Automating End-to-End Functional Testing of Web Applications based on Augmented Workflows and LLMs

Anonymous Author(s)

Abstract

End-to-End (E2E) functional testing validates web applications by executing user-level workflows across frontend and backend components. The effectiveness of E2E testing is commonly assessed through coverage metrics; however, existing approaches compute coverage over interaction spaces derived from models, specifications, or runtime exploration artifacts. As a result, coverage remains bounded by abstraction choices or exploration policies rather than by the executable behavior encoded in the frontend implementation itself. This paper reframes E2E coverage as a property of a finite, constraint-aware workflow space constructed directly from frontend source code. We introduce augmented workflows, which enrich navigation paths with implementation-level guard and state constraints, and define coverage strictly over the resulting feasible workflow set. Our approach statically extracts a navigation–interaction graph, enumerates bounded entry-to-terminal paths, aggregates and normalizes constraints, prunes infeasible workflows, and then dynamically realizes feasible workflows into executable Selenium scenarios using validated assignments. Coverage is computed over the fixed workflow space, ensuring a shared denominator across approaches. We evaluate the approach on two Angular applications and compare it against AUTOE2E, a state-of-the-art LLM-based E2E test generation technique. When both methods are evaluated over the same statically derived workflow space, our approach improves average workflow coverage from 61% to 87%. These results suggest that meaningful E2E coverage evaluation requires explicit construction of a finite, implementation-grounded reference space, and that separating representation from realization enables more complete exercise of executable frontend behavior.

ACM Reference Format:
Anonymous Author(s). 2026. Automating End-to-End Functional Testing of Web Applications based on Augmented Workflows and LLMs. In *Proceedings of The 30th International Conference on Evaluation and Assessment in Software Engineering (EASE 2026).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

End-to-End (E2E) functional testing verifies that an application behaves correctly from the user interface to the backend by executing sequences of user interactions. E2E testing exercises the application as a whole from the perspective of the end user [2]. For web applications, user-visible behavior is primarily defined at the frontend [3, 9, 13], so defects at this level directly affect user experience and are unlikely to be detected through unit or backend testing alone. E2E testing is therefore essential to validate functional behavior from the user's perspective.

E2E testing proceeds by defining workflows over the application's interaction behavior. A workflow is a structured sequence of user actions (e.g., clicks, inputs, navigations) that follows UI states and transitions to achieve a functional objective [24]. Executable tests are concrete instantiations of such workflows. The effectiveness of E2E testing therefore depends on how workflows are defined and exercised.

Coverage in this context denotes the proportion of frontend workflows exercised by executed tests. As a ratio, coverage is computed with respect to a reference interaction space that defines which workflows count as part of the system's behavior. If this reference space does not faithfully reflect executable frontend behavior, coverage claims become ambiguous and difficult to compare across tools. Achieving high coverage is challenging because workflows must correspond to behaviors that are structurally reachable, but also executable under real authorization, feature-flag, parameter, and state constraints. A path may appear valid in the UI structure while remaining infeasible in practice due to unsatisfied guard or state conditions. Thus, if such constraints are ignored, infeasible workflows inflate the reference space and distort coverage measurements. However, constructing an executable interaction space is difficult because interaction logic is distributed across routing configurations, components, templates, event handlers, guards, and state-dependent updates. Moreover, workflows are often derived from external artifacts (e.g., requirements or user stories) that evolve independently from the frontend implementation [18, 19], introducing drift between specified scenarios and actual behavior.

Traditionally, workflows are manually defined by inspecting requirements and exploring the application. This process is time-consuming, dependent on tester expertise, and difficult to reproduce [18, 19]. Automation frameworks like Selenium [1] facilitate execution but assume relevant workflows are already specified.

