Enhancing the Resiliency of Automated Web Tests with Natural Language

Maroun Ayli
Center for Computer Science, Saint Joseph University of Beirut
Lebanon
maroun.ayli1@usj.edu.lb

Youssef Bakouny
Center for Computer Science, Saint Joseph University of Beirut
Lebanon
youssef.bakouny@usj.edu.lb

Nader Jalloul
Murex
France
njalloul@murex.com

Rima Kilany
Center for Computer Science, Saint Joseph University of Beirut
Lebanon
rima.kilany@usj.edu.lb

Abstract

Web application testing has traditionally been the domain of specialized software professionals, with a significant portion still relying on manual execution by individuals with limited programming expertise. This research introduces a novel proof-of-concept tool that democratizes the creation of automated web tests through a restricted natural language interface. Leveraging the GPT-4 language model and introducing "smart web element locators, our tool enables both technical and non-technical professionals to design comprehensive test cases without explicit programming knowledge. The tool comprises three key components: (1) a pseudo-language definition mapping into Selenium actions, (2) a new concept for resilient locators, and (3) a semantic understanding system translating natural language into software assertions. This approach enhances test production speed by offering a no-code interface and improves test resiliency through non-fragile locators. While introducing a slight increase in execution time (approximately 15% on average), the benefits in creation speed and script resilience far outweigh this trade-off. Empirical evaluation demonstrates the tool's effectiveness in real-world scenarios, enabling non-technical team members to contribute meaningfully to the testing process. Results show a significant reduction in test suite creation and maintenance time, as well as improved test coverage due to increased participation of domain experts. This research contributes to software testing by combining natural language processing, smart locators, and automated test generation. By lowering the barrier to entry for test automation, our tool has the potential to revolutionize web application testing practices, leading to more efficient, comprehensive, and reliable testing processes across the software development industry.

AI2A 2024, September 27–29, 2024, Shanghai, China

ACM ISBN 979-8-4007-1784-0/24/09
https://doi.org/10.1145/3700523.3700536

CCS Concepts

• **Software and its engineering** → **Empirical software validation**; • **Information systems** → **Web applications**; **Web interfaces**; • **Computing methodologies** → **Natural language processing**; • **Human-centered computing** → *Graphical user interfaces.*

Keywords

Web automation testing, Restricted natural language, Test case generation, Web application testing, Natural language processing in testing, Automated test scripts, Behavior-driven development, Web UI testing, Test automation frameworks, Domain-specific language for testing

ACM Reference Format:
Maroun Ayli, Youssef Bakouny, Nader Jalloul, and Rima Kilany. 2024. Enhancing the Resiliency of Automated Web Tests with Natural Language. In *2024 4th International Conference on Artificial Intelligence, Automation and Algorithms (AI2A 2024), September 27–29, 2024, Shanghai, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3700523.3700536

1 Introduction

The web industry is changing rapidly, with ongoing improvements in rendering techniques, web frameworks, database systems, architecture, and UI testing. Many studies are published each year on web testing to improve the efficiency, quality, and effectiveness of web UI tests. While the software industry is a major source of innovation in web testing, academic research sometimes overlooks its contributions.

Recently, there have been advances in web testing where companies are using Artificial Intelligence (AI) to achieve more effective testing. Some features in these applications include smart test creation using Natural Language Processing (NLP), self-fixing tests, and visual testing. Companies like Functionize [6], TestRigor [23], and Autify [30] are leading this effort. However, since these are for-profit companies, their algorithms are not open-source.

Much research has focused on using Machine Learning (ML) and AI for web testing. This research covers various areas: web navigation [9] [16], model inference [37], and image-based deep learning [5] are some of the methods proposed by researchers.

We believe that easy test creation and test durability are two key aspects of effective web testing. Our goal is to create a proof of concept for an algorithm that can help improve the durability of web automation tests. This algorithm should be developer-friendly, meaning it is easy to understand, implement, and cost-effective.

This paper will first explain locator-based web testing, then discuss the problems it faces, review current research on these issues, and finally present our proposed solution.

