Diffing E2E Tests against User Traces for Continuous Improvement

Xavier Blanc, Thomas Degueule, Jean-Rémy Falleri

Diffing E2E Tests against User Traces for Continuous Improvement

Xavier Blanc
xavier.blanc@labri.fr
Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI
Bordeaux, France

Thomas Degueule
thomas.degueule@labri.fr
Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI
Bordeaux, France

Jean-Rémy Falleri
jean-remy.falleri@labri.fr
Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI
Bordeaux, France
Institut Universitaire de France
France

ABSTRACT

End-to-end testing (E2E) is a popular approach to uncover faults in web applications by mimicking the behavior of real users. Using dedicated frameworks (*e.g.*, Playwright, Cypress), testers translate high-level validation scenarios into concrete browser interactions to evaluate the application's behavior. Testers are forced to make implementation choices when writing their tests: choosing one of several valid selectors to click a button, focusing an input field using a *Click* or *Tab* event, *etc.*

In this paper, we show that the resulting tests may not always represent the behavioral of real users—some action sequences may even be infeasible for users. These discrepancies between tests and users hurt the very premise of E2E testing.

To address this issue, we propose a novel method that enables testers to understand the true behavior of their E2E tests, and help them improve their tests by pinpointing when, where, and in which way they differ from the behavior of real users.

Using a set of E2E tests provided by our industrial partner, we show that our tool is indeed able to identify discrepancies between the tests and the users and that, in several cases, the testers decided to update their tests to better match the behavior of their users.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

end-to-end testing, web application testing, trace differencing

ACM Reference Format:
Xavier Blanc, Thomas Degueule, and Jean-Rémy Falleri. 2022. Diffing E2E Tests against User Traces for Continuous Improvement. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A popular approach to web application testing is to define and execute end-to-end (E2E) tests to uncover faults by mimicking the behavior of real users [1]. E2E testers typically leverage the capabilities of popular testing frameworks such as Playwright, Cypress, or Selenium to translate high-level validation scenarios (*e.g.*, purchasing a product in a webshop) into concrete interactions within the browser where the web application runs.

Using these frameworks, simple steps in a scenario (*e.g.*, clicking a button or filling a form) can be realized in multiple ways. Indeed, a modern stylized button may consist of several DOM elements, each with its particular CSS selector, and the click itself may be realized in various ways: clicking the button directly, focusing it and pressing the *Enter* key, *etc.* Similarly, one may jump from one field in a web form to the other by pressing the *Tab* key or by focusing each field with a *Click* event before inputting a value. Although there are many ways to realize the same scenario, only one (or a handful) is eventually implemented and verified.

For E2E testing to be successful, it must simulate the behavior of real users. However, since simple interactions may be realized in many different ways, it is difficult for testers to assess whether the interactions realized by their tests match what a user would do. Worse, our exploratory analysis reveals that, out of the box, test frameworks sometimes interact with web applications in a way that can never be reproduced by actual users (Section 2). For instance, when filling a web form, Playwright may use the JavaScript API without performing any *Click* or *Keystroke*. This contrasts with real user interactions that would inevitably require the use of the mouse or the keyboard to focus on the fields and to set their values. We believe that these discrepancies between tests and users hurt the very objective of E2E testing and may prevent the tests from revealing faults encountered by the users.

In this paper, our aim is twofold: assist the testers in understanding the true behavior of their E2E tests, and help them improve their tests by pinpointing when, where, and in which way they differ from the behavior of a real user. To this end, we first present a non-intrusive tool that automatically records every relevant interaction on a web application (clicks, keystrokes, etc.). We use this tool to record the traces resulting from test executions, as well as the traces of real users interacting with the application (Section 3). Then, we present a diff algorithm and interface that pinpoints and highlights the differences between the execution(s) of a test and the same scenario realized by one or several users (Section 4).

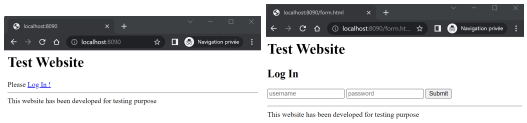
We hypothesize that whenever the trace produced from an automated test and from a user realizing the same scenario differ, the testers may decide to update their tests to match the user's trace better. In our evaluation (Section 5), we find that, indeed, our tool can identify new interactions that are of higher quality than those used in the tests (*e.g.*, because the user clicked on a different area

Conference'17, July 2017, Washington, DC, USA
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

or used some keyboard shortcuts) and identify ill-formed traces that prompted the testers to reconsider their tests (*e.g.*, because the test filled two input fields in a form without switching focus in between). This leads us to conclude that our tool can nicely complement the usual E2E testing toolkit by improving the quality of E2E tests implemented using the most popular frameworks.

2 BACKGROUND & EXPLORATORY ANALYSIS

Let us consider a straightforward web application composed of two pages: a landing page and a login page (see Figure 1).



(a) Landing Page. **(b) Login Page.**

Figure 1: A simple web application with two pages: a landing page and a login page.

Alice, a junior tester, is tasked with testing this simple web application by implementing a new E2E test. Her objective is to assess the following scenario:

(1) Go to the landing page
(2) Navigate to the login page
(3) Fill the username and password fields
(4) Submit the form
(5) Check that the user has been successfully logged in

To implement her scenario, Alice uses the Playwright framework and writes the source code presented in Listing 1. She implements all the scenario steps and makes some design choices to specify how the test interacts with the browser. For instance, she decides that moving from the landing page to the login page is performed by clicking the DOM element identified by `goto-login-page-link`. She also decides to rely on the `fill()` method of the Playwright framework to implement step (3).