To reduce manual effort, prior work constructs interaction spaces either statically, by deriving models from code or external artifacts, or dynamically, by exploring the running system. Static approaches include model-based testing and artifact-driven scenario derivation, including recent LLM-based synthesis of E2E scripts from textual specifications [4, 8, 10–12, 20, 24, 26]. Dynamic approaches include crawling and exploration techniques, as well as LLM-driven web agents that select actions step-by-step over observed DOM state [2, 5, 16, 17, 22, 27]. Across both families, coverage is evaluated over derived artifacts (e.g., abstract models, specifications, discovered states, inferred features), that is, over representations produced by the abstraction or exploration strategy itself rather than over a

EASE 2026, Glasgow, United Kingdom

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

fixed, implementation-grounded workflow space. As a result, coverage remains bounded by artifact completeness, abstraction choices, or exploration budget. Existing approaches do not construct a finite, constraint-aware workflow reference space directly grounded in the frontend implementation and use that space itself as the denominator for coverage. Consequently, coverage reflects how well an approach explores or interprets its chosen artifact, not how completely it exercises executable implementation-level workflows.

In this paper, we address this limitation by constructing a closed, constraint-aware workflow reference space directly from source code. We statically extract navigation and interaction structure, aggregate implementation-level guard and state constraints, and prune infeasible paths to obtain a finite feasible workflow set. Coverage is defined strictly over this workflow space, while dynamic execution and LLM assistance are used to realize and exercise workflows so as to maximize coverage over that fixed space.

In summary, this paper makes the following contributions:

- We introduce *augmented workflows*, a finite, constraint-aware representation of frontend interaction behavior grounded in implementation-level navigation structure and guard/state constraints, and define E2E coverage over this representation as a stable, implementation-level reference space.
- We propose an automated approach to construct and realize augmented workflows by statically extracting navigation and constraint information from source code, pruning infeasible workflows, and instantiating the feasible ones into executable Selenium scenarios, enabling coverage to be computed over a fixed, constraint-aware workflow space.

We evaluate our approach on two Angular applications and compare it against AUTOE2E [2]. For each subject, coverage is computed over the same statically derived workflow reference space W, and test cases produced by both approaches are mapped onto this space prior to measurement. Across both applications, our method improves workflow coverage from an average of 61% to 87%. This demonstrates that defining coverage over a finite, constraint-aware, implementation-grounded workflow space exposes differences in realization capability that remain hidden when coverage is computed over abstraction- or exploration-derived artifacts.

2 Related Work

Maximizing E2E testing coverage depends on how the frontend interaction space is constructed and how coverage is defined over that space. Existing approaches construct this interaction space either (i) statically, by deriving models from code or external artifacts, or (ii) dynamically, by exploring the running system. Their limitations stem from how faithfully the constructed space reflects executable frontend behavior.

Static Construction of Interaction Spaces. Static approaches derive interaction models without executing the system. Model-Based Testing (MBT) techniques construct behavioral models (e.g., finite state machines) and generate workflows satisfying criteria such as state or transition coverage [8, 24, 25]. When reverse engineered from source code, such models typically capture routing structure and control flow. However, guard semantics, role constraints, feature flags, and state-dependent enabling conditions are often abstracted away or partially encoded. Coverage is therefore measured over reachable paths that may not correspond to executable workflows under realistic guard and state constraints.

Other static approaches derive workflows from external artifacts (e.g., requirements, use cases, or UML models) [7, 20, 26]. More recently, LLM-based techniques synthesize E2E test scripts directly from textual artifacts (e.g., user stories, Gherkin scenarios, form descriptions), translating them into executable Selenium scenarios [4, 10–12]. In these approaches, the interaction space is defined by the specification rather than by the implemented frontend logic. As a result, generated scripts may target scenarios that are text-plausible yet unexecutable under real authorization, feature-flag, or state constraints. Coverage is thus bounded by artifact completeness and alignment with the implementation.

Dynamic Construction through Runtime Exploration. Dynamic approaches construct interaction models by executing the application and abstracting observed runtime behavior. Crawling techniques such as AJAX Crawling and Crawljax [16, 17], GUI ripping and scriptless inference [6, 15, 21], and search- or reinforcement-learning-based exploration [14, 23, 27] aim to maximize discovered states, transitions, or coverage metrics.

