Tree-Based Synthesis of Web Test Sequences From Manual Actions

Pengkun Jiang[1,2], Sinan Wang[1], and Yepang Liu[1,2,(✉)]

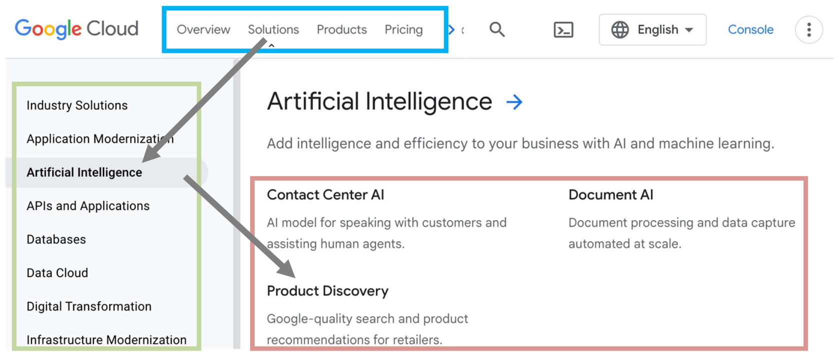
[1]Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China
[2]Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
{12132334,wangsn}@mail.sustech.edu.cn, liuyp1@sustech.edu.cn

Abstract. The thrive of web technologies and applications demands effective testing methods for quality assurance. For this purpose, researchers have proposed various testing techniques to automate the interaction with web applications in recent years. While such techniques help save human testers from repetitive and tedious manual testing tasks, they often only explore limited usage scenarios due to the lack of domain knowledge. In practice, constructing test sequences that align with application business logic still requires substantial manual efforts. In this work, we explore a new paradigm that leverages the synergy between humans and machines. Our key observation is that in many web applications, interactive elements within or across web pages often exhibit similarities in functionality and interaction style. Leveraging this observation, we propose a novel testing method, Fret , which combines manual recording of an example test sequence with automated synthesis of multiple new test sequences that have similar intentions through action mutation. To detect similar elements to the example, Fret employs a programming-by-example technique, in which similar elements are described as the intersection within a well-defined version space. To avoid redundant mutations, Fret builds a tree representation of the mutation process, which helps preserve the logical dependence between actions and monitor the progress of test generation. The experiment results on 32 top-ranking websites show that, given an example sequence, Fret can effectively generate an average of 138 new sequences to interact with the web application under test with a false positive rate of only 0.69% on similar element searching, significantly outperforming the baseline method in terms of covered elements and generated valid test sequences. Additionally, Fret has also led to the discovery of 86 real errors in 19 web applications, which demonstrates its practical usefulness.

Keywords: Web Testing, Sequence Generation, Record and Replay.

1 Introduction

To cope with the growing number of end-users and minimize losses caused by errors, many web applications require thorough testing before being launched



Add intelligence and efficiency to your business with AI and machine learning.

AI model for speaking with customers and assisting human agents.

Document processing and data capture automated at scale.

Google-quality search and product recommendations for retailers.

Fig. 1: Example test sequence on Google Cloud

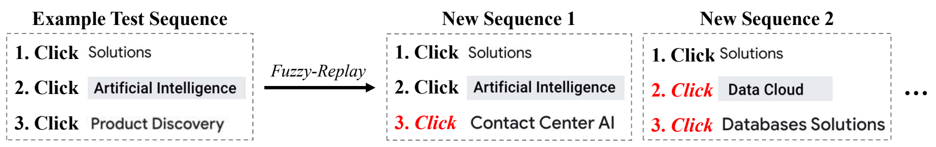

Example Test Sequence
1. Click Solutions
2. Click Artificial Intelligence
3. Click Product Discovery
Fuzzy-Replay
New Sequence 1
1. Click Solutions
2. Click Artificial Intelligence
3. Click Contact Center AI
New Sequence 2
1. Click Solutions
2. Click Data Cloud
3. Click Databases Solutions
...

Fig. 2: Test generation via fuzzy-replay

online. In practice, *automatic end-to-end testing* [8, 39] is often adopted, whose testing objective is maximizing the functional coverage. However, such techniques do not involve domain knowledge of the web under testing (WUT). This limitation can easily trap the testing agents in a local optimal state, preventing them from exploring deeper functionalities of the WUTs.

Many test automation frameworks support recording user actions on a web browser and replaying the recorded actions. Such a *record-and-replay* technique is widely used [20], especially for regression testing purposes. In existing practices of record-and-replay, a recorded user action sequence is usually saved as a definite test script that can be played back later. Compared with automatic testing, record-and-replay allows testers to use their domain knowledge during recording to test some important but hard-to-reach features, so as to improve the effectiveness of regression testing.

In order to thoroughly test a webpage, testers should manually record many action sequences, which can be labor-intensive. Take the Google Cloud homepage in Figure 1 as an example. Testers may record the sequence of click actions as shown in Figure 2 (see Example Test Sequence) and replay the sequence when needed. While this facilitates regression testing, it is not efficient. For instance, to test whether users can still visit the "Contact Center AI" page or pages related to "Data Cloud" after the website evolves, the testers need to record all corresponding action sequences in advance. Due to combinatorial explosion, testers will need to record hundreds of similar sequences to the example one (Figure 2) to ensure complete functional coverage.

