VNU Journal of Science: Comp. Science & Com. Eng., Vol. 41, No. 1 (2025) 97–111

VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce

Original Article

An Efficient End-to-End User Interface Testing Method for Web Applications

Trinh Le-Khanh, An Pham-Hoang, Quyen Hoang-Van, Cong Bui-The, Tam Cao-Thi-Minh, Pham Ngoc Hung*

VNU University of Engineering and Technology, Hanoi, Vietnam

Received 18th September 2024
Revised 9th January 2025; Accepted 18th June 2025

Abstract: This paper proposes a novel end-to-end UI testing method by introducing a domain-specific language named User Behavior Language (UBL) for specifying UI test scenarios. The UBL provides a concise and intuitive syntax for describing UI interactions, enabling rapid test script creation and reducing the learning curve for non-technical testers. Additionally, UBL allows for the specification of constraints between interactions, facilitating the modeling of complex user workflows and enforcing application-specific rules. We also propose a test case generator that utilizes the UBL and the test data to automatically generate a comprehensive suite of test cases, covering both expected and abnormal user behavior. The effectiveness of the proposed UI testing method was validated through a comparative analysis with Katalon Studio by testing some functionalities of SauceDemo, a benchmark Website for UI testing.

Keywords: Domain-specific language, End-to-end UI testing, Test case generation, Web application testing.

1. Introduction

User interface (UI) testing is one of the major challenges in modern software development processes. It is a complex problem that requires significant effort and often depends on technologies provided by third parties. Nevertheless, interface testing is an indispensable requirement for every software company and project and is also one of the most time-consuming and labor-intensive tasks in the development process. Particularly in software evolution, interface testing needs to be performed repeatedly, further increasing the complexity and cost of this process. Therefore, developing and providing an effective tool for UI testing is an urgent need to optimize the development process and improve the quality of software products.

*Corresponding author.
E-mail address: hungpn@vnu.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.3629

End-to-end UI testing is employed to verify the seamless functionality of a web application throughout the user experience [1]. However, contemporary testing tools rely on manual execution, which requires testers to possess extensive prior knowledge of the application's structure and functionality [2]. As applications evolve and expand, the time and resources required for comprehensive manual testing can become prohibitively expensive. Moreover, the subjective nature of manual testing can lead to inconsistencies in test coverage and quality across different testers or testing cycles.

Various approaches and tools have been developed to streamline the UI testing process. One popular method is *model-based testing* [3], which involves creating a formal model of the UI to generate test cases automatically [4–7]. While this method can provide a comprehensive view of the application's interface, it often results in redundant test cases and an expansive set of states to be evaluated. The complexity of managing and interpreting these layout graphs outweighs their benefits, particularly for large-scale applications with dynamic interfaces. Another commonly used technique is *Record & Replay*, where testers manually interact with the UI to record their actions, which are then replayed to verify the application's behavior. By automating repetitive user interactions, these tools significantly enhance testing efficiency, reducing human error and accelerating development cycles. Some popular tools support this approach such as Cypress [8], Katalon Studio [9], Ranorex [10], or Selenium [11]. This technique is relatively easy and effective for simple UI testing scenarios. However, *Record & Replay* is limited in handling dynamic UI elements, complex workflows, and data-driven testing.

This paper proposes a novel end-to-end UI testing method for Web applications to address the limitations of existing UI testing approaches. The proposed method consists of two key components: (1) a domain-specific language (DSL) for test script specification and (2) a test cases generator. The proposed DSL provides a concise and intuitive way to describe UI test scripts. Each user interaction is represented by a 2-unit phrase: ⟨*action*⟩ ⟨*description*⟩. For example, "*Click add to cart the backpack*" describes a user clicking the *Add to Cart* button for a backpack on an e-commerce Website. This simple syntax facilitates rapid test script creation and reduces the learning curve for non-technical testers. Furthermore, the DSL allows for the specification of constraints between interactions, enabling the modeling of complex user workflows and enforcing application-specific rules. For instance, the constraint "*Login requires username and password*" ensures that users must enter both a username and password before successfully logging in.

The proposed method provides a test case generator to enhance test coverage and efficiency. Users input test data corresponding to each interactable UI element declared in the test script. Then, considering the specified constraints and the provided test data, the test cases generator generates a comprehensive suite of test cases that cover both expected and abnormal user behavior when interacting with the UI. Separating test scripts from test data reduces the effort required to create and maintain test cases. To validate the effectiveness and efficiency of the proposed UI testing method, we conducted a comparative analysis with Katalon Studio, a widely used commercial UI testing tool. The comparison focused on key aspects such as test script creation, test case generation, and learning curve.

The rest of this paper is organized as follows. Section 2 introduces the User Behavior Language (UBL) for specifying the UI test scenario. Section 3 presents the method for test case generation. Section 4 designs an experiment to prove the advantages of our proposed method. A comparison of learning time and test cases geneartion time is conducted in Section 5 to

validate the effectiveness and efficiency of the proposed method. The drawbacks and threats to validity are discussed in Section 6. Some related works are presented in Section 7 to compare our research. Finally, Section 8 concludes our research and discusses future directions.