2 State of the Art

There are many web testing tools available in both open source and commercial domains. We will group some of the most popular tools into three categories: script-based tools, capture and replay tools, and AI-based tools.

2.1 Script-Based Web Testing

Besides manual testing, script-based testing is a standard technique for testing web applications. Developers write automation scripts that run and perform actions and checks on the application under test (AUT). One of the most popular tools for script-based testing is Selenium [28], a cross-platform tool that can test on almost all browsers, is open source, and is available in many popular programming languages. There are several alternatives to Selenium, such as Cypress, TestCafe, Puppeteer, Playwright, and JMeter.

To interact with web elements, scripts use locators. A locator is a data point that allows the script or webdriver to find the needed element. Examples of locators include ID, Name, ClassName, CSS Selector, and XPATH. In automation scripts, whether using Selenium or other frameworks and libraries, locators are essential for reaching elements.

2.1.1 Problems with Locator-Based Web Testing. The locator approach has several well-documented problems. Biagiola et al. [2] highlight the need for generating robust locators. Stocco et al. [34] present a design pattern for web tests where locators can be easily changed if needed. Nass et al. [21] present a detailed survey of web testing challenges. The problem of "Application changes breaking test execution" is recurring in publications from 2001 to 2019. Ricca et al. [33] name "the fragility problem" a fundamental issue.

The main problem with locators is their susceptibility to change. Class locators might change if a developer alters the CSS related to an element. The XPATH of an element might change if any change occurs in the DOM tree. ID and name locators might also change depending on the developers' needs. No locator is guaranteed to remain unchanged when the code changes, requiring time-consuming updates by developers.

2.1.2 Single Locator Algorithms. Many publications have addressed the problem of web test fragility/robustness. Some research has focused on generating more robust locators. Montoto et al. [18] aim to provide an algorithm for generating resilient XPATHs. Leotta et al. propose an algorithm named Robula+ [13] to generate robust XPATH expressions. Robula+ is a well-regarded state-of-the-art algorithm with a detailed published description and a JavaScript implementation [31].

2.1.3 MultiLocator Solutions. Some approaches handle locator fragility by relying on multiple locators. For example, during test recording, the software could register all locators pointing to the interacted element. Leotta et al. [12] present a multi-locator solution combined with a voting procedure. Multiple locators are generated for the same elements by six different algorithms, with a voting procedure to choose the best locator. Nass et al. [20] published a new algorithm called Similo, which stands for Similarity-based web element localization. Similo aims to provide robust testing when updating website versions by quantifying the similarity between multiple attributes of candidate web elements and a target element.

2.2 Capture and Replay Tools

Capture and replay programs help with test generation by recording user interactions with the website. The recording software saves every interaction, such as mouse movements, widget clicks, and text input. It can then replay the exact sequence of actions later. This approach faces similar problems to manually written tests, as interface changes still require maintenance or re-recording of tests. Another issue is the absence of oracles; a correct execution in capture and replay tools is often defined as one that ends without errors, which may not be sufficient. Some industrial capture and replay tools include SeleniumIDE [28], Squish [29], Rapise [27], Ranorex [26], and AutoIT [24]. Academic publications like WATER [4], VISTA [35], and SITAR [7] have addressed capture and replay problems, but these tools have not been widely adopted in industry. According to Leotta et al. [11], manually written scripts are easier to maintain than recorded ones.

2.3 AI-Based Tools

With the recent surge in AI companies, AI has been applied to web testing. Several companies advertise "AI-based web testing" with advanced features such as:

- Self-healing web tests: minor interface changes do not break the tests.
- Natural language test creation: web tests can be created using plain English.
- Integration with CI pipelines
- Data-driven testing

Companies like Functionize [6], TestRigor [23], and Autify [30] offer such features. However, their algorithms are not open-source, and it remains unclear how much of their work is truly based on ML and AI algorithms. This work aims to explore their potential inner workings or achieve similar features with open-source work.

2.4 Concerns with Visual-Based Testing

Recent advancements in computer vision have led to its application in UI testing. Publications such as [8], [5], [36], and [15] have tackled this problem with some success. Visual testing shows promise in accurately finding elements to interact with. However, several concerns arise when using these techniques.

