# Learning Correct Behavior from Examples: Validating Sequential Execution in Autonomous Agents

Reshabh K Sharma[♠1], Gaurav Mittal[♡], Yu Hu[♡]
[♠]University of Washington, Seattle, WA, USA [♡]Microsoft, Redmond, WA, USA
reshabh@cs.washington.edu, Gaurav.Mittal@microsoft.com, yuhu@microsoft.com



As autonomous agents become increasingly sophisticated, validating their sequential behavior presents a significant challenge. Traditional testing approaches require manual specification, exact sequence matching, or thousands of training examples. We present a novel algorithm that automatically learns correct behavior from just 2–10 passing execution traces and validates new executions against this learned model. Our approach combines dominator analysis from compiler theory with multimodal large language model-powered semantic understanding to identify essential states and handle non-deterministic behavior. The system constructs a generalized ground truth model using Prefix Tree Acceptors, merges traces through multi-tiered equivalence detection, and validates new executions via topological subsequence matching. In controlled experiments, our system achieved high accuracy in detecting product bugs and false successes using only 3 training traces. This approach provides explainable validation results with coverage metrics and works across diverse domains including UI testing, code generation, and robotic processes.

## 1 Introduction

As AI agents become more sophisticated, from computer use agents that interact with user interfaces Hu et al. (2024) to coding agents that generate and refactor software Jimenez et al. (2024), an important question emerges. How do we know if an agent's behavior is correct? This challenge is particularly acute because autonomous systems rarely follow the exact same sequence of states and actions between executions. Loading screens may appear or disappear based on timing, alternative UI paths can accomplish the same goal, and different but equally valid code solutions can solve the same problem.

Traditional testing approaches exhibit limitations in handling these scenarios. Assertion-based testing requires manually writing assertions for every check, validates internal data but misses visual state issues, and cannot handle alternative execution paths Pezzè and Young (2008). Record-and-replay tools are brittle, failing on minor rendering differences or timing variations Hammoudi (2016). Visual regression testing compares individual screenshots in isolation without understanding execution flow or semantic meaning vit; spa. Machine learning oracles require thousands of training examples and provide no explainability Fontes and Gay (2021); Braga et al. (2018).

The core problem is *non-determinism* Weyuker (1982). Autonomous systems exhibit variations in execution sequences due to timing differences, environmental factors, and legitimate alternative paths. Consider testing a computer use agent that needs to open VS Code and search for text in files. One execution might proceed by opening the Start Menu, typing "VS Code," launching the application, displaying a loading screen, opening the search dialog, and finally showing results. Another execution might open the Start Menu, type "VS Code," launch the application, and then proceed directly to the search dialog and results without any loading screen. The loading screen appears or disappears based on timing, but both executions are correct.

Traditional testing would either fail the second execution, being too brittle, or miss when truly essential steps are skipped, being too permissive. Consequently, there is a need for a system that can distinguish between acceptable variations and actual failures.

[1]Work done while at Microsoft.

### 1.1 Our Contribution

We present a novel algorithm that automatically constructs a generalized ground truth model from a small number of passing execution traces and validates new executions by checking if they follow the essential structure of this model. Our key contributions are as follows.

- A three-phase algorithm combining Prefix Tree Acceptors Angluin (1987) with dominator analysis Lengauer and Tarjan (1979) to automatically identify essential versus optional states from 2–10 example traces
- A multi-tiered state equivalence detection system that combines visual metrics with semantic LLM analysis to handle non-deterministic behavior
- Topological subsequence matching that validates execution correctness while tolerating acceptable variations

Our approach is broadly applicable to any domain where sequential state transitions need validation, including UI testing, coding agents, robotic automation, and business process validation.

## 2 Motivating Example

Consider a computer use agent that automates the task of opening VS Code and searching for specific text across files. We want to validate that the agent correctly completes this task across multiple executions.