2. The User Behaviors Language

This section outlines our methodological approach, which revolves around developing a domain-specific language (DSL) specifically tailored for the specification of user interactions within a web environment. These scenarios are described in an abstract manner, incorporating a predefined action and a corresponding word for the element's information. Figure 1 outlines our proposed approach for automated web UI testing, which consists of five main steps:

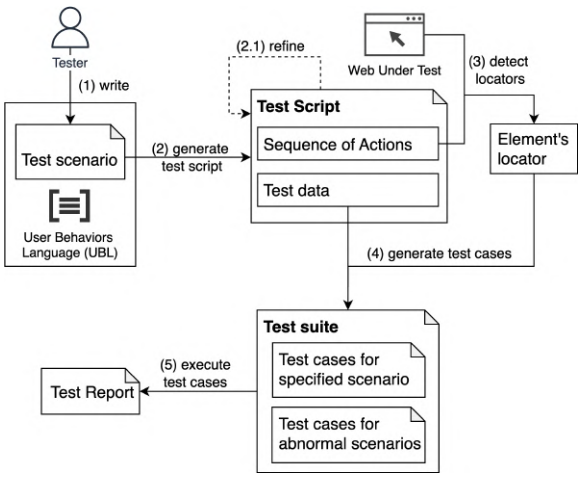


Figure 1. The proposed automated Web testing process.

1. **Test Scenario Creation**: Testers initially create UI test scenarios using the DSL named User Behavior Language (UBL). These scenarios provide a high-level description of the desired user interactions for specific functionalities.
2. **Test Script Generation**: The test scenario is subsequently transformed into a test script, which details the specific actions and data involved in the test. Testers can refine the test script as needed to ensure accuracy and completeness.
3. **Element Location Detection**: Once the test script is finalized, the element's locations on the Web Under Test are detected. This involves identifying the elements' unique identifiers (e.g., XPath) that correspond to the actions specified in the script. This information is crucial for automating the test execution.
4. **Test Case Generation**: Test cases are generated with the element locations and test data in place. These test cases represent individual scenarios that need to be tested, encompassing both normal and abnormal conditions.
5. **Test Case Execution and Report Generation**: The generated test cases are then executed against the Web Under Test, and the results are captured in a comprehensive test report. This report provides valuable insights into the quality of the web application and identifies any defects or issues that need to be addressed.

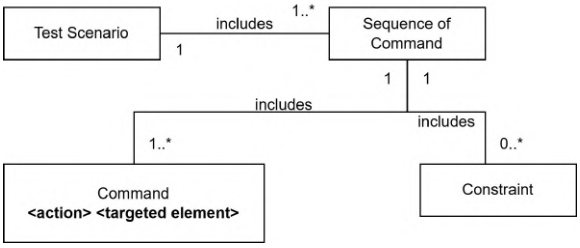


Figure 2. The metamodel of the UBL.

Figure 2 presents the metamodel to construct the UBL. This metamodel contains concepts and their relationships to specify test scenarios. In UI testing, each functionality is a specific aspect of the Web UI, such as form submission or page navigation, and can be represented by *Test scenarios*. These scenarios are created to verify

the corresponding expected user behaviors on the Web Under Test. Each scenario includes one or more *Sequences of Command (SoC)*, representing users' steps to express a functionality. Each SoC contains one or many *Commands* and might contain *Constraints*.

Table 1. Supported actions

Actions	Meaning
Open	Access a website via opening the link
Hover	Hover on an element
Fill	Enter text content into a text input or a text area element
Click	Click on a clickable element, such as button, link, or image
Select	Select an item from a list, such as checkbox, dropdown, or list
Verify	An assertion to check whether the current link or the elements' value appear as expected

A *Command* describes a specific action using two phrases: *action* and *targeted element*. The UBL supports several user *actions* to facilitate web automation (see Table 1). The "`Open`" action enables access to a website by opening the provided link. The "`Hover`" action allows positioning the cursor over a specified element without activating it. The "`Fill`" action permits entering textual content into a text input or text area element. The "`Click`" action activates clickable elements such as buttons, links, or images. The "`Select`" action facilitates choosing an item from a list, such as a checkbox, dropdown menu, or list. Furthermore, the "`Verify`" action asserts whether the current link or the value of specified elements appears as expected, enabling validation of the desired state.

When a *SoC* expression comprises more than one *Command*, these *Commands* are linked together through *Constraints*, which are binary logical characters, such as conjunction (&) or disjunction (|). In conjunction with *n Commands* within a SoC, a test template is generated to execute all *Commands* as expected (i.e., the Command's element receives the expected value from the input data set). The *Constraint* can not be used to link different SoCs together.

If the SoC represents a disjunction of *n Commands*, there are $n + 1$ generated test templates, including:

- n templates where each *Command* is executed as expected (i.e., the *Command* elements receive the anticipated value from the input data set), the remaining *Commands* are free.

- 1 template where all *Commands* are executed as expected.

The disjunction of *n Commands* produces $2^n - 1$ combinations. However, considering all combinations is unnecessary and would consume computational resources inefficiently. Thus, $n + 1$ test cases are appropriate, as they allow for a balance between achieving essential test cases and minimizing computational overhead. For example, consider the formula $A|B|C$. All possible combinations where at least one subset is valid, such as $A\&B$, $B\&C$, $C\&A$, must be considered to ensure their validity. However, validating A, B, and C simultaneously satisfies all these subset combinations, eliminating the need to check each subset individually and reducing computational overhead. The specification permits users to combine conjunction (&) and disjunction (|). To streamline evaluation, all formulas are converted to Disjunctive Normal Form (DNF), where the formula is expressed as a disjunction of conjunctive clauses. The test case generation process then accounts for $n + 1$ disjunction combinations, ensuring comprehensive coverage while optimizing efficiency.

3. UI Test Cases Generation

This section presents the method for generating UI test cases, encompassing specified scenarios that describe expected actions and abnormal scenarios that delineate potential abnormal actions of users.

