eBAT: An Efficient Automated Web Application Testing Approach Based on Tester's Behavior

Mridha Md. Nafis Fuad
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
bsse0920@iit.du.ac.bd

Kazi Sakib
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh
sakib@iit.du.ac.bd

***Abstract*—Web application failure detection relies mostly on the tester's creativity, leaving test automation to only ease executing repetitive tasks. Existing automated testing techniques opt for test path diversity or input generation but not the tester's behavioral patterns. For example, testing deeply nested business logic, proper form submission, or non-redundant navigation are not considered. This paper proposes eBAT, an automated testing approach that considers those testers' interaction patterns from observation. A behavior-driven action selection strategy is derived from these patterns to interact with the system. Actionable elements (buttons, links, inputs, etc.) obtained through state abstraction and interaction pattern-wise grouping are operated in a tree-based manner. The effectiveness and efficiency of eBAT are evaluated as the unique number of failures detected and the detection rate respectively. Results compared against the state-of-the-art indicate significant improvement in failure detection with similar code coverage. Moreover, eBAT outperforms the baseline failure detection rate in 5 out of 6 benchmark projects.**

***Index Terms*—web testing, behavior, tester's behavior, automated testing, redundancy reduction**

I. INTRODUCTION

In manual web application testing, a tester always tries to test a complete functionality at a time. They design test cases in such a way that fulfills each of the unique functionality. Their goal is to detect bugs with minimum redundant action execution [1] (such as clicking the same link multiple times is redundant). These common practices can be defined as the tester's behavior. Automated web testing approaches [2], [3] focus mostly on testing metrics like code, branch coverage or path diversity, etc., which may fail to explore the complete functionality (e.g., filling the inputs without submitting the form). As a result, these approaches may lead to partial but ineffective functional execution sequences.

An automated testing approach based on the tester's behavior can efficiently test a complete functional action sequence, combining the benefits of both manual and automated testing. However, determining such human-level behavioral patterns is challenging as manual testing practices may vary based on the tester or organization. A generic set of behavior needs to be designed such that it is applicable to any web application. These patterns might be obtained from examining manually written automation test scripts. Test scripts from the open-source community can be selected as a baseline to remove any developer or organization-specific practices. The resulting set of behaviors incorporated into an automated testing approach may make the failure detection process efficient.

Automated testing is broadly categorized into model-based and model-free approaches. Model-based techniques [3]–[6] are one of the most popular types but limited to generating test cases using a static navigation model. These test cases often require human domain knowledge for inputs or domain-specific description [4], [7]. Model-free approaches that select actions in a pseudo-random manner [8] are termed as random-based. They consider the dynamic nature of the **A**pplication **U**nder **T**est (**AUT**) but generate redundant and invalid test cases. Zheng et al. [9] proposed a model-free reinforcement learning based fully automated approach to explore diverse application scenarios. However, the action selection policy accounts for much redundancy and randomness, which affects the test suite efficiency. Despite recent advancements, no automated testing approach incorporates the tester's exploration strategies for efficiency in failure detection.

This paper proposes an **e**fficient **B**ehavior based **A**utomated **T**esting approach (**eBAT**), incorporating strategies employed in manual testing such as non-redundant exploration and effective action interaction (e.g., non cyclic navigation, form submission, etc.). The behavioral patterns considered in this approach do not represent the entire set of tester's interaction patterns, rather a subset observed in manually written test cases. These test cases were extracted from popular open-source projects which represent the most popular frontend web frameworks. These patterns were used to devise an action selection strategy, focusing on - i) non-redundant actions in **E**xecution **T**race (**ET**) and ii) interdependent action grouping. Each test case starts from the root page of AUT and selects actionable elements (buttons, links, etc.) based on the strategy. State abstraction mechanism is applied to group functionally similar application behaviors and extract the set of operable actions. The actions that testers execute together to accomplish a business logic are grouped (for example, inputs and submit button inside web form). Redundant actions that are encountered previously in the current trace are avoided. The execution of the resulting actions in each state generates a

Fellowship from ICT Division, Government of Bangladesh – No.: 56.00.0000.052.33.002.22-42, date: 12.06.2022.