### 2.1 The Challenge

Traditional testing approaches would struggle with this scenario. Exact sequence matching would fail because the loading screen may or may not appear depending on system performance. Manual assertions would require specifying every possible valid path, which is impractical. Pixel-perfect screenshot comparison would fail on minor variations in window decorations or font rendering.

### 2.2 Our Approach

We collect 3–5 passing execution traces where the agent successfully completes the task. Each trace captures sequential screenshots showing the UI state at each step along with actions taken between states such as clicks and keystrokes.

For example, traces might show different paths. In Trace 1, the agent opens the Start Menu, types "VS Code," launches the application, sees a loading screen, reaches the main window, opens the search dialog, and displays results. In Trace 2, the agent follows the same sequence but proceeds directly from launch to the main window without any loading screen. Trace 3 resembles Trace 1, again showing the loading screen before the main window.

Our system automatically performs four key operations. First, it merges the traces into a unified graph structure with branches for alternative paths. Second, it identifies that the loading screen is optional because it appears in some traces but not others. Third, it extracts the dominator tree showing essential states: Start Menu, Launch, Main Window, Search Dialog, and Results. Fourth, it validates new executions by checking if they contain these essential states in order.

When a new test execution arrives, the system checks whether it follows the essential structure. If the agent skips directly from Launch to Search Dialog without ever showing the Main Window, this would be flagged as a failure because Main Window is an essential state. However, if the Loading Screen is missing, this is acceptable because it has been identified as optional.

This example illustrates the core challenge our work addresses. We aim to automatically learn which states are essential versus optional and validate new executions against this learned model without manual specification.

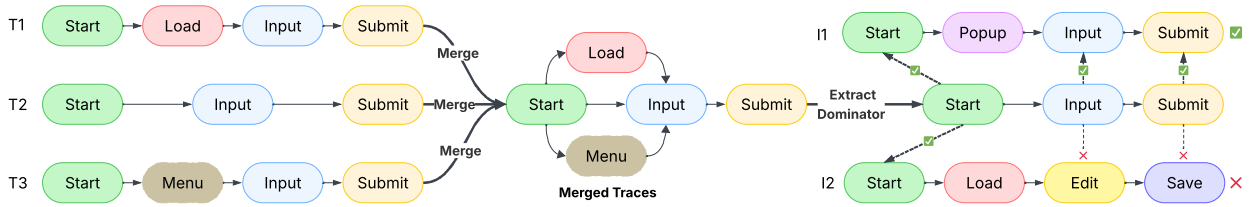


Figure 1: We illustrate the workings of our algorithm using three passing traces, T1, T2, and T3 (represented as PTAs), which are merged into a graph from which a dominator tree is extracted. The incoming traces I1 and I2 are then checked for a topological match with the dominator tree.

## 3 Design

Our algorithm consists of three phases as shown in Figure 1. The first phase captures execution traces and constructs Prefix Tree Acceptors. The second phase merges traces and extracts dominator structure. The third phase validates new executions via topological subsequence matching.

### 3.1 Phase 1: Capture Execution Traces

We begin by collecting 2–10 execution traces from runs that are known to be correct. Each trace contains sequential state observations such as screenshots for UI agents, code snapshots for coding agents, or sensor readings for robotic systems. Additionally, each trace records actions or transitions including clicks, keystrokes, API calls, or other operations between states.

We convert each trace into a *Prefix Tree Acceptor* (PTA) Angluin (1987), a directed graph where nodes represent observable states and edges represent actions. The state representation is flexible and domain-dependent, making the approach applicable across diverse domains. For this work, we focus on UI testing where states are captured as screenshots.

### 3.2 Phase 2: Merge and Generalize

Our primary contribution in this phase is merging multiple PTAs to handle non-determinism and identify essential versus optional states.

#### 3.2.1 Multi-Tiered State Equivalence Detection

Determining when different screenshots represent the same logical UI state is challenging. We employ a three-tiered system.

