substantially higher mean PER of 23.7%, reflecting a high frequency of false negatives and confirming its limited suitability for NL test case execution.

Comparison with SimpleNLTestRunner:

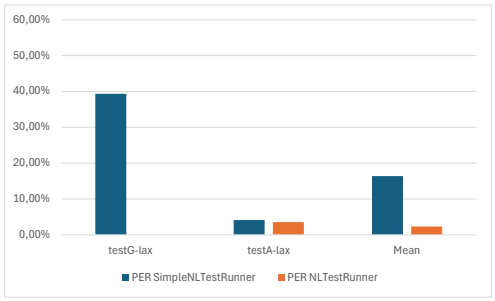
This comparison highlights the impact of the guardrails implemented in NLTestRunner. With TestG-lax, the latter completely eliminates false negatives due to the inclusion of the readiness actions. With SimpleNLTestRunner, all evaluated LLMs exhibit high PER values because the navigation agent ($agent_{nav}$) reports successful execution even when the requested interaction is not feasible. Whatever the LLM used, it attempts to infer an alternative interaction to complete the action, leading to false negatives that are undesirable in a testing context. With TestA-lax, no difference is observed between NLTestRunner and SimpleNLTestRunner. Although NLTestRunner initially applies a strict assertion evaluation, assertions that do not hold are subsequently re-evaluated by an LLM agent. This design choice was motivated by the potential ambiguity of natural language assertions, which may not always be reliably interpreted by the strict evaluation mechanism. However, this fallback reintroduces laxness in the presence of incorrect expected behaviours.

Variability of execution of the LLM agents:

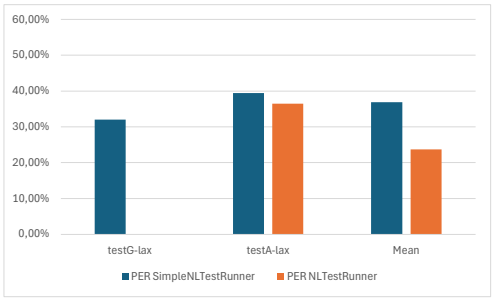
Figure 9 presents box-plots of the PER distributions across the NL test cases, confirming the previous observations regarding PER. In particular, NLTestRunner consistently achieves a lax test ratio close to 0% when using either Llama3.3 70B or Qwen3 14B, indicating highly stable behaviour across executions. By contrast, SimpleNLTestRunner exhibits substantially greater dispersion in PER values, reflecting higher variability in the outcomes of the generated test cases. Consistent with the findings from RQ1, all three LLMs produce several extreme values, indicating that certain NL test cases systematically result in pass verdicts despite the implementation being non-conformant. A manual inspection revealed that these cases predominantly originate from assertions in which only minor lexical variations were introduced, such as replacing lowercase letters with uppercase letters. These observations suggest that $agent_{assert}$ is insufficiently sensitive to such subtle syntactic modifications, whatever the underlying LLM used.



(a) Llama3.3 70B



(b) Qwen3 14B



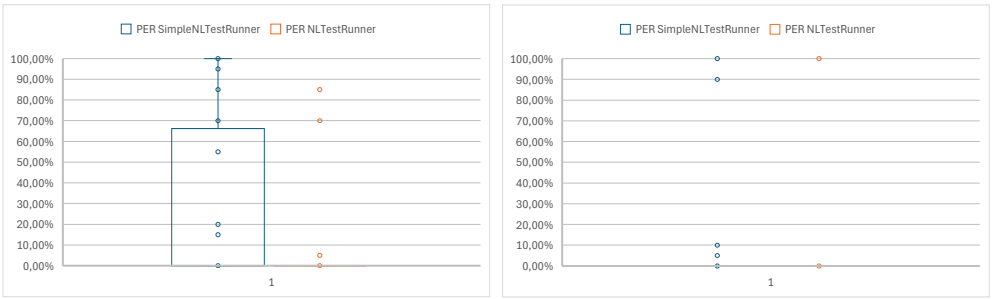
(c) Mistral Nemo 12B

Fig. 8: PER

Initial conclusions: 1) With high-capability LLMs, the ratio of false negatives is low when NL test cases that satisfy the constraints of Proposition 3 are executed with NLTestRunner. 2) The results show that NL test case execution with NLTestRunner is highly robust to incorrect expected navigation when enforced through GUI readiness checks. 3) Robustness to incorrect expected assertions is strongly dependent on LLM capability, with weaker models exhibiting higher rates of false negatives. 4) Our algorithm could benefit from an additional guardrail in the assertion evaluation, relying on the strict assertion mechanism when applicable to reduce laxness in the presence of incorrect expected behaviours.

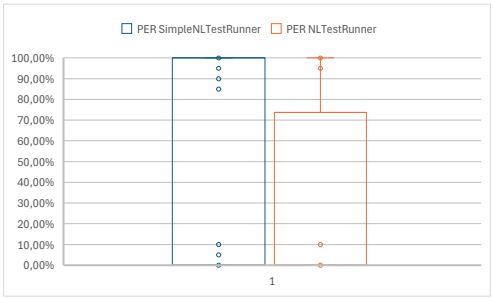
4.3 RQ3: What is the average execution time of NL test cases executed by LLM agents compared to traditional test scripts ?

Setup: We measured the execution times of NL test cases on a local server to better reflect industrial deployment contexts, where LLMs are often hosted locally due to privacy and cost considerations. We evaluated the execution times of the TestG and TestA test suites using the three considered LLMs. To provide a baseline for comparison, we also measured the execution times of 10 test scripts manually implemented with the Playwright framework. This comparison aims to quantify the runtime overhead introduced by LLM requests during NL test execution.



(a) Llama3.3 70B

(b) Qwen3 14B



(c) Mistral Nemo 12B

Fig. 9: Box-plots illustrating the PER variability accross 20 NL test case executions

Table 5: Execution times (in seconds) for the evaluated models.

Models / Execution times (seconds)	TestG	TestA	Average execution time per test case	Average execution time per step
Llama3.3 70B	6921	1646	133	27
Qwen3 14B	12460	7954	318	65
Mistral Nemo 12B	1831	430	35	7

Results: Table 5 reports, for the three evaluated LLMs, the total execution times of the TestG and TestA test suites, as well as the average execution time per test case and per execution step. A Playwright test script requires on average 3.95 seconds for an 8-step scenario, corresponding to approximately 0.5 seconds per step.