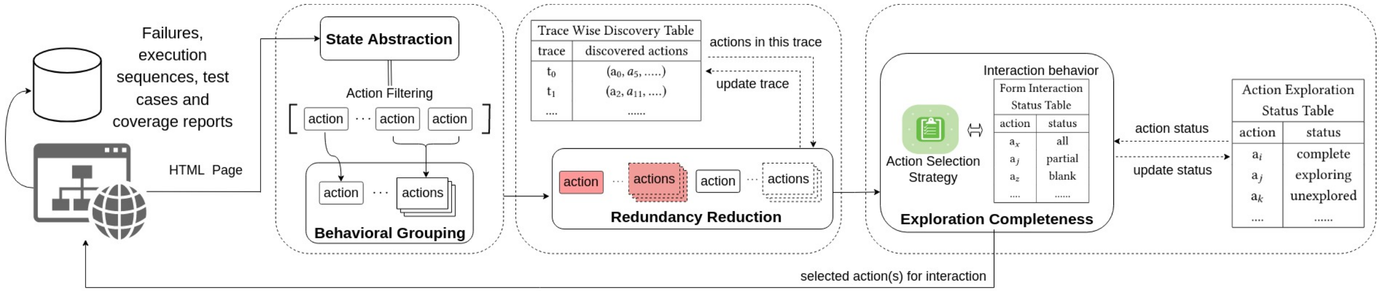


Fig. 1: Overview of proposed behavior-based automated testing approach

navigation model (state flow graph). Test cases are generated by executing a partially explored action sequence until all actions are completely explored using tree-based traversal [2].

Evaluation is done using a time budget of 30 minutes on six open-source benchmark web applications taken from prior work [3], [9]. In terms of effectiveness and efficiency of failure detection, eBAT outperforms the current state-of-the-art, WebExplor [9]. Their results were statistically compared using Mann–Whitney U test. An early rise in failure detection rate in eBAT indicates the impact of non-redundant exploration and behavioral action interaction. The average number of failures detected in WebExplor and eBAT are 8.1 and 17.8 respectively with similar code coverage. Moreover, eBAT detected the same number of failures within only 5 minutes of execution against 30 minutes for WebExplor in 5 out of 6 subjects.

II. RELATED WORK

Automated web application testing gained much research attention due to the challenges imposed by the dynamic nature of web applications [10], [11]. Approaches in literature can be broadly classified into model-based and model-free techniques.

Model-based approaches are the most common type in literature as test cases can be generated without executing the application. They rely on a static navigation model, requiring external assistance or the tester's domain knowledge. Such examples are domain-specific language in the approach ATA [7], user session behavior to generate statistical navigation model in [12], manually crafted inputs for diverse paths and navigation model in DIG [3] and SUBWEB [4] respectively. However, static models cannot capture the full essence of AUT as they fail to detect the dynamic changes from any event [9]. Pages that are not included in the navigation model are never explored and thus not tested by these approaches.

This limitation of model-based approaches is addressed in model-free approaches where the AUT is crawled to generate test cases on-the-fly. This enables interaction with actions resulting from dynamic changes. Fard et al. proposed FeedEx [2] where the exploration prioritizes actions that maximize the effect on functionality coverage, navigation coverage, page structure coverage, and test suite size. However, FeedEx only generates a dynamic abstract test model rather than actual test cases. On the other hand, random-based action selection strategy [8] generates test cases from a random or pseudo-random interaction with AUT. They generate a large number of redundant test cases along with invalid action sequences, suffering from infeasibility problems [4].

TESTAR [13] is a scriptless model-free testing approach that can utilize random, Q-learning, or evolutionary computing based action selection strategies. Redundant actions are avoided only if they are present in their immediate parent state without considering cyclic navigation. Moreover, manual intervention is needed to blacklist undesirable actions and predefined grouped actions for the action selection rule.

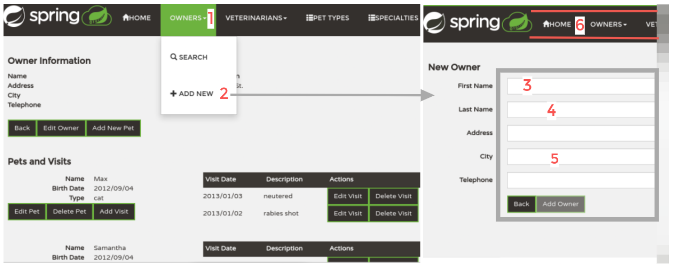
Zheng et al. [9] proposed WebExplor, an RL-based model-free automated web testing approach evaluated to be the current state-of-the-art. However, redundant test cases are generated as the action selection policy adds noise sampled from Gumbell distribution, accounting for exploring new actions generated dynamically. This makes the process inefficient even with high-level DFA guidance to explore less visited features.