Recognizing this limitation, our work aims to enhance the efficiency of conventional record-and-replay techniques. Our key observation is that UI elements within or across web pages may have similarities in their interaction style. For example, in Figure 1, the UI elements in the box of the same color are similar to each other: clicking the elements in the blue box will all lead to the appearance of a dropdown menu such as the one in the green box; clicking the green box elements will all lead to the appearance of a frame that contains more clickable elements (i.e., the red box). By leveraging such element similarity, it is possible to create many new diverse sequences based on one example test sequence. Consider the example test sequence in Figure 2. By replacing the last action "Click 'Product Discovery'" with a click on a similar element "Contact Center AI", we can obtain a new test sequence. By replacing the second action "Click 'Artificial Intelligence'" with a click on a similar element "Data Cloud", we can create more test sequences to explore the pages related to "Data Cloud". We call this process *fuzzy-replay*, in which we obtain new test sequences from an example test sequence by replacing user actions with similar ones while keeping the interaction order. Fuzzy-replay expands an existing manually recorded test sequence into multiple logically-consistent ones, thus significantly improving web test efficiency with the least human effort.

In this paper, we propose a web test generation framework FRET, which stands for Fuzzy-REply for web Testing. FRET formulates the web element selector with *version space algebra* (VSA) [13], which has been adopted in various *programming by example* (PBE) tasks [10, 28]. This formulation describes the selection of similar elements as the intersection operation of directed graphs in a well-defined space. To address the challenge of dynamic content loading in web applications, FRET replays the actions with the selected similar elements (called mutation) in an online manner. In addition, FRET maintains a tree-structure of the mutation process during test generation, which helps preserve the logical relationships between generated actions and monitor the progress of fuzzy-replay. With little manual effort in the recording stage, FRET can quickly learn the interaction logic with domain knowledge, generate new test sequences similar to the example, and thus comprehensively test a web page.

We evaluated FRET on 32 top-ranking websites under different categories. Based on the experimental results, **Fret can generate an average of 138 new test sequences and detect 86 web errors**, with a false positive rate of only 0.69% on similar element searching, which significantly outperforms other baseline methods. In summary, this paper makes the following contributions:

- We propose FRET, which employs PBE-based similar element selection and tree-coloring based action sequencing to generate new test sequences from a manually-recorded web action sequence.
- We have evaluated FRET on 32 top-ranking web applications of different kinds, demonstrating its effectiveness in boosting functional coverage and exposing real web errors based on the manually-recorded examples.

```
{ "url": "https://cloud.google.com",
  "tests": [{
    "commands": [
      { "command": "open" },
      { "command": "click", "target": "xpath=//body/tab[2]/a" },
      { "command": "click", "target": "xpath=//tab[2]/div/div/ul/li[3]/a" },
      { "command": "click", "target": "xpath=//div[3]/ul[4]/li/a/div" },
      { "command": "input", "target": "xpath=//header[1]//input", "value": "AI solution"}
    ]
}]}
```

Listing 1: An snippet of Selenium IDE test script

2 Background and Challenges

Selenium IDE [33] enables human testers to record their web test sequences and replay them as unit test cases using Selenium's WebDriver API, thereby achieving record-and-replay tasks. Listing 1 is a snippet of the Selenium IDE test script for a test sequence in Figure 2. Typically, a test script (or test sequence) consists of a series of test actions:

$$Seq_{\text{example}} := \{act^{(1)}, act^{(2)}, ..., \ act^{(n)}\} \tag{1}$$

There are two types of actions: browser actions and element actions. Browser actions configure the state of the web driver, including opening a URL, resizing the window, and running some JavaScript program, etc. An element action consists of three key components: the action type, the targeting DOM element, and an (optional) interaction value. They correspond to the `command`, `target` and `value` attributes in Listing 1:

$$act^{(i)} := \langle action\_type^{(i)},\ target\_element^{(i)},\ interaction\_value^{(i)} \rangle \tag{2}$$

Here, the *interaction_value* provides the necessary message for executing the given *action_type*. For example, for an input action, this attribute stores the textual input value. As for a selection action, it can be the selected option in the particular `<select>` menu.

A web application usually contains elements that are similar in functionality and interaction pattern. Usually, an element action can also be applied to interact with other similar elements. Such similarity in human-computer interaction patterns provides the opportunity to reuse manual tests to generate new test sequences. Also, we can obtain new test actions by altering the original *interaction_value*. We call this process as **mutation**. As such, the similar actions generated by mutation are called **mutated actions**, as shown below:

$$act\langle typ_0, tgt_0, val_0\rangle \xrightarrow{\text{mutate}} \begin{cases} \hat{act}\langle typ_0, tgt_1, val_0\rangle, \\ \hat{act}\langle typ_0, tgt_2, val_0\rangle, \\ \hat{act}\langle typ_0, tgt_2, val_1\rangle, \\ ... \end{cases} \tag{3}$$

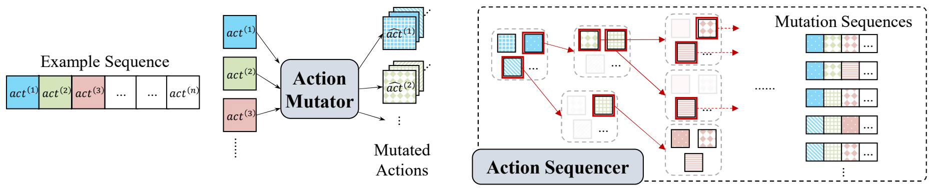


Fig. 3: The overview of Fret

For each element action in the example test sequence, we can obtain multiple mutated actions. By recombining mutated actions in the order of the example test sequence, we can generate new test sequences, which we call **mutation sequences**. Ideally, the mutation sequences will retain the domain knowledge in the example test sequence, thus achieving higher functional coverage for WUT:

$$Seq_{\text{mutation}} := \{\hat{act}^{(1)}, \hat{act}^{(2)}, \dots \hat{act}^{(n)}\} \tag{4}$$

Technically, fuzzy-replay generates new test sequences by mutating and recombining the example test sequence. However, it faces two major challenges:

1. **How to find similar elements?** Due to the diverse implementation styles, we cannot directly obtain the interaction types and functionalities of web elements. Determining the similarity of web elements is hard.
2. **How to recombine mutated actions?** The logical relation between actions in the example test sequence requires recombining the mutated actions in a meaningful order. Otherwise, the mutation sequences can be invalid.