Listing 1: Source code of the simple E2E test (version 1).

```
const browser = await chromium.launch({
    slowMo:1000,
    headless:false
});
const page = await browser.newPage();
await page.goto('http://localhost:8090/');
await page.click('#goto-login-page-link');
await page.fill('#username', 'Jane');
await page.fill('#password','pass');
await page.click('form > div.button');
const testResult = await page.$eval('#logname',
        el => el.value === 'Jane');
```

Alice's design choices aim at realizing the scenario in a similar manner as a real user. Her choices are also driven by her knowledge of the framework and her willingness to make the source code as clean as possible.

Bob, a senior tester who is also familiar with the playwright framework, explains that the `fill()` method relies on the internal JavaScript API of the browser to focus the text fields and set their values. As a consequence, the test does not actually perform any realistic user interaction (clicks or keystrokes) in step (3) of the scenario. He then proposes a new version where the `fill()` methods are each replaced by a `click()` and a `type()` (see Listing 2). This new version is therefore much closer to the behavior of a real user.

Listing 2: Source code of the simple E2E test (version 2).

```
const browser = await chromium.launch({
    slowMo:1000,
    headless:false
});
const page = await browser.newPage();
await page.goto('http://localhost:8090/');
await page.click('#goto-login-page-link');
await page.click('#username');
await page.type('#username', 'Jane');
await page.click('#password');
await page.type('#password','pass');
await page.click('form > div.button');
const testResult = await page.$eval('#logname',
        el => el.value === 'Jane');
```

Finally, Jenny, another senior tester, proposes a third version claiming that some users usually use the *Tab* key to navigate through the fields in a form. She also argues that the `form > div.button` selector is too vague and may accidentally match another button in the form if it were to evolve in the future. She then proposes another selector that uniquely identifies the submit button. Listing 3 shows the relevant lines in this third version.

Listing 3: Source code of the simple E2E test (version 3).

```
await page.click('#username');
await page.type('#username', 'Jane');
await page.keyboard.press('Tab');
await page.type('#password','pass');
await page.click('#submit');
```

This simple illustrative example shows that there are many ways to implement the same E2E test. Some implementations, such as the first version of this simple test, emulate a behavior that cannot be realized by real users, which directly contradicts the purpose of E2E testing. We argue that these implementations should be avoided in tests, but that testers do not have the necessary tool support to easily detect these cases.

There might also be several implementations that represent the behavior of real users. It then becomes difficult for testers to decide which design choices should be taken (*e.g.*, here, a *Click* or a *Tab*). In such a context, we argue that tests should be as close as possible to real users, with the main motivation to uncover faults that may be encountered by them. For instance, if most users use the *Tab* key to switch among the fields, then the test should do the same.

In this paper, we aim at helping testers such as Alice, Bob, and Jenny towards that objective. We propose an approach that identifies the real interactions made when executing an E2E test. We then propose to compare these interactions with the ones realized

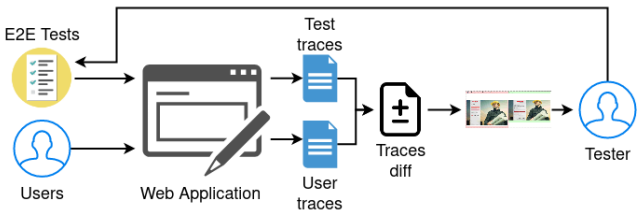


Figure 2: Approach overview.

by real users of the application. We argue that highlighting the differences in interactions between tests and users enables testers to improve their E2E tests and, ultimately, their application's quality. This overall process is depicted in Figure 2 and detailed in the following sections.

3 RECORDING INTERACTIONS TRACES

In this section, we explain how we record the interactions performed by tests and real users when using a web application. The section starts by presenting the set of interactions we are interested, and then presents how we build sequences of interactions (*i.e.*, traces) by monitoring the browser and the web application.

3.1 User Interactions

In any web application, most interactions are performed thanks to a pointing device (*e.g.*, a mouse) and a keyboard. An interaction mainly means to (1) focus on a specific element and (2) actuate it.

Some interactions simultaneously focus on an element and actuate it (*e.g.*, a mouse click). Others only focus an element (*e.g.*, the *Tab* key), or only actuate (the *Enter* key). Interactions have a *target*, which is the element on which the interaction occurs and that can be uniquely identified by a CSS selector [2].

Several interactions can be performed with the mouse: left-clicking, double-clicking, right-clicking, selecting a text, moving the mouse, *etc.* We choose to only consider left-click on an element (mouse down), as the other mouse interactions rarely lead to application-specific interactions in most web applications. We further choose to abstract any left-click interaction by the word `Click$TargetCSSSelector` (where `TargetCSSSelector` is the CSS selector of the target element of the interaction). For instance, a click on an element identified by the CSS selector `#goto-login-page-link` is abstracted by the word `Click$#goto-login-page-link`.