3.1. Test Cases for Specified Action

In particular, if a user performs a specific action, the system should respond in a certain way. These responses could be in the form of data returned, changes in the system state, or even the triggering of certain events. The test cases are generated to verify these expected outcomes. After executing test cases, the actual outcomes are compared with the expected outcomes to determine if the system is functioning as intended. A test scenario containing k sequence of actions and having l test data will generate the number of test cases as equation (1):

$$\mathrm{T} = l \times \prod_{i=1}^{k} \mathrm{t}_i \tag{1}$$

Where t_i is the number of test templates for the i^{th} *SoA*.

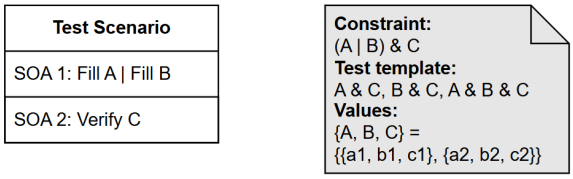
The algorithm 1 represents our method for generating executable test scripts from a specified set of actions and datasets. It iterates through each dataset, initializing an empty list, `listLineTemplates`, to store test templates for each line. Each line is treated as a boolean expression and is converted into Disjunctive Normal Form (DNF), which is a standard format consisting of a disjunction of conjunctions. Subsequently, each conjunction, representing a sequence of multiple actions, is transformed into a linear list of actions, with data for each action extracted directly from the dataset. This linear list of actions serves as a candidate template for the current line and is added to `currentLineTemplates`. If multiple conjunctions exist, the algorithm combines all corresponding linear lists into a single linear list and adds this combination to `currentLineTemplates`. Once all candidate templates for current line are generated, the `currentLineTemplates` is added to the `listLineTemplates`. Finally, a backtracking process is applied to `listLineTemplates` to generate all complete test templates, which are then transformed into executable test scripts.

Algorithm 1: Test cases generation algorithm for specified actions

```
Input: lines, dataSets
Output: testScript
testScript ← emptyString;
for dataSet in dataSets do
    listLineTemplates ← emptyList;
    for line in lines do
        currentLineTemplates ← emptyList;
        dnf ← convertToDNF(line);
        templateAllActions ← emptyList;
        for conjunction in dnf do
            template ← emptyList;
            for singleAction in conjunction do
                singleAction ←
                  replaceData(singleAction);
                template.add(singleAction);
                templateAllActions.add(singleAction);
            end
            currentLineTemplates.add(template);
        end
        if dnf.size() > 1 then
            currentLineTemplates.add(templateAllActions);
        end
        listLineTemplates.add(currentLineTemplates);
    end
    testCases ←
      createTestCases(listLineTemplates);
    currentTestScript ←
      createTestScript(testCases);
    testScript.append(currentTestScript);
end
return testScript;
```

Figure 3 illustrates an example of generating test cases following the given test scenario, which involves two lines: LINE 1 presents an expression that the user would enter a value to element A or element B, LINE 2 presents the expected result. There are three possible test templates

for the specified scenario, including: `Fill A & Assert C`, `Fill B & Assert C`, and `Fill A & Fill B & Assert C`. Then, six test cases are generated from three test templates and two sets of values.



Test case 1: Fill a1 Verify c1	**Test case 4:** Fill a2 Verify c2
Test case 2: Fill b1 Verify c1	**Test case 5:** Fill b2 Verify c2
Test case 3: Fill a1 Fill b1 Verify c1	**Test case 6**: Fill a2 Fill b2 Verify c2

Figure 3. Generated test cases for the specified scenario.

3.2. Test Cases for Abnormal Actions

This section presents our approach to determining and testing abnormal actions not performed as specified in the scenario. Algorithm 2 depicts the process of test templates generation for abnormal situations. The algorithm takes the targeted assertion block index as the input. For all assertion blocks prior to the targeted block, they will be added to the *validBlockList* for the valid templates generation process. The targeted block will be used as the input for the invalid templates generation process which will is described in Figure 3 and then both the *validTemplates* and *invalidTemplates* will be used as input for the templates combination process in Figure 4 and return the *finalTemplates*.

Algorithm 3 describes the algorithm for invalid templates generation, which takes the

Algorithm 2: Abnormal actions test templates generation

```
Data: assertionBlockList, targetedBlockIndex > 0
Result: finalTemplates
validBlocksList ← list;
targetedBlock ←
  assertionBlockList.get(targetedBlockIndex);
for i = 0 to targetedBlockIndex do
    validBlocksList ← assertionBlockList.get(i)
end
validTemplates ←
  validTemplatesGeneration(validBlocksList);
invalidTemplates ←
  invalidTemplatseGeneration(targetedBlock);
finalTemlates ←
  combineTemplates(validTemplates, invalidTemplates);
```

targetedBlock script as input and return the *invalidTemplates*. The algorithm loops through each line in the targeted block, then check if the line is an action line and not an assertion line. After which, the line will be hashed as ”invalid” so that in the data replacing phase, the system will know to replace invalid data for the line. Then, all other line get valid hashed as ”valid” for the data replacing phrase. The algorithm will loop until there are no action line left in the targeted block and then return the *invalidTemplates*.

Algorithm 3: Invalid templates generation

```
Data: targetedBlock
Result: invalidTemplates
invalidTemplates ← list;
for i = 0 to targetedBlock.size() do
    if isActionLine(targetedBlock.get(i)) then
        invalidTemplate ← list;
        invalidTemplate ←
          invalidHash(targetedBlock.get(i));
        for j = 0 to targetedBlock.size() do
            if j ≠ i then
                invalidTemplate ←
                  validHash(targetedBlock.get(j));
            end
        end
        invalidTemplates ← invalidTemplate;
    end
end
```

Algorithm 4 shows the algorithm for combining the *validTemplates* and

invalidTemplates to obtain the *finalTemplates* ready for data replacing. The algorithm loops through the list of valid templates and for each valid template, it will be matched against each of the invalid templates to create the final templates list. The number of final templates can be calculated by multiplying the number of valid template with the number of invalid templates.