In this paper, we propose Fret, the first framework that generates web GUI test sequences with fuzzy-replay. It explicitly addresses the above challenges with two techniques: PBE-based searching and coloring-based action sequencing. We will illustrate them in the next section.

3 The Fret Test Generation Framework

3.1 Overview

Figure 3 shows the overview of our proposed web GUI test generation framework, Fret. Similar to traditional record-and-replay techniques, Fret also accepts a manually recorded test sequence as input. Then, it tries to generate new test sequences that experience other functionalities of WUT. This is accomplished by two core components: an *Action Mutator* that mutates each action in the example test sequence to generate mutated actions, and an *Action Sequencer* that combines mutated actions with respect to their original ordering in the example. They will be illustrated in detail in the following parts.

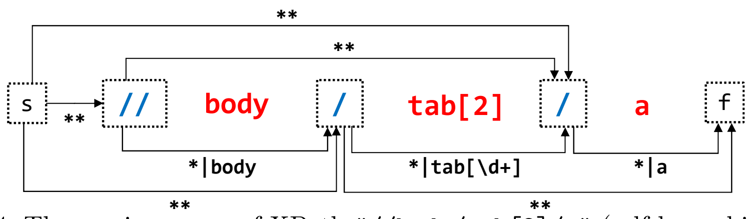


Fig. 4: The version space of XPath "//body/tab[2]/a" (self-loops hidden)

3.2 The Action Mutator

Programming By Example At high-level, the Action Mutator mutates an action by replacing its target element with similar elements. Technically, this is accomplished by selecting elements whose XPaths share common patterns with the original one. Similar elements usually exhibit spatial locality in the webpage (Figure 1), which enables a wildcard element selector to assign common visual property or page layout to all of them. Such intuition makes it possible to formulate our task as a *programming by example* (PBE) problem [28]: given a single element selector (XPath in our task), construct all possible selectors that maximize the similarity criterion on the current web state. We define the syntax of the element selector as follows:

XPath	xp	$\longrightarrow$	$/lvl \mid xp/lvl$
Level	lvl	$\longrightarrow$	$cond \mid$ * $\mid$ **
Condition	$cond$	$\longrightarrow$	regular expression

This syntax can be converted from the original XPaths [21] via simple text processing. Each selector has multiple levels separated by slash symbols (/), while each level has a condition represented by a regular expression or wildcard symbol. Here, a single asterisk (*) matches exactly one level, while a double asterisk symbol (**) matches zero or more levels. To simplify representation, we transform a double slash (//) into a wildcard condition (/**), which preserves its semantics [21]. The level condition *cond* is stored as a regular expression, which is usually a constant string of the node's tag name (see Listing 1). In particular, we convert the indexing operator (e.g., [1]) into a regular expression [\d+], which can match all elements at the same DOM level.

Version Space Algebra of Element Selector The XPath selector syntax above allows a powerful tool *version space algebra* (VSA) to encode an even more general representation of our desired web elements. VSA has been successfully applied in commercial text processing products [10, 28], demonstrating its viability in related tasks. Our version space definition of XPath selectors follows the widget path VSA employed by Jigsaw [17]. However, our XPath selector is more general, as Jigsaw only supports `id` selector as the level condition. As an example, Figure 4 shows the version space representation of an XPath in Listing

Algorithm 1: PBE-based similar element selection

```
   Input: target element $\omega$, all elements $E$
   Output: set of similar elements $E_{\text{similar}}$
1  $G_\Omega \longleftarrow$ version_space($\omega$)
2  while $\exists e \in E.\neg trivial(G_\Omega \sqcap G_e)$ do
3      $e^* \longleftarrow \underset{e \in E \setminus \{\omega\}}{\arg\max} \sum_{x \in \mathcal{X}(G_\Omega \sqcap G_e)}$ match_score($G_\Omega, x$)
4      $E \longleftarrow E \setminus \{e^*\}$
5      $E_{\text{similar}} \longleftarrow E_{\text{similar}} \bigcup \{e^*\}$
6      $G_\Omega \longleftarrow G_\Omega \sqcap$ version_space($e^*$)
```

1. Given the XPath of an element from the example sequence, we first transform it into a directed graph by creating edges:

$$\begin{aligned} E \; = & \{(v_i, v_j, \texttt{**}) \mid 1 \leq i \leq j < n\} \cup \\ & \{(v_i, v_{i+1}, \texttt{*}) \mid 1 \leq i < n\} \cup \\ & \{(v_i, v_{i+1}, cond_i) \mid 1 \leq i < n\} \end{aligned}$$

Here, $cond_i$ is the condition of the i-th level of the XPath, represented by a regular expression mentioned above. Each node v_i is a slash (`/`) in the XPath. There are also two special nodes, s and f, the source and sink nodes. As such, an XPath can be accepted by this directed graph, if each of its level conditions matches regular expressions along a path from the source node s to the sink node f.