Several interactions can be performed with the keyboard: editing a text field, pressing the *Enter* key to submit a form, pressing the *Tab* key to focus the next element, *etc.* When a user is editing a text field, we consider that all the interactions they perform can be abstracted by only one interaction: `Edit$CSSSelector`, as it is uncommon for web applications to have custom interactions triggered by entering a specific text value. When a user is not editing a text field, the keyboard may be used to focus and actuate an element thanks to keyboard shortcuts. Unfortunately, there is no standard defining which keyboard shortcuts are generally available and supported. The *WAI-ARIA Authoring Practices 1.1* defines some principles and some guidances but does not specifies a list of keyboard shortcuts that any browser and any web application should respect.[1] The W3C defines a list of keyboard shortcuts that are commonly supported by browsers, but these shortcuts are not for focusing on and actuating elements.[2] Finally, the gray literature contains some lists of keyboard shortcuts for interacting with any web application.[3]

We then consider that the following shortcuts are supported by most the browsers and web applications:

key	interaction
Tab	focus on the next element in the DOM
ShiftTab	focus on the previous element in the DOM
Space	actionate the focused element
Enter	actionate the focused element
Arrow	select an option in a select box
Esc	close a popup or a dialog

We choose to abstract these interactions using the word `Key$TargetCSSSelector`, where `Key` represents the keyboard shortcut and `TargetCSSSelector` the target's CSS Selector. In our example, `Tab$#username` abstracts the fact that the *Tab* key has been pressed when the element identified by `#username` was focused. Note that this interaction makes the `#password` element focused.

Our set of interactions is relatively small but captures the majority of the interactions that are performed by a user. For example, running the test depicted in Listing 3 will yield the following trace:

nb	interaction
1	`Click$#goto-login-page-link`
2	`Click$#username`
3	`Edit$#username`
4	`Click$#password`
5	`Edit$#password`
6	`Click$#submit`

By contrast, running the test depicted in Listing 1 highlights that the test is not well designed as it does interact with the browser like a real user (the *fill()* method uses the JavaScript API and does not emulate clicks nor keyboard keys):

nb	interaction
1	`Click$#goto-login-page-link`
2	`Click$#submit`

3.2 Recording traces

To record a trace of interactions, we monitor the web application and listen to the mouse and keyboard events. More precisely, we built a simple JavaScript script that listens to the *mousedown* and *keydown* events. Each time such an event occurs, the script creates the corresponding interactions and sends them to a server responsible for storing the trace. Thanks to the Chrome extension API, this script is injected into any page and executed just after loading.

Listing 4 presents the source code of the script for mouse interactions. This script generates an interaction for each *mousedown* event and sends it to the server.

Listing 4: A script for recording mouse interactions

```
document.addEventListener('mousedown', mouseDown, true);
```

[1] https://www.w3.org/TR/wai-aria-practices-1.1/
[2] https://www.w3schools.com/tags/ref_keyboardshortcuts.asp
[3] https://www.barrierbreak.com/have-you-ever-tried-to-access-the-webpage-using-keyboard/

```
mouseDown(event) {
    let prefix = 'Click';
    let suffix = generateCSSSelector(event.target);
    let interaction = prefix + '$' + suffix;
    sendAction(interaction, SERVER_URL);
}
```

Listing 5 presents the source code of the script for keyboard interactions. This script checks whether the target is editable or not. It then generates an interaction (a word) depending on the key that has been pressed and whether the target is editable. Finally, the word that abstracts the interaction is sent to the server.

Listing 5: A script for recording keyboard interactions

```
document.addEventListener('keydown', keyDown, true);
keyDown(event) {
    let prefix = 'Edit';
    let isEditable = false;
    if (event.target instanceof HTMLInputElement
        && !event.target.disabled
        && !event.target.readOnly) {
        isEditable = true;
    }

    switch (event.code) {
        case 'Tab':
            if (event.shiftKey){
                prefix = 'ShiftTab';
            } else {
                prefix = 'Tab';
            }
            break;
        case 'Enter':
            if (isEditable) {
                prefix = 'Edit';
            } else {
                prefix = 'Enter';
            }
            break;
        case 'Space':
            if (isEditable) {
                prefix = 'Edit';
            } else {
                prefix = 'Space';
            }
            break;
        case 'ArrowUp':
        case 'ArrowDown':
        case 'ArrowLeft':
        case 'ArrowRight':
            prefix = event.code;
            break;
        case 'Escape':
            prefix = 'Escape';
            break;
        default:
            prefix = 'Edit';
    }
    let suffix = generateCSSSelector(event.target);
    let interaction = prefix + '$' + suffix;
    if (lastInteraction !== interaction) {
        lastInteraction = interaction;
        sendAction(interaction, SERVER_URL);
    }
}
```

To ease the understanding of a recorded trace, we further attach a screenshot of the page at the time the interaction took place. This can be done thanks to the chrome extension API. We also attach the coordinates of the bounding rectangle of the target element. The screenshot and the bounding rectangle allow us to build a visualization of the traces highlighting the targets of the interactions with red rectangles. Figure 3 presents our visualization for three interactions of the trace obtained by running the test of Listing 2.

4 DIFFING TRACES

A trace captures the interactions performed through the web interface (clicks and keyboard strokes). It can be recorded when a test is run (test trace) or when a user browses the web application (user trace). Diffing a test trace with a user trace highlights their differences in terms of interaction. It therefore pinpoints the interactions that are missed by the test, the ones that are done by the test whereas the user does not perform them, and any other kinds of discrepancies.

This section presents our approach for diffing two traces. It starts by explaining how we use Myers' algorithm to diff two traces. Then, it describes our graphical visualization of the diff that uses screenshots attached to the trace. Our approach is designed to help testers understand the differences between the tests and the users to help them decide whether—and how—the tests should be updated to match the users.

