CrawLabel: Computing Natural-Language Labels for UI Test Cases

Yu Liu*
yuki.liu@utexas.edu
University of Texas at Austin
Austin, TX, USA

Rahulkrishna Yandrapally
rahulyk@ece.ubc.ca
University of British Columbia
Vancouver, BC, Canada

Anup K. Kalia†
akalia@dataminr.com
Dataminr, Inc
NY, USA

Saurabh Sinha
sinhas@us.ibm.com
IBM Research
Yorktown Heights, NY, USA

Rachel Tzoref-Brill
rachelt@il.ibm.com
IBM Research
Haifa, Israel

Ali Mesbah
amesbah@ece.ubc.ca
University of British Columbia
Vancouver, BC, Canada

ABSTRACT

End-to-end test cases that exercise the application under test via its user interface (UI) are known to be hard for developers to read and understand; consequently, diagnosing failures in these tests and maintaining them can be tedious. Techniques for computing natural-language descriptions of test cases can help increase test readability. However, so far, such techniques have been developed for unit test cases; they are not applicable to end-to-end test cases.

In this paper, we focus on the problem of computing natural-language labels for the steps of end-to-end UI test cases for web applications. We present two techniques that apply natural-language processing to information available in the browser document object model (DOM). The first technique is an instance of a supervised approach in which labeling-relevant DOM attributes are ranked via manual analysis and fed into label computation. However, supervised approach requires a training dataset. So we propose the second technique, which is unsupervised: it leverages probabilistic context-free grammar learning to compute dominant DOM attributes automatically. We implemented these techniques, along with two simpler baseline techniques, in a tool called CrawLabel (available as a plugin to Crawljax, a state-of-the-art UI test-generation tool for web applications) and evaluated their effectiveness on open-source web applications. Our results indicate that the supervised approach can achieve precision, recall, and F1-score of 83.38, 60.64, and 66.40, respectively. The unsupervised approach, although less effective, is competitive, achieving scores of 72.37, 58.12, and 59.77. We highlight key results and discuss the implications of our findings.

1 INTRODUCTION

End-to-end tests play an important role in functional testing in practice. The goal of end-to-end testing is to exercise the Application Under Test (AUT) through its user interface (UI) in a manner similar to how an end-user would navigate the application. Because of their focus on end-user perspective and coverage of end-to-end application flows, these tests are also useful for acceptance testing. Such tests can be written manually, created using test automation tools such as record-replay [1, 15], or generated through automatic/semi-automatic test generation techniques (e.g., [6, 19, 25, 32, 33, 46]).

Listing 1: End-to-end test (without labels)

```
1 def Test1():
2   driver.loadURL("Base_URL")
3   driver.findElement("//*[@id='Content']/p[1]/a").click()
4   driver.findElement
5     ("//*[@id='SearchContent']/form/input[1]").enter("Bulldog)
6   driver.findElement
7     ("//*[@id='SearchContent']/form/input[2]").click()
8   driver.findElement("//*[@id='BackLink']/a").click()
9   driver.findElement("//*[@id='SidebarContent']/a[1]").click()
```

Regardless of how they are created, end-to-end tests are notoriously hard to maintain as they are fragile [17, 20] and can break in response to minor changes to the UI. Such test breakages require costly human effort in practice despite research efforts to design techniques that automatically make tests more resilient to changes [25, 31, 49] and repair broken tests [12, 23, 45]. A key challenge for a human tester to fix breakages is the difficulty in understanding the tests. Existing research efforts to improve maintainability are limited to creation of page objects [44], which helps reduce duplication in human maintenance effort but does not address test comprehension directly.

Test comprehension in general is a known challenge [16] that hinders adoption of automated testing techniques as it makes test maintenance a challenging task for developers. Several techniques have been developed to facilitate test comprehension such as summarizing or documenting test cases [26] and creating meaningful test case names [13]. However, existing techniques focus on unit test cases; they are not applicable to end-to-end tests, which is the focus of this work.

Unlike a unit test, which targets at program's methods or functions, an end-to-end test consists of actions (e.g., clicks) performed on the UI of the program, often via APIs of a testing framework such as Selenium [42]. These interactions with the UI can make such tests particularly hard to understand.