Given two VSA directed graphs $G = (V, E)$ and $G' = (V', E')$, an important operation is their *intersection* $G \sqcap G' = (V_\sqcap, E_\sqcap)$, which selects common XPaths accepted by both graphs. Their definitions are:

$$V_\sqcap = \{\langle v, v' \rangle \mid v \in V, v' \in V'\}$$
$$E_\sqcap = \{(\langle u, u' \rangle, \langle v, v' \rangle, l \sqcap l') \mid (u, v, l) \in E, (u', v', l') \in E'\}$$

where the intersection of level conditions is: $l \sqcap l' = l$, if l and l' are identical; or $l \sqcap \texttt{*} = l$; the remaining intersections result in an invalid condition. It has been proven by Li et al. [17] that, this VSA intersection is well-defined and can soundly capture all XPaths acceptable to both G and G'.

Similar Element Selection Algorithm 1 describes how the Action Mutator finds similar elements. All available elements E can be obtained via trivial element selectors on the current web page, while Action Mutator will select those similar to the target element w among them. G_Ω is the directed graph that accepts all desired similar elements. It is initialized with the version space of the target element ω in the example sequence (line 1), and will grow by taking its intersection with the selected similar elements (line 6). Similar to the widget selector synthesis of Jigsaw [17], the Action Mutator also employs an iterative process, until the remaining elements only match a trivial selector (line 2). Here,

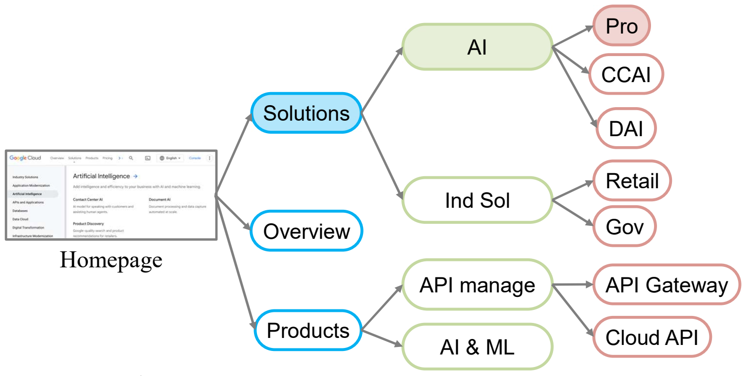


Fig. 5: A mutation tree of the example on Google Cloud

"trivial" means the resultant graph is equivalent to a wildcard selector "`/**/*`". Each time, an element is selected based on the evaluation of a *match score*. It adopts the idea of `FlashFill`'s compatibility score [10], such that matching elements should be selected earlier to avoid computational cost. For an element e, $\mathcal{X}(G_\Omega \sqcap G_e)$ denotes all XPaths selected by its intersection with G_Ω. Intuitively, these XPaths should evaluate to a higher match score with the original element selector G_Ω. Therefore, we prefer the one that achieves the maximum sum of match scores (line 3). Here, `match_score` counts the number of *exact matching* of level conditions in an XPath.

3.3 The Action Sequencer

With the mutated actions, the next step is to combine them into new sequences. FRET employs *online action sequencing*, which is executed during the interaction with WUT. Online sequencing validates the mutated actions in real-time. As a result, all the generated test sequences will be valid.

Mutation Tree In a manually recorded sequence, the execution order of actions describes the use case scenario from the testers. This logical dependence encodes domain knowledge such that subsequent actions could be executed after the preceding action, while the preceding action should be executed before its succeeding actions. We model such logical dependence as a tree structure, which we call a *mutation tree*.

In a mutation tree (Figure 5), each node saves an action from the example test sequence, or its mutated actions discovered by the Action Mutator. The descendants of each node compose all possible subsequent actions that can be executed afterwards. In particular, the root node indicates the test entry and does not save any action. The mutation tree has two characteristics: first, the target elements between each layer of the mutation tree are similar to each other; second, a path from the root node to any leaf node is a legal test sequence. During test generation, the mutation tree is expanded from the current web

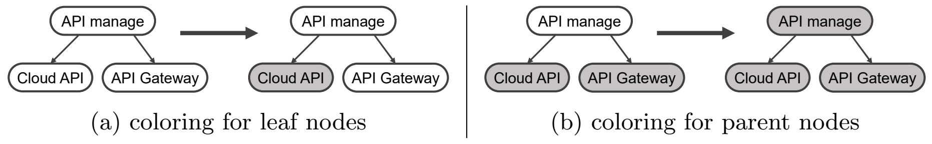


Fig. 6: Tree-coloring based action sequencing

state, such that all similar elements of the current action's target element will become the children nodes of the current node. To avoid repeating elements or loops, each node and its incoming edge will be added to the tree by exactly once. This constraint ensures maintaining the mutation tree property. Once all actions from the example sequence are mutated and the mutation tree along this path is fully expanded, the Action Sequencer will start from the root node and continue to visit other paths, until the mutation tree can no longer grow. Therefore, the mutation tree provides a cache for mutated actions and quantifies the test sufficiency criterion.

Tree-coloring Based Sequencing As action sequencing progresses, the proportion of unselected mutated actions decreases. To eliminate the potential of generating duplicate sequences, Action Sequencer adopts *tree-coloring based action sequencing* to expand the mutation tree. The key idea is that, each path from the root node to a leaf node in the mutation tree can form a valid test sequence. Since each leaf node is unique, there is only one path from the root node to a specific leaf node. When a leaf node is selected to generate a test sequence, it is colored to indicate that this node should not be selected again in the future. There are two coloring strategies:

– Figure 6a: for a leaf node, it will be colored every time it is executed, to indicate it should not be selected again.
– Figure 6b: for a non-leaf node, once its children nodes have all been colored, selecting it again will produce duplicate test sequences. Therefore, we color this node to avoid such cases.

When selecting the next mutated action to execute, an uncolored node in the mutation tree shall be selected to avoid duplicate sequences. When the root node is colored, it indicates a corresponding test sequence has been generated for each functional point in WUT, and the test generation shall terminate. All newly generated sequences will be output as test sequences.