4.1 Myers' Algorithm for Diffing Interactions Traces

A trace represents a sequence of interactions. From a syntactical point of view, each interaction is represented by a token, and the trace can therefore be seen as simply a sequence of tokens. Diffing two traces is, therefore, the same as applying a sequence differencing algorithm such as the one proposed by Myers [3]. The principle of Myers's algorithm is to find the longest common subsequence (LCS) between the two sequences. Using this longest common subsequence, it deduces the tokens that are conserved between the two traces (those belonging to the LCS), the deleted tokens (tokens of the left trace not belonging to the LCS), and the inserted tokens (tokens of the right trace not belonging to the LCS). For instance, if we diff the traces obtained by running the tests of Listing 1 and Listing 2, we obtain the diff shown in Table 1. In this table, the conserved tokens are represented by the white line containing two equal (=) signs. The inserted tokens are represented by the green lines with a plus (+) sign. The deleted tokens are represented by the red lines with a minus (-) sign. In the diff terminology, a sequence of consecutive red and green lines between white lines is called a *hunk*. In Table 1, there is only one such hunk consisting of 4 green lines (lines 2 to 5). This hunk indicates that in the second trace,

Figure 3: Visualizing the interactions realized by the test depicted in Listing 2. The red rectangles highlight which element, identified by the corresponding CSS selector, is the target of the interaction.

	First trace		Second trace
=	Click$#goto-login-page-link	=	Click$#goto-login-page-link
		+	Click$#username
		+	Edit$#username
		+	Click$#password
		+	Edit$#password
=	Click$#submit	=	Click$#submit

Table 1: Myers' algorithm output for the traces of 1 and 2

there are four interactions that were not performed in the first trace.

If we now compute a diff between test runs of Listing 2 and Listing 3, we obtain the diff shown in Table 2. This diff contains

	Second trace		Third trace
=	Click$#goto-login-page-link	=	Click$#goto-login-page-link
=	Click$#username	=	Click$#username
=	Edit$#username	=	Edit$#username
-	Click$#password		
		+	Tab$#username
=	Edit$#password	=	Edit$#password
-	Click$#submit		
		+	Click$form > div.button

Table 2: Myers' algorithm output for the traces of 2 and 3

two hunks. The first hunk contains one green line and one red line (lines 4 and 5). This hunk indicates a *Click* interaction in the second trace, while in the third trace, there was a *Tab* interaction. The second hunk also contains one green and one red line (lines 7 and 8). This hunk indicates a *Click* interaction in both traces but on a different target, as we can deduce from the different selector value.

More generally, there are three types of hunks. Firstly, hunks consisting only of green lines, such as the one in Table 1. These hunks represent interactions performed in the right trace but not in the left trace, *i.e.*, interactions performed by the users only, not by the tests. Conversely, there are also hunks consisting only of red lines. These hunks represent interactions performed in the left trace but not in the right trace, *i.e.*, interactions performed by the tests only, not by the users. Finally, there are hunks composed of both red and green lines, such as the ones in Table 2. These hunks represent interactions performed differently by tests and users, for instance because they interacted with different elements or because they interacted with the same element with a different action.

4.2 Visualizing the diff

In the previously described presentation of the diff, using the tokens is not very readable for testers. We then propose a graphical visualization that uses the tokens and the screenshots that were recorded at the time the interaction took place as well as the bounding rectangle highlighting which specific element was the target of the interaction. We still use the green and red notations to display the hunks. As an example, Figure 4 displays the beginning (lines 1 and 2) of the diff shown in Table 1. This visualization shows that the first interaction is the same for the two traces and that the trace of the second version has an interaction that is not present in the trace of the first version. The main difference with Table 1 is that the tester now clearly sees in the screenshot which actions took place when the two traces diverged.

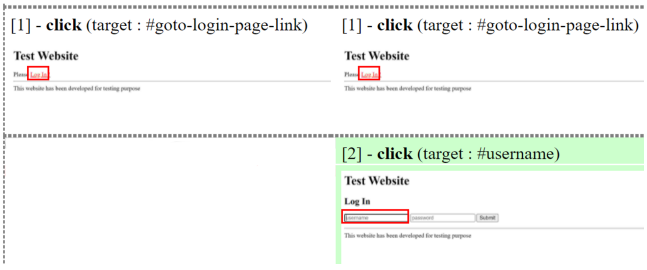


Figure 4: Visualization of lines 1 and 2 of the diff shown in Table 1

Figure 5 displays another example based upon the lines 3 to 5 of Table 2. One notable difference to the table is that we pack the red and green lines to save vertical space. The hunk displayed in Figure 5 clearly shows that in the second trace, a click was made on the password input, while in the third trace, the tab key was pressed while the focus was on the username input.

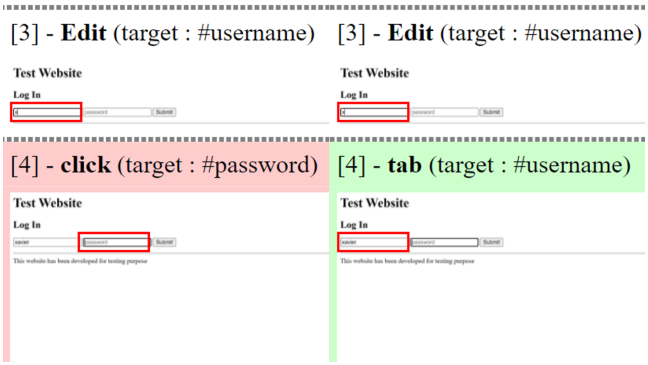


Figure 5: Visualization of lines 3 to 5 of the diff shown in Table 2

5 EVALUATION

This section presents the evaluation of our approach using a field experiment [4]. We conducted this experiment with our industrial partner Mr Suricate, a company specialized in implementing and running E2E tests for external customers.[4] The goal was to use our approach on real test scenarios chosen by the company to answer this research question: *is our approach effective in pinpointing relevant differences in the test traces?*

5.1 Context

Mr Suricate covers the entire scope of its clients' test campaigns, including creating the E2E test suites and scenarios, alert notification, test execution, and the creation of incident sheets in the event of a test failure. It is a young growing company with many clients in different business domains (eCommerce, banking, energy, etc.). In a nutshell, their clients express their test objectives, and Mr Suricate creates the test scenarios, implements them, runs them every day, and reports anomalies whenever they arise.