2.4.1 Visual Regression Testing. Visual Regression Testing (VRT) involves capturing baseline screenshots of the original web page and comparing them to screenshots of updated versions. This process uses traditional comparison algorithms or deep learning approaches to detect changes in visual elements like color schemes, layout, typography, and alignment. VRT is technology-independent and

can be applied to GUIs or web components rendered in various technologies. However, it can produce false positives or negatives, which is common in image similarity comparisons. Companies like Percy (part of BrowserStack) offer VRT services.

2.4.2 Headless Tests. Continuous integration and delivery (CI/CD) pipelines are essential for faster time to market, improved quality, and better reliability. These pipelines often run on remote machines without display servers, and web UI tests are executed in headless mode. Headless mode offers advantages like faster test execution, reduced resource consumption, and easier integration for continuous integration pipelines. However, visual tests require website rendering and display, which introduces two major drawbacks:

- Setting up the pipeline's execution environment to have a display server can be costly, complex, and introduce significant computational overhead.
- Reduced performance due to the need for rendering the website to apply visual tests.

While visual testing may help in finding elements, it risks over-engineering a solution that could be solved with less costly computations.

3 Proposed Approach: Smart Locators

Our goal is to move away from the concept of hard-coded locators. Standard locators are fixed representations of elements in the DOM. We aim to represent elements using natural language descriptions, which can be more robust than traditional locators. For example, a test looking for "the red sign in button" should not fail when the button moves slightly or when the text inside the button changes to a similar meaning, e.g., "The login button". To achieve this, we reimagine tests as a sequence of natural language sentences composed of actions, descriptions, and values.

3.1 Why Restrict Natural Language?

Mapping natural language to software is a challenging problem, partly due to the ambiguity of natural languages [1]. A restricted language can help reduce the complexity of mapping natural language into computation instructions, as detailed in [19]. The number of actions that can be executed on a web interface is limited (clicking an element, sending keys, hovering over an element, etc.). A small set of tokens is sufficient to cover all possible actions. An explicit mapping between basic natural language instructions and web actions can be achieved through simple manual configuration.

3.2 Test Structure

We define the natural language test as a sequence of well-formed sentences. Each sentence consists of three parts: an action to be executed, a description of the element to be accessed, and an optional value to be used for input actions. For example:

```
Type smartwatch in the searchbox
Press the searchbox button
Press the applewatch label
```

The UI tool guides the tester in selecting the actions they need and generates the natural language sentences required to automate the test.

3.3 A New Perspective?

3.3.1 Why Locators Might Not Be the Ideal Solution. This section discusses the motivation for a new approach to web testing that presents a radical shift in how we think about web locators and how we can maximize their usefulness when designing a web test. Locators are optimal for machine-to-machine or software-to-software communication. For example, when using Selenium to find an element with a specific XPATH, the browser driver will try to find an element that exactly matches the XPATH and, if present, will always return it. The statement "locators are fragile" should be rephrased as: "locators are fragile in their representation of a web element when they are not updated in synchronization with the change of the corresponding web element". A locator's function is to tell the script the location of the element on a web page. However, that location is statically persisted and not updated in real-time. When the website evolves and elements change, the locator at time t_0 may no longer point to the correct element. In its current form, locators will always need to be updated to remain correct across versions, regardless of how smart the algorithm (such as Robula+) is.

3.3.2 A New Usage for Locators. Consider a web element as a person and the XPATH of an element as coordinates of their address on a map. Selenium WebDriver would simulate mapping software that gives directions to the person's address. Our end goal is to find the person. Using static locators would be like using the person's address to find them, which works well until the person changes their address. Instead of collecting the address, which is not directly linked to the essence of the person, we should collect data points that help us find the person in a more resilient way. The question becomes: **what other representations can we use for a web element so that we can identify it without relying on the notion of a locator?** Whatever changes are applied to a web element, it should remain semantically equivalent (e.g., a submission button should remain a submission button regardless of style, position, or bound JavaScript function). If it becomes semantically different, the test should fail because the meaning itself has changed, and the test **must** be updated.