Using Llama3.3 70B, the average execution time remains below 27 seconds per step, compared to approximately 0.5 seconds per step for Playwrights. While this represents a significant overhead compared to traditional automation frameworks such as Playwright, this difference is expected since our approach performs runtime NL interpretation, reasoning, GUI observation, and decision-making through LLM agents rather than executing predefined deterministic scripts. Furthermore, NL test cases mainly target high-level functional scenarios and are typically executed less frequently than low-level unit tests, making longer execution times more acceptable in practice.

The lighter Qwen3 14B model exhibits execution times approximately 2.4 times higher than Llama3.3 70B. Manual observations suggest that this behaviour is mainly caused by longer reasoning phases during certain navigation and assertion-evaluation steps, despite the smaller model size.

The Cost of Guardrails:

Our algorithm introduces several guardrails designed to reduce the unpredictable behaviour commonly observed in standard LLM agents. Overall, we observed that these mechanisms introduce only limited overhead compared to the cost of LLM inference. The clarity mechanism first checks whether a navigation action complies with a predefined grammar. This verification is purely rule-based and therefore does not require any LLM invocation. Only ambiguous actions trigger an additional LLM call to rewrite the instruction into an unambiguous navigation action, which may add a few seconds to the execution time. In practice, such situations remain relatively infrequent for well-formed NL test cases. Readiness evaluation introduces negligible overhead, since it is implemented through formula evaluations, implemented with regular expressions. Consequently, the main execution cost of our algorithm remains dominated by the LLM agent calls.

These results should be interpreted with caution. During our experiments, we observed several limitations that make it difficult to derive stable conclusions regarding execution costs. First, our implementation relies on rapidly evolving frameworks such as Ollama and Stagehand, whose performance characteristics significantly improve across versions. Second, our experiments were conducted on a server equipped with an NVIDIA L40 GPU, whose memory capacity is only marginally sufficient to host a 70B model together with its execution context. Consequently, we believe that the reported measurements should primarily be viewed as indicative observations rather than definitive benchmarks, especially given the rapid evolution of LLM infrastructures and instantiation engines. A more comprehensive and long-term study of execution costs, scalability, and optimisation strategies therefore constitutes an important direction for future work.

4.4 Threats To Validity

We now address potential threats to the validity of our evaluation. We organise them into internal and external threats.

Internal threats. The first set of threats concerns factors that may have affected our experimental results. These include limitations in our prototype tools, the design of the prompts and the repeated execution limited to 20 runs.

1) Implementation: The evaluation of LLM agents depends on our tools, which could be improved to better handle response timeouts, capture structured data from GUIs, and verify the correct execution of navigation actions. For navigation, we used the Stagehand framework, which is relatively new and still evolving. Future versions of Stagehand may yield better results, particularly for smaller LLMs. Alternative frameworks could also be explored; for instance, we conducted preliminary experiments with BrowserUse [4], a framework oriented toward general scenario execution, but initial results were not promising. We performed initial experiments to set the configutations of the LLMs (temperature, top-k, etc.). We performed initial calibration experiments to select appropriate LLM configuration parameters (e.g., temperature, top-k sampling). Nevertheless, further model-specific tuning could potentially

[4] https://browser-use.com/

improve the performance of individual agents beyond the results reported here. Additionally, the effectiveness of the guardrails may depend on implementation details, making our results sensitive to the specific choices in the tool design.
2) Prompt design: The prompts used in our experiments may also limit the validity of our conclusions, particularly when applied to other LLMs. Despite following a strict protocol and applying prompts consistently across models, they may still be better suited to some LLMs than to others.
3) Repeated Execution Limitation: Each test case was executed 20 times. While this provides some statistical confidence, it may not fully capture rare execution failures.

External threats. The second set of threats concerns factors that may affect the generalisability of our findings. External threats include the choice of the GUI applications, the test suites built to perform the experiments, the selection of LLMs, the use of mutated test cases instead of non-conformant AUT for RQ2, and test environment variability.
1) Limited GUI application Diversity: The implementations we considered in our experiments consist of 6 different web sites, which are not supposed faulty. To strengthen external validity, further kinds of applications, such as mobile applications, should be considered.
2) Design of the NL Test Suites: The test suites used in the experiments are made up of common interactions but do not cover all cases, such as hovering over or dragging elements. Furthermore, larger or more complex suites may behave differently. The NL test suites used in the evaluation were manually designed by the authors, which may introduce bias. However, their construction was constrained by the assumptions required to study RQ1 and RQ2, including disambiguability and controlled navigation semantics. Test cases that do not satisfy the assumptions formalised in the propositions 1 and 3 cannot be used to rigorously evaluate weak unsoundness or weak laxness. To mitigate potential author bias, we nevertheless designed test suites covering different applications, interaction patterns, navigation complexities, and assertion styles.
3) LLM Selection Bias: We considered 3 LLMs that can be deployed on local servers, but excluded cloud-based LLMs that might provide better results for GUI interaction. Future work could extend our evaluation to additional LLMs to assess the broader applicability of our approach.
4) Use of Mutated NL Test Cases in RQ2: Incorrect expected behaviours were simulated by mutating test cases rather than injecting faults into AUT. This might not fully capture real-world non-conformant behaviours, limiting the realism of the evaluation. To limit the impact of this threat, we designed mutated NL test cases that simulate realistic specification violations, as a non-conformant AUT would.
5) Test Environment Variability: The execution times reported in RQ3 were obtained using locally deployed LLMs on a specific hardware configuration. Different deployment environments (e.g., cloud-hosted LLMs or more powerful GPUs) may exhibit significantly different latency characteristics. Consequently, the reported timings may not generalise to all industrial settings.

5 Conclusion

In this paper, we investigated the feasibility of executing NL test cases for GUI applications using LLM agents, with the objective of considering NL test cases not only as informal documentation but as executable testing artifacts.

To this end, we introduced a novel NL test case execution algorithm that orchestrates three specialised LLM agents responsible for navigation, readiness validation, and assertion evaluation. It also incorporates guardrail mechanisms, including grammar-based disambiguation of test actions and strict rule-based GUI readiness verification, enabling controlled interaction with the application under test.

We provided a foundation for reasoning about NL test case properties in the presence of LLM agents. We revisited and refined the notion of weak unsoundness and introduced weak laxness, together with propositions that characterise the conditions under which NL test case are weakly unsound or weakly lax with respect to the ioco conformance relation.