Algorithm 4: Templates combine algorithm

```
Data: validTemplates, invalidTemplates
Result: finalTemplates
finalTemplates ← list;
for i = 0 to validTemplates.size() do
    for j = 0 to invalidTemplates.size() do
        finalTemplate ← list;
        finalTemplate ← validTemplates.get(i);
        finalTemplate ← invalidTemplates.get(j);
        finalTemplates ← finalTemplate
    end
end
```

So what is considered an Assertion Block in the test script. An Assertion Block is the combination of normal action lines and assertion lines with the condition that the assertion block must be started with normal action lines and end with an assertion line or a formula of assertion and there are no assertion line in between of the normal action lines. Figure 4 shows the examples for valid and invalid assertion blocks for better understanding of how we breakdown the test script for this section. An Assertion Block can be considered a subset of SoA as not all SoA end with or contain a assertion line.

4. Experiment

This section shows an example that illustrates the application of the proposed approach in UI testing for Web applications. The methodology employs a test scenario in the proposed UBL language. It also provides an end-to-end toolchain to facilitate users in crafting test scenarios, generating test cases, and executing them. The implementation process is structured into four phases, each contributing to the testing procedure:

- **Test scenario creation**: Testers describe the user actions on UI for a functionality using the UBL syntax.
- **Test data creation**: Defining expected values representing the anticipated outcomes or results the Web page should produce when executing the generated test cases.
- **Test case generation**: In this step, test cases are generated automatically by applying the methodology in Section 3. These test cases are derived from the instructions and specifications outlined in the scenario written in the UBL, ensuring thorough Web functionality coverage.
- **Test case execution**: The final step is to execute the generated test cases against the target web page. Currently, generated test cases are written in the Robot framework. However, test cases could be generated in any common UI testing framework.

4.1. Experimental Design

The experiment employed SauceDemo, a web application developed by SauceLabs, a provider of automation testing solutions. This platform served as an environment for practicing and evaluating browser automation techniques. The experiment was designed to assess the functionality of the *login* and *buy products* features.

Figure 5 illustrates the translation of user interactions into the test scenario written in the UBL language. The left image describes user interactions on the Web under test. The center images illustrate how those steps are described as commands in UBL. Writing the scenario in natural language commands significantly reduces the learning curve and composes scripts with minimal effort. Then, the proposed system

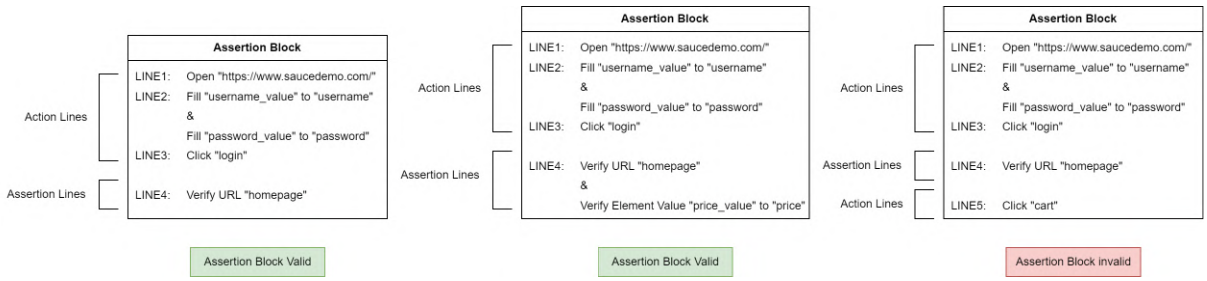


Figure 4. Examples for valid and invalid assertion blocks.

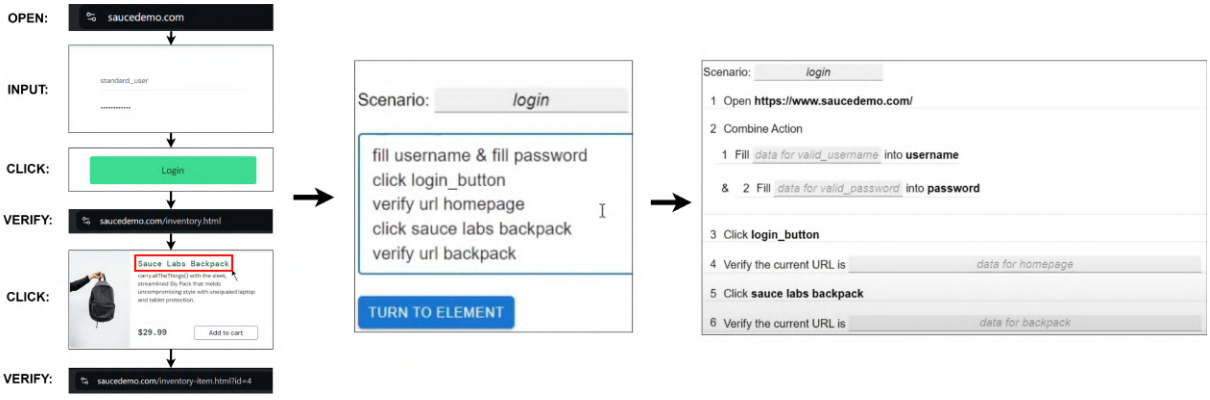


Figure 5. The translation of user interactions into the proposed UI test script.

converts the described scenario into an interactive UI, which allows users to define expected data values for UI elements. The right image in Figure 5 represents the interactive UI. The bold texts are elements used to identify corresponding element locators (e.g., XPath information), and text input boxes define the desired data values to be used during the automated testing process.