**Tier 1** involves visual metrics. We compute fast perceptual comparisons including perceptual hash similarity Zauner (2010) to measure visual similarity, structural similarity index (SSIM) Wang et al. (2004) to determine structural alignment, and pixel change ratio to quantify differences. If all metrics clearly indicate equivalence by exceeding predefined thresholds, we merge the states.

**Tier 2** performs semantic analysis via LLM. When visual metrics are ambiguous, we invoke a multimodal large language model such as GPT-5.1 with a side-by-side comparison. We ask the model to analyze whether differences are semantically meaningful. Differences that are not meaningful include different window decorations, minor font rendering differences, and timestamp changes. Meaningful differences include different form validation errors, different data displayed, and different UI controls available. This semantic layer enables the system to understand functional equivalence beyond pixel-level comparison.

**Tier 3** handles smart merging with branches and convergence. As we merge PTAs, we build a graph structure that captures branches representing alternative execution paths such as those with or without loading screens. The structure also identifies convergence points where different paths rejoin, such as when all paths reach a "save complete" state.

#### 3.2.2 Dominator Extraction

After merging, we compute which states "dominate" others Lengauer and Tarjan (1979). A state $d$ dominates state $s$ if every path from the initial state to $s$ must pass through $d$. For example, the initial state dominates everything because every execution starts there. A "save" action dominates the "complete" state because completion cannot occur without saving. Loading screens do not dominate anything because they are optional.

We extract a dominator subtree containing only essential states through a three-step process. We start with all terminal states representing successful endpoints. We then trace backward through immediate dominators to the initial state. Finally, we build a tree structure capturing the "must-have" execution flow. This automatically filters out optional variations while preserving the essential execution structure.

### 3.3 Phase 3: Validate New Executions

When a new test trace arrives, we extract its sequence of states and check whether it contains a topological subsequence that matches a path in the dominator tree.

Topological subsequence matching works as follows. If reference states are $A \to B \to C \to D$ and the test trace is $A \to X \to B \to Y \to Z \to C \to D$, this is a MATCH because $A, B, C, D$ appear in correct order with $X, Y, Z$ as allowed extras.

We compute coverage metrics using the following formula.

$$\text{coverage} = \frac{\text{matched states}}{\text{total reference states}} \times 100\% \tag{1}$$

The result is a **PASS** if coverage meets or exceeds the threshold (typically 100%) and the terminal state matches. The result is a **FAIL** if essential states are missing or the wrong final state is reached. The system provides explainable results including coverage percentage, matched states, missing states, and a detailed explanation of the validation outcome.

## 4 Algorithm

This section presents the complete algorithm for learning correct behavior from execution traces and validating new executions. The algorithm consists of two main sub-algorithms: (1) extracting the dominator tree from passing traces (Algorithm 1), and (2) evaluating test executions against the extracted dominator tree (Algorithm 2).

### 4.1 Complexity Analysis

The time complexity of the algorithm is dominated by the merging phase. For $n$ traces with average length $k$, PTA construction runs in $O(n \cdot k)$ time, which is linear in trace length. State equivalence checking requires $O(n^2 \cdot k^2)$ time in the worst case, but typically performs much better with early termination in Tier 1. Dominator extraction runs in $O(|V_G| + |E_G|)$ time using standard dominator tree algorithms Lengauer and Tarjan (1979). Validation requires $O(m \cdot n)$ time where $m$ is the test trace length and $n$ is the number of reference states.

The space complexity is $O(n \cdot k)$ for storing the merged graph and dominator tree.

## 5 Case Study: VS Code Extension Bug Detection

While a large-scale empirical study is reserved for future work, we designed a targeted, controlled case study to validate the core mechanics of our algorithm. This preliminary evaluation uses a controlled synthetic benchmark comprising 28 agent executions that mimic real-world testing conditions, allowing precise measurement of detection accuracy across failure categories.