To assess the effectiveness of our approach, we designed dedicated NL test suites and implemented prototype tools. Our experimental study, conducted using three publicly available LLMs of varying sizes and four test suites targeting six web applications, shows that the proposed algorithm can effectively execute NL test cases when paired with high-capability models. In particular, with Llama3.3 70B, NL test cases rarely reject conformant implementations (FER under 1%), providing empirical support for the weak unsoundness guarantees given in Proposition 1. The guardrails used in our algorithm effectively mitigate false positives by producing inconclusive verdicts when environment issues or agent misbehaviour are detected. Moreover, our results show that NL test cases executed with our algorithm and high-capability LLMs, are weakly lax. Our algorithm is highly robust to incorrect expected navigation and to incorrect expected assertions (PER under 4%) under the conditions on NL test cases given in Proposition 3. In contrast, a simpler NL test case execution algorithm without guardrail or the use of lower-capability models exhibit substantially higher rates of false positive and false negative verdicts. These tools, the test suites along with our detailed experimental results are freely available in [38].

This work opens several directions for future research. First, NL test cases could be automatically derived from higher-level artefacts such as user scenarios or navigation models, thereby further reducing manual effort in test design. The execution algorithm could be extended to better balance between weak unsoundness, weak laxness and efficiency, potentially through adaptive strategies that would adjust guardrail strictness or agent coordination policies. NL test cases could also be designed to assess non-functional properties, such as security or usability, broadening the applicability of NL-driven testing beyond functional conformance. The paper does not quantitatively analyse the runtime and efficiency of calling LLM agents for test case execution. Future work should focus on quantitative analysis of runtime overhead, including per-test execution time, latency introduced by guardrail mechanisms, and scalability in CI/CD environments with large and frequently executed test suites. In this context, future research might explore hybrid agent configurations in which smaller, fine-tuned LLMs specialise in well-defined subtasks (e.g., navigation or readiness verification), while larger models are invoked only for semantically complex reasoning. Dynamically optimising agent composition according to GUI complexity or execution context could improve efficiency. Another promising direction concerns the execution of adaptive NL test cases,

whose execution can dynamically adjust to variations in the observed GUI. Rather than following a linear sequence of actions, such test cases would explicitly incorporate conditional branches based on runtime observations (e.g., "If a confirmation popup appears, close it; otherwise proceed to the dashboard"). Such flexibility would enable the specification of more complex expected behaviours, while improving robustness to UI variations. However, adaptive NL test case execution requires either additional safeguards to ensure that branching conditions hold, or LLM agent evaluations to guarantee that condition evaluation and branch selection do not introduce unsoundness or laxness during test execution.

Acknowledgements. Research supported by the Industrial Chair on Reliable and Confident Use of LLMs (https://uca-fondation.fr/les-chaires/) and MIAI Cluster, France 2030 (ANR-23-IACL-0006)

References

[1] Alégroth, E., Feldt, R.: On the long-term use of visual gui testing in industrial practice: a case study. Empirical Softw. Engg. **22**(6), 2937–2971 (2017) https://doi.org/10.1007/s10664-016-9497-6

[2] Di Meglio, S., Starace, L.L.L., Pontillo, V., Opdebeeck, R., De Roover, C., Di Martino, S.: Investigating the adoption and maintenance of web gui testing: Insights from github repositories. Information and Software Technology **189**, 107928 (2026) https://doi.org/10.1016/j.infsof.2025.107928

[3] Augusto, C., Morán, J., Bertolino, A., Riva, C., Tuya, J.: Software system testing assisted by large language models: An exploratory study. In: Testing Software and Systems: 36th IFIP WG 6.1 International Conference, ICTSS 2024, London, UK, October 30 – November 1, 2024, Proceedings, pp. 239–255. Springer, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-3-031-80889-0_17 . https://doi.org/10.1007/978-3-031-80889-0_17

[4] Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D., Wang, Q.: Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639180 . https://doi.org/10.1145/3597503.3639180

[5] Yoon, J., Feldt, R., Yoo, S.: Intent-Driven Mobile GUI Testing with Autonomous Large Language Model Agents . In: 2024 IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 129–139. IEEE Computer Society, Los Alamitos, CA, USA (2024). https://doi.org/10.1109/ICST60714.2024.00020 . https://doi.ieeecomputersociety.org/10.1109/ICST60714.2024.00020

[6] Tomic, S., Alégroth, E., Isaac, M.: Evaluation of the choice of llm in a multi-agent solution for gui-test generation. In: 2025 IEEE Conference on Software Testing, Verification

and Validation (ICST), pp. 487–497 (2025). https://doi.org/10.1109/ICST62969.2025.10989038

[7] Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Comput. Surv. **51**(4) (2018) https://doi.org/10.1145/3212695

[8] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., Wang, H.: Large language models for software engineering: A systematic literature review. ACM Trans. Softw. Eng. Methodol. **33**(8) (2024) https://doi.org/10.1145/3695988

[9] Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., Tan, S.H.: Automated Repair of Programs from Large Language Models (2023)

[10] Jiang, N., Liu, K., Lutellier, T., Tan, L.: Impact of Code Language Models on Automated Program Repair (2023)

[11] Yasunaga, M., Liang, P.: Graph-based, self-supervised program repair from diagnostic feedback. In: Proceedings of the 37th International Conference on Machine Learning. ICML'20. JMLR.org, online (2020)

[12] Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L., Poshyvanyk, D., Monperrus, M.: Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering **47**(09), 1943–1959 (2021) https://doi.org/10.1109/TSE.2019.2940179

[13] Maniatis, P., Tarlow, D.: Large Sequence Models for Software Development Activities. Google Brain. https://blog.research.google/2023/05/large-sequence-models-for-software.html

[14] Kanade, A., Maniatis, P., Balakrishnan, G., Shi, K.: Learning and evaluating contextual embedding of source code. In: III, H.D., Singh, A. (eds.) Proceedings of the 37th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 119, pp. 5110–5121. PMLR, online (2020). https://proceedings.mlr.press/v119/kanade20a.html

[15] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., Sutton, C.: Program Synthesis with Large Language Models (2021)

[16] Helander, V., Ekedahl, H., Bucaioni, A., Nguyen, T.P.: Programming with chatgpt: How far can we go? Machine Learning with Applications **1**, 1–34 (2024)

[17] Salva, S., Sue, J.: Dynamic mitigation of restful service failures using llms. In: Mecella, M., Rensink, A., Maciaszek, L.A. (eds.) Proceedings of the 20th International Conference on Software Technologies, ICSOFT 2025, June 10-12, 2025, pp. 27–38. SCITEPRESS, Bilbao, Spain (2025). https://doi.org/10.5220/0013460700003964 .