To optimize the creation and the implementation of the test scenarios, Mr Suricate has developed a graphical domain-specific language (DSL), built atop Blockly,[5] with a dedicated runtime engine. The DSL mainly consists of basic building blocks (*e.g.*, for clicking on elements, filling text fields, navigating to a page) and composite blocks (*e.g.*, for executing actions in sequence or in parallel). Based on the Playwright framework, the runtime engine compiles the scenarios written in this DSL into JavaScript files and executes them.

[4]https://www.mrsuricate.com/
[5]https://developers.google.com/blockly

Participant	Age	Exp. (y)	Rôle	Scenario
T_A	27-32	1.5	Tester	*S_A*
T_B	27-32	2	Tester	*S_B*
T_C	23-27	1.5	Tester	*S_C*
T_D	27-32	2.5	Tester	*Manitou*
T_E	23-27	3	Product owner	*Zadig*
T_F	18-23	0.5	Tester	*Zadig*

Table 3: The six employees who participated to our experiment

5.2 Corpus and Methodology

We worked closely with two test managers of Mr Suricate to conduct our field experiment. They gave us five real test scenarios coming from five different external clients. They also asked the clients to open their scenario and to accept to publish some information: two accepted (Manitou Group[6] and Zadig & Voltaire[7]) and the three others preferred to remain anonymous.

The test managers provided us with a virtual machine running the runtime engine which allowed us to run the tests at any time. After instrumenting their runtime with our trace recorder (cf. Section 3.2), we then ran the five scenarios to obtain the five corresponding traces generated from the execution of the tests.

Then, six employees of Mr Suricate (five testers and one product owner) participated in our experiment. Each of them acted as a real user that would attempt to realize the actions described in the scenarios. They carried out the scenarios using an instrumented version of the Chrome browser that captured their traces in the same way we recorded the tests. We decided to assign each tester to a different scenario and the PO to the Zadig & Voltaire scenario. We thus obtained six user traces. Table 3 lists the participants, their age, their experience, their role in the company, and the scenario they were assigned (*S_A*, *S_B*, and *S_C* denote the three scenarios for which the corresponding company preferred to remain anonymous).

Finally, we invited the testers and the PO to use our diff visualization to compare their own trace with that of the corresponding test, to know whether the diff helps them to better understand and possibly improve the automated test. We asked the testers and the PO to mark each interaction in each hunk as either "Of Interest" or "Not Interesting" (cf. Figure 6). If an interaction is marked as "Of Interest", it means that the difference triggered the tester to investigate the test and potentially improve it.

Our field experiment is then composed of six cases (one for each of the six employees). A case is composed of three parts: a user trace, the corresponding test trace, and the diff with the marks of interest. Thanks to these six cases, we first answer our research question through quantitative analysis. When a hunk is marked as "Of Interest", it means that the diff visualization pushed the testers to question their tests and, potentially, to improve them so that they can be closer to the behavior of a real user. The more hunks end up being marked as "Of Interest", the more tests may be improved.

[6]https://www.manitou.com/
[7]https://zadig-et-voltaire.com/

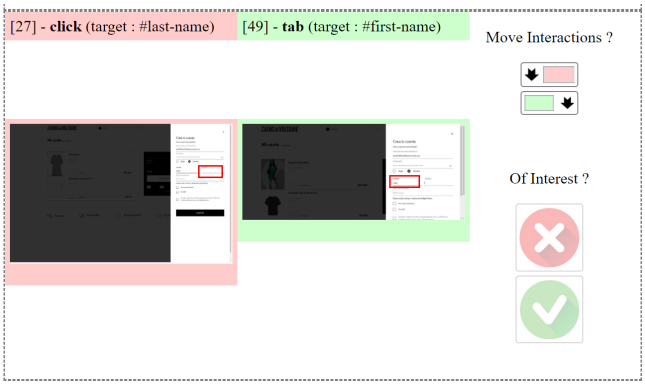


Figure 6

Section 5.3 presents the metrics we measured and the results of our quantitative analysis.

To better understand the differences between the test traces and the user traces, two of the authors inspected all the hunks and built a taxonomy of their underlying causes. As it is a subjective task, the two authors worked together to reach a consensus. Finally, an interview session was conducted with the two test managers of Mr Suricate where we shared with them the results of our experiment and asked them to give their opinion on the taxonomy. Section 5.4 presents this analysis.

5.3 Quantitative analysis

We first made some descriptive statistics by counting:

- the number of interactions in the test trace (#TestI),
- the number of interactions in the user trace (#UserI).

Table 4 presents all the metrics we measured. First, it shows that the number of interactions in the test traces (#TestI) ranges from 21 to 69. This highlights the fact that the five scenarios are quite complex as they consists of a significant number of steps. It should be noted that the Zadig & Voltaire scenario is the most complex, with a total of 69 interactions. This scenario consists in choosing several products in the webshop and then buying them after creating a user account.

The descriptive statistics shows that some user traces have more interactions than the test traces, while others have less. This depends on the scenario but also on the experience of the participants. As a matter of fact, the less experienced participant did not strictly follow the steps of the scenario, and performed much more interactions than the other participants. Two other participants, S_A and S_E, performed less interactions than the test. In the case of S_A, the tester did not manage to finish the scenario before the session was over. In the case of S_E, the test performs interactions that can be avoided by a real user; we discuss this point later in Section 5.4.