The broad two categories have their own set of advantages and disadvantages. Model-based approaches can incorporate the tester's behavior via manual intervention, i.e., laborious. On the other hand, model-free approaches are significantly automated but suffer from redundancy and the inability to test the complete functionality. Combining the benefits from both categories, i.e, incorporating the testers' behavior in a model-free approach may improve the testing efficiency.

III. METHODOLOGY

This paper proposes eBAT, incorporating the tester's behavioral patterns in model-free testing. By doing so, it minimizes execution having no impact on failure detection. The entire approach is divided into four components, namely - State Abstraction, Behavioral Grouping, Redundancy Reduction, and Exploration Completeness, as indicated in Fig. 1.

State Abstraction. The underlying representation (such as HTML page, screenshot, etc.) of a web application at a particular time is its state. To achieve the required functionality, states contain a set of operable elements such as links, buttons, input fields, dropdowns, etc. The interaction with such elements is called actions in web testing. These interactions may lead to a change in the URL or functionality of the current DOM. An exponentially increasing number of states affects the efficiency of test suite generation [14]. This problem is solved by employing state abstraction to detect the functionally similar states, reducing redundant testing of the same behavior.



Name Max
Birth Date 2012/09/04
Type cat

Visit Date	Description	Actions
2013/01/03	neutered	Edit Visit Delete Visit
2013/01/02	rabies shot	Edit Visit Delete Visit

Name Samantha
Birth Date 2012/09/04

Visit Date	Description	Actions

Fig. 2: Petclinic sample project (ineffective execution scenario *owner-detail* -> *add-owner* illustrated by actions 1-6)

Motivation for the abstracted state calculation process has been taken from [9]. On each ET, the approach starts by loading the home page of AUT (URL provided as a parameter) and extracts the HTML representation. The sequence of HTML tag names along with corresponding nested attributes is listed recursively. Digits within the attributes are removed for consistency as many modern front-end frameworks dynamically generate attribute IDs on each page load. The Gestalt pattern matching algorithm is used to calculate the similarity of the listed sequence after each change in the application representation. However, to differentiate the abstracted states a threshold of 0.8 is used based on prior work [9]. The resulting state is considered unique if - i) the difference between the two states is greater than the predefined threshold, or ii) the URLs of the two states are different. The operable HTML tags such as $\langle button \rangle$, $\langle a \rangle$, $\langle input \rangle$, etc. are parsed and mapped as the state actions. Only the visible actions are filtered using the area bounded by the element's box model.

Behavior Identification. A set of most commonly employed testers' behavior is needed for an efficient action section strategy. Since no prior work lists such behaviors, manually written front-end automation test suites are analyzed to identify them. The open source community is preferred for baseline project selection due to public availability and to avoid organization or developer specific behaviors. Firstly, the top six popular frontend frameworks are selected from prior work [3] (based on stars). Secondly, similar to [15], for each framework a popular GitHub repository that contains a list of projects is selected. The projects within these repositories are included for analysis if - i) GitHub star count $\geq$ 50, and ii) automation test scripts mocking the tester's interactions are available (selected resources are made available [16]). Finally, test scripts from the resulting 51 projects are analyzed manually to determine the behaviors that occur in every scenario. The subset of the tester's behaviors identified are - i) executing non-redundant actions in a particular trace with non-cyclic navigation, ii) proper form submission, and iii) form submission with complete or partial input. These behaviors are incorporated in eBAT to devise the exploration strategy.

The Petclinic open source project, as shown in Fig. 2, is used to demonstrate the impact of the identified behaviors during testing. Petclinic contains most of the failures in functionalities accessible only from the deeply nested *owner-detail* page [9]. Suppose a tester reaches this page and instead of focusing on its functionalities, follows the action sequence as shown in Fig. 2. This would result in a redundant ineffective execution. Employing the identified behaviors in this scenario would - i) prevent cyclic navigation (Actions 1-2) since navbar actions are present on all pages, ii) prevent ineffective form interaction (Actions 3-6), and iii) test form submissions with missing data (e.g., exploiting forms in *add-visit* page)