4.2. Experimental Result

4.2.1. Test Cases for the Specified Scenario

The experimental scenario consisted of six SoCs, which were processed to generate corresponding test templates as presented in Figure 6.

Each SoC template is combined to form a complete, end-to-end test template for the scenario. This combinatorial process uses the Multiplication Product Rule [12], a basic counting principle that takes each element in each

Test template lists
1. [Open "https://www.saucedemo.com/"]
2. [Fill "username_value" to "username" Fill "password_value" to "password"]
3. [Click "login"]
4. [Verify current URL is "homepage"]
5. [Click "sauce labs backpack"]
6. [Verify current URL is "backpack"]

Figure 6. Lists of test templates for each Sequence of Actions.

SoC and combines it with another SoC. Applying this rule in the current context creates a single test template illustrated in Figure 7.

Finally, a test suite was generated by integrating the generated test templates with each record within the dataset to validate the

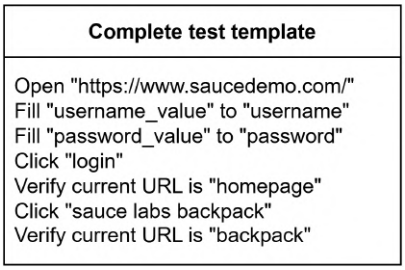


Figure 7. Complete test template.

basic flow of user behavior across the specified input scenarios. This test suite ensured that the application's functionality was robustly assessed.

4.2.2. Test Cases for the Abnormal Actions

This section describes the process of generating test cases for abnormal actions. This process consists of (1) generating test templates for preceding valid assertion blocks if they exist, (2) generating test templates for the invalid assertion blocks, and (3) combining the valid and invalid templates.

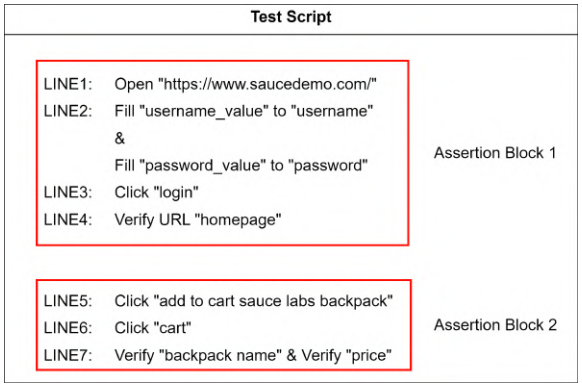


Figure 8. The two assertion blocks of the specified scenario.

Figure 8 illustrates two assertion blocks named `Assertion Block 1` and `Assertion Block 2`. As mentioned in Section 3.2, abnormal test templates were generated by negating each SoC within each assertion block. `Assertion Block 1`, with two SoCs at LINE2 and LINE3, produced four abnormal test templates. Three abnormal test templates were generated by negating LINE2, while one was created by negating LINE3. These abnormal test templates for `Assertion Block 1` are illustrated in Figure 9 where the line with the color red indicated the process of invalid hashing mention in Algorithm 3 which marked the line so that the system will replace the data used for the line with invalid data.

As shown in Figure 10, the generation of abnormal test templates for `Assertion Block 2` was initiated only after confirming that no abnormal actions were present within `Assertion Block 1`, the blue color being used here indicated the valid hashing process mention in Algorithm 3 which specified the line will be replaced with valid data in the data replacing process. Specifically, to generate test cases for abnormal user actions related to product selection (e.g., `Assertion Block 2`), it was essential that the user successfully logged into the system without encountering any errors (e.g., `Assertion Block 1`). Given an SoC in LINE5, one abnormal test template is generated for `Assertion Block 2`.

Five abnormal test templates were generated: four for `Assertion Block 1` and one for `Assertion Block 2`. These test templates were the foundation for generating test cases for abnormal user actions. The test cases were generated according to formula (1), utilizing the test data set provided by the users. This process ensured that the generated test cases adequately covered a variety of abnormal scenarios and effectively validated the application's behavior under unexpected conditions.

5. Evaluation

This research conducts a comparative analysis with Katalon Studio [1], a widely recognized commercial UI testing tool, to

[1] `https://katalon.com/`

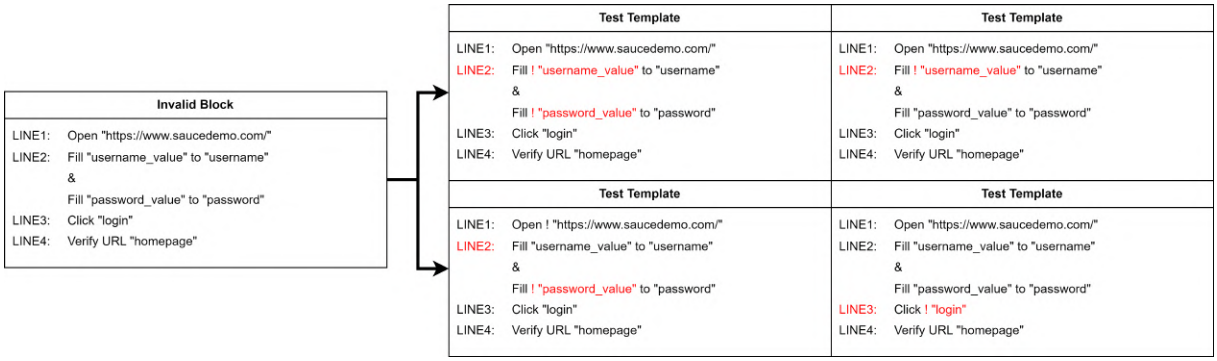


Figure 9. The test templates generation for `Assertion Block 1`.