**Algorithm 1** Extract Dominator Tree

**Require:** $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$: set of $n$ passing execution traces
**Ensure:** Dominator tree $D = (V_D, E_D)$ representing essential execution flow
**function** EXTRACTDOMINATORTREE($\mathcal{T}$)
{Step 1: Construct Prefix Tree Acceptors for each trace}
$PTAs \leftarrow \emptyset$
**for all** trace $T_i \in \mathcal{T}$ **do**
  $PTA_i \leftarrow$ CONSTRUCTPTA($T_i$)
  $PTAs \leftarrow PTAs \cup \{PTA_i\}$
**end for**
{Step 2: Merge all PTAs into unified graph using multi-tiered state equivalence}
$G \leftarrow$ MERGEPTAS($PTAs$)
{Step 3: Compute dominator relationships using iterative dataflow analysis}
$dom \leftarrow$ COMPUTEDOMINATORS($G$)
{Step 4: Extract essential states by tracing back from terminal states}
$s_0 \leftarrow$ initial state of $G$
$T \leftarrow$ set of terminal states in $G$
$essential\_states \leftarrow \{s_0\}$
$V_D \leftarrow \{s_0\}$
$E_D \leftarrow \emptyset$
**for all** terminal state $t \in T$ **do**
  $current \leftarrow t$
  **while** $current \neq s_0$ **do**
    $idom \leftarrow$ immediate dominator of $current$ in $dom$
    **if** $idom \notin essential\_states$ **then**
      $essential\_states \leftarrow essential\_states \cup \{idom\}$
      $V_D \leftarrow V_D \cup \{idom\}$
    **end if**
    $E_D \leftarrow E_D \cup \{(idom, current)\}$
    $current \leftarrow idom$
  **end while**
**end for**
{Step 5: Build and return dominator tree}
$D \leftarrow (V_D, E_D)$
**return** $D$

### 5.1 Experimental Setup

We created a synthetic bug test using a custom VS Code extension. The passing configuration consisted of one VM with the custom extension installed, producing correct execution traces. The failing configuration used another VM without the extension installed, simulating a product bug scenario. For model training, we used 3 passing traces to build the dominator tree that serves as the trace model.

We compare our approach against a baseline where the Computer Use Agent (CUA) self-reports its own success or failure. This baseline represents the common practice of relying on an agent's internal assessment rather than independent structural validation.

### 5.2 Research Questions

We designed our evaluation to answer the following research questions.

**RQ1: Can the system accurately detect different types of failures?** We evaluate the system's ability to distinguish between passing traces, false successes, agent issues, product bugs, and missed bugs.

**RQ2: How does structural validation compare to agent self-assessment?** We compare the accuracy, precision, recall, and F1-score of our dominator tree approach against the CUA's internal success reporting.

**RQ3: Can the system identify "not a bug" scenarios?** Beyond detecting failures, we investigate whether the

**Algorithm 2** Validate Execution

**Require:** Test trace $T_{test}$, dominator tree $D$, coverage threshold $\theta$
**Ensure:** Validation result: PASS/FAIL with coverage metrics and explanation
**function** VALIDATEEXECUTION($T_{test}$, $D$, $\theta$)
{Extract states from test trace}
$S_{test} \leftarrow \langle s_1^{test}, s_2^{test}, \ldots, s_m^{test} \rangle$ from $T_{test}$
{Get reference states from dominator tree in topological order}
$S_{ref} \leftarrow$ TOPOLOGICALORDER($D$)
{Perform topological subsequence matching using state equivalence}
$(matched, missing) \leftarrow$ TOPOLOGICALSUBSEQUENCEMATCH($S_{test}$, $S_{ref}$)
{Compute coverage metrics}
$coverage \leftarrow |matched|/|S_{ref}| \times 100\%$
$terminal_{ref} \leftarrow$ terminal state in $D$
$terminal_{test} \leftarrow$ last state in $S_{test}$
$terminal\_match \leftarrow$ STATESEQUIVALENT($terminal_{test}$, $terminal_{ref}$)
{Make validation decision}
**if** $coverage \geq \theta$ **and** $terminal\_match$ **then**
  **return** (PASS, $coverage$, $matched$, $\emptyset$)
**else**
  $explanation \leftarrow$ "Missing essential states: " + $missing$ + ". Coverage: " + $coverage$
  **return** (FAIL, $coverage$, $matched$, $missing$, $explanation$)
**end if**

system can correctly identify when a test failure is due to agent execution error rather than an actual product regression.