Behavioral Grouping. The process of grouping actions that act interdependently is termed behavioral grouping in eBAT. Actions are grouped in eBAT that collaborate in a specific sequence to achieve a business objective. Testing these functionalities, requiring selection of such interdependent actions would otherwise rely on "luck". Web forms are considered as they most naturally encapsulate interdependent nested actions (input, dropdown, radio button, submit button, etc. [17]) to complete the intended functionality. The form encapsulated inputs and buttons are combined to always interact together as a compound action during on-the-fly testing. This grouping is done by determining the actions residing within a $\langle form \rangle$ element in the current state. These composite actions are added to the set of valid actions in the state, making actions inside the form no longer defined or executed individually. The detected submit button (using type="submit" attribute) is always executed at the end of different possible input action interactions in the case of composite form actions. This pattern of interaction ensures proper form submissions and avoids unnecessary interaction with input actions separately. The behavioral grouping of table rows and list items is not considered as the diversity in the content may exploit different failures, e.g., the *owner-detail* page with and without pets need to be tested in Petclinic open-source project.

Redundancy Reduction. Employing state abstraction and action grouping algorithms is not sufficient for efficiency as the actions are often duplicated [10]. Tracking redundant actions within a single trace enables the exploration of the system through diverse paths. For instance, the actions in the layout components (navbar, sidebar, and footer) are present in multiple states. Such actions, although usually unchanged in their functionality, are marked as valid actions in each state. The comparison of actions is done based on the action's tag and attribute representation used to calculate the state similarity. An action is called redundant if it is encountered previously in the trace [1]. Actions discovered in the current ET are tracked to detect duplicates. On encountering a new state in a particular trace, redundant actions are filtered (as depicted in Fig. 1). The resulting set of actions within the new state is considered for interaction in that particular trace.

Exploration Completeness. Web applications rely on a State-Flow Graph based navigation model [3], [5] where the nodes and edges represent various states of AUT and transitions between states due to action events respectively. This model can be updated dynamically based on business logic [9], e.g., the same URL containing a different set of actions based on authorization. The discovery of diverse states in AUT is devised using tree-based traversal of the dynamic navigation model [9]. This dynamic model is constructed using the states as nodes discovered through exploration. Edges indicate transitions $\langle s, a, s' \rangle$ between states using action a from state s. Depth-first traversal is used to prioritize deeply nested states that require a specific sequence of interaction [18]. Moreover, this strategy ensures incremental discovery of all states and tests the completeness of a particular path.

Instead of backtracking to immediate parent states, the traversal is done starting from the root state for every trace. On each change, the state abstraction mechanism is employed to match similarity and extract executable actions. Afterward, the actions are grouped and redundant actions are removed through the behavioral grouping and redundancy reduction steps respectively. Actions are executed automatically based on the element's type (using Puppeteer [19]), such as clicking links or buttons, filling up input fields with random values based on the type attribute, etc. Actions leading to external resources are blacklisted from further interaction.

Initially, starting from the root page, all actions are marked as "unexplored". An unexplored unique action is selected at random from the current web state and marked "active". This action exploration state mapping is maintained across ETs. For a different trace starting from the root, the exploration strategy needs to continue from the previous active action sequence. Partially explored actions are marked "complete" recursively when no actions in the resulting state are left incomplete. On the other hand, grouped form actions are marked as complete after considering possible interaction patterns - i) all inputs left blank, ii) interact with all inputs, and iii) interact with a subset of inputs. Finally, the entire process is repeated when all actions from the root state attain exploration completeness. The repetition promotes the exploration of new actions in nested pages that might be unavailable before, e.g., the delete functionality is exposed only after successful creation.

IV. EVALUATION AND RESULTS

To evaluate the effectiveness and efficiency in terms of failure detection, eBAT is compared against the state-of-the-art WebExplor [9], as it outperforms other model-based, random, and model-free automated testing approaches. Upon encountering a failure during trace execution (tracked via browser console errors), it is added to the set of failures unique to each state. The sequence of actions taken to exploit the failure is stored as a failing test case. Both approaches were compared using the same environment, parameters, failure definition, initial login script, and AUT version. Code coverage is measured in both approaches using a python wrapper of Google's Puppeteer library [19]. The source codes and results of both eBAT and WebExplor are made publicly available [16].