https://doi.org/10.5220/0013460700003964

[18] Endres, M., Fakhoury, S., Chakraborty, S., Lahiri, S.K.: Can large language models transform natural language intent into formal method postconditions? Proc. ACM Softw. Eng. **1**(FSE) (2024) https://doi.org/10.1145/3660791

[19] MA, W., WU, D., SUN, Y., WANG, T., LIU, S., ZHANG, J., XUE, Y., LIU, Y.: Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. In: Proceedings - 2025 IEEE/ACM 47th International Conference on Software Engineering, ICSE 2025, pp. 1742–1754. IEEE Computer Society, United States (2025). https://doi.org/10.1109/ICSE55347.2025.00027 . Publisher Copyright: © 2025 IEEE.; 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025 ; Conference date: 26-04-2025 Through 06-05-2025

[20] Jesse, K., Ahmed, T., Devanbu, P.T., Morgan, E.: Large language models and simple, stupid bugs. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), pp. 563–575 (2023). https://doi.org/10.1109/MSR59073.2023.00082

[21] Schuster, R., Song, C., Tromer, E., Shmatikov, V.: You autocomplete me: Poisoning vulnerabilities in neural code completion. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1559–1575. USENIX Association, online (2021). https://www.usenix.org/conference/usenixsecurity21/presentation/schuster

[22] Wu, Y., Jiang, A.Q., Li, W., Rabe, M.N., Staats, C., Jamnik, M., Szegedy, C.: Autoformalization with large language models. ArXiv **abs/2205.12615** (2022)

[23] Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In: Enea, C., Lal, A. (eds.) Computer Aided Verification, pp. 383–396. Springer, Cham (2023)

[24] Su, W., Wu, X., Zhao, Y.: Nl2acsl : Interactively translating natural language to ansi c specification language with large language models. Knowledge-Based Systems **335**, 115177 (2026) https://doi.org/10.1016/j.knosys.2025.115177

[25] Just, R., Jalali, D., Ernst, M.D.: Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ISSTA 2014, pp. 437–440. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2610384.2628055 . https://doi.org/10.1145/2610384.2628055

[26] Sinha, A., Sutton, S.M., Paradkar, A.: Text2test: Automated inspection of natural language use cases. In: 2010 Third International Conference on Software Testing, Verification and Validation, pp. 155–164 (2010). https://doi.org/10.1109/ICST.2010.19

[27] Ayenew, H., Wagaw, M.: Software test case generation using natural language processing (nlp): A systematic literature review. Artificial Intelligence Evolution **5**(1), 1–10 (2024) https://doi.org/10.37256/aie.5120243220 . [Online; cited 2026-01-07]

[28] Salva, S., Zafimiharisoa, S.R.: Model reverse-engineering of mobile applications with exploration strategies. In: Ninth International Conference on Software Engineering Advances, ICSEA 2014, Nice, France (2014). https://uca.hal.science/hal-02019705

[29] Lu, Y., Yao, B., Gu, H., Huang, J., Wang, J., Li, L., Gesi, J., He, Q., Li, T.J.-J., Wang, D.: Uxagent: An llm agent-based usability testing framework for web design. In: Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '25). ACM, Yokohama, Japan (2025). https://doi.org/10.1145/3706599.3719729

[30] Deng, Y., Xia, C.S., Peng, H., Yang, C., Zhang, L.: Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2023, pp. 423–435. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3597926.3598067 . https://doi.org/10.1145/3597926.3598067

[31] Arruda, F., Barros, F., Sampaio, A.: Automation and consistency analysis of test cases written in natural language: An industrial context. Science of Computer Programming **189**, 102377 (2020) https://doi.org/10.1016/j.scico.2019.102377

[32] Wang, C., Pastore, F., Goknil, A., Briand, L.C., Iqbal, Z.: Umtg: a toolset to automatically generate system test cases from use case specifications. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015, pp. 942–945. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2803187 . https://doi.org/10.1145/2786805.2803187

[33] Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., Blackburn, M.: Nat2testscr: Test case generation from natural language requirements based on scr specifications. Science of Computer Programming **95**, 275–297 (2014) https://doi.org/10.1016/j.scico.2014.06.007 . Special Section: ACM SAC-SVT 2013 + Bytecode 2013

[34] Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., Yin, J.: Chatunitest: A framework for llm-based test generation. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering. FSE 2024, pp. 572–576. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3663529.3663801 . https://doi.org/10.1145/3663529.3663801

[35] Sapozhnikov, A., Olsthoorn, M., Panichella, A., Kovalenko, V., Derakhshanfar, P.: Testspark: Intellij idea's ultimate test generation companion. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, April 14-20, 2024, pp. 30–34. ACM, Lisbon, Portugal (2024). https://doi.org/10.1145/3639478.3640024 . https://doi.org/10.1145/3639478.3640024

[36] Salva, S., Taguelmimt, R.: On the Soundness and Consistency of LLM Agents for Executing Test Cases Written in Natural Language (2025). https://arxiv.org/abs/2509.19136

[37] Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software – Concepts and Tools **17**(3), 103–120 (1996)

[38] Salva, S.: NLTestCaseRunner, A Natural Language Test Case Executor Using LLM Agents. https://github.com/FondationUCA-Chair-LLM/NL-test-case-runnerv2 (2026). https://github.com/FondationUCA-Chair-LLM/NL-test-case-runnerv2

[39] White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C.: A prompt pattern catalog to enhance prompt engineering with chatgpt. In: Proceedings of the 30th Conference on Pattern Languages of Programs. PLoP '23. The Hillside Group, USA (2023)

[40] Dalpiaz, F., van der Schalk, I., Brinkkemper, S., Aydemir, F.B., Lucassen, G.: Detecting terminological ambiguity in user stories: Tool and experimentation. Information and Software Technology **110**, 3–16 (2019) https://doi.org/10.1016/j.infsof.2018.12.007

[41] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22. Curran Associates Inc., Red Hook, NY, USA (2022)

[42] Jard, C., Jéron, T., Morel, P.: In: Ural, H., Probert, R.L., Bochmann, G. (eds.) Verification of Test Suites, pp. 3–18. Springer, Boston, MA (2000). https://doi.org/10.1007/978-0-387-35516-0_1 . https://doi.org/10.1007/978-0-387-35516-0_1

[43] Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014, pp. 643–653. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635920 . https://doi.org/10.1145/2635868.2635920