Recent web agents employ LLMs operating over observed DOM state, selecting actions step-by-step during execution. AutoE2E [2] uses LLMs to infer application features and generate E2E tests executed dynamically against the system. Related work evaluates LLM-driven autonomous agents that execute natural-language test cases and produce verdicts through runtime interaction [5], as well as functionality-guided exploration agents navigating toward target behaviors using iterative reasoning over page state [22].

In these approaches, the interaction space is defined by observed runtime behavior and exploration policy rather than by a statically closed representation derived from source code. Coverage reflects the subset of behavior discovered under a given exploration budget or reasoning strategy, potentially omitting feasible but unvisited workflows encoded in the implementation. Reported metrics (e.g., task success, feature discovery) therefore depend on exploration itself rather than a fixed, implementation-level workflow space.

Coverage Semantics and Research Gap. Across both static and dynamic families, coverage is measured over an interaction space that is either structurally under-constrained (static models abstracting guards and state conditions) or exploration-bounded (dynamic models limited to discovered behavior). Static structure-centric models risk over-approximating executability, while dynamic exploration risks under-approximating implemented behavior. To the best of our knowledge, prior work does not explicitly construct a finite, constraint-aware workflow reference space grounded directly in the frontend implementation and use that space itself as the denominator for coverage computation.

3 Approach

Overview and Research Questions. The objective of our approach is to maximize E2E test coverage for web applications. To achieve this objective, we address three research questions:

- **RQ1 (Representation):** *How can frontend interaction behavior be represented such that the resulting workflow space* *is complete with respect to the implementation, constraint-aware, finite, and executable?*

- **RQ2 (Construction):** *How can such a representation be systematically and deterministically derived from frontend source code?*
- **RQ3 (Realization):** *How can workflows in this representation be instantiated and executed so as to maximize coverage over that space?*

Figure 1 summarizes our process to address these RQs. Phase A constructs a finite workflow reference space W from frontend source code by extracting the navigation–interaction graph G (A1), enumerating bounded entry-to-terminal paths (A2), and aggregating constraints to prune infeasible workflows (A3). Phase B then maximizes coverage over this fixed space by translating workflows in W into executable Selenium scenarios via validated assignments (B1), executing them with feedback-driven retries and mutation (B2), and computing coverage as $|E|/|W|$ where E is the subset of workflows that succeed at least once (B3).

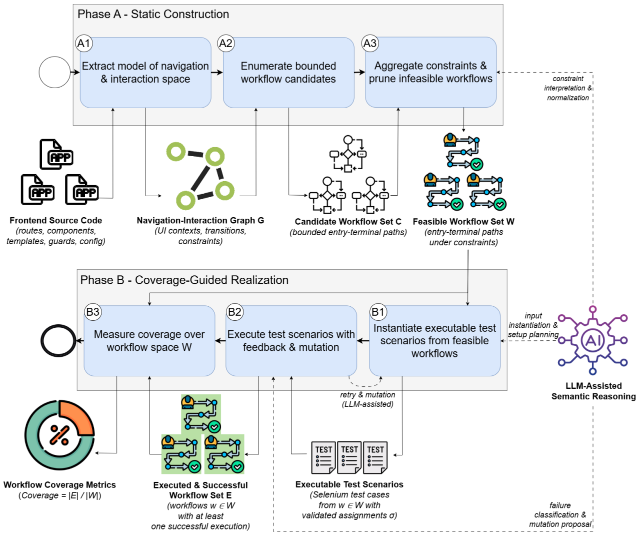
Phase A - Static Construction
A1
Extract model of navigation & interaction space
A2
Enumerate bounded workflow candidates
A3
Aggregate constraints & prune infeasible workflows
constraint interpretation & normalization
Frontend Source Code
(routes, components, templates, guards, config)
Navigation-Interaction Graph G
(UI contexts, transitions, constraints)
Candidate Workflow Set C
(bounded entry-terminal paths)
Feasible Workflow Set W
(entry-terminal paths under constraints)
Phase B - Coverage-Guided Realization
B3
Measure coverage over workflow space W
B2
Execute test scenarios with feedback & mutation
B1
Instantiate executable test scenarios from feasible workflows
input instantiation & setup planning
LLM-Assisted Semantic Reasoning
retry & mutation (LLM-assisted)
Workflow Coverage Metrics
(Coverage = |E| / |W|)
Executed & Successful Workflow Set E
(workflows w ∈ W with at least one successful execution)
TEST
TEST
TEST
Executable Test Scenarios
(Selenium test cases from w ∈ W with validated assignments σ)
failure classification & mutation proposal