To illustrate this, consider the end-to-end test shown in Listing 1, implemented using the Selenium WebDriver API. After loading the AUT's URL in line 2, each test step locates a UI element using an XPath-based locator and clicks the identified element. Without

* Author was an intern at IBM Research at the time of this work.
† Author was with IBM Research when this work was done.

AST '22, May 17–18, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9286-0/22/05…$15.00
https://doi.org/10.1145/3524481.3527229

Listing 2: End-to-end test (with labels)

```
def Test1():
  driver.loadURL("Base_URL")
  # enter the store
  driver.findElement("//*[@id='Content']/p[1]/a").click()
  # search product
  driver.findElement
    ("//*[@id='SearchContent']/form/input[1]").enter("Bulldog)
  driver.findElement
    ("//*[@id='SearchContent']/form/input[2]").click()
  # return to main menu
  driver.findElement("//*[@id='BackLink']/a").click()
  # view category
  driver.findElement("//*[@id='SidebarContent']/a[1]").click()
```

suitable comments that describe the actions performed by the test steps, such a test case can be hard to understand; the user essentially has to identify the target UI element for each action from the XPath locator of the element (e.g., using browser developer tools) and piece together the exercised UI flow. This is a cumbersome manual task that can be especially tedious for tests that navigate long and complex navigational flows. For test steps with broken locators, this manual task becomes even harder as the target element cannot be manually identified even using browser tools. Tests that lack proper comments (or obsolete comments) pose comprehension challenges.

The goal of our work is to automatically generate labels, such as the labels shown in Listing 2, for end-to-end tests. The idea is to make such tests more comprehensible for developers by adding descriptive information for each test step. In this work, we focus on automatically generated tests, but our technique can also be applied to tests created by record–replay tools or written manually.

Unit-level test-summarization techniques [26, 38] attempt to generate natural-language descriptions of test steps by analyzing the called application methods (their names, descriptions, and code). Such approaches are not applicable to end-to-end tests. Summarizing the steps of a UI test must be done in terms of the visual actions performed by the steps on the application UI. For instance, for web applications this involves analyzing the browser document object model (DOM) and computing natural-language descriptions of events triggered on the DOM elements by the test steps.

Overall, our approach applies natural-language processing (NLP) to the information available in the browser DOM and computes natural-language labels that describe the actions performed in UI test steps. It takes as input an end-to-end test represented as a path in which nodes represent DOM states and edges represent events that cause transitions from the source DOM state to the target DOM state. It produces as output a set of natural-language labels, where each label is associated with an edge of the input path. To compute the label for an edge, the technique analyzes DOM elements, attributes and their contents using standard NLP methods, such as tokenization, lemmatization, and removal of non-English words, followed by identification of keywords.

A key challenge in extracting a meaningful label for a test action concerns the importance assigned to keywords extracted from DOM analysis. Inclusion of keywords unrelated to the semantics of the test action can generate labels that are meaningless and confusing for developers, impacting the end goal of making UI tests comprehensible. To tackle this challenge, we present two techniques for identifying the DOM attributes on which NLP should be applied for label computation.

The first technique is a supervised approach in which the labeling-relevant DOM attributes are manually identified and ranked on a set of web applications (the training set), and then used for computing labels. An important limitation of the supervised approach is that it may not generalize to novel web frameworks. Thus, we present the second technique, which is unsupervised. The approach computes the probability of verb/verb phrases and noun/noun phrases by first applying parts-of-speech (POS) tagging to each DOM attribute and then generating the parse trees for all tagged attributes of clickable elements via probabilistic context-free grammar learning [11, 21, 22]. The verb/verb phrase and noun/noun phrase with highest probability are extracted as the label.

We implemented our labeling techniques in a tool called CrawLabel. CrawLabel leverages various open-source tools—KeyBERT [3], the Stanford CoreNLP toolkit [29], and the NLTK library [28]—to generate test labels. To add comments in test code, CrawLabel uses the plugin architecture of Crawljax [2].