Theoretically, the Action Sequencer performs a *post-order depth-first search* (DFS) on the mutation tree, in which visiting the web state corresponds to the coloring operation on the tree node. Meanwhile, a backtrack operation should be implemented to ensure the proper execution of DFS. To achieve this goal, the Action Sequencer will store the current web state, which will be loaded back once all its child nodes are colored. Nevertheless, the store and reload operations will not be recorded in the generated test sequences.

An important objective of testing is to discover errors. Action Sequencer will record errors captured by the web browser during the online sequencing process. These recorded errors indicate the ability of FRET to expose real web bugs. What's more, they can form the test oracles [27] in the generated test cases. However, there is one type of error that should not be regarded as a real bug, that is, the target element is not found. We will drop such errors and consider the corresponding test sequences invalid.

4 Evaluation

4.1 Research Questions

To evaluate the performance of FRET, we investigated three research questions:

- **RQ1 (Similar Element Detection)**: Can the Action Mutator effectively identify similar elements?
- **RQ2 (Mutation Sequence Legality)**: How many valid mutation sequences can be generated by the Action Sequencer?
- **RQ3 (Mutation Sequence Redundancy)**: To what extent can FRET avoid generating redundant test sequences?
- **RQ4 (Tool Usefulness)**: Can FRET outperform the baseline methods in generating unique valid tests and triggering real web errors?

4.2 Subject Websites

To select subject WUTs, we referred to Semrush[1]. The website provides traffic rankings for global public websites under 32 different categories. Within its indexed websites, we selected our subjects based on the following criteria:

- The website should not have an anti-robot system like reCAPTCHA. FRET generates test sequences and automatically replays them with Selenium. It, however, cannot automatically pass the anti-robot system.
- The website's core functionality should not rely on specific text input. For example, the Google homepage does not contain too many interactive elements. Most of its functionalities should be accessed through inputs on the search bar. However, currently FRET cannot purposefully mutate text inputs.

Finally, we selected 32 top-ranking websites for our experiment.

4.3 Experimental Setup

FRET was implemented on top of Selenium and Selenium IDE [33]. It supports the mutation of element click actions and input texts (random modification). We recorded one example test sequence for each of the 32 WUTs. We ensure all recorded sequences can effectively experience the WUTs' core functionalities. All experiments were run with a Chromium browser and corresponding Webdriver, on a Macbook with a 2GHz i5 processor and 16GB RAM.

[1] `https://www.semrush.com/website/top/global/all/`

4.4 RQ1: Similar Element Detection

Metric We evaluated the performance of the Action Mutator using *false positive rate* (FPR). During test generation, once the Action Mutator generates invalid actions (or dissimilar actions), the subsequent example actions will not be able to playback. Therefore, its precision is crucial for preserving the legality of the generated sequences. We saved the results of similar element selection and manually determined whether they were valid similar elements. In other words, we shall manually check whether the mutated actions can experience similar functionalities to their original example actions.

Result We manually identified 6,275 similar elements selected by Action Mutator, which correspond to 116 elements across 32 example test sequences. Among them, 43 elements from four WUTs (Advertising, Airlines, Entertainment, and Restaurants) were determined to be dissimilar to their corresponding original elements, resulting in an FPR of 0.69% for the similar element searching algorithm. By investigating the misjudging elements, we found that 34 of them were due to regular expression simplification of the element indexes, while the remaining were selected due to the same special tag names.

Comparison For comparison, we also implemented a naive similar element selection algorithm, that is, **only selecting the elements in the same DOM level that acquire identical tag names and class names**. This simple rule only reveals 5,034 valid similar elements being selected (FPR=19.8%), which is a significant performance downgrade. Moreover, this simple rule could not identify similar elements in eight example sequences. Compared to the PBE-based similar element selection, the naive algorithm fails to capture the inter-relationship between similar elements. Nevertheless, we still found that the simple algorithm discovers non-overlap valid similar elements compared to the PBE-based approach. The main reason is that similar elements may have same class yet different tag names. In the future, we will improve the precision of the Action Mutator by cooperating with the DOM attributes in the algorithm.

4.5 RQ2: Mutation Sequence Legality

Baseline To compare with FRET's Action Sequencer, we implemented a baseline method, called *offline action sequencer*. This baseline simply takes a Cartesian product over all mutated actions with respect to their parents' ordering in the example sequence. That is, for the sequence in (1), and the mutated actions for each of its actions $A^{(i)} = \{\hat{act}_k^{(i)} \mid 1 \leq k \leq \left|E_s^{(i)}\right|\}$, the offline action sequencer will output all sequences produced by $A^{(1)} \times A^{(2)} \cdots \times A^{(n)}$. It is realized by performing Algorithm 1 during the manual recording stage, thus no Internet connection is needed during offline action sequencing.

Table 1: Experiment result of RQ2

WUT	**example seq. length**	**#seq. (offline)**	**#sample seq. (offline)**	**#legal seq. (offline)**	**#seq. (Fret)**
Agoda	5	74,030	382	0	63
Doubleclick	4	35,420	380	0	372
Ryanair	6	2,826,252	384	0	109
Chase	3	168,480	384	0	102
Sephora	3	55,680	381	0	138
Openai	4	58,562	381	2	289
Stackoverflow	4	23,440	377	1	93
Onliner	3	3,386,812	384	0	89
Youtube	4	1,695,200	384	0	162
Archive.	4	162,150	383	0	341
Shein	4	1,617,000	384	0	197
Paypal	3	37,352	380	1	213
All Recipes	4	74,700	382	0	289
Steam	3	603,720	383	0	52
NIH	3	90,896	382	0	74
Wordpress	3	617,760	383	0	20
Robobank	4	169,664	383	0	292
Upstox	3	304,128	383	0	55
ADP	4	44,649	380	0	119
Qualtrics	3	26,895	378	1	161
Genius	3	41,040	380	0	81
Wikipedia	4	509,184	383	0	236
Google Cloud	3	31,140	379	1	84
Redfin	4	58,590	381	0	123
Uber eats	3	110,700	382	1	125
Sciencedirect	4	536,130	383	0	42
Marca	4	55,440	381	0	96
Samsung	4	57,456	381	0	59
Usps	3	32,480	379	1	105
Bahn	3	54,272	381	0	121
Av.by	4	78,064	382	0	85
Mayoclinic	5	25,984	378	0	52