Figure 1: Our process for constructing a closed workflow space and maximizing E2E coverage over it.

RQ1: Augmented Workflows as a Constraint-Aware Representation. The issue underlying RQ1 is representational inadequacy. Classical user workflows capture reachability but not executability. A path such as */login* → */dashboard* → */admin* may exist in the navigation structure because links or navigation calls connect these routes. However, this path may be unexecutable if the `/admin` route requires authentication and a specific role. Thus, reachability does not imply executability. The representation must therefore incorporate implementation-level constraints.

To resolve this issue, we introduce *augmented workflows*. An augmented workflow w is a path whose transitions are annotated with constraints extracted from the implementation. These constraints originate from route guards and resolvers, UI visibility and enablement predicates, handler-level conditional checks, required route parameters, and feature flags. All encountered constraints are aggregated into a workflow-level constraint set $C(w)$.

A workflow is considered feasible if there exists at least one frontend state assignment σ satisfying $C(w)$: w is feasible $\iff$ $\exists\sigma$ such that $C(w)(\sigma) = \texttt{true}$. For example, the previous path becomes feasible only if $\sigma = \{\texttt{authenticated = true}, \texttt{role = admin}, \texttt{profileExists = true}\}$. If no consistent assignment exists (e.g., conflicting role requirements), the workflow is discarded. The reference space W is therefore the set of all feasible augmented workflows. This representation resolves the executability gap while preserving structural completeness.

RQ2: Systematic Construction of the Closed Workflow Space. The issue underlying RQ2 is methodological: even with a suitable representation, constructing it from frontend code must be systematic, deterministic, and finite.

A1: Navigation–interaction multigraph extraction. Frontend interaction logic is distributed across routing configurations, templates, guards, event handlers, and service calls. Without a principled extraction strategy, the resulting workflow space may be incomplete, inconsistent, or unbounded, and transitions and their associated constraints would remain disconnected. To address this issue, we must consolidate scattered interaction logic into a single coherent structure. We therefore statically analyze the codebase via AST inspection and construct a navigation–interaction multigraph G. Nodes represent UI contexts (e.g., routes, components), while edges represent user-triggered transitions (e.g., clicks, submissions) or programmatic navigation calls (e.g., `router.navigate()`, `routerLink`). Each edge is annotated with constraints extracted at its origin (e.g., guard bindings, visibility predicates, conditional navigation logic). This step produces a deterministic structural model of implemented interaction behavior; given the same codebase and configuration, the resulting graph G is identical.

A2: Bounded workflow enumeration. The issue addressed here concerns unboundedness. Even if interaction logic is systematically extracted, unrestricted path enumeration over G may lead to an infinite workflow space due to cycles and recursive navigation patterns (e.g., dashboard → settings → dashboard). We therefore impose explicit structural bounds during enumeration. To ensure finiteness, we enumerate only entry-to-terminal paths bounded by a maximum path length k and restrict repeated visits to the same context. Entry contexts correspond to initial application states (e.g., root or public routes), while terminal contexts correspond to leaves or designated completion states (e.g., logout success, pages with no outgoing user-triggered transitions). This yields a finite set $\text{Paths}(G, k)$.