We then measured the following metrics on the diff:

- the number of interactions common to the test and user traces (#Sim),
- the number of hunks (#Hunks),
- the number of hunks that contain interactions of the test and the user traces (#HTU),
- the number of hunks that contains only interactions of the test trace (#HT_),
- the number of hunks that contains only interactions of the user trace (#H_U),
- the number of interactions contained in the biggest hunk (HChurn).

The metric #Sim shows that the test trace and the user trace share a lot of interactions. The ratio #Sim / #TestI ranges from 0.40 (Zadig & Voltaire) to 0.86 (S_A), which is not surprising as Zadig & Voltaire is the most complex scenario. The metric #Hunks shows that the discrepancy is high. The metrics #HTU, #HT_ and #H_U give more information. First of all, #HTU shows that most of the hunks contain interactions of the test and the user traces. This means that the traces diverge in a lot of ways. Some hunks (#HT_) only contain interactions of the test trace. We show in Section 5.4 that they highlight cases where the test performs interactions that are not useful anymore. Some hunks (#H_U) only contain interactions of the user trace. We show in Section 5.4 that they either identify cases where the user performed additional extra interactions or highlight cases where the test misses to perform some interactions. The metric HChurn show that the biggest hunk belongs to the Zadig & Voltaire scenario, which is not surprising as it is the most complex scenario. Finally, the #Mark metric shows that there are hunks that contain at least one interaction of interest. If we consider that the object of interest is the hunk, 12% of the hunks (8/64) are interesting. Their existence asks for a deeper investigation of the tests to check whether they may be improved. This low score shows that our approach lacks precision as it produces many hunks that are not interesting. If we now consider that the object of the interest is the couple (user trace, test trace), then 66% are interesting. Finally, the two cases of the Zadig & Voltaire scenario show that the results of our approach also vary with the participant.

As a short conclusion, this quantitative analysis shows that our approach pinpoints discrepancies between the user and test traces. Participants expressed that some of these discrepancies need more investigation and may end with some test improvement. The analysis also shows that our approach identifies many hunks that are not considered to be of interest (86%) and that the results are pretty subjective.

5.4 Hunks analysis

Two authors analyzed the 64 hunks of the experiment and identified six main causes of discrepancy. They then interviewed the test managers and asked them to get their opinion on the causes.

5.4.1 Tab vs click. Focusing on an element of a web application can be done either by clicking on it or using the Tab key. The test traces of our experiment mostly contain clicks; tabs rarely happened. The user traces contain clicks and tabs, depending on the participants. This difference in practice is, therefore, a cause of the discrepancy. For example, Figure 7 shows such a hunk in the Manitou Group scenario. The test focuses on the password field by clicking on it, whereas the user uses the *Tab* key. The two test managers agreed that this case of discrepancy is important as the test should be as close as possible to the user's behavior.

Tester - Scenario	#TestI	#UserI	#Sim	#Hunks	#HTU	#HT_	#H_U	HChurn	#Mark
T_A - S_A	54	43	29	9	5	2	2	10	1
T_B - S_B	22	22	19	3	3	0	0	4	0
T_C - S_C	61	68	45	14	8	3	3	8	4
T_D - *ManitouGroup*	21	22	17	5	4	0	1	4	1
T_E - *Zadig&Voltaire*	69	63	28	16	13	1	2	18	0
T_F - *Zadig&Voltaire*	69	114	41	17	10	0	7	30	0

Table 4: Experimental quantitative results

Figure 7: A Click vs Tab hunk in the Manitou Group scenario

5.4.2 Same Widget. Each page of a web application presents graphical widgets that are technically composed of several HTML elements. For instance, a button is frequently composed of a surrounding bloc (`<div>`), an image (`<img>`) and a legend (`<span>`). However, depending on which element is clicked, our approach returns different interactions (the CSS selector is unique for each element), which is a cause of discrepancy. For example, Figure 8 presents such a difference in the Manitou Group scenario. The test clicked on the legend, whereas the user clicked on the surrounding block. The two test managers agreed that this case discrepancy is not as important. They argue that the widget itself is of importance, not the technical details of its implementation.

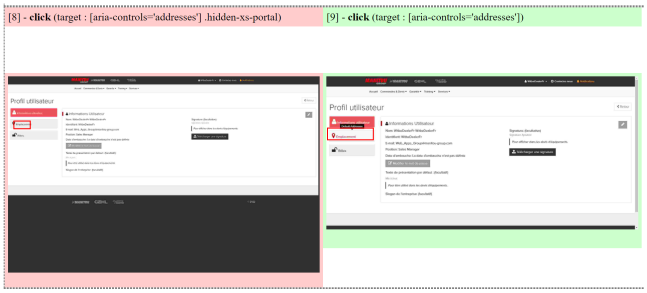


Figure 8: A example of hunk in the Manitou Group scenario with a click on a same widget that yields two different interactions.

5.4.3 Missing Interaction. A testing framework may sometimes use the JavaScript API to focus and interact with the elements. The test trace does not contain the corresponding interactions in such a case, whereas the user trace does. Indeed, a user cannot use the JavaScript API and always uses the mouse or the keyboard. Figure 9 presents an example on the Zadig & Voltaire scenario. The Figure shows a form with several fields. The user trace (on the right) registered a click on the element, whereas the test trace (on the left) did not (there is no interaction). The two test managers said that this cause of discrepancy is very important and asked for an improvement of the testing framework. Their framework already extends the `fill()` method of the Playwright Framework. Their extension modifies the default behavior of this method by making an explicit click on the fields and by using the keyboard to fill them. Our experiment made it possible to identify new situations where their framework still uses the JavaScript API. Therefore, they wanted to investigate these situations with the objective to potentially improve their framework.