3.4 Towards Natural Language Descriptions

Locators are a major barrier to achieving natural language testing. If we still need to specify locators in tests, non-technical professionals will have difficulty creating tests. Natural language descriptions provide a universal interface. The problem becomes: how can we map a natural language description to a locator that can be used by a webdriver?

3.4.1 Intuition. A natural language description is a new type of locator. It is not a simple string that tells the browser what to look for. Natural language descriptions are inspired by human intuition when navigating a website. An average web user can create an account and log in on a website relatively easily without guidance because they can easily identify what to click and interact with. The visual aspect of websites is intuitive for humans but not for machines. Image-based reinforcement learning has been attempted in automated testing by Eskonen et al. [5]. However, we believe that image processing techniques for UIs are best used when access

to the DOM tree is not allowed or flawed, such as in the case of embedded GUIs inside a web application like Webswing.

3.4.2 Attribute Extraction. The natural language description provided by the tester is converted into a key-value collection which will be used by the algorithm. We use OpenAI's Text Completion API [3], one of the most advanced NLP tools powering GPT-4. For all descriptions, we feed the text completion API the same question: *"Write me a Python dictionary of HTML attributes that corresponds to the following description (description). If you can't find any, give me an empty string"*. This helps us convert a natural language description into a dictionary of attributes that we can use to find the needed HTML element. For example:

```
The green submit button at the bottom of the screen ->
 {
 "type": "submit",
 "value": "Submit",
 "style": "background-color:green,
 position:fixed; bottom:0;"
 }
```

Since OpenAI can generate code, it has knowledge of HTML attributes and can help generate high-quality attributes.

4 Smart Web Element Locators

A locator based on natural language descriptions is the missing piece of the puzzle that allows the creation of natural language tests. Using standard locators, it is very difficult for a non-technical tester to communicate with a tool to specify which element to interact with.

```
Click on

//div[@class='container']//a[@href='/about-us']
and check if

//td[contains(text(), 'John')]}
```

At this point, the tester might as well write the test themselves.

The primary aim of the algorithm is to find elements on the web page without using locators. We thus need to **search** for the elements. The input for the algorithm is a multidimensional vector of data points that the test creator **supposes are true** about the element. We are not trying to exactly match the element based on the input vector, but instead, we're trying to find the most probable element. Similar to the work by Nass et al. [20], we aim to give each candidate element on the page a similarity score. The probability of likelihood is directly proportional to the similarity score. The similarity score of elements will depend on the following:

- textual and semantic similarity of attributes
- tag similarity
- textual and semantic similarity of element value

This calculation is sufficient to give the desired element the highest similarity to the description. A well-designed website should be intuitive enough for a human to navigate without much guidance, which means that in practice, elements are distinguishable to a degree by basic assumptions. The distribution of similarity should be rather uniform and peak only at the highest likely elements (ideally one). The following paragraph describes the algorithm in detail.

4.1 Algorithm explanation

Algorithm 1 Smart Web Element Location

procedure ComputeSim($desc, elt$)
  $sim \leftarrow 0$
  $sim \leftarrow sim + T(desc_{tag}, elt_{tag})$
  $sim \leftarrow sim + \alpha_1.FuzzyMatch(desc_{value}, elt_{value})$
  $sim \leftarrow sim + \alpha_2.SemanticSim(desc_{value}, elt_{value})$
  for $attr$ in elt_{attrs} **do**
    $sim \leftarrow sim + \alpha_3.FuzzyMatch(desc_{attr}, elt_{attr})$
    $sim \leftarrow sim + \alpha_4.SemanticSim(desc_{attr}, elt_{attr})$
  end for
  return sim
end procedure
procedure Locate($desc, src$) ▷ src is the source HTML code
  $elem \leftarrow None$ ▷ elem is the highest likely element
  $maxsim \leftarrow -\infty$
  for elt_i in E **do**
    $sim \leftarrow$ ComputeSim($desc, elt_i$)
    if test **then**
      $maxsim \leftarrow sim$
      $elem \leftarrow elt_i$
    end if
  end for
  return $elem$
end procedure