A3: Constraint aggregation and pruning. The final issue concerns constraint comparability and consistency. Extracted guard logic may contain nested boolean expressions, for example:

```
return this.authService.isLoggedIn() &&
(this.userService.hasRole('admin') ||
this.featureService.isEnabled('beta'));
```

Such expressions mix authentication state, role membership, and feature flags in a way that is difficult to compare and reason about across workflow steps. To enable systematic feasibility analysis, we normalize such expressions into a fixed atomic schema comprising authentication requirement, required roles, feature flags, route parameters, and required entities. The LLM is used only to translate complex expressions into this schema. For example the expression above is summarized into `[authRequired=true, rolesRequired={’admin’}, featureFlags={’beta’}]`. This enables deterministic contradiction detection across aggregated constraints. As such, workflows whose constraint sets are unsatisfiable are pruned. For example, if one transition requires an `’admin’` role and another transition along the same workflow requires `’user’`, then no consistent assignment exists and the workflow is discarded. The resulting feasible workflow set $W = \{w \in \mathrm{Paths}(G, k) \mid \mathrm{SAT}(C(w))\}$ is finite because enumeration is bounded and infeasible paths are eliminated.

RQ3: Coverage-Guided Realization over the Closed Workflow Space. The issue underlying RQ3 is operational: although Phase A yields a finite feasible workflow space W, feasibility does not guarantee execution success under concrete runtime conditions. For a workflow $w \in W$, realization requires producing at least one concrete assignment σ that satisfies $C(w)$ and translating the workflow's transitions into executable Selenium actions. Phase B therefore performs coverage-guided realization while keeping W fixed, so that execution progress changes only the numerator of coverage.

B1: Translation into Selenium scenarios via validated assignments. For each workflow w, the system generates one or more candidate assignments σ and emits a Selenium scenario whose steps implement the transitions of w (locate element, click, type, submit) under σ. The LLM acts as a constrained planner: given the workflow steps, the normalized atomic schema from A3, and the available seeded test data, it proposes concrete values and setup actions (e.g., pick an admin account, choose a valid identifier). Each proposal is deterministically validated against $C(w)$ before code generation, and only schema-consistent assignments are accepted.

B2: Feedback-driven execution with bounded retry and mutation. Scenarios are executed in a controlled environment with a fresh browser session, fixed configuration and feature flags, and seeded backend data. If execution succeeds, the workflow is added to the executed-and-successful set $E \subseteq W$. If execution fails, the run is classified (e.g., authorization mismatch, missing entity, invalid parameter, UI unavailability, unexpected navigation). The system performs bounded retries by mutating σ through deterministic rules (e.g., alternate role/account, alternate seeded entity, alternate valid parameter) and LLM-assisted repair proposals constrained by the same atomic schema. Retries affect only the assignment and scenario instance, but W remains unchanged.

B3: Coverage computation. Coverage is computed over the fixed denominator W as Coverage $= |E|/|W|$. During evaluation, workflows produced by baselines are mapped onto the same reference space W before measurement, so coverage differences reflect realization effectiveness rather than differences in the underlying interaction space. This construction operationalizes coverage over executable implementation-level workflows rather than over abstraction- or exploration-bounded artifacts.

4 Evaluation

The goal of our evaluation is to assess whether our approach achieves higher E2E coverage over a fixed, implementation-grounded workflow space W compared to an established baseline. We therefore answer the following research question:

RQ (Coverage Maximization over W). Does our approach achieve higher workflow coverage compared to an existing baseline when both are evaluated over the same workflow space W?

Baseline and Dataset. We compare against AUTOE2E [2], a publicly available LLM-based E2E test generation approach. We evaluate on two Angular applications: (i) *PetClinic*[1], a full-stack healthcare management system with guarded routes, role-based navigation, and multi-step workflows involving backend entities, and (ii) *Tour of Heroes*[2], a canonical Angular application featuring client-side navigation, conditional UI rendering, and parameterized routing. PetClinic contains deeper business workflows with backend dependencies, whereas Tour of Heroes emphasizes client-side interaction patterns.