### 5.3 Results

Using the dominator tree built from 3 passing traces, we evaluated the system on the remaining 25 traces comprising 14 failing traces (3 agent issues and 11 product bugs) and 11 passing traces. Additionally, we assessed the system's ability to catch CUA misclassifications: cases where the CUA incorrectly reported success on a failing trace (false success) or incorrectly reported failure on a passing trace (missed bug). The system achieved 100% detection accuracy across all categories: false successes (1/1), agent issues (3/3), product bugs (11/11), and missed bugs (1/1). Table 1 compares our approach against CUA self-assessment, and Table 2 shows classification accuracy when distinguishing failure root causes.

| Metric | CUA | Ours |
|---|---|---|
| Accuracy | 82.2% | 100% (+17.8pp) |
| Precision | 83.3% | 100% (+16.7pp) |
| Recall | 60.0% | 100% (+40.0pp) |
| F1-Score | 69.8% | 100% (+30.2pp) |

Table 1: CUA self-assessment vs. dominator tree validation

| Trace Type | Accuracy |
|---|---|
| Agent Issue | 33.3% (1/3) |
| Product Bug | 72.7% (8/11) |

Table 2: Classification accuracy for failure root causes

**RQ1 Results:** The system achieved 100% accuracy on our synthetic product bug subset (11/11) and 100% accuracy across all other categories. This demonstrates that the approach can effectively learn correct behavior from just 3 passing traces and use this model to validate new executions.

**RQ2 Results:** Our structural validation approach substantially outperforms CUA self-assessment across all metrics (Table 1). The CUA frequently misreported failures as successes, often due to timing out or misinterpreting its own state, achieving only 82.2% accuracy and 60.0% recall. In contrast, the dominator tree achieved perfect differentiation by focusing on whether essential milestones were actually reached rather than relying on the agent's internal assessment.

**RQ3 Results:** The most significant impact for developers is in reducing false alarms. When a test fails,

high-signal feedback is needed to determine if the product code is broken or if the agent simply stumbled due to environmental noise. The CUA's internal self-assessment was completely unable to identify "not a bug" scenarios (0% F1-score), demonstrating that agents cannot yet reliably assess their own performance in non-deterministic environments. By using state and action equivalence within the dominator model, our approach achieved a 52.2% F1-score in correctly identifying when a failure was an agent execution error rather than a product regression. The system correctly identified 1 of 3 agent issues and 8 of 11 product bugs (Table 2). While there is room for improvement, this capability significantly reduces manual review time wasted on flaky test results and false positives in CI pipelines.

### 5.4 Threats to Validity

Our evaluation uses a synthetic bug scenario with a controlled VS Code extension, which allows precise measurement but may not capture the complexity of real-world failure patterns; additionally, the small sample sizes (e.g., 1 false success, 1 missed bug) limit statistical confidence for some categories. Results from UI testing with computer use agents may not generalize to all domains, and the reliance on visual state representation assumes screenshot-based observation, meaning domains with non-visual state such as backend services would require alternative representations. Finally, our accuracy metric treats all failure types equally, though in practice some failures such as missed bugs may be more costly than others, and the distinction between agent issues and product bugs depends on ground truth labeling that may be ambiguous in edge cases.

## 6 Discussion

The strong performance on product bug detection makes this approach particularly valuable for regression testing Yoo and Harman (2012) and continuous integration pipelines Hilton et al. (2016), where quickly identifying genuine failures is important. The ability to learn from just 3 passing traces significantly reduces setup cost compared to traditional machine learning approaches that require hundreds or thousands of examples. The system's detection of all false successes and missed bugs in our test set demonstrates its ability to catch subtle failures that might pass traditional assertion-based tests—a capability particularly relevant for validating autonomous agents where the execution path matters as much as the final outcome.