We evaluated the effectiveness of the supervised and unsupervised labeling techniques, along with two baseline techniques, using a corpus of 13 open-source web applications that have been used in prior empirical studies of web application testing techniques [5, 7]. We generated end-to-end tests using Crawljax and created a ground truth of labels for the steps of each test case. We divided the 13 applications into a training set of five applications and a testing set of eight applications; the first set was used for creating rules for the supervised approach; the second set was used for evaluating the four competing techniques. We used a set of metrics (precision, recall, F1 score, and edit similarity) to measure the accuracy of the four techniques against the ground truth.

Our results show that both the supervised and unsupervised approaches can be quite effective in computing labels; moreover, they both outperform the baseline techniques with a difference that is statistically significant. On average, the supervised approach achieves precision, recall, F1 score, and edit-similarity score of 83.38, 60.64, 66.40, and 73.27, respectively. The unsupervised approach, although less effective, is quite competitive, achieving scores of 72.37, 58.12, 59.77, and 66.97, respectively. These results are promising because the unsupervised approach does not require a training set of web applications for computation and ranking of labeling-relevant DOM attributes. We note, however, that there is also scope for improvement in both techniques.

Our results also emphasize the key roles played by clickable elements—often, analyzing just the clickable element is sufficient to get useful labels about test steps—and certain HTML attributes (e.g., `text` and `href`) that are more likely to contain labeling-relevant information than other HTML attributes. We discuss the implications of our findings for further research on the topic of summarizing end-to-end UI test cases.

The contributions of this work are:

- A first investigation of the readability and understandability aspects of automatically generated end-to-end test cases.
- Novel supervised and unsupervised techniques that apply NLP to DOM attribute to compute labels for test steps that interact with the UI.
- Empirical assessment of the effectiveness of the labeling techniques, with comparison against baseline techniques.

We provide a replication package consisting of the CrawLabel implementation, benchmark applications, UI test cases, and evaluation artifacts [4].

Figure 1: Sample states of the Jpetstore web application

Figure 2: UI path for the example test case

Figure 3: Example web page, DOM, and clickable with XPath locator

2 BACKGROUND

In this section, we provide background information on UI testing and NLP techniques leveraged in our approach.

2.1 UI Testing for Web Apps

UI testing for web apps is typically performed through automated UI test cases (e.g., the example shown in Listing 1) that employ browser automation tools such as selenium in order to perform user interactions on the web application. UI test cases can be either manually written, created using record-replay tools or even automatically generated using test generation tools.

Definition 1. [**Application State** ($\mathcal{S}$)] is a tuple ($\mathcal{D}$, $\mathcal{V}$, [α...]) where $\mathcal{D}$ is the dynamic DOM of the page, $\mathcal{V}$ is the screenshot of the page and [α..] is the list of clickable elements in the page.

Definition 2. [**State Transition** ($\mathcal{A}_x$)] is a tuple ($\mathcal{S}^x_{src}$, α_x, $\mathcal{S}^x_{tgt}$) where exercising a clickable α_x in a state $\mathcal{S}^x_{src}$ produces a transition to state $\mathcal{S}^x_{tgt}$.

Definition 3. [**Path** ($\mathcal{P}$)] A sequence of transitions ($\mathcal{A}_0$...$\mathcal{A}_n$) is a $\mathcal{P}$ if for 0<=i<n, $\mathcal{A}_i$, $\mathcal{A}_{i+1} \in \mathcal{P} \implies \mathcal{A}_i(\mathcal{S}_{tgt}) == \mathcal{A}_{i+1}(\mathcal{S}_{src})$.

Definition 4. [**TestCase** $\mathcal{T}$] Given a path $\mathcal{P}$ = ($\mathcal{A}_0$...$\mathcal{A}_n$), a test case can be represented as [$\mathcal{S}^0_{src}$, α_0, $\mathcal{S}^0_{tgt}$, α_1, $\mathcal{S}^1_{tgt}$, …, α_n,$\mathcal{S}^n_{tgt}$], where $\mathcal{S}^0_{src}$ is the source state of the path, and α_x, $\mathcal{S}^x_{tgt}$ are the clickable and target state of $\mathcal{A}_x$, respectively.