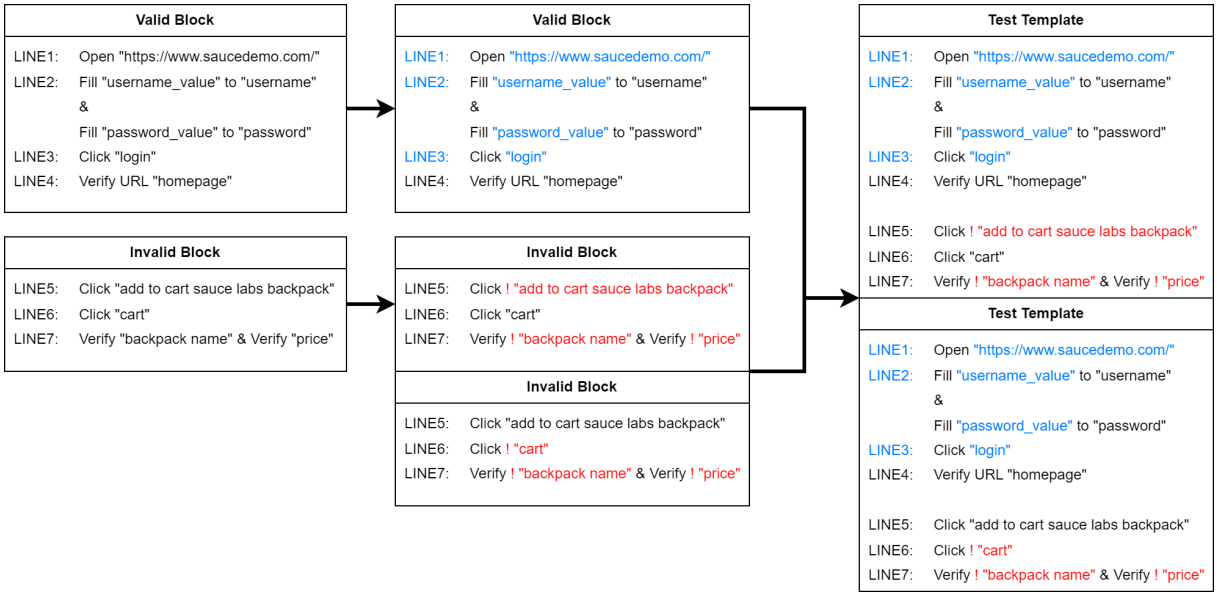


Figure 10. The test templates generation for `Assertion Block 2`.

validate the effectiveness and efficiency of the proposed UI testing method. The comparison was performed on SauceDemo [2], an e-commerce Web application that provides well-known functionality in real-world e-commerce applications such as user authentication, product selection, and transaction processing, on Demoqa[3], a website which allows for interaction simulation on multiple type of elements and BKCAD[4], a real world e-commerce management website. The proposed UBL was implemented in a Web application [5] and was used for comparison with Katalon Studio. There are 44 test cases for the three functionalities, testing a range of user interactions within the SauceDemo application. The results are presented in Table 2 to provide valuable information on the potential benefits and drawbacks of each approach, demonstrating the effectiveness of the proposed method.

[2] `https://www.saucedemo.com/`
[3] `https://demoqa.com/`
[4] `http://103.138.113.158:1012/account/login`
[5] `http://109.123.233.95:8082/`

In Saucedemo, the proposed method generates *44* test cases in *660* seconds, in which it takes *175* seconds to write the specification and *525* seconds to finish the test cases generation process. Meanwhile, it takes *1450* seconds for the user to write the specification manually. In which the total creation time was calculated using the formula:

$$T_{creation} = T_{spec} + T_{gen}$$

Where:

- $T_{creation}$: The time taken to complete the whole test case creation process, containing the specification writing process and the test case generation process.
- T_{spec}: Time to write the test specification.
- T_{gen}: Time to automatically generate test cases from the specification.

Thus, the performance of the proposed method is twice as high, with an average time of *15* seconds to generate a test case when using the proposed method and *33* seconds if manually generated in Katalon Studio. The result for Demoqa and BKCAD follow the same trend, the proposed method outperformed heavily in the writing time due to the ability to generate multiple test cases from one specification, however it comes with a low overhead cost in generation time.

6. Threats to Validity

Despite the promising results demonstrated by our algorithm, several limitations need to be acknowledged to provide a comprehensive understanding of its capabilities and constraints.

6.1. Rigid Invalid Data Generation Process

The process employed for generating invalid data in the proposed method relies solely on basic string manipulation techniques. This approach lacks the flexibility to adapt the generated data to the specific characteristics or requirements of the input elements within the system. As a result, the generated invalid data may fail to effectively challenge or invalidate the targeted elements, reducing the robustness of the testing process. This limitation restricts the method's ability to comprehensively evaluate the system's handling of diverse or unexpected inputs, potentially overlooking critical edge cases or failure scenarios.

6.2. Absence of Core Programming Concepts

The metamodel of the Universal Behavior Language (UBL) incorporates only Scenario, Sequence of Commands (SoCs), Commands, and Constraints, as defined Section 2. However, it does not support the integration of fundamental programming concepts [13], such as variables, conditions, and loops. This absence imposes significant restrictions on the scripting capabilities within the UBL structure. Without variables, scripts cannot store or manipulate dynamic data, limiting their ability to handle varying inputs or maintain state across operations. The lack of conditions prevents scripts from implementing decision-making logic, resulting in linear execution paths that cannot adapt to different GUI responses or system states. Similarly, the absence of loops prohibits repetitive operations, forcing scripts to rely on manual, hard-coded repetition for tasks requiring iteration. Collectively, these limitations render scripts static and inflexible, incapable of supporting complex workflows or dynamic interactions with the graphical user interface, thereby constraining the overall functionality and adaptability of the system.