T is the element similarity matrix. The developer might not guess the tag of the element correctly every time. However, they can guess what the element might be. The tag similarity matrix T is a way to tell the algorithm how close tags are to each other. For example, one could say that buttons are 60% similar to links. These values vary per domain and how developers write their UI, and should be tuned according to needs. $FuzzyMatch(a, b)$ is a collection of fuzzy matching functions implemented in the FuzzyWuzzy Python library [25]. This module is based on the Levenshtein distance [14]. The Levenshtein distance, also known as the edit distance, can help the algorithm find values which are syntactically close to the one provided in the description. In particular, we are using the *token_set_ratio* function of fuzzywuzzy.

$SemanticSim(a, b)$ is the semantic similarity between two strings. Currently, we are using BERT by Reimers et al. [32]. The idea is to use sentence and word similarity to best describe an element in a more probabilistic way. Suppose we are clicking on a button with a value "Submit". We would want our locator to be smart enough to understand that the button is about "submission" or "sending" or "finalizing". This allows us to still find the element even if its content has been changed. Instead of looking for an element with value "Submit", the developer can write something like "element that is about sending the data" in the description. Since the elements on a web page have more or less different meanings, the distribution of semantic similarities should be broad enough to highlight the most likely element with a generally larger than normal value.

4.2 Tuning the parameters

As mentioned earlier, the similarity score calculation depends on both semantic and syntactic similarity between the description and each element. Placing too much emphasis on either type of similarity could lead to mistakes. Take the following example: A web page contains a sign-in button and a create account button. The tester provides the description, "Click the sign-up button". If the weight of the syntactic similarity is large compared to the semantic similarity, the highest likely element will be the sign-in button, which is a miss for the algorithm. Semantic similarity may not be entirely accurate as well, as the number of tokens involved in web elements is rather small, especially with interactable elements like buttons, links, and inputs. Generalizing the weights for the general case might not be ideal. Tuning the weights would yield the best result when looking at the problem from an organizational perspective, with a solid set of practices to build around.

5 Evaluation and Results

To evaluate the tool comprehensively, we specifically examine crucial aspects such as the simplicity of creating test cases, the time required for test executions, the distribution of scores across multiple runs, the tool's resilience against changes in websites (structural, syntactical, semantic), and finally, the ease of integrating this tool into an existing automation pipeline.

5.1 Ease of test creation

Since the tests are created via a simple UI, and the elements to be accessed are described in natural language, the tests are much faster to create. They can also be used by non-technical professionals who can simply write the test by translating their mental scenarios directly into the UI. It is difficult to determine the average time it takes to create Selenium scripts due to many parameters such as the complexity of the task and number of elements to interact with. Measuring the time it takes to write the part of the script that identifies a single element is easier to do, and our experience shows that using a smart locator for test creation is at least twice as fast as a regular locator approach due to the fact that we do not have to look for the locators using browser extensions.

5.2 Test execution time

It is undeniable that the smart locator approach introduces an overhead in terms of executions. The advantage, however, in web testing is that the bottleneck usually comes from the fact that we need to wait for elements to load and not locating the elements once they are loaded. In our study, we looked at pages with up to 5000 elements and saw that the smart locator approach introduced a minimum of 5% and up to 20% additional execution time. Considering the speed-up in creation and the test resiliency against website changes, we accept this compromise. Table 1 presents the overhead of the execution of the tests introduced by the smart locators.

5.3 Accuracy

The smart locator algorithm is designed on the premise that elements should not be difficult to find, due to the fact that web pages are fairly intuitive for humans to navigate. For that reason, we conjecture that for each element we are trying to find, there should be a very small (ideally unique) number of elements for which the similarity score is greater than the rest by a significant margin. We ran the experiment of trying to locate elements that were required during multiple tests on different popular websites (Twitter, LeetCode, Discord, Amazon, Wikipedia and more). A simple algorithm is sufficient to locate elements on the web page. The algorithm is thus locator independent and will work on any version of the website that still contains a valid representation based on the description. Minor changes in CSS and in XPath, and even in ID and name will not have a significant impact on the similarity value between the description and the desired element. Should the similarity score for a certain description be very low for all the elements, then the element is considered not found, and the test failure is in this case desired as a significant change has occurred.