Our approach provides several key advantages over existing validation techniques. Traditional assertion-based testing requires developers to manually write assertions for every check and cannot handle alternative execution paths; our approach eliminates this manual effort by automatically learning from examples and naturally accommodates multiple valid paths through merged graph structures. Machine learning-based test oracles require thousands of training examples and operate as black boxes, while our approach achieves effective validation with just 2–10 traces and produces an interpretable dominator tree with explainable results. Record-and-replay tools fail on minor rendering differences and require exact matches; our multi-tiered equivalence detection combines visual metrics with LLM semantic analysis to tolerate acceptable variations while generalizing from multiple traces. Finally, symbolic execution and formal verification methods require source code access and suffer from path explosion; our black-box approach learns from observed executions and uses dominator extraction to reduce state space complexity.

While our evaluation focuses on UI testing with computer use agents, the approach applies broadly to other domains. For coding agents Jimenez et al. (2024), the system can learn essential development patterns (e.g., creating tests, implementing features, verifying tests pass) and validate that AI-generated solutions follow these checkpoints even when implementation details vary. For robotic process automation Van der Aalst et al. (2018), the dominator tree captures mandatory compliance checkpoints and approval gates that must occur regardless of conditional branching. For reinforcement learning from demonstrations Argall et al. (2009), the approach provides quality assurance for training data by filtering demonstration traces that skip essential waypoints.

The integration of multimodal LLM-based semantic analysis addresses ambiguous cases where visual metrics alone are insufficient. This semantic layer distinguishes functionally irrelevant differences (e.g., window decorations) from meaningful ones (e.g., error messages). As LLM capabilities continue to improve,

equivalence detection accuracy should increase accordingly.

## 7 Limitations and Future Work

The system requires passing traces and cannot learn from failures alone, though obtaining a few successful runs is typically straightforward in practice. The current implementation relies on visual state representation, making it ideal for UI testing but less effective for pure backend services; non-visual domains would require alternative state representations such as API responses or database states. Semantic equivalence checking introduces an LLM dependency with associated API costs, though visual metrics alone provide a fallback. The current implementation also does not model timing information or temporal constraints.

Several directions could extend this work. Temporal constraint modeling would enable validation of performance-critical behavior by learning acceptable time bounds for operations. Learning from negative examples could improve discrimination by identifying divergence points where failures occur. Hierarchical state abstraction could cluster low-level states into high-level concepts (e.g., abstracting multiple launch screenshots into a single "application launch" state). Multi-modal state representation combining screenshots with DOM structure, accessibility trees, or network traffic could improve equivalence detection. Online learning that updates the dominator tree from new validated traces would enable continuous model refinement.

## 8 Related Work

This research bridges classical state-machine inference and modern AI validation. We contextualize our dominator-based approach against three relevant domains: software testing, automata learning, and compiler theory. While these fields provide foundational techniques for structural analysis and behavioral modeling, we highlight their limitations in handling the non-determinism inherent in autonomous agent execution.

### 8.1 Software Testing and Validation

Traditional software testing approaches include unit testing with assertions, integration testing, and end-to-end testing Pezzè and Young (2008). These methods require manual specification of expected behavior and struggle with non-deterministic systems Weyuker (1982). Our work complements these approaches by providing automatic learning of expected behavior from examples.

Record-and-replay testing tools capture user interactions and play them back to detect regressions Hammoudi (2016). However, these tools are brittle and fail on minor variations. Our approach generalizes from multiple traces and uses semantic equivalence to tolerate acceptable variations.

Visual regression testing tools compare screenshots to detect UI changes spa; vit. Unlike these tools that compare states in isolation, our approach validates execution flow and understands the sequential dependencies between states.