6.3. Testing Responsive Web User Interfaces

Responsive user interfaces (UIs) are designed to adapt seamlessly to a wide range of screen resolutions and device types, ensuring consistent functionality and presentation across diverse platforms [14]. This adaptability, however,

Table 2. Comparison with Katalon Studio

Web Applications	**Method**	**Number of Test Cases**	**Writing Time**	**Generation Time**	**Avg. Creation Time**
Saucedemo	Katalon Studio	44	1450 s	N/A	~33 s
	Proposed Method	44	175 s	525 s	~15 s
Demoqa	Katalon Studio	40	3350 s	N/A	~169 s
	Proposed Method	40	212 s	588 s	~20 s
BKCAD[4]	Katalon Studio	35	2630 s	N/A	~152 s
	Proposed Method	35	114 s	306 s	~12 s

introduces significant challenges to the UI testing process. At different screen sizes, the UI may exhibit variations in layout, behavior, or interaction patterns, potentially leading to distinct functional outcomes or user experiences. The proposed method currently lacks the capability for testers to define or configure specific screen resolutions for testing purposes. This limitation restricts the ability to thoroughly evaluate the UI's performance under varying conditions, potentially overlooking critical workflows or edge cases that could result in system failures or degraded user experiences on certain devices or resolutions.

7. Related Works

Many researchers have explored automating test script creation for web applications to reduce development costs. Sikuli [15] is a commercial tool that identifies UI elements through image matching. However, the Sikuli script is restricted to interacting with graphical elements currently displayed on the screen. It cannot access GUI components that are obscured or hidden from view. Moreover, Sikuli is more affected by theme variations.

The most significant challenge in maintaining end-to-end user interface testing for web applications is the fragility of the tests. Research presented in [16] addresses the issue of instability in visual GUI testing. The authors report that approximately 20% to 30% of the testing methodologies required at least one modification as the application underwent development and changes. To address this issue, the authors in [17] developed a proof-of-concept library that makes test cases independent of the internal structure of UIs. Their methodology incorporates natural language descriptions of user interactions and NLP algorithms for analysis. However, a limitation of their study is the relatively small dataset.

Rahulkrishna [18] proposed a method for identifying web elements through contextual relationships with other prominent elements on the page. This approach involves establishing a sequence of contextual cues that precisely identify web elements without relying on deeply embedded page information. However, the algorithm exhibits high computational complexity. Furthermore, DOM restructuring can hinder the playback phase, rendering it infeasible.

Besides, some researchers write test cases in natural language without test scripts or using a previously defined domain-specific language. Pasupat [19] proposed a machine learning approach to translating natural language instructions (like "*click on apple deals*") into actions performed on web pages. While this method shows promise for automated testing,

it struggles to interpret complex or indirect commands accurately. Kirinuki [20] introduced a cutting-edge approach that combines natural language processing and heuristic search algorithms to locate web elements based on descriptions written in a domain-specific language. However, this method can be inefficient when dealing with intricate or large-scale user interfaces. WebPuppet [21] is an automated web UI testing tool that uses DSL formulation with JSON schema to generate test scripts. However, this self-defined domain language is still a bit complicated compared to natural language, and users still need to write another type of code.

In addition to the above approaches, a method for generating new test scenarios for widgets and test specifications is presented in the paper [6]. Besides, the paper contributes a technique to detect hidden widgets, considered one of the most challenging problems in user interface testing. One of the drawbacks is that this method is semiautomatic - testers need to specify user interactions. The authors in study [22] employed deep learning techniques to analyze graph data derived from user interactions on e-commerce platforms. By examining URL patterns, they sought to understand user behavior. However, the limitations of this approach become apparent when user interactions do not result in URL changes, as the analysis would be restricted to surface-level engagement.

Beyond the limitations mentioned earlier, most of these approaches employ scenarios comprising simple sequences of consecutive actions, lacking inter-relational constraints. This results in a limited diversity of generated test cases. In contrast, our methodology proposes constraints as binary logical characters for commands to represent their relationship. Consider the example of a website where users can log in using conventional login methods or via Google, GitHub, or Facebook. Users only need to describe that functionality in the following scenario: "*Click Google | Github | Facebook*" instead of writing three different scenarios as the mentioned approaches. Such inter-commands constraints significantly reduce user scenario writing time, particularly in complex scenario development. Our proposed method enhances efficiency and streamlines the process of creating comprehensive test cases. Furthermore, with the proposed method, the generated test cases also cover abnormal user interactions on UI that cause the scenario to fail.

8. Conclusion

This paper has introduced a novel end-to-end UI testing method for Web applications, integrating a domain-specific language called User Behavior Language (UBL) and a test case generator. The proposed approach addresses critical challenges in UI testing by simplifying test script creation and enhancing test coverage and efficiency.

With its 2-unit phrase format, the UBL significantly reduces the complexity of test script specification, making it more accessible to non-technical testers. This language simplifies the testing process and diminishes the learning curve associated with traditional testing frameworks. Furthermore, the language's capability to define constraints between interactions enables the modeling of complex user workflows and the enforcement of application-specific rules, thereby enhancing the fidelity of the testing scenarios.

The test case generator, leveraging the specified constraints and test data, produces a comprehensive set of test cases that encompass both expected and abnormal user behaviors. This automated approach ensures thorough coverage of application functionality while optimizing testing efficiency. Comparative analysis with Katalon Studio has validated the effectiveness and efficiency of our proposed method.