Regardless of the mode of creation, an automated UI test case ($\mathcal{T}$, definition 4) essentially is a programmatic representation of what we call a UI path ($\mathcal{P}$, definition 3). A UI Path ($\mathcal{P}$) is a series of state transitions ($\mathcal{A}$, definition 2) that are observed in the browser as a result of user interactions. We refer to interactive elements such as buttons available in each application state ($\mathcal{S}$, definition 1) as clickables (α).

Consider the example UI test case $\mathcal{T}_{ex}$ shown in Listing 1. The UI path ($\mathcal{P}_{ex}$) shown in Figure 2 represents the transitions for the path. The application states in $\mathcal{P}_{ex}$ are shown in Figure 1. We use text labels in the path diagram for easier comprehension.

When a test case is being created, human testers typically extract the locator required to identify the target clickable using the DOM hierarchy from browser developer tools. In Figure 3, we show an example XPath for the clickable *'Enter the Store'* extracted from the DOM of the application state $\mathcal{S}_1$. Our example test case contains 4 state transitions in total, each after a user interaction (typically a click) that causes a state change in the browser.

As the example in Listing 1 demonstrates, it is challenging for a human to comprehend the semantics of a given automated UI test case when there are no labels. Given the susceptibility of UI test cases for breakages [20], the human effort required for test maintenance only increases when they are not comprehensible.

2.2 AI Frameworks

We provide the background on KeyBERT and Probabilistic Context-Free Grammar (PCFG). KeyBERT and PCFG are the building blocks for our supervised and unsupervised approaches, respectively.

2.2.1 KeyBERT. KeyBERT [18] is a keyword extracting technique based on the popular Bidirectional Encoder Representation of Transformers (BERT) technique. It has three steps for extracting keywords from a set of documents. In the first step, it creates candidate keywords or phrases in terms of n-grams from a document. N-grams represent the sequence of word representation in a document. In the second step, it creates a BERT-based embedded representation for documents and the candidate keywords or phrases. In the third step, it finds the candidates that are most similar to documents using the cosine similarity.

For a clickable,

```
<a href="/jpetstore/actions/Account.action?
newAccountForm=">Register Now!</a>
```

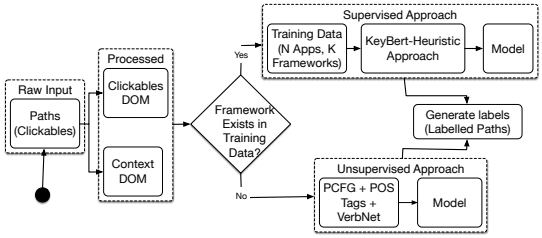

Figure 4: Overview of our approach for label generation

with html tags as stop words, KeyBERT outputs a list of tuples. In each tuple, the first element is the keyword/keyphrase, and the second element is the probability: [('account newaccountform', 0.5779), ('actions', 0.2787), ('jpetstore', 0.4154), ('now', 0.1297), ('register now', 0.4638)].

2.2.2 Probabilistic Context-Free Grammar (PCFG). PCFG extends context-free grammars by assigning a probability to each production rule. A PCFG is represented by a tuple $\langle N, \Sigma, R, S, \theta \rangle$. Here N is a finite set of non-terminal symbols, Σ is a finite set of terminal symbols, R is a finite set of production rules of the form $X \rightarrow Y_1, Y_2, ..., Y_n$ and $Y_i \in (N \cup \Sigma)$, $S \in N$ is a distinguished start symbol. $\theta(X \rightarrow Y_1)$ represents the probability that captures the conditional probability of choosing a production rule $X \rightarrow Y_1$.

For a label "register user" we obtain the following grammar using the Standford parser: S $\longrightarrow$ VP, VP $\longrightarrow$ (VB, NP), VB $\longrightarrow$ register, NP $\longrightarrow$ NN, and NN $\longrightarrow$ user. Here, S represents the start symbol, VP, NP, VB, and NN represent non-terminals and register and user represent terminals. Using PCFG, we obtain the following parameters and their probabilities bounded by the constraint: q(S $\longrightarrow$ VP) = 1, q(VP $\longrightarrow$ VB) + q(VP $\longrightarrow$ NP)=1, q(NP $\longrightarrow$ NN) =1, q(VB $\longrightarrow$ register) = 1, q(NN $\longrightarrow$ user) = 1.