Experimental Protocol. For each subject, we first apply Phase A to construct the workflow reference space W by extracting the navigation–interaction graph G, enumerating bounded entry-to-terminal paths (with k=5), normalizing constraints, and pruning infeasible workflows. We set k=5 to capture common multi-step user tasks while ensuring W remains finite in the presence of navigation cycles. This W serves as the fixed denominator for coverage. We then execute Phase B to realize workflows in W into Selenium scenarios and collect the successfully executed set E_{ours}. For constraint normalization (A3) and assignment generation during realization (B1/B2), we use OpenAI's GPT5.2 LLM. Separately, we run AUTOE2E using its default configuration. Each generated test case is mapped onto W by extracting its UI action sequence (navigate, click, type, submit) and matching it to an entry-to-terminal workflow in W over the same action alphabet. Unmatched tests are treated as outside W and do not contribute to coverage. If the sequence corresponds to a workflow in W, it contributes to E_{auto}, otherwise it is treated as outside W and do not contribute to coverage. Coverage for both approaches is computed as Coverage $= \frac{|E|}{|W|}$. All experiments are executed under identical runtime conditions for both approaches, including the same browser version, application configuration, and initialized backend state. For both subjects, required entities (e.g., users, roles, and domain objects) are pre-seeded to ensure that workflows deemed feasible in W are realizable. No manual intervention is performed during execution.

Results. Table 1 reports the size of W and the number of successfully realized workflows for both approaches. Across both subjects, our approach achieves higher coverage over W.

Subject	$\lvert W\rvert$	$\lvert E_{\text{ours}}\rvert$	$\lvert E_{\text{auto}}\rvert$	Coverage (%)
PetClinic	84	71	49	84.5 / 58.3
Tour of Heroes	36	32	23	88.9 / 63.9

Table 1: Workflow coverage over the statically derived space W. Coverage values are reported as (ours / AUTOE2E).

[1] https://github.com/parsaalian/autoe2e/tree/main/benchmark/pet-clinic/spring-petclinic-angular

[2] https://github.com/johnpapa/heroes-angular

Analysis. On PetClinic, our approach covers 71 out of 84 workflows (84.5%), compared to 49 (58.3%) for AUTOE2E. On Tour of Heroes, coverage increases from 63.9% to 88.9%. The difference is more visible in PetClinic, which contains guarded routes and multi-step workflows requiring consistent role assignments and valid backend entities. AUTOE2E frequently generates interaction sequences that correspond to structurally valid navigation paths but fail to satisfy constraint combinations required for feasible execution, and therefore do not map to valid elements of W. In contrast, our realization phase instantiates workflows under validated constraint assignments derived during Phase A. This leads to a higher proportion of successfully executed workflows, particularly for longer or constraint-dependent paths.

Discussion. The results indicate that coverage defined over an implementation-grounded workflow space exposes differences in realization capability that are not visible when coverage is computed over abstraction-derived artifacts. AUTOE2E focuses on feature inference and dynamic test generation, which may generate diverse interaction sequences but does not explicitly reason over a closed, constraint-aware workflow space. Because our approach separates representation (Phase A) from realization (Phase B), it systematically enumerates feasible workflows before execution and ensures that realization respects aggregated constraints. This design particularly benefits applications with guard logic, role-based routing, and parameterized transitions. Additionally, because both methods are evaluated against the same W, the observed coverage gap reflects differences in executable workflow realization rather than differences in exploration policy or artifact construction.

Threats to Validity. Subject selection. We evaluate on two Angular applications, which limits generalizability to other frontend frameworks or architectural styles. However, the selected subjects differ in size and workflow complexity, partially mitigating this risk.

Bounded path length. The workflow space W depends on the chosen bound k. A different bound could alter $|W|$ and thus absolute coverage values. We fixed k=5 consistently across subjects and methods.