The results demonstrate superior performance in test scenario creation, test

case generation, and ease of learning. However, further comparative analyses incorporating broader evaluation criteria are necessary to assess the solution's efficacy comprehensively. Future research will expand the comparison to include other prominent user interface testing tools available in the market, providing a more holistic assessment of the method's capabilities. While the current implementation significantly enhances UI testing automation and efficiency, the manual identification of Web element locations remains a limitation. Our future research will focus on developing an automated solution for determining the locations of web elements based on natural language descriptions. This improvement further saves time in the UI testing process, reduces manual intervention, and increases overall testing productivity.

References

[1] M. Niranjanamurthy, A. Nagaraj, H. Gattu, and P. K. Shetty, "Research Study on Importance of Usability Testing/User Experience (UX) Testing," *International Journal of Computer Science and Mobile Computing*, vol. 3, no. 10, pp. 78–85, 2014.

[2] N. Minh, "Developing Automated UI Testing," 2023.

[3] L. J. White, "Regression Testing of GUI Event Interactions," in *icsm*, vol. 96, 1996, pp. 350–358.

[4] L. K. Trinh, V. D. Hieu, P. N. Hung *et al.*, "A Method for Automated User Interaction Testing of Web Applications," *Journal on Information Technologies & Communications*, pp. 28–28, 2015.

[5] R. Narkhede, S. Korde, A. Darda, and S. Sharma, "An Industrial Research on GUI Testing Techniques for Windows Based Application Using UFT," in *2015 International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM)*. IEEE, 2015, pp. 466–471, doi:10.1109/ICSTM.2015.7225455.

[6] D. T. Dinh, P. N. Hung, and T. N. Duy, "A Method for Automated User Interface Testing of Windows-Based Applications," in *Proceedings of the 9th International Symposium on Information and Communication Technology*, 2018, pp. 337–343, doi:10.1145/3287921.3287973.

[7] I. Prazina, Š. Bećirović, E. Cogo, and V. Okanović, "Methods for Automatic Web Page Layout Testing and Analysis: A Review," *IEEE Access*, vol. 11, pp. 13 948–13 964, 2023, doi:10.1109/ACCESS.2023.3243285.

[8] E. Pelivani, A. Besimi, and B. Cico, "A Comparative Study of UI Testing Framework," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 2022, pp. 1–5, doi:10.1109/MECO55406.2022.9797217.

[9] R. B. Bahaweres, E. Oktaviani, L. K. Wardhani, I. Hermadi, A. Suroso, I. P. Solihin, and Y. Arkeman, "Behavior-Driven Development (BDD) Cucumber Katalon for Automation GUI Testing Case CURA and Swag Labs," in *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*. IEEE, 2020, pp. 87–92, doi:10.1109/ICIMCIS51567.2020.9354292.

[10] S. R. Mallick, R. K. Lenka, S. Sudershana, A. Sahoo, S. Palei, and R. K. Barik, "An Investigation into the Efficacy of RANOREX Software Test Automation Tool," in *2023 3rd International Conference on Innovative Sustainable Computational Technologies (CISCT)*. IEEE, 2023, pp. 1–5, doi:10.1109/CISCT57197.2023.10351286.

[11] H. A. Thooriqoh, T. N. Annisa, and U. L. Yuhana, "Selenium Framework for Web Automation Testing: A Systematic Literature Review," *Jurnal Ilmiah Teknologi Informasi*, vol. 19, no. 2, pp. 65–76, 2021.

[12] K. H. Rosen, *Discrete Mathematics and Its Applications*, 5th ed. McGraw-Hill Higher Education, 2002.

[13] P. Van Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*. MIT press, 2004.

[14] M. H. Baturay and M. Birtane, "Responsive Web Design: A New Type of Design for Web-Based Instructional Content," *Procedia-Social and Behavioral Sciences*, vol. 106, pp. 2275–2279, 2013, doi:10.1016/j.sbspro.2013.12.258.

[15] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using GUI Screenshots for Search and Automation," in *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, 2009, pp. 183–192, doi:10.1145/1622176.1622213.

[16] R. Coppola, L. Ardito, and M. Torchiano, "Fragility of Layout-Based and Visual GUI Test Scripts: An Assessment Study on a Hybrid Mobile Application," in *Proceedings of the 10th acm sigsoft international workshop on automating test case design, selection, and evaluation*, 2019, pp. 28–34, doi:10.1145/3340433.3342822.

[17] H. Pirzadeh and S. Shanian, "Resilient User Interface Level Tests," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 683–688, doi:10.1145/2642937.2642992.

[18] R. Yandrapally, S. Thummalapenta, S. Sinha,

and S. Chandra, "Robust Test Automation Using Contextual Clues," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 304–314, doi:10.1145/2610384.2610398.

[19] P. Pasupat, T.-S. Jiang, E. Z. Liu, K. Guu, and P. Liang, "Mapping Natural Language Commands to Web Elements," *arXiv preprint arXiv:1808.09132*, 2018.

[20] H. Kirinuki, S. Matsumoto, Y. Higo, and S. Kusumoto, "NLP-Assisted Web Element Identification toward Script-Free Testing," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 639–643, doi:10.1109/ICSME52107.2021.00070.

[21] R. Queirós, "WebPuppet-A Tiny Automated Web UI Testing Tool," in *Third International Computer Programming Education Conference (ICPEC 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022.

[22] R. F. Oguz, I. Erdem, E. Olmezogullari, and M. S. Aktas, "End-to-End Automated UI Testing Workflow for Web Sites with Intensive User–System Interactions," *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no. 10, pp. 1477–1497, 2022, doi:10.1142/S0218194022500462.