Table I lists the benchmark projects used in literature [3], [9], sampled from the most popular web frameworks based on GitHub stars. To align with the baseline approach [9] and remove any statistical bias, both approaches were run 15 times with a defined time budget of 30 minutes and a state abstraction threshold of 0.8. Functionally duplicate failures were filtered manually to get the unique number of failures, e.g., in Splittypie, the same errors caused by separate transaction entities were ignored. The evaluation was carried out with the help of the following questions:

TABLE I: Comparison of failure detection effectiveness. (Values in bold and star indicate best average results along with standard deviations over 15 trials and statistically significant differences respectively)

Projects	Unique Failures (#)		Code Coverage (%)	
	eBAT	WebExplor	eBAT	WebExplor
Dimeshift	6.9 (1.3)	**8.3** (1.4)	**0.9*** (0.6)	0.3 (1.4)
Pagekit	**10.2*** (4.6)	7.5 (1.2)	**90.2** (1.1)	89 (8.8)
Splittypie	**20.6*** (3.4)	6.3 (0.7)	**65.1*** (0.7)	57 (0.04)
Phoenix	**3.9** (2.5)	3.2 (0.4)	50.4 (3.3)	**50.9** (0.04)
Retroboard	**22.5*** (4.4)	5.3 (0.7)	**51.7** (7.1)	51.3 (9.1)
Petclinic	**43*** (9.9)	18.2 (1.8)	**45.8*** (0.3)	41.2 (1.2)
Average	17.8	8.1	50.7	48.3

Effectiveness: *How effective is eBAT in terms of failure detection in web applications?* The effectiveness of automated web testing is measured by the unique number of failures detected and code coverage [3], [9]. Table I shows the comparison in which eBAT significantly outperforms WebExplor in 4 out of 6 projects separately using Mann-Whitney U test at 0.05 confidence level (similar to [3], [9]).

Failures located in deeply nested pages in these projects are exploited by eBAT through non-redundant exploration and proper form submission. For instance, in Petclinic, exploiting the forms (add/edit pets and add/edit visits) in the deeply nested *owner-detail* page. Detecting such failures is difficult for WebExplor due to the randomness in the action selection policy, even with high-level DFA guidance. An exceptional case is observed in Dimeshift where WebExplor is able to detect "waking up" script errors due to redundant navigation to *wallet-detail* page. As shown in Table I, WebExplor attains more code coverage in Phoenix whereas eBAT fails to explore them focusing on depth-first traversal. However, eBAT can detect more failures due to its behavior-driven action selection strategy as failure detection is not dependent on code coverage [9]. Therefore, eBAT is evaluated to be more effective in terms of failure detection compared to the baseline.

Efficiency: *How efficient is eBAT in exploiting web applications?* The efficiency is measured in terms of failure detection rate similar to [9]. The plots in Fig. 3 demonstrate the comparison in terms of efficiency where the x-axis indicates the time elapsed and the y-axis indicates the number of unique failures detected. Based on the plots, eBAT outperforms WebExplor in 5 out of 6 projects within 5 minutes of execution. An early jump in discovery rate is seen within 5 minutes in Pagekit, Petclinic, Splittypie, and Retroboard (as shown in Fig. 3) as they contain deeply nested pages. Proper form submission saves time by avoiding unnecessary execution resulting in the faster discovery of nested update functionalities in Trello. Errors in Splittypie are discovered only after the creation of a transaction event and navigating to its details.

WebExplor fails to exploit these scenarios in most cases because - i) proper form submission gets interrupted by other actions, and ii) selects actions that cause redundant