Mapping accuracy. Mapping AUTOE2E test cases to workflows in W requires interpreting action sequences under our representation. We mitigate mapping bias by restricting the mapping to the same action alphabet used to construct G (navigate, click, type, submit) and by matching only to entry-to-terminal workflows in W. Although this mapping is deterministic, mismatches in abstraction granularity could still influence measured coverage.

Execution environment. Coverage results depend on controlled runtime configuration and seeded data. Different environment setups may affect realizability outcomes.

LLM involvement. Our approach uses OpenAI's GPT5.2 LLM for schema-level normalization of extracted constraints (A3) and for proposing candidate assignments during realization (B1/B2). All outputs are deterministically validated against the extracted constraint set $C(w)$ before execution. Nonetheless, model version, configuration, or nondeterminism could affect realized coverage if validation criteria or prompting change.

Despite these limitations, the consistent coverage improvement across both subjects suggests that defining coverage over a finite, constraint-aware, and implementation-grounded workflow space provides a stronger basis for evaluating E2E effectiveness.

5 Conclusion

This paper reframes E2E test coverage as a property of a finite, constraint-aware workflow space grounded directly in frontend implementation logic. We introduced augmented workflows as an executable representation of navigation behavior enriched with guard and state constraints, and defined coverage strictly over this reference space. By separating representation (static construction of W) from realization (constraint-aware instantiation), our approach enables coverage to be measured over executable implementation-level workflows rather than abstraction- or exploration-bounded artifacts. Empirical evaluation on two Angular applications demonstrates that this design leads to higher workflow coverage compared to a state-of-the-art LLM-based baseline when both are evaluated over the same W. These results suggest that precise denominator construction is central to meaningful E2E coverage evaluation. Future work includes extending the approach to additional frontend frameworks, exploring adaptive bounds for workflow enumeration, and integrating richer semantic reasoning over constraint schemas.

References

[1] Selenium, 2004.

[2] Alian, P., Nashid, N., Shahbandeh, M., Shabani, T., and Mesbah, A. Feature-Driven End-to-End Test Generation. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada, Sept. 2025), ICSE '25, IEEE Press, pp. 450–462.

[3] Balsam, S., and Mishra, D. Web Application Testing - Challenges and Opportunities. *Journal of Systems and Software 219* (Jan. 2025), 112186.

[4] Cavalcanti, A. R., Accioly, L., Valença, G., Nogueira, S. C., Morais, A. C., Oliveira, A., and Gomes, S. Automating Test Design Using LLM: Results from an Empirical Study on the Public Sector. *Conference on Digital Government Research 26* (May 2025). Conference Name: Conference on Digital Government Research.

[5] Chevrot, A., Vernotte, A., Falleri, J.-R., Blanc, X., Legeard, B., and Cretin, A. Are Autonomous Web Agents Good Testers? *Proceedings of the ACM on Software Engineering 2*, ISSTA (June 2025), 206–228.

[6] Domínguez Osorio, J. M. *Automated GUI Ripping for Web Applications.* PhD thesis, Universidad de los Andes, 2019. Accepted: 2020-09-03T14:56:04Z Publisher: Uniandes.

[7] García, B., and Dueñas, J. C. Automated Functional Testing based on the Navigation of Web Applications. *Electronic Proceedings in Theoretical Computer Science 61* (Aug. 2011), 49–65. arXiv:1108.2357 [cs].

[8] Garousi, V., Keles, A. B., Balaman, Y., Guler, Z. O., and Arcuri, A. Model-based Testing in Practice: An Experience Report from the Web Applications Domain. *Journal of Systems and Software 180* (Oct. 2021), 111032.

[9] Garousi, V., Mesbah, A., Betin-Can, A., and Mirshokraie, S. A Systematic Mapping Study of Web Application Testing. *Information and Software Technology 55*, 8 (Aug. 2013), 1374–1396.