[44] Uluskan, M., Godfrey, A.B., Joines, J.A.: Integration of six sigma to traditional quality management theory: an empirical study on organisational performance. Total Quality Management & Business Excellence **28**(13-14), 1526–1543 (2017) https://doi.org/10.1080/14783363.2016.1150173 https://doi.org/10.1080/14783363.2016.1150173

Declarations

5.1 Funding

Research supported by the Industrial Chair on Reliable and Confident Use of LLMs (https://uca-fondation.fr/les-chaires/) and MIAI Cluster, France 2030 (ANR-23-IACL-0006)

Appendix A Prompts Used By Our Agents

Navigation Agent Prompt

```
You are an autonomous agent whose role is to perform interactions on
 Web pages with a web browser.

# ROLE AND OBJECTIVE
You will be given an instruction that describes an interaction to be
performed on the web page. Your task is to execute the corresponding
interaction on the page.

# PROCESS
1. Identify the element in the instruction
2. Find the exact corresponding element in the page.
3. Use a function to interact with the page
4. Generate a set of facts that are contained in the output

# Common functions:
click, fill, type, press, or any other playwright locator method

# OUTPUT FORMAT
You must respond with valid JSON strictly using the
following format:
{{ "facts": ["fact 1", "fact 2", "..."], "task_accomplished" :
                "Success|Failed|Unknown" }}
```

Fig. A1: System prompt for the agent $agent_{nav}$

Appendix B NL Test Case Weak Unsoundness

We use the classical following notations of IOLTS:

Definition 6 Let $S =< Q, L \cup \mathcal{T}, \rightarrow, q_0 >$ be an IOLTS and $\mu_i \in L_I \cup L_O \cup \mathcal{T}$, $a_i \in L_I \cup L_O$, $q, q', q_i \in Q$.

- $q \xrightarrow{\mu_1 \dots \mu_n} q' =_{def} \exists q_0, q_n : q = q_0 \xrightarrow{\mu_1} q_1 \dots \xrightarrow{\mu_n} q_n = q'$
- $q \stackrel{\varepsilon}{\Rightarrow} q' =_{def} q = q' \vee q \xrightarrow{\tau_1 \dots \tau_n} q'$
- $q \stackrel{a}{\Rightarrow} q' =_{def} \exists q_1, q_2 : q \stackrel{\varepsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \stackrel{\varepsilon}{\Rightarrow} q'$
- $q \stackrel{a_1 \dots a_n}{\Longrightarrow} q' =_{def} \exists q_0, q_n : q = q_0 \stackrel{a_1}{\Rightarrow} q_1 \dots \stackrel{a_n}{\Rightarrow} q_n = q'$
- $S \stackrel{\sigma}{\Rightarrow} q' =_{def} \exists q' : q_0 \stackrel{\sigma}{\Rightarrow} q'$
- $traces(q) =_{def} \{\sigma \in L^* \mid \exists q' : q \stackrel{\sigma}{\Rightarrow} q'\}$
- $traces_{\mathcal{T}}(q) =_{def} \{\sigma \in (L \cup \mathcal{T})^* \mid \exists q' : q \xrightarrow{\sigma} q'\}$
- q after $\sigma = \{q' \mid q \stackrel{\sigma}{\Rightarrow} q'\}$
- $out(q) =_{def} \{a \in L_O \mid \exists q' \in Q : q \stackrel{a}{\Rightarrow}\}$. $out(Q) =_{def} \bigcup_{q \in Q} out(q)$

Clarity Agent Prompt

```
You are a converter agent tasked with converting an ambiguous test step
into an unambiguous test step list.

#ROLE AND OBJECTIVE
You will be given a test step and a grammar given below.
Your task is to convert this test step into new test steps that must be
compliant with the grammar. Let think step by step and Respond "only"
with the new step or the sequence of steps that can be recognized by
the grammar.

# PROCESS
1. Split the step into one or several substeps that are close to the
commands of the grammar
2. Convert each substep
3. Check if the substeps meet the grammar
4. Return the list of generated steps

#OUTPUT FORMAT
You must respond using this format
'Converted Step(s): step1, step2, etc. '\n

Step: {step}\n
EBNF Grammar:\n
<COMMAND> ::= <OPEN> | <CLICK> | <CHECK> | <UNCHECK> | <SELECT> |
<SCROLL> | <PRESS> | <TYPE> | <FILL> | <ENTER>
<OPEN>    ::= "open" <WS> <QUOTE>
<CLICK>   ::= "click" [ <WS> "on" ] <WS> <QUOTE>
<CHECK>   ::= "check" <WS> <QUOTE>
<UNCHECK> ::= "uncheck" <WS> <QUOTE>
<SELECT>  ::= "select" <WS> <QUOTE> <WS> "on" <WS> <QUOTE>
<SCROLL>  ::= "scroll"
<PRESS>   ::= "press" <WS> <QUOTE>
# type variants:
<TYPE>    ::= "type" <WS> "in" <WS> <QUOTE> <WS> "in" [ <WS> "the" <WS>
 "field" ]
<WS> <QUOTE>
# fill variants:
<FILL>    ::= "fill" [ <WS> "the" <WS> "field" ] <WS> <QUOTE> <WS>
"with" <WS> <QUOTE>
# enter variants:
<ENTER>   ::= "enter" <WS> <QUOTE> <WS> "in" [ <WS> "the" <WS> "field"
] <WS> <QUOTE>
# terminals:
<QUOTE>   ::= "'" <TEXT> "'"
<TEXT>    ::= (any character except a single quote)*
<WS>      ::= (one or more whitespace characters)
```

Fig. A2: System prompt for the agent $agent_{clarity}$

Definition 7 (*ioco* implementation relation) Let AUT and S be two IOLTS. AUT *ioco* $S \Leftrightarrow_{def} \forall \sigma \in traces(S) : out(AUT \text{ after } \sigma) \subseteq out(S \text{ after } \sigma)$

We rewrite Definition 4 with these notations (adapted from the definitions given in [42]):

Assertion Evaluation Prompt

```
You are an evaluation agent tasked to evaluate an Assertion of a test
case.

#ROLE AND OBJECTIVE
You will be given a page content and an assertion. Your task is to
evaluate the assertion on the page content.
The page content is a list of elements formatted as {{id, description,
type'}}
Read the descriptions and the types of the elements carefully
Respond 'True' if the Assertion is True and 'False' otherwise
Let think step by step and return the final response.

# PROCESS
1. Identify all the elements related to the assertion (link, statictext,
 etc.) in the given page content
2. Extract the descriptions and types of the elements
2. Check if some elements meet the assertion
3. Conclude on the result of the assertion based on your observations

#OUTPUT FORMAT
You must respond with valid JSON strictly
 using the following format:
{{ "facts": ["fact 1", "fact 2", "..."], "Verdict": true|false }}
```