Metric We evaluate the effectiveness of the Action Sequencer using *legality rate of action sequences*. To be specific, if a generated test sequence produces errors like “fail to locate the target element”, we consider it illegal. Since the number of sequences generated by the offline action sequencer is exponentially large, validating all of them is impossible. Therefore, we randomly sampled the results using a 95% confidence interval and a 5% margin of error [37], and ran them to compute the legality rate under statistical significance.

Result Table 1 shows the results. For all WUTs, the offline action sequencer generated more than 20,000 action sequences. Among our randomly-sampled sequences, most of them cannot be exactly replayed. Moreover, in 25 WUTs, the offline action sequencer did not output any legal sequences, indicating that its legality rate is extremely low. On the other hand, FRET’s Action Sequencer pro-

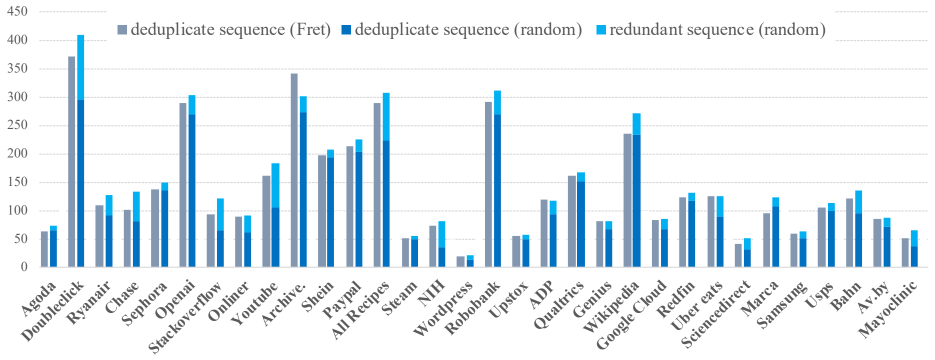


Fig. 7: Experiment result of RQ3

duced only legal action sequences, as they were generated in an online manner. Online sequencing ensures that only interactable web elements are selected as target elements in the mutation sequences. In conclusion, FRET's online action sequencing algorithm, compared to the offline version, can produce significantly more legal test sequences.

4.6 RQ3: Mutation Sequence Redundancy

Baseline We introduce a *random online action sequencer* as the baseline to compare with FRET's Action Sequencer in their generated redundant sequences. The random sequencer will randomly pick a mutated action from all available ones during the online replay stage. Unlike FRET, it cannot determine whether the replay is finished; thus, it will endlessly (and repeatedly) mutate all available actions from the example sequence.

Metric We count identical sequences as the *number of redundant sequences* generated by the random sequencer to investigate redundancy in test generation. The coloring-based sequencer can determine whether the replay is complete, while the random sequencer cannot. Hence, for each WUT, we record the duration spent by FRET's Action Sequencer, and run the random sequencer for the same duration for fair comparison. In RQ2, we have already shown that the offline sequencer can only produce a few legal sequences. Therefore, in RQ3, we will not evaluate it due to its poor performance.

Result Figure 7 shows the total number of sequences and redundant sequences generated by the random sequencer. The random sequencer generated a certain amount of redundant sequences for all 32 WUTs. Among them, in NIH, it generated the highest proportion of redundant sequences, which accounts for 55%. An analysis of the WUTs with a high redundancy rate revealed that some of their elements may result in states with no subsequent actions. Due to the lack of memorization, the random sequencer will repeatedly select such elements and produce a large amount of redundancy (e.g., repeated short sequences). False positive elements in "Archive." caused unhandled exceptions. Handling these web errors consumes time. Without a coloring mechanism, the random sequencer will trigger these errors repeatedly, thus wasting more time but generating fewer new sequences. Except for "Archive.", the total number of randomly generated sequences is greater than that of tree-coloring based sequencing.

Figure 7 also shows the number of generated action sequences by two strategies after deduplication. Among the four out of 32 WUTs, FRET generated more than 300 action sequences, while in Doubleclick, it generated the most unique sequences, with a number of 372. In 13 WUTs, FRET generated less than 100 action sequences, and Wordpress had the fewest unique sequences. There are mainly two reasons for its poor performance: 1) the slow loading speed of web pages reduces the number of interactive elements; 2) the functionalities involved in the example test sequence are limited and there are no sufficient similar elements to produce more action sequences. It indicates that the number of action sequences generated by FRET is greatly influenced by the subject websites and the example test sequences provided by the human testers. Nevertheless, FRET still generated a considerable amount of valid and unique test sequences.

A deep analysis of the number of generated action sequences by the two strategies shows that, within 2 hours, the coloring-based online sequencer successfully traversed the mutation tree of three WUTs (NIH, Sciencedirect, and Mayoclinic) and generated all possible test sequences based on the example test sequences. In the three WUTs, the number of action sequences generated by the random online sequencer is significantly lower. This indicates that while FRET's tree-coloring based action sequencer generates all possible test sequences, the random sequencer has a significant disadvantage. Among 12 WUTs, the differences in the number of generated action sequences between the two strategies did not exceed 10, indicating there is no significant performance difference between them. Among 30 WUTs, the coloring-based online sequencer generated more new test sequences than the random online sequencer. Due to the redundancy of random online sequencing, the coloring mechanism has better performance, even though it has overhead of maintaining the mutation tree.