Figure 9: An example of hunk in the Zadig & Voltaire scenario with no focus on the element.

5.4.4 Web Application Changes. There are some changes, which are performed on a web application, that do not break the tests. For example, this is the case when the changes aim to ease the user experience by automatically filling the fields. These changes do not have any impact on the test trace as the test always fills the fields. They, however, have an impact on the user trace as the user is not required to fill the fields anymore. Figure Figure 10 presents an example on the Zadig & Voltaire scenario. The form is now automatically filled, but the test still explicitly fills it. This is to be contrasted with the user trace, where no such action happened. The two test managers were interested by this case of discrepancy, as the tests should remain consistent with the web application's behavior, even when it does evolve.

5.4.5 User-Side Interactions. Participants frequently clicked or used their keyboard aside from the scenario. The test never performs these interactions. Figure 11 presents an example on the Manitou Group scenario: the figure shows that the participant clicked somewhere on the form, even though it was not required

Figure 10: An example of an evolution of the Zadig & Voltaire web application that may ask for an update ot the test.

by the test scenario. The two test managers are not interested in this case of discrepancy.

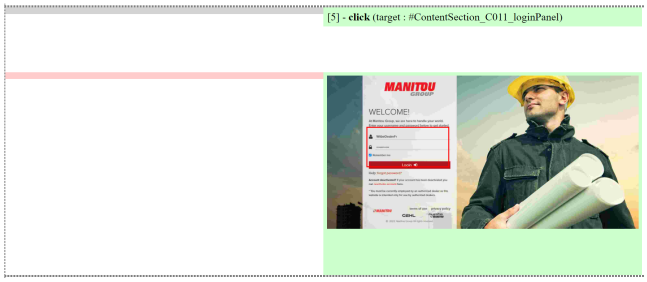


Figure 11: An example of a side click made on the Manitou Group scenario.

5.4.6 User Deviations. Participants of our evaluation sometimes deviated from the scenario, leading to hunks that may be large. For example, Figure 12 presents an example on the Zadig & Voltaire scenario. The figure shows that the participant chose a different value than the test for a text field. This difference is due to the sometimes loose steps in a scenario (here, "create an account"). The two test managers said that this case might be interesting even if they agreed that most of the hunks would lead to nothing. They argue that this kind of hunks may help to update a fuzzy test in some very rare situations.

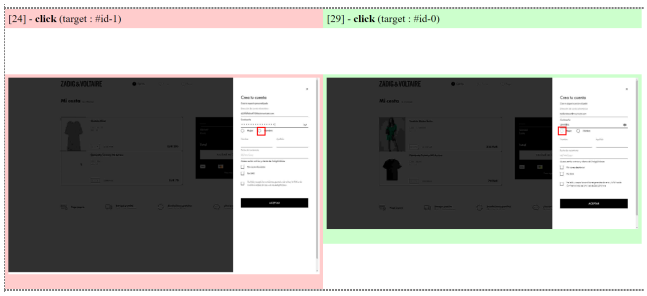


Figure 12: An example of a use deviation in the Zadig & Voltaire scenario. The test and the user chose different values for the same field.

5.5 Threat to validity

Our approach aims at measuring and comparing the clicks and the keyboard strokes performed by automated tests and by real users. Our field experiment on five industrial scenarios with six testers confirms that our approach accurately abstracts the interactions, and there is no threat to construct validity.

In total, 629 interactions have been realized in our evaluation. We cannot assume that all kinds of interactions have been realized. However, we can consider that all of the commonly performed interactions have been observed, limiting the threat to content validity.

We agree that the results we obtained are influenced by the scenarios, the participants, and the managers, which is a threat to internal validity. Furthermore, the results we obtained cannot be generalized much beyond the specific web applications we have been studying, which is a threat to external validity. To limit these threats, we conducted a field experiment with a company that built a considerable experience in E2E testing. We also worked on large industrial scenarios representing complex E2E tests.

6 RELATED WORK

In this section, we describe the related work on the domain of E2E web testing. E2E web tests are known to be fragile [5], therefore researchers have extensively worked on means to mitigate this issue. To the best of our knowledge, no other approaches have been introduced to precisely compare two interactions traces like ours. However, there is a body of complementary work in three main domains, as follows.

6.1 Automated web test repair

Researchers have investigated how to repair broken web tests [6–10]. The objective of these approaches is to automatically repair web tests that are broken due to a variety of reasons, such as layout changes in the web application. They usually use a DOM analysis or a visual analysis (using computer vision algorithms) to identify and repair the cause of the breakage. In contrast to these approaches, we do not aim at automatically repair web tests, but rather to pinpoint subtle differences between the execution of a test and the behavior of a real user.

6.2 Improving the robustness of web tests

Several approaches have been described to decrease the odds of a test being broken by a change in the web application. There are two main categories of approaches. Approaches from the first category aim at improving the robustness of the CSS selectors used in the tests [2, 7, 11–14]. Approaches from the second one aim at relying on visual cues to guide the test instead of CSS selectors [15–17]. In our work, we leverage these approaches to synthesize selectors when the test or the user are interacting with the web application.

6.3 Web test debugging

Many approaches have been introduced to help developers to debug web pages [18–21]. Two main challenges are addressed by these approaches. First, they aim at recording as precisely as possible what happened during a browsing session to enable developers to replay the session and reproduce the behavior. Second, they usually

try to establish links between the DOM elements manipulated and the source code written by the developers. Nevertheless, they are not designed to quickly compare two different browsing sessions, like we do. However, in our approach we borrow the idea of the recording web session traces with a lower level of precision to support comparing two different sessions.