Fig. A3: System prompt for the $agent_{assert}$

Definition 8 (Weak Unsoundness) Let $S =< Q^S, L^S \cup \mathcal{T}, \rightarrow^S, q_0^S >$ be a deterministic IOLTS and $tc = a_1 \dots a_k \mathcal{A}_{k+1} \dots \mathcal{A}_l$ be a NL test case such that $\exists ?a_1 ! g_1 \dots ?a_k ! g_k \in traces(S)$ and $\mathcal{A}_j (k+1 \leq j \leq l)$ are true on g_k.

tc is weakly unsound w.r.t. S for *ioco* iff
$\forall AUT, AUT$ ioco $S, \forall c \in \Omega \setminus \Omega_r, \forall \sigma \in traces(S) \cap traces(AUT) \cap traces(tc|AUT), \forall ! o \in L_O^S$ such that $\sigma ! o \in traces(tc|AUT), ! o \in out(AUT \text{ after } \sigma), ! o \in out(S \text{ after } \sigma), tc|AUT \text{ after } \sigma ! o \neq fail$, $tc|AUT$ after $\sigma ! o \neq$
inconclusive $\wedge \exists AUT, AUT$ ioco $S, \exists c \in \Omega_r, \exists \sigma \in traces(S) \cap traces(AUT) \cap traces(tc|AUT), \exists ! o \in L_O^S$ such that $\sigma ! o \in traces(tc|AUT), ! o \in out(AUT \text{ after } \sigma), ! o \in out(S \text{ after } \sigma), tc|AUT \text{ after } \sigma ! o \subseteq \{fail, inclonclusive\}$.

B.1 Proofs of Proposition 1 and 2

Let $\sigma = ?a_1 ! g_1 \dots ?a_i (i > 0) \in traces(S)$ and $! o \in L_O^S$ such that both σ and $! o$ satisfy the condition of Definition 8. The actions *clarity*, $?a_i$ and $\mathcal{A}_j$ can be performed by the agents $agent_{clarity}$, $agent_{nav}$, and $agent_{assert}$, having the following behaviours: A) the agents are not faulty (correct responses, no hallucination, no errors) B) the agents return errors; C) the agents hallucinate (wrong responses).

We now list all the traces and verdicts that can be obtained with $\sigma ! o | AUT$ in A) B) and C):
A)

$traces_{\mathcal{T}}(\sigma ! o | AUT)$ in $\begin{cases} (1)\sigma\theta \text{ (environment issues, inc, unsound)} \\ (2)\sigma ! error \text{ (environment issues, inc, unsound)} \end{cases}$

$tc|AUT$ after $\sigma ! o = inconclusive$

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in $\begin{cases} (3)\sigma!o \text{ clarity} \\ (4)\sigma!o\neg\text{clarity (next } ?a_{i+1} \text{ ambiguous, inc,unsound)} \\ (5)\sigma!o \text{ clarity readiness (next } ?a_{i+1} \text{ can be done)} \\ (6)\sigma!o\mathcal{A}_{k+1}\ldots\mathcal{A}_l, \text{ (as } o = g_k \text{ and } \mathcal{A}_{k+1}\ldots\mathcal{A}_l \quad \text{true on } g_k) \end{cases}$

Hence, $tc|AUT$ after $\sigma!o = inconclusive$ in 1), 2) 4) and $tc|AUT$ after $\sigma!o = pass$ in 5). If tc is not ambiguous (trace 4 not doable) and environment issues happen under rare circumstances $c \in \Omega_r$ (traces 1 and 2 not doable), tc is weakly unsound.

B)

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in $\begin{cases} (1)\sigma\theta \text{ (environment issues, or } agent_{nav}\text{faulty, inc, unsound)} \\ (2)\sigma!error \text{ (environment issues, or } agent_{nav}\text{faulty, inc, unsound)} \end{cases}$

$tc|AUT$ after $\sigma!o = inconclusive$

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in $\begin{cases} (3)\sigma!o \text{ clarity} \\ (4)\sigma!o\neg \text{ clarity (next } ?a_{i+1} \text{ ambiguous inc,unsound)} \\ (5)\sigma!o \text{ clarity readiness (next } ?a_{i+1} \text{ can be done)} \\ (6)\sigma!o!error \text{ (}agent_{assert} \text{ produces an error, inc,unsound)} \\ (7)\sigma!o\mathcal{A}_{k+1}\ldots\mathcal{A}_l, \text{ (use of assert\_strict)} \end{cases}$

Hence, $tc|AUT$ after $\sigma!o = inconclusive$ in 1) 2) 4) and 6). If tc is not ambiguous (trace 4 not doable) and environment issues happen under rare circumstances $c \in \Omega_r$ (traces 1, 2 not doable), and $agent_{nav}$, $agent_{assert}$ are faulty under rare circumstances c (traces 1,2 and 5 not doable), tc is weakly unsound (Proposition 1 holds). If $agent_{nav}$ is faulty under rare circumstances (traces 1, 2 not doable) and assertions are evaluated by $assert\_strict$ only (trace 6 not doable), tc is weakly unsound (Proposition 2 holds).

C)

Firstly, let us assume that $agent_{nav}$ operates always correctly, but both $agent_{clarity}$ and $agent_{assert}$ return incorrect results.

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in $\begin{cases} (1)\sigma\theta \text{ (environment issues, inc, unsound)} \\ (2)\sigma!error \text{ ()environment issues, inc, unsound)} \end{cases}$

$tc|AUT$ after $\sigma!o = inconclusive$

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in

$\begin{cases} (3)\sigma!o\neg\text{clarity next } a_{i+1}\text{ambiguous or modifiedincorrectly, inc, unsound)} \\ (4)\sigma!o \text{ clarity readiness (next } ?a_{i+1} \text{ is notambiguous and can be done)} \\ (5)\sigma!o\neg\mathcal{A}_{k+1} \text{ (}agent_{assert} \text{ hallucinate, fail,unsound)} \\ (6)\sigma!o\mathcal{A}_{k+1}\ldots\mathcal{A}_l \text{ (use of assert\_strict)} \\ (7)\sigma!o\mathcal{A}_{k+1}\ldots\neg\mathcal{A}_{k+j}, \text{ (use of assert\_strict and useof } agent_{assert} \text{ fail, unsound)} \end{cases}$

Hence, $tc|AUT$ after $\sigma!o = inconclusive$ in 1) 2) 4), $tc|AUT$ after $\sigma!o = fail$ in 5) and 7). If tc is not ambiguous (trace 4 not doable) and environment issues happen under rare circumstances $c \in \Omega_r$ (traces 1, 2 not doable), and if we suppose that $agent_{assert}$ behaves incorrectly under rare circumstances c (traces 5, 7 not doable), then tc is weakly unsound (Proposition 1 holds). Instead of the last assumption, if we suppose that $assert\_strict$ is used instead of $agent_{assert}$ (traces 5, 7 not doable) then tc is weakly unsound (Proposition 2 holds).