4.7 RQ4: Tool Usefulness

Following previous web testing work [34, 39], we also count the *number of web errors* triggered by different strategies to measure their ability to detect bugs. There were a total of 86 unique web errors found by FRET on 19 different WUTs during our experiment, while the random sequencer only triggered 73 web errors. Since the offline sequencer is poor at producing valid test sequences, we cannot further count the number of web errors it can trigger.

5 Threats To Validity

There are several potential threats to the validity of our experiments. Firstly, we conducted our experiments on 32 WUTs; however, our selection process focused on high-traffic webs without anti-robot systems within each web category. This means that FRET may not be directly applicable to all web applications. In such cases, it could potentially introduce unknown errors or lead to an increased false positive rate when identifying similar elements. Secondly, Action Mutator extends the example element's XPath selector to other syntactically similar elements within the same web state. However, due to their inherent complexity and diversity, it is unlikely that the generated tests can cover all elements. This makes it challenging to achieve full functional coverage. Moreover, FRET currently can only handle limited types of web actions (clicking, mouseover, and random text input). In order to improve the logical coherence of the generated mutation actions, it is essential to purposefully mutate the interaction value.

6 Related Work

• **Record-and-replay.** Record-and-replay method has shown its practical uses on diverse platforms, including web applications [1, 33] and mobile apps on Android [18,26] or iOS [29]. However, most GUI tests are prone to break [22,31], and many techniques were proposed to improve the robustness of GUI element locators [14–16,25]. There were also techniques leveraging machine learning methods to fix the broken recorded test cases [6, 11, 20, 35]. FRET directly utilizes the recorded test script and generates new tests with PBE-based similar element selection and tree-coloring based action sequencing.
• **Test generation.** The techniques generate web test scripts based on the DOM structure of WUT. To measure the sufficiency of test generation, a common approach is to utilize state flow graphs (SFGs) of WUTs. An SFG abstracts web pages into different states, and uses user events that trigger state transition as edges. Based on SFG, search-based algorithms generate test paths as unit test cases [2–5, 7, 23, 36]. However, it is difficult to determine whether different generated paths have logical dependencies. FRET also generates new tests, from which the business logic (domain knowledge about WUT) is represented by the human-recorded examples.
• **Similar element detection.** Web GUI testing techniques use varying criteria to determine the similarity between web elements. For example, [3, 7] compare the similarity between web states and select states with significant differences in similarity to achieve higher coverage. [6, 20, 24, 30] find elements corresponding to the target element in the WUT's new version. They rely on the elements' attributes to determine element correspondence. FRET involves Action Mutator, which detects similar GUI elements in the current web page using a PBE-based element selection algorithm.
• **Test reuse.** Test reuse is a technique that uses existing test scripts to generate new tests. One application scenario is to extend tests of a web application to another WUT with similar functionality [12,30,32]. Test reuse can also enhance automated testing. It often builds an initial SFG based on the example tests, and then uses a web crawler to extend the SFG. When searching for new test paths, the web page state is abstracted and the similarity between the newly selected

state and the corresponding state of the example test will be compared. The new test path will be constructed based on the most similar state [19, 24]. Test reuse is similar to FRET's approach. However, test reuse relies on human-written scripts, while FRET does not require explicit test scripting.

7 Conclusions and Future Work

We proposed FRET, a fuzzy-replay framework to facilitate web testing by synthesizing test sequences from a human-recorded example. With such an example, FRET expands the test actions to similar target elements through PBE-based element selection, and connects these selected elements with a tree-coloring based sequencing algorithm, thus generating new test sequences similar to the example one. Our experiments show that FRET , compared to two baseline methods, can generate a considerable number of valid test sequences for 32 top websites under different categories. FRET also successfully improves the number of covered elements during testing and identifies many real web errors.

In the future, we will further improve FRET's similar element searching algorithm. We plan to add visual attributes [38] or use large language models (LLMs) to assist in determining similar elements [9] and find more similar elements that the current Action Mutator may miss. Additionally, FRET can be combined with other test generation techniques [3,7,23] to access hard-to-reach web states, thus increasing the overall functional coverage.

Acknowledgements. We thank all reviewers for their valuable comments, which have helped improve this paper. This work is supported by the National Natural Science Foundation of China (Grant Nos. 61932021 and 62372219).

References

1. Andrica, S., Candea, G.: Warr: A tool for high-fidelity web application record and replay. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN). pp. 403–410. IEEE (2011)
2. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in dynamic web applications. In: Proceedings of the 2008 international symposium on Software testing and analysis. pp. 261–272 (2008)
3. Biagiola, M., Ricca, F., Tonella, P.: Search based path and input data generation for web application testing. In: Search Based Software Engineering: 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings 9. pp. 18–32. Springer (2017)
4. Biagiola, M., Stocco, A., Ricca, F., Tonella, P.: Diversity-based web test generation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 142–153 (2019)
5. Choudhary, S.R., Prasad, M.R., Orso, A.: Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. ICST 12, 171–180 (2012)