### 8.2 Machine Learning for Testing

Machine learning-based test oracles (e.g., neural network classifiers Fontes and Gay (2021); Braga et al. (2018); Aggarwal et al. (2004)) learn pass/fail classification from large datasets but require extensive training data and provide no explainability. Our work achieves effective validation with minimal examples and produces interpretable structural models. Metamorphic testing Segura et al. (2016); Chen et al. (2018) generates test cases using system properties without requiring a test oracle, but demands domain-specific metamorphic relations; our approach learns automatically from examples.

### 8.3 Formal Methods and Model-Based Testing

Symbolic execution and model checking enumerate execution paths and verify properties Cadar and Sen (2013); Clarke (1997). These approaches suffer from path explosion and require source code access. Our black-box approach learns from observed behavior and uses dominator analysis to manage state space complexity.

Model-based testing requires manual construction of state machine models Utting and Legeard (2010); Utting et al. (2012). Our work automates model construction from execution traces, eliminating the need for manual modeling expertise.

### 8.4 Automata Learning

Automata learning techniques for inferring state machine models fall into two categories, each with distinct limitations that our dominator-based approach addresses.

**Active learning approaches** such as L* Angluin (1987) and EFSM inference Walkinshaw et al. (2016) require extensive system interaction through membership and equivalence queries. This assumption breaks down for autonomous agents where querying is expensive, slow, or impractical—for instance, a computer-use agent interacting with production UIs cannot be queried thousands of times to infer a model. Our approach instead learns passively from 2–10 observed traces, requiring no system interaction.

**Passive learning approaches** construct models from traces without interaction but face different challenges. SAT-based inference Avellaneda and Petrenko (2018) builds FSMs iteratively from trace subsets, while invariant-enhanced mining Krka et al. (2014) augments traces with program invariants. Both produce complete state machines that grow with trace complexity and do not distinguish essential states from optional variations. In contrast, our dominator extraction automatically identifies which states are structurally necessary—states that every successful execution must visit—versus optional states like loading screens that may or may not appear. This distinction, adapted from compiler control-flow analysis Dietl and Müller (2007), enables validation that tolerates acceptable non-determinism while enforcing essential behavior.

### 8.5 Compiler Theory and Program Analysis

Dominator analysis is a well-established technique in compiler optimization and program analysis Lengauer and Tarjan (1979); Cooper et al. (2001). We adapt this technique to the domain of execution trace validation, using it to identify essential states in behavioral sequences rather than control flow in programs.

### 8.6 Autonomous Agents and AI Testing

As autonomous agents become more prevalent, validating their behavior becomes increasingly important. Recent work on testing AI systems focuses on adversarial testing and robustness evaluation Goodfellow et al. (2015); Carlini and Wagner (2017). Our work addresses the complementary problem of validating correct sequential behavior in non-adversarial settings.

Several recent approaches address agent validation through specification-based checking. ContextCov Sharma (2026) synthesizes executable checks from natural language agent instructions (e.g., AGENTS.md files) to enforce coding conventions and architectural constraints; however, this approach requires explicit natural language specifications, which do not exist for the visual sequential behaviors we target. AgentPex Sharma et al. (2026) extracts behavioral rules from agent prompts and evaluates chat agent traces for compliance, demonstrating that agents frequently violate their own instructions—a finding that strengthens the case for automated validation approaches like ours. AgentRx Barke et al. (2026) contributes a cross-domain failure taxonomy and diagnostic framework for tool-using agents, synthesizing constraints from tool schemas and policies to localize the first unrecoverable failure; however, AgentRx relies on manually annotated failure benchmarks and domain-specific tool schemas, whereas our approach learns validation models automatically from a small number of passing traces without manual annotation.

## 9 Conclusion

The proliferation of autonomous agents Wang et al. (2024) demands validation techniques that can accommodate non-deterministic execution without sacrificing rigorous correctness. This paper demonstrates that reliable behavioral validation does not require massive training corpora, brittle exact-match specifications, or manual assertions.