Now, let us assume that $agent_{nav}$ does not operate correctly, it returns wrong GUIs.

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in $\begin{cases} (1)\sigma\theta \text{ (environment issues, or } agent_{nav}\text{faulty, inc, unsound)} \\ (2)\sigma!error \text{ (environment issues, or } agent_{nav}\text{faulty, inc, unsound)} \end{cases}$

$tc|AUT$ after $\sigma!o = inconclusive$

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in

$\left\{\begin{array}{l}(3)\sigma!o\neg\text{clarity (next } a_{i+1}\text{ambiguous ormodified incorrectly, inc, unsound)}\\(4)\sigma!o\text{ clarity }\neg\text{readiness (next }?a_{i+1}\text{ is notambiguous but cannot be done, fail, unsound)}\\(5)\sigma!o\text{ clarity readiness (next }?a_{i+1}\text{ is notambiguous and can be performed on next incorrect GUI)}\\(6)\sigma!o\neg\mathcal{A}_{k+1}\text{ (use of assert\_strict that evaluates to fail, unsound)}\\(7)\sigma!o\mathcal{A}_{k+1}\ldots\mathcal{A}_l\text{ (wrong assertion evaluation by}agent_{assert})\\(8)\sigma!o\mathcal{A}_{k+1}\ldots\neg\mathcal{A}_{k+j}\text{ wrong assertion evaluationby }agent_{assert}\text{ followed by a correct evaluation of assert\_strict, fail, unsound)}\\(9)\sigma!o\mathcal{A}_{k+1}\ldots\mathcal{A}_l,\text{ simple assertions that evaluate to true even on wrong GUI)}\end{array}\right.$

Hence, $tc|AUT$ after $\sigma!o = inconclusive$ in 1) 2) 3), $tc|AUT$ after $\sigma!o = fail$ in 4), 6) 8).

If tc is disambiguable (trace 3 not doable) and environment issues happen under rare circumstances $c \in \Omega_r$ (traces 1, 2 not doable), and if we suppose that both $agent_{assert}$ and $agent_{nav}$ behave incorrectly under rare circumstances c (traces 1, 2, 4, 6, 8 not doable), then tc is weakly unsound (Propositions 1 and 2 hold).

In summary, Proposition 1 holds if tc is not ambiguous and and if both $agent_{nav}$ and $agent_{assert}$ behave incorrectly under rare circumstances. Proposition 2 holds if tc is not ambiguous and environment issues happen under rare circumstances and if $agent_{nav}$ behave incorrectly under rare circumstances and assertions can be evaluated by $assert\_strict$ only.

Appendix C NL Test Case Weak Laxness

C.1 Proofs of Propositions 3 and 4

Let $\sigma = ?a_1!g_1\ldots?a_i(i>0) \in traces(S)$ and $!o \in L_O^S$ such that both σ and $!o$ satisfy the condition of Definition 5. As previously, we consider : A) the agents are not faulty (correct responses, no hallucination, no errors) B) the agents return errors; C) the agents hallucinate (wrong responses).

We now list all the traces and verdicts that can be obtained with $\sigma!g|AUT$ in A) B) and C):

A) $traces_{\mathcal{T}}(\sigma!o|AUT)$ in $\left\{\begin{array}{l}(1)\theta\text{ environment issues, Inc, lax}\\(2)!error\text{ environment issues, Inc, lax}\end{array}\right.$

$tc|AUT$ after $\sigma!o = inconclusive$

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in

$\left\{\begin{array}{l}(3)\sigma!o\neg\text{clarity next }?a_{i+1}\text{ ambiguous, Inc, lax}\\(4)\sigma!o\text{ clarity readiness next }?a_{i+1}\text{ can be done on incorrect GUI, not fail, lax}\\(5)\sigma!o\text{ clarity }\neg\text{ readiness next }?a_{i+1}\text{ cannot be done on incorrect GUI}\\(6)\sigma!o\neg\mathcal{A}_{k+1}\text{first assertion evaluates to false as expected}\\(7)\sigma!o\mathcal{A}_{k+1}\ldots\neg\mathcal{A}_{k+j},\text{at least one assertion evaluates to false as expected}\\(8)\sigma!o\mathcal{A}_{k+1}\ldots\mathcal{A}_l,\text{ all assertions evaluates to true on incorrect GUI. weak assertions, pass, lax}\\(9)\sigma!o!o_2\text{unexpected new output}\\(10)\sigma!o\text{ clarity }!o_2\text{unexpected new output}\\(11)\sigma!o\text{ clarity readiness}!o_2\text{unexpected new output}\end{array}\right.$

If tc is not lax, we expect $tc|AUT$ after $\sigma!o = fail$. $tc|AUT$ after $\sigma!o = inconclusive$ in 3). $tc|AUT$ after $\sigma!o \neq fail$ in 4), 8). If tc is not ambiguous (trace 3) not doable), environment issues happen under rare circumstances $c \in \Omega_r$ (traces 1,2 under c), the navigation actions can

only be enabled on one GUI (trace 4) not doable), and the assertions $\mathcal{A}_{k+1} \ldots \mathcal{A}_l$ hold only on a unique GUI (trace 8) not doable), then tc is weakly lax. The propositions 3 and 4 hold.