[10] Júnior, E., Valejo, A., Valverde-Rebaza, J., and Neves, V. d. O. GenIA-E2ETest: A Generative AI-Based Approach for End-to-End Test Automation. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software (SBES 2025)* (Sept. 2025), pp. 282–292. arXiv:2510.01024 [cs].

[11] Leotta, M., Yousaf, H. Z., Ricca, F., and Garcia, B. AI-Generated Test Scripts for Web E2E Testing with ChatGPT and Copilot: A Preliminary Study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno Italy, June 2024), ACM, pp. 339–344.

[12] Li, T., Cui, C., Huang, R., Towey, D., and Ma, L. Large Language Models for Automated Web-Form-Test Generation: An Empirical Study — RCR Report. *ACM Transactions on Software Engineering and Methodology* (Feb. 2026), 3797275.

[13] Li, T., Huang, R., Cui, C., Towey, D., Ma, L., Li, Y.-F., and Xia, W. A Survey on Web Application Testing: A Decade of Evolution, Dec. 2024. arXiv:2412.10476 [cs].

[14] Liu, C.-H., You, S. D., and Chiu, Y.-C. A Reinforcement Learning Approach to Guide Web Crawler to Explore Web Applications for Improving Code Coverage. *Electronics 13*, 2 (Jan. 2024), 427. Publisher: Multidisciplinary Digital Publishing Institute.

[15] Memon, A., Banerjee, I., and Nagarajan, A. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *Proceedings of the 10th Working Conference on Reverse Engineering* (USA, Nov. 2003), WCRE '03, IEEE Computer Society, p. 260.

[16] Mesbah, A., Bozdag, E., and van Deursen, A. Crawling AJAX by Inferring User Interface State Changes. In *2008 Eighth International Conference on Web Engineering* (July 2008), pp. 122–134.

[17] Mesbah, A., van Deursen, A., and Lenselink, S. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web 6*, 1 (Mar. 2012), 3:1–3:30.

[18] Nass, M. *On Overcoming Challenges with GUI-based Test Automation.* PhD thesis, Blekinge Institute of Technology, Karlskrona, 2024. Publisher: Blekinge Tekniska Högskola.

[19] Nass, M., Alégroth, E., and Feldt, R. Why Many Challenges with GUI Test Automation (Will) Remain. *Information and Software Technology 138* (Oct. 2021), 106625.

[20] Nebut, C., Fleurey, F., Le Traon, Y., and Jezequel, J.-M. Automatic Test Generation: A Use Case Driven Approach. *IEEE Transactions on Software Engineering 32*, 3 (Mar. 2006), 140–155.

[21] Pastor Ricós, F., Slomp, A., Marín, B., Aho, P., and Vos, T. E. J. Distributed State Model Inference for Scriptless GUI Testing. *Journal of Systems and Software 200* (June 2023), 111645.

[22] Shahbandeh, M., Alian, P., Nashid, N., and Mesbah, A. NaviQAte: Functionality-Guided Web Application Navigation, 2024. Version Number: 1.

[23] Sherin, S., Muqeet, A., Khan, M. U., and Iqbal, M. Z. QExplore: An Exploration Strategy for Dynamic Web Applications Using Guided Search. *Journal of Systems and Software 195* (Jan. 2023), 111512.

[24] Utting, M., and Legeard, B. *Practical Model-based Testing: A Tools Approach.* Elsevier, 2010.

[25] Utting, M., Pretschner, A., and Legeard, B. A Taxonomy of Model-Based Testing Approaches. *Software Testing, Verification and Reliability 22*, 5 (2012), 297–312. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.456.

[26] Wang, C., Pastore, F., Goknil, A., and Briand, L. C. Automatic Generation of Acceptance Test Cases From Use Case Specifications: An NLP-Based Approach. *IEEE Transactions on Software Engineering 48*, 2 (Feb. 2022), 585–616.

[27] Zheng, Y., Liu, Y., Xie, X., Liu, Y., Ma, L., Hao, J., and Liu, Y. Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (May 2021), pp. 423–435. ISSN: 1558-1225.