7 CONCLUSION

E2E tests are meant to reflect the behavior of real users to uncover the faults they might encounter using a web application. As testers are forced to make implementation choices when translating high-level validation scenarios into concrete automated tests using popular E2E frameworks such as Playwright or Cypress, the resulting tests may not always represent the behavioral of real users. In particular, we show that some interactions resulting from these tests may not even be feasible by real users. These discrepancies hurt the very premise of E2E testing.

In this paper, we proposed a novel method to record the interaction traces of both real users and E2E tests. Using a novel diff algorithm and visualization for interaction traces, we have shown that it was able to pinpoint when, where, and in which way the tests and the users differ.

Our field experiment with the Mr Suricate company showed that, while our approach may sometimes show information to testers that they do not find interesting, it was also able to pinpoint in several places that the tests were either out-of-sync with the application or too far from a real user's behavior. This prompted managers at the company to investigate some of their E2E tests, as well as their testing framework, which highlights the usefulness of our approach and tool.

For future work, we would like to expand our field experiment by including more users for each scenario to soften the subjectivity of our results. Ideally, we would like to record the behavior of real users interacting with the web application in production, and use these traces in the comparison with the E2E tests. To lessen the number of false positives in the diff, we also plan to exploit the information contained in the screenshots to match widgets more accurately. When the user and the test click on two different element, and thus two different selectors, the screenshots should allow us to recognize that they actually click on the same widget, and that no diff should be reported.

REFERENCES

[1] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Chapter Five - Approaches and Tools for Automated End-to-End Web Testing," in *Advances in Computers*, A. Memon, Ed. Elsevier, Jan. 2016, vol. 101, pp. 193–237. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0065245815000686

[2] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Robula+: an algorithm for generating robust XPath locators for web testing," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 177–204, 2016, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1771. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1771

[3] E. W. Myers, "An O(ND) Difference Algorithm and Its Variations." in *Algorithmica*, 1986, pp. 251–266.

[4] K.-J. Stol and B. Fitzgerald, "The ABC of Software Engineering Research," *ACM Transactions on Software Engineering and Methodology*, vol. 27, pp. 1–51, 2018.

[5] M. Hammoudi, G. Rothermel, and P. Tonella, "Why do Record/Replay Tests of Web Applications Break?" in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2016, pp. 180–190.

[6] A. Stocco, R. Yandrapally, and A. Mesbah, "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 - ESEC/FSE 2018*. Lake Buena Vista, FL, USA: ACM Press, 2018, pp. 503–514. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3236024.3236063

[7] S. Brisset, R. Rouvoy, L. Seinturier, and R. Pawlak, "Erratum: Leveraging Flexible Tree Matching to repair broken locators in web automation scripts," *Information and Software Technology*, vol. 144, p. 106754, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921002020

[8] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "WATER: Web Application TEst Repair," in *Proceedings of the First International Workshop on End-to-End Test Script Engineering - ETSE '11*. Toronto, Ontario, Canada: ACM Press, 2011, pp. 24–29. [Online]. Available: http://portal.acm.org/citation.cfm?doid=2002931.2002935

[9] M. Hammoudi, G. Rothermel, and A. Stocco, "WATERFALL: An Incremental Approach for Repairing Record-replay Tests of Web Applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 751–762. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950294

[10] W. Chen, H. Cao, and X. Blanc, "An Improving Approach for DOM-Based Web Test Suite Repair," in *Web Engineering*, M. Brambilla, R. Chbeir, F. Frasincar, and I. Manolescu, Eds. Cham: Springer International Publishing, 2021, pp. 372–387.

[11] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Reducing web test cases aging by means of robust XPath locators," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 2014, pp. 449–454.

[12] ——, "Using Multi-Locators to Increase the Robustness of Web Test Cases," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2015, pp. 1–10, iSSN: 2159-4848.

[13] Y. Zheng, S. Huang, Z.-w. Hui, and Y.-N. Wu, "A Method of Optimizing Multi-Locators Based on Machine Learning," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Jul. 2018, pp. 172–174.

[14] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Synthesizing Web Element Locators (T)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 331–341.

[15] E. Alégroth, M. Nass, and H. H. Olsson, "JAutomate: A Tool for System- and Acceptance-test Automation," in *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*, Mar. 2013, pp. 439–446, iSSN: 2159-4848.

[16] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Automated generation of visual web tests from DOM-based web tests," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing - SAC '15*. Salamanca, Spain: ACM Press, 2015, pp. 775–782. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2695664.2695847

[17] ——, "Pesto: Automated migration of DOM-based Web tests towards the visual approach," *Software Testing, Verification and Reliability*, vol. 28, no. 4, p. e1665, 2018, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1665. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1665

[18] B. Burg, A. J. Ko, and M. D. Ernst, "Explaining Visual Changes in Web Interfaces," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ser. UIST '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 259–268. [Online]. Available: https://doi.org/10.1145/2807442.2807473

[19] P.-Y. P. Chi, S.-P. Hu, and Y. Li, "Doppio: Tracking UI Flows and Code Changes for App Development," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3173574.3174029

[20] S. Oney and B. Myers, "FireCrystal: Understanding interactive behaviors in dynamic web pages," in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sep. 2009, pp. 105–108, iSSN: 1943-6106.

[21] J. Mickens, J. Elson, and J. Howell, "Mugshot: Deterministic Capture and Replay for Javascript Applications," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11, event-place: San Jose, California. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855722