3 OUR TECHNIQUE

Automated UI tests, such as the one shown in Listing 1, consist of actions performed on UI elements present in web pages. Each click α_x results in a state transition ($\mathcal{A}_x$). To assist users in understanding automatically generated test cases, we want to generate labels for each clickable α_x, consisting of an action phrase and its target phrase that represent α_x and its target state $\mathcal{S}^x_{tgt}$, respectively.

For label generation, we consider two sources of information: the first is the DOM of a clickable and the second is its context. Formally, the context of a clickable is defined as follows:

Definition 5. Context (C) Given a clickable α and dynamic DOM of the page $\mathcal{D}$, context C is the largest sub-DOM that contains only one clickable which is α.

To obtain the context of a clickable α, we traverse upward in the DOM, starting at the node for α, until a subtree that contains another clickable (different from α) is reached. We obtain the DOM and the context from a UI path: for each clickable α_i, we get the DOM attributes and its context in the source state $\mathcal{S}^i_{src}$.

Figure 4 shows the overview of our approach. Once a user generates paths using Crawljax, we preprocess the clickables in the paths to extract the DOM and the context for each clickable. Then, we check if the AUT uses a framework(e.g., AngularJS) present in our training data. If it does, the user can run the supervised approach to generate labels from the clickables. If the framework is not present, the user can choose the unsupervised approach to generate the labels. In the following, we provide the details of the supervised and the unsupervised approaches.

3.1 Supervised approach

We propose a Supervised approach to extract meaningful semantic information from a DOM tree. First, we consider applications developed using different frameworks such as JQuery, AngularJS, React, etc. as our training data. From the training data, we identify attributes that tend to be the most representative source of a clickable α. For example, in the application petclinic, the most frequent attributes are *text*, *href*, *title*, and *aria-label*. In addition, we find frequent framework-specific (AngularJS) features such as *ng-reflect-router-link* and *routerlink*. We create an *attributeList* considering the frequent attributes. For a new application, serving as our test data, we extract the attribute values using the *attributeList* as shown in Algorithm 1.

In some cases, we observed that simply extracting the value of an attribute might generate less meaningful keywords that are not indicative of an action. For example, consider the clickable

```
<a href="/jpetstore/actions/Catalog.action?viewProduct=&amp;productId
=FI-SW-01">FI-SW-01</a>
```

Here, according to the *attributeList*, the first checked attribute is "text", however, the text "FI-SW-01" is not meaningful. In such cases, we consider the value under `href` as shown below, as it seems to be more meaningful than "FI-SW-01".

```
/jpetstore/actions/Catalog.action?viewProduct=&amp;productId=FI-SW-01
```

However, the value might have multiple subsequences of words that could reflect the true action representation for the clickable. To infer the true action we take the following steps. First, we preproceess the value of an attribute. It includes tokenizing the value to generate tokens, lemmatizing the tokens, removing stop word tokens, and finally removing non-English tokens. Second, we directly use the phrase as a keyword, or use KeyBERT to extract a meaningful keyword from the value if the value is complicated. For example, for the above value, we obtain the tokens "view product product id". Then, applying KeyBert, we further obtain relevant keywords and their probabilities as follows: ('product id', 0.8667), ('view', 0.4029), ('product', 0.6253), ('view product', 0.7894), ('id', 0.4704). We choose the one with the highest probability, i.e., "product id", as the label.