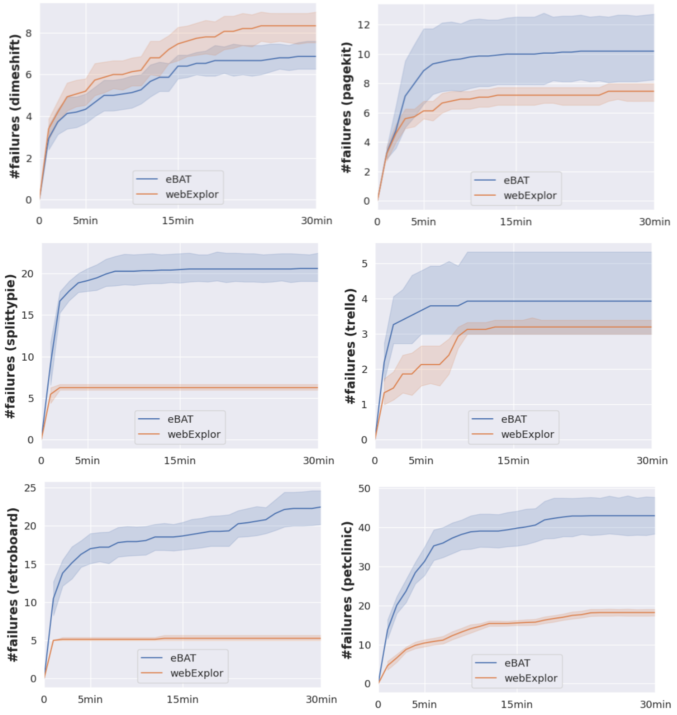


Fig. 3: Comparison of failure detection rate. Shaded areas represent lower and upper bounds along with their average

exploration, even after the transition to less-visited areas using DFA guidance. Therefore, eBAT is more efficient to discover failures, employing the design decision of redundancy reduction, behavioral grouping, and tree-based exploration.

V. Threats to Validity

Evaluation on a limited number of projects (taken from prior work [3]) poses an *external validity* threat as results may vary based on projects. To mitigate this threat, open source projects from the top 6 popular frontend frameworks are taken for generalizability. The selected behaviors may have some confounding variables that are not considered in this preliminary study, affecting the *internal validity*. However, these patterns are identified from popular open-source projects to remove developer-specific practices. The most common behaviors are considered in this paper but there is scope to incorporate other behaviors that may change the results. The replication package [16] is provided for evaluation reproducibility.

VI. Conclusion and Future Work

The approach presented in this paper, eBAT, bridges the gap in automated testing by incorporating the tester's behavioral decisions taken during manual testing, such as non-redundant exploration and grouped action interaction. This devised action selection strategy executes non-redundant actions to generate the execution trace. Redundancy reduction and behavioral grouping of interdependent actions significantly improve failure detection against the state-of-the-art approach. Results show that even simple behavior-driven decisions significantly outperform the current state-of-the-art in the effectiveness and efficiency of failure detection. Thus, focusing on enriching the set of tester's behavior rather than metric-based strategies can make failure detection more efficient. Our future plan includes the investigation of factors that affect approaches focusing on metric maximization against human-level decisions.

Acknowledgment

This study is supported by the Fellowship from ICT Division, Government of Bangladesh – No.: 56.00.0000.052.33.002.22-42, date: 12.06.2022

References

[1] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2007, pp. 291–305.

[2] A. M. Fard and A. Mesbah, "Feedback-directed exploration of web applications to derive test models." in *ISSRE*, vol. 13, 2013, pp. 278–287.

[3] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 142–153.

[4] M. Biagiola, F. Ricca, and P. Tonella, "Search based path and input data generation for web application testing," in *International Symposium on Search Based Software Engineering*. Springer, 2017, pp. 18–32.

[5] A. Mesbah and A. Van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 210–220.

[6] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah, "Leveraging existing tests in automated test generation for web applications," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 67–78.

[7] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 881–891.

[8] "Crawljax," 2017. [Online]. Available: https://github.com/crawljax/crawljax

[9] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 423–435.

[10] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 571–579.

[11] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Approaches and tools for automated end-to-end web testing," in *Advances in Computers*. Elsevier, 2016, vol. 101, pp. 193–237.

[12] S. E. Sprenkle, L. L. Pollock, and L. M. Simko, "Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour," *Software Testing, Verification and Reliability*, vol. 23, no. 6, pp. 439–464, 2013.

[13] T. E. Vos, P. Aho, F. Pastor Ricos, O. Rodriguez-Valdes, and A. Mulders, "testar–scriptless testing through graphical user interface," *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1771, 2021.

[14] R. Yandrapally, A. Stocco, and A. Mesbah, "Near-duplicate detection in web app model inference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 186–197.

[15] F. S. Ocariza, K. Pattabiraman, and A. Mesbah, "Detecting unknown inconsistencies in web applications," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 566–577.

[16] ebat, "ebat replication package," 2022. [Online]. Available: 10.6084/m9.figshare.19906264.v1

[17] mozilla.org, "The input (form input) element," 1998. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input

[18] A. V. Singh and A. M. Vikas, "A review of web crawler algorithms," *International Journal of Computer Science & Information Technologies*, vol. 5, no. 5, pp. 6689–6691, 2014.

[19] Google, "Puppeteer — tools for web developers," 2010. [Online]. Available: https://developers.google.com/web/tools/puppeteer