B) $traces_{\mathcal{T}}(\sigma!o|AUT)$ in $\begin{cases} (1)\sigma\theta \text{ environment issues, or agent\_nav faulty, Inc, lax} \\ (2)\sigma!error \text{ environment issues, or agent\_nav faulty, Inc, lax} \end{cases}$

$tc|AUT$ after $\sigma!o = inconclusive$

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in

$\begin{cases} (3)\sigma!o\neg\text{clarity next } ?a_{i+1} \text{ ambiguous, Inc, lax} \\ (4)\sigma!o \text{ clarity readiness next } ?a_{i+1} \text{ can be done on incorrect GUI, not fail lax} \\ (5)\sigma!o \text{ clarity } \neg \text{ readiness next } ?a_{i+1} \text{ cannot be done on incorrect GUI} \\ (6)\sigma!o!error\ agent_{assert} \text{ produces an error, Inc, lax} \\ (7)\sigma!o\theta\ agent_{assert} \text{ produces an error, Inc, lax} \\ (8)\sigma!o\neg\mathcal{A}_{k+1} \text{ use of assert\_strict, first assertion evaluates to false as expected} \\ (9)\sigma!o\mathcal{A}_{k+1} \ldots \neg\mathcal{A}_{k+j}, \text{ use of assert\_strict, at least one assertion evaluates to false as expected} \\ (10)\sigma!o\mathcal{A}_{k+1} \ldots \mathcal{A}_l, \text{ use of assert\_strict, all assertions evaluate to true on incorrect GUI,, weak assertions, pass, lax} \\ (11)\sigma!o!o_2 \text{unexpected new output} \\ (12)\sigma!o \text{ clarity } !o_2 \text{unexpected new output} \\ (13)\sigma!o \text{ clarity readiness} !o_2 \text{unexpected new output} \end{cases}$

If tc is not lax, we expect $tc|AUT$ after $\sigma!o = fail$. $tc|AUT$ after $\sigma!o = inconclusive$ in 1) 2) 3) 6) 7). $tc|AUT$ after $\sigma!o \neq fail$ in 4), 10). Let suppose that tc is not ambiguous (trace 3) not doable), environment issues happen under rare circumstances $c \in \Omega_r$ and $agent_{nav}$ fails under c (traces 1,2 under c), the navigation actions can only be enabled on one GUI (trace 4 not doable)and the assertions $\mathcal{A}_{k+1} \ldots \mathcal{A}_l$ hold only on a unique GUI (trace 10 not doable). If $agent_{assert}$ fails under rare circumstances c (traces 6,7 under c) then tc is weakly lax. Proposition 3 holds. Secondly, f the assertion evaluation is exclusively performed by assert_strict instead of $agent_{assert}$ ((traces 6,7 not doable), then Proposition 4 holds.

C) Firstly, let us assume that $agent_{nav}$ operates always correctly, but both $agent_{clarity}$ and $agent_{assert}$ return incorrect results.

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in $\begin{cases} (1)\sigma\theta \text{ environment issues} \\ (2)\sigma!error \text{ environment issues} \end{cases}$

$tc|AUT$ after $\sigma!o = inconclusive$

$traces_{\mathcal{T}}(\sigma!o|AUT)$ in

$\begin{cases} (3)\sigma!o\neg\text{clarity next } ?a_{i+1} \text{ ambiguous or modified incorrectly, Inc, lax} \\ (4)\sigma!o \text{ clarity readiness next } ?a_{i+1} \text{ can be done on incorrect GUI, not fail, lax} \\ (5)\sigma!o \text{ clarity } \neg \text{ readiness next } ?a_{i+1} \text{ cannot be done on incorrect GUI} \\ (6)\sigma!o\mathcal{A}_{k+1}agent_{assert} \text{ hallucinate, not fail, lax} \\ (7)\sigma!o\mathcal{A}_{k+1} \ldots \mathcal{A}_l, agent_{assert} \text{ hallucinate, pass, lax} \\ (8)\sigma!o\neg\mathcal{A}_{k+1} \text{ use of assert\_strict, first assertion evaluates to false as expected} \\ (9)\sigma!o\mathcal{A}_{k+1} \ldots \neg\mathcal{A}_{k+j}, agent_{assert} \text{ hallucinate and use of assert\_strict} \\ (10)\sigma!o\mathcal{A}_{k+1} \ldots \mathcal{A}_l, \text{ use of assert\_strict, all assertions evaluate to true on incorrect GUI, weak assertions, pass, lax} \\ (11)\sigma!o!o_2 \text{unexpected new output} \\ (12)\sigma!o \text{ clarity } !o_2 \text{unexpected new output} \\ (13)\sigma!o \text{ clarity readiness} !o_2 \text{unexpected new output} \end{cases}$

If tc is not lax, we expect $tc|AUT$ after $\sigma!o = fail.\ tc|AUT$ after $\sigma!o = inconclusive$ in 1) 2) 3). $tc|AUT$
after $\sigma!o \neq fail$ in 4), 6), 7), 10).

Let suppose that tc is not ambiguous (trace 3) not doable), environment issues happen under rare circumstances $c \in \Omega_r$ (traces 1, 2 under c), the navigation actions can only be enabled on one GUI (trace 4 not doable) and the assertions $\mathcal{A}_{k+1} \ldots \mathcal{A}_l$ hold only on a unique GUI (trace 10 not doable). If $agent_{assert}$ fails under rare circumstances c (traces 6,7 under c) then tc is weakly lax. Proposition 3 holds. Secondly, f the assertion evaluation is exclusively performed by assert_strict instead of $agent_{assert}$ ((traces 6,7 not doable), then Proposition 4 holds.

Now, let us assume that $agent_{nav}$ does not operate correctly. If it returns wrong GUIs we obtain the same traces as previously where we considered that AUT returns an incorrect GUI. But, in an extremely rare case, $agent_{nav}$, which does not operate correctly, could also interact with the faulty AUT and returns the correct GUI. In this case, the additional traces would be:

$$traces_{\mathcal{T}}(\sigma!o|AUT) \text{ in } \begin{cases} (14)\sigma!o\neg\mathcal{A}_{k+1}, agent_{assert} \text{ hallucinate} \\ (15)\sigma!o\mathcal{A}_{k+1} \ldots \mathcal{A}_l, \text{ use of assert\_strict, pass, lax} \\ (16)\sigma!o\mathcal{A}_{k+1} \ldots \neg\mathcal{A}_{k+j}, \text{ use of assert\_strict and use of } agent_{assert} \end{cases}$$

If tc is not lax, we still expect $tc|AUT$ after $\sigma!o = fail.\ tc|AUT$ after $\sigma!o = inconclusive$ in 1) 2) 3). $tc|AUT$
after $\sigma!o \neq fail$ in 4), 6), 7), 10), 15). As previously, if $agent_{nav}$ fails under rare circumstances c, $agent_{assert}$ fails under rare c (traces 6,7 under c) then tc is weakly lax. Proposition 3 holds. Secondly, if the assertion evaluation is exclusively performed by assert_strict instead of $agent_{assert}$ ((traces 6,7 not doable) and if $agent_{nav}$ fails under c, then Proposition 4 holds.