6. Choudhary, S.R., Zhao, D., Versee, H., Orso, A.: Water: Web application test repair. In: Proceedings of the First International Workshop on End-to-End Test Script Engineering. pp. 24–29 (2011)
7. Dincturk, M.E., Jourdan, G.V., Bochmann, G.V., Onut, I.V.: A model-based approach for crawling rich internet applications. ACM Transactions on the Web (TWEB) 8(3), 1–39 (2014)
8. Fan, Y., Wang, S., Wang, S., Liu, Y., Wen, G., Rong, Q.: A comprehensive evaluation of q-learning based automatic web GUI testing. In: Tenth International Conference on Dependable Systems and Their Applications (2023)
9. Gilardi, F., Alizadeh, M., Kubli, M.: Chatgpt outperforms crowd workers for text-annotation tasks. Proceedings of the National Academy of Sciences 120(30), e2305016120 (2023)
10. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM Sigplan Notices 46(1), 317–330 (2011)
11. Hammoudi, M., Rothermel, G., Stocco, A.: 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. pp. 751–762 (2016)
12. Hu, G., Zhu, L., Yang, J.: Appflow: using machine learning to synthesize robust, reusable UI tests. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 269–282 (2018)
13. Lau, T.A., Domingos, P.M., Weld, D.S.: Version space algebra and its application to programming by demonstration. In: ICML. pp. 527–534 (2000)
14. Leotta, M., Ricca, F., Tonella, P.: Sidereal: Statistical adaptive generation of robust locators for web testing. Software Testing, Verification and Reliability 31(3), e1767 (2021)
15. Leotta, M., Stocco, A., Ricca, F., Tonella, P.: Using multi-locators to increase the robustness of web test cases. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). pp. 1–10. IEEE (2015)
16. Leotta, M., Stocco, A., Ricca, F., Tonella, P.: Robula+: An algorithm for generating robust XPath locators for web testing. Journal of Software: Evolution and Process 28(3), 177–204 (2016)
17. Li, C., Jiang, Y., Xu, C.: Push-button synthesis of watch companions for Android apps. In: 2022 International Conference on Software Engineering (ICSE). IEEE (2022)
18. Li, X., Jiang, Y., Liu, Y., Xu, C., Ma, X., Lu, J.: User guided automation for testing mobile apps. In: 2014 21st Asia-Pacific Software Engineering Conference. vol. 1, pp. 27–34. IEEE (2014)
19. Lin, J.W.: Advancing Automated Software Testing Through Test Reuse. University of California, Irvine (2021)
20. Long, Z., Wu, G., Chen, X., Chen, W., Wei, J.: Webrr: self-replay enhanced robust record/replay for web application testing. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1498–1508 (2020)
21. MDN: XPath - MDN - developer.mozilla.org. `https://developer.mozilla.org/en-US/docs/Web/XPath` (2023)
22. Memon, A.M., Soffa, M.L.: Regression testing of GUIs. ACM SIGSOFT software engineering notes 28(5), 118–127 (2003)

23. Mesbah, A., Van Deursen, A., Roest, D.: Invariant-based automatic testing of modern web applications. IEEE Transactions on Software Engineering 38(1), 35–53 (2011)
24. Milani Fard, A., Mirzaaghaei, M., Mesbah, A.: Leveraging existing tests in automated test generation for web applications. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 67–78 (2014)
25. Montoto, P., Pan, A., Raposo, J., Bellas, F., López, J.: Automated browsing in AJAX websites. Data & Knowledge Engineering 70(3), 269–283 (2011)
26. Nguyen, B.N., Robbins, B., Banerjee, I., Memon, A.: Guitar: an innovative tool for automated testing of GUI-driven software. Automated software engineering 21, 65–105 (2014)
27. Pezze, M., Zhang, C.: Automated test oracles: A survey. In: Advances in computers, vol. 95, pp. 1–48. Elsevier (2014)
28. Polozov, O., Gulwani, S.: Flashmeta: A framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 107–126 (2015)
29. Qin, Z., Tang, Y., Novak, E., Li, Q.: Mobiplay: A remote execution based record-and-replay tool for mobile applications. In: Proceedings of the 38th International Conference on Software Engineering. pp. 571–582 (2016)
30. Rau, A., Hotzkow, J., Zeller, A.: Transferring tests across web applications. In: International Conference on Web Engineering. pp. 50–64. Springer (2018)
31. Ricca, F., Leotta, M., Stocco, A.: Three open problems in the context of e2e web testing and a vision: Neonate. In: Advances in Computers, vol. 113, pp. 89–133. Elsevier (2019)
32. Roy Choudhary, S., Prasad, M.R., Orso, A.: Cross-platform feature matching for web applications. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 82–92 (2014)
33. SeleniumHQ: Selenium: A browser automation framework and ecosystem. `https://github.com/SeleniumHQ/selenium/` (2004)
34. Sherin, S., Muqeet, A., Khan, M.U., Iqbal, M.Z.: Qexplore: An exploration strategy for dynamic web applications using guided search. Journal of Systems and Software 195, 111512 (2023)
35. Stocco, A., Yandrapally, R., Mesbah, A.: 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. pp. 503–514 (2018)
36. Thummalapenta, S., Lakshmi, K.V., Sinha, S., Sinha, N., Chandra, S.: Guided test generation for web applications. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 162–171. IEEE (2013)
37. Wang, S., Wen, M., Liu, Y., Wang, Y., Wu, R.: Understanding and facilitating the co-evolution of production and test code. In: 2021 IEEE International conference on software analysis, evolution and reengineering (SANER). pp. 272–283. IEEE (2021)
38. Yu, S., Fang, C., Yun, Y., Feng, Y.: Layout and image recognition driving cross-platform automated mobile testing. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1561–1571. IEEE (2021)
39. Zheng, Y., Liu, Y., Xie, X., Liu, Y., Ma, L., Hao, J., Liu, Y.: Automatic web testing using curiosity-driven reinforcement learning. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 423–435. IEEE (2021)