We introduced a novel automated validation framework that extracts generalized ground-truth models from just 2–10 passing execution traces. By integrating multimodal LLM semantic equivalence with dominator analysis Lengauer and Tarjan (1979); Cooper et al. (2001), our system successfully isolates essential execution states from acceptable, non-deterministic variations. Our preliminary evaluation demonstrates the efficacy of this approach: using a model built from only three passing traces, the system achieved 100% accuracy in detecting product bugs and successfully identified false successes while tolerating valid UI variations.

Furthermore, the reliance on dominator trees ensures that our validation results remain fully explainable, generating precise coverage metrics rather than black-box classifications. As autonomous agents are increasingly deployed in complex production environments, our approach provides a practical, low-shot, and interpretable mechanism for ensuring their reliability.

## References

Visual regression testing in design systems. https://sparkbox.com/foundry/design_system_visual_regression_testing. Accessed: 2026-01-21.

Visual regression testing. https://main.vitest.dev/guide/browser/visual-regression-testing. Accessed: 2026-01-21.

KK Aggarwal, Yogesh Singh, Arvinder Kaur, and OP Sangwan. A neural net based approach to test oracle. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–6, 2004.

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2): 87–106, 1987.

Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

Florent Avellaneda and Alexandre Petrenko. Fsm inference from long traces. In *International Symposium on Formal Methods*, pages 93–109. Springer, 2018.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. Agentrx: Diagnosing ai agent failures from execution trajectories. *arXiv preprint arXiv:2602.02475*, 2026.

Ronyérison Braga, Pedro Santos Neto, Ricardo Rabêlo, José Santiago, and Matheus Souza. A machine learning approach to generate test oracles. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, pages 142–151, 2018.

Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks, 2017. URL https://arxiv.org/abs/1608.04644.

Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1): 1–27, 2018.

Edmund M Clarke. *Model checking*. 1997.

Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

Werner Dietl and Peter Müller. Runtime universe type inference. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, pages 72–80, 2007.

Afonso Fontes and Gregory Gay. Using machine learning to generate test oracles: A systematic literature review. In *Proceedings of the 1st International Workshop on Test Oracles*, pages 1–10, 2021.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015. URL https://arxiv.org/abs/1412.6572.

Mouna Hammoudi. Regression testing of web applications using record/replay tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1079–1081, 2016.

Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 426–437, 2016.

Siyuan Hu, Mingyu Ouyang, Difei Gao, and Mike Zheng Shou. The dawn of gui agent: A preliminary case study with claude 3.5 computer use. *arXiv preprint arXiv:2411.10323*, 2024.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? 2024. URL https://arxiv.org/abs/2310.06770.

Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 178–189, 2014.

Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.

Reshabh K Sharma. Contextcov: Deriving and enforcing executable constraints from agent instruction files. *arXiv preprint arXiv:2603.00822*, 2026.

Reshabh K Sharma, Shraddha Barke, and Benjamin Zorn. Willful disobedience: Automatically detecting failures in agentic traces. *arXiv preprint arXiv:2603.23806*, 2026.

Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012.

Wil MP Van der Aalst, Martin Bichler, and Armin Heinzl. Robotic process automation. *Business & Information Systems Engineering*, 60(4):269–272, 2018.

Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical software engineering*, 21(3):811–853, 2016.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

Christoph Zauner. Implementation and benchmarking of perceptual image hash functions. 2010. URL https://api.semanticscholar.org/CorpusID:17075066.

# A Appendix: Semantic Equivalence Prompt

## A.1 LLM Prompt for State Equivalence Detection

```
Compare these two UI screenshots side-by-side.
Are the differences semantically meaningful?
Examples of NOT meaningful:
- Different window decorations
- Minor font rendering differences
- Timestamp changes
Examples of MEANINGFUL:
- Different form validation errors
- Different data displayed
- Different UI controls available
Please analyze the images and respond with:
1. Whether the differences are semantically meaningful (Yes/No)
2. A brief explanation of the key differences
3. Your confidence level in this assessment
Response format:
{
  "equivalent": true/false,
  "explanation": "...",
  "confidence": "high/medium/low"
}
```