Algorithm 1: Supervised Approach

```
Function getLabel(clickable: DOM, context: DOM, attribute_list: List) : str
begin
    foreach attribute ∈ attribute_list do
        if attribute ∈ clickable.keySet then
            if attribute == "href" then
                if "?" exists then
                    label = KeyBert(href.split("?")[1]) return label
                else
                    label = href.split("/")[-1].split(".")[0] return label
                end
            else if is_English(clickable[attribute]) then
                label = preprocess(clickable[attribute]) return label
    end
    label = KeyBert(preprocess(element))
    if label is NaN then
        label = KeyBert(preprocess(context))
    return label
end
```

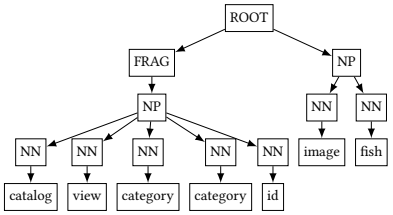

Figure 5: Parse tree representation for values of attributes associated with a clickable

3.2 Unsupervised Approach

An application that uses a novel framework might contain different domain knowledge from the frameworks used for training. As a result, the application might not contain attributes present in our maintained attribute list. To support such cases, we propose an unsupervised approach. The approach uses PCFG, which leverages parts-of-speech (POS) tag analysis to extract relevant keywords that might represent an action invoked when clicking the clickable element, and the target of the action. Algorithm 2 depicts our Unsupervised approach. It consists of the following steps.

Algorithm 2: Unsupervised Approach

```
Function getLabel(clickable: DOM, context: DOM) : str
begin
    attributes = preprocess(clickable)
    trees = parser(attributes)
    pcfgs = induce_pcfg(trees)
    v_list = extract_verbs(pcfgs)  #POS tag: VB, VBG, VBD, VBN, VBP, VBZ
    verb, _ = sorted(v_list)[-1]     #extract the verb with highest probability
    n_list = extract_nouns(pcfgs)              #POS tag: NN, NNS, NNP, NNPS
    noun, _ = sorted(n_list)[-1]   #extract the noun with highest probability
    if verb is NaN then
        verb = match_verb(clickable)              #use verbnet to match verb
    if verb is NaN then
        verb = match_verb(context)                #use context to match verb
    label = verb + noun
    return label
end
```

First, given a clickable as an input, for example, consider the clickable 4 in Figure 1

```
<a href="/jpetstore/actions/Catalog.action;jsessionid=
CBE38E7BFD1E93E99C2B87D154700A6F?viewCategory=&amp;categoryId=FISH">
    <img src="../images/fish_icon.gif"/>
</a>
```

The clickable can contain a start symbol such as <a> or <button> along with attributes such as `class` and `href` as the children of the start symbol. The attributes have values such as "`/jpetstore/actions ...`" for `href` and "`../images/fish _icon.gif`" for `src`.

We preprocess the value by first tokenizing it and then removing stop words and non-English words. Next, we generate the parse tree representation with POS tags using the Stanford CoreNLP parser, as shown in Figure 5. Then, we input all the parse tree representations into PCFGs, and obtain production rules with their probabilities using the NLTK library, as shown in Table 1.

In some cases, we observed that POS tag recommendations from the Stanford parser were incorrect for certain verbs. For example, in Figure 5, we find that the word "view" has been tagged as a noun (NN) whereas it should have been tagged as a verb (VB). To address this limitation, we use VerbNet to replace the incorrectly identified POS tags. VerbNet [41] is a verb lexicon that links verbs' syntactic and semantic patterns. VerbNet would replace the POS tag for "view" with a verb (VB).

Table 1: Production rules where a parent yields a child along with their probabilities

Parent	Child	Probability
FRAG	NP	0.6
NP	NN NN NN NN NN NN NN	0.0909091
NN	'category'	0.0927152
NN	'id'	0.13245
NN	'fish'	0.013245
NP	NN	0.254545
NN	'image'	0.0529801

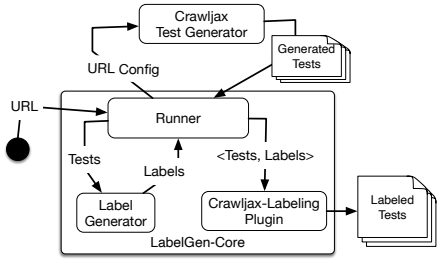

Figure 6: The workflow of CrawLabel

For extracting an action label, we use a similar approach as suggested by Kalia et al. [22]. Given a clickable, we first extract the verb with the highest probability and then extract a noun with the highest probability, thereby creating an action label as a pair of the verb and the noun.

If VerbNet cannot detect the verb, we use the context by mapping HTML tags to verbs. <nav>, <li> and <menu> are mapped to "navigate to"; <option> and <select> are mapped to "select". The default verb is set to "view" if none of the tags mentioned above exists.