If our reasoning about the nature of websites is correct, then when we describe an element using natural language, the similarity score of that same element on the page should be very high and the rest of the elements should have a drastically low score in comparison. We ran the experiment on finding elements on different websites. We calculated the mean μ, the median Med, the difference between the mean and the median $d_{\mu,Med}$ and the ratio between the maximum score and the second maximum score $ratio_{max1,max2}$ calculated by the formula : $\frac{(max1-max2)}{max2}$.

We ran multiple experiments of trying to find web elements using their natural language description. For each run, we calculate the similarity scores for each element in the web page. The resulting similarity score distribution indeed conforms to our assumption: mostly uniform with low values with very few outliers, and one outlier that is larger than the rest of the outliers. The maximum similarity score corresponds to the element we're trying to find. Table 2 reflects those results: The average is significantly higher than the median in most distributions, and the ratio between the maximum and the second maximum is very large, indicating the uniqueness of the maximum value (the element we're trying to find).

5.4 Healing and Adaptability

The strength of our tool is in the independence of locators and the ability to semantically understand content based on pretrained language models. To demonstrate the healing potential that we have, we are comparing our smart locator approach with standard selenium scripts. We will take template websites[1], apply some modifications and see if the scripts still hold after the changes.

Indeed, table 3 clearly shows that UI tests that use smart locators are significantly more resistant to shallow modifications of the UI interface, while traditional XPATH-based locators require significantly more maintenance even after trivial modifications. Smart locators also avoid false positives by failing if the similarity score is too low.

5.5 From an academic standpoint

To the best of our knowledge, this work is the first that tackles the full pipeline of generating web tests from natural language. It is important nonetheless to highlight some of the related work that can be used to potentially improve our tool. The paper by

[1]Websites A, B, C are template websites downloaded from free-css.com

Table 1: Speed of execution of the smart locator algorithm across different websites

Website	element location (s)	No. elements	Additional Time Overhead
Amazon	3.83	3450	26%
Chess.com	0.80	143	6%
Leetcode	0.55	620	12%
Wikipedia	0.23	953	5%
Discord	0.92	650	10%
Twitter	0.96	1450	12%

Table 2: Statistics about the distribution of similarity scores

Website	*Element*	μ	*Med*	$d_{\mu,Med}$	$ratio_{max1,max2}$
Amazon	The search bar	0.68	0.20	0.48	422.23
Leetcode	The sign-in button	6.92	0.20	6.72	20.48
Discord	The sign-up button	10.07	0.20	9.87	178.88
Wikipedia	The Wikibooks links	10.26	0.20	10.06	57.60
Wikipedia	The search bar	10.30	0.20	10.10	57.60
Twitter	Sign Up with Google	16.10	1.00	15.10	718.38
Chess.com	Play Online	16.36	0.50	15.86	93.13
Chess.com	Play as Guest	15.93	0.50	15.43	2.82

Table 3: Test Robustness Comparison Between Traditional Locators and Smart Locators

Tool	Site	Version	Locator	Success	Type of Change
Selenium	A	1.0	XPATH	PASS	N/A
SL	A	1.0	Smart Locator	PASS	N/A
Selenium	A	2.0	XPATH	FAIL	Changed input placeholder
SL	A	2.0	Smart Locator	PASS	Changed input placeholder
Selenium	A	3.0	CSS-Selector	FAIL	Restyled buttons
SL	A	3.0	Smart Locator	PASS	Restyled buttons
Selenium	B	1.0	XPATH	PASS	N/A
SL	B	1.0	Smart Locator	PASS	N/A
Selenium	B	2.0	CSS-Selector	FAIL	Changed form layout
SL	B	2.0	Smart Locator	PASS	Changed form layout
Selenium	B	3.0	XPATH	FAIL	Altered elements text
SL	B	3.0	Smart Locator	PASS	Altered elements text
Selenium	C	1.0	XPATH	PASS	N/A
SL	C	1.0	Smart Locator	PASS	N/A
Selenium	C	2.0	CSS-Selector	FAIL	Added dropdown menu
SL	C	2.0	Smart Locator	PASS	Added dropdown menu
Selenium	C	3.0	XPATH	FAIL	Removed search bar
SL	C	3.0	Smart Locator	PASS	Removed search bar
Selenium	C	4.0	CSS-Selector	FAIL	Altered header style
SL	C	4.0	Smart Locator	PASS	Altered header style
SL	C	5.0	name and ID	FAIL	Major overhaul
SL	C	5.0	Smart Locator	FAIL	Major overhaul