4 IMPLEMENTATION

Our technique is implemented in a tool called CrawLabel that is publicly available [4]. Figure 6 presents an overview of the workflow implemented in CrawLabel, which takes as input the URL of the web application under test and generates labeled UI test cases. CrawLabel first invokes Crawljax [2], a state-of-the-art automatic UI test generator for applications. Then, it analyzes the generated tests to infer natural-language labels for test steps and outputs labeled UI tests.

For each clickable exercised in a UI test case, *Label Generator* retrieves the corresponding application state (captured in the DOM) and invokes the appropriate NLP engine (supervised or unsupervised) to generate a natural-language label. Finally, using the generated label, *Crawljax-Labeling Plugin* modifies the original test case to incorporate the generated label as a code comment associated with the corresponding test step (as illustrated in Listing 2).

The "Runner" and "Label Generator" components are implemented in Python and the "Crawljax-Labeling Plugin" is implemented in Java using the plugin framework provided by Crawljax. A more detailed flowchart of label generation is provided in Figure 4.

For the Supervised approach, we use the open-source tool KeyBERT [3] to compute the label in cases where the attribute is complicated or the clickable does not contain any of the attributes included in the rules. For the Unsupervised approach, we use the Stanford CoreNLP toolkit [29] to perform POS tagging and the

Table 2: Crawling results of Crawljax

Application	# Paths	# States	# Edges	# URLs	Exit Status
Dimeshift	1	12	14	3	Exhausted
Pagekit	42	105	113	23	Maximum Time
Petclinic	61	36	84	14	Exhausted
Retroboard	10	12	18	6	Exhausted
Splittypie	47	28	77	10	Exhausted
Addressbook	41	35	74	19	Exhausted
Claroline	86	91	200	63	Maximum Time
Collabtive	39	16	25	14	Exhausted
Jpetstore	20	19	39	16	Exhausted
Mantisbt	119	104	199	18	Maximum Time
Mrbs	3	36	70	36	Maximum Time
Phoenix	1	205	404	5	Maximum Time
Ppma	43	23	46	14	Exhausted

NLTK library [27] to construct PCFGs from the parse trees. If a verb is not recognized by the Stanford parser, we use the VerbNet module provided in the NLTK library [28] to search for verbs.

Although our current implementation generates labels for Crawljax generated tests, with an appropriate parser and adapter modules for the source code of test cases, it is possible to extend it for other kinds of Web UI tests. For example, in order to adapt CrawLabel for manually written UI tests, instrumentation or event-listener frameworks could be used to track the browser events and gather required information for label generation. Test suites intended to be maintainable (by using design patterns such as page objects) require substantial manual effort to create and adding labels to such test suites could further enhance their understandability and maintainability.

5 EMPIRICAL EVALUATION

Our evaluation focuses on investigating the research question of the effectiveness of the Supervised and Unsupervised approaches, along with two baseline techniques, in computing labels for UI test cases. We measure effectiveness using multiple metrics and compute them with respect to a manually constructed ground truth of test-case labels. After describing the benchmark open-source web apps (§5.1), the evaluation metrics (§5.2), and the baseline techniques (§5.3), we present the results of the evaluation (§5.4). We then discuss key findings and their implications (§5.5) and conclude this section with a discussion of threats to the validity of our results (§5.6).

5.1 Benchmark Applications

We use 13 open-source web applications (Table 2) that have been used in prior work on UI testing [5, 7]. For each application, we provide Crawljax with appropriate configuration (e.g., form-fill data) to maximize its state exploration and a time limit of 60 minutes, similar to prior studies [48].

In Table 2, # Paths represents the number of paths crawled by Crawljax during the exploration, which equals the number of generated test cases. # States represents the number of abstract UI states. # Edges represents the number of clickables that cause state transitions. # URLs represents the number of unique URLs. Column 6 shows whether Crawljax completed its run within the one-hour limit ("Exhausted") or terminated after reaching the time limit ("Maximum time passed").

We created a manual label to summarize each clickable and the state transition it causes. Two of