Kirinuki et al. [10] presents a step towards script-free testing by using NLP and heuristic search algorithms to identify web elements and determine test procedures from test cases written in a domain-specific language, achieving a 94% success rate in identifying web elements in two open-source web applications. The work by Long et al. [17] presents a robust record and play tool named WebRR which extracts many data points about web elements during the recording phase of information about the web elements to improve the robustness of the test cases. Additional data can be fed into our algorithm using WebRR to build a description on the go by recording the script.

5.6 From an industrial standpoint

To the best of our knowledge, this tool is the first open source tool that covers the full pipeline of generating web tests via a user-friendly interface. This tool resembles TestRigor and Functionize the most in the way it presents the tester with a restricted natural language to create the tests. Quoting TestRigor's list of features:

- Usage of plain English to allow anyone to build/understand tests purely from end-user's point of view, avoiding relying on locators.
- Support for web testing on Desktop and Mobile almost all browsers on multiple OSes such as, for example, Internet Explorer on Windows and Safari on Mac and on iOS

And Functonize's:

- Create tests in minutes by writing in plain English.
- Visual Comparaison of elements on the website

6 Conclusion

This work is a proof of concept that showcases that we can achieve web UI testing using natural language descriptions and without using locators. This allows for quicker test creation, and the democratization of web tests creation, as well as more resilient scripts. Our tool is a web application written in React and Flask that takes sentences that describe the actions to be executed and outputs Selenium scripts which are enhanced using our Smart Web Element Locator module. A performance overhead of 5-10% is introduced on small pages with less than 1000 elements, and 20% for pages bigger than that. We believe that the minor loss in performance is overshadowed by the gain in test creation speed as the resiliency of scripts that are totally locator free. Code for replication is available at [22].

References

[1] Daniel M. Berry. 2008. Ambiguity in Natural Language Requirements Documents. In *Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs*, Barbara Paech and Craig Martell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–7.

[2] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. 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 (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 142–153. https://doi.org/10.1145/3338906.3338970

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[4] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. 2011. WATER: Web Application TEst Repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering* (Toronto, Ontario, Canada) *(ETSE '11)*. Association for Computing Machinery, New York, NY, USA, 24–29. https://doi.org/10.1145/2002931.2002935

[5] Juha Eskonen, Julen Kahles, and Joel Reijonen. 2020. Automating GUI Testing with Image-Based Deep Reinforcement Learning. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, -, 160–167. https://doi.org/10.1109/ACSOS49614.2020.00038

[6] Functionize. 2022. Agile Automation Testing Framework with machine learning. https://www.functionize.com/

[7] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M. Memon. 2016. SITAR: GUI Test Script Repair. *IEEE Transactions on Software Engineering* 42, 2 (2016), 170–186. https://doi.org/10.1109/TSE.2015.2454510

[8] Luke Harries, Rebekah Storan-Clarke, Timothy Chapman, Levent Ozgur, Shuktika Jain, Alex Leung, Steve Lim, Aaron Dietrich, José Miguel Hernández-Lobato, Tom Ellis, Cheng Zhang, and Kamil Ciosek. -. DRIFT: Deep Reinforcement Learning for Functional Software Testing. - -, - (-), 10.

[9] Sheng Jia, Jamie Kiros, and Jimmy Ba. 2019. DOM-Q-NET: Grounded RL on Structured Language. https://doi.org/10.48550/ARXIV.1902.07257

[10] Hiroyuki Kirinuki, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. 2021. NLP-assisted Web Element Identification Toward Script-free Testing. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 639–643. https://doi.org/10.1109/ICSME52107.2021.00072

[11] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2013. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, -, 272–281. https://doi.org/10.1109/WCRE.2013.6671302

[12] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Using Multi-Locators to Increase the Robustness of Web Test Cases. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, -, 1–10. https://doi.org/10.1109/ICST.2015.7102611

[13] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2016. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process* 28, 3 (2016), 177–204.

[14] V. I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (Feb. 1966), 707.

[15] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, -, 1070–1073. https://doi.org/10.1109/ASE.2019.00104

[16] Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. 2018. Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration. https://doi.org/10.48550/arXiv.1802.08802 arXiv:1802.08802 [cs].

[17] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. 2020. WebRR: Self-Replay Enhanced Robust Record/Replay for Web Application Testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1498–1508. https://doi.org/10.1145/3368089.3417069

[18] Paula Montoto, Alberto Pan, Juan Raposo, Fernando Bellas, and Javier López. 2011. Automated browsing in AJAX websites. *Data 'I&' Knowledge Engineering* 70, 3 (2011), 269–283. https://doi.org/10.1016/j.datak.2010.12.001

[19] Jesse Mu and Advait Sarkar. 2019. Do We Need Natural Language? Exploring Restricted Language Interfaces for Complex Domains. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI EA '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3290607.3312975

[20] Michel Nass, Emil Alégroth, Robert Feldt, Maurizio Leotta, and Filippo Ricca. 2022. Similarity-Based Web Element Localization for Robust Test Automation. *ACM Trans. Softw. Eng. Methodol.* -, - (nov 2022), –. https://doi.org/10.1145/3571855 Just Accepted.

[21] Michel Nass, Emil Alégroth, and Robert Feldt. 2021. Why many challenges with GUI test automation (will) remain. *Information and Software Technology* 138 (Oct. 2021), 106625. https://doi.org/10.1016/j.infsof.2021.106625

[22] Authors of this paper. 2024. Smart Web Locators. https://anonymous.4open.science/r/ACCMI2024Submission-EE34/README.md.

[23] Online. 2016. Testing from end-user's perspective. https://app.testrigor.com/

[24] Online. 2022. *Autoit*. -. https://www.autoitscript.com/sites/autit

[25] Online. 2022. FuzzyWuzzy. https://pypi.org/project/fuzzywuzzy/

[26] Online. 2022. *Ranorex*. -. https://www.ranorex.com

[27] Online. 2022. *Rapise*. -. https://www.inflectra.com/Rapise

[28] Online. 2022. Selenium. https://www.seleniumhq.org

[29] Online. 2022. *Squish*. -. https://www.frologic.com/squish

[30] Online. 2023. autify. https://www.autify.com

[31] Andreas Plewnia. 2020. Robula+ JS implementation. https://github.com/cyluxx/robula-plus

[32] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, -. https://arxiv.org/abs/1908.10084

[33] Filippo Ricca, Maurizio Leotta, and Andrea Stocco. 2019. Chapter Three - Three Open Problems in the Context of E2E Web Testing and a Vision: NEONATE. In *Advances in Computers*, Atif M. Memon (Ed.). Vol. 113. Elsevier, -, 89–133. https://doi.org/10.1016/bs.adcom.2018.10.005

[34] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2017. APOGEN: automatic page object generator for web testing. *Software Quality Journal* 25, 3 (Sept. 2017), 1007–1039. https://doi.org/10.1007/s11219-016-9331-9

[35] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual Web Test Repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 503–514. https://doi.org/10.1145/3236024.3236063

[36] Faraz YazdaniBanafsheDaragh and Sam Malek. 2021. Deep GUI: Black-box GUI Input Generation with Deep Learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, -, 905–916. https://doi.org/10.1109/ASE51524.2021.9678778

[37] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. -, -, 423–435. https://doi.org/10.1109/ICSE43902.2021.00048 ISSN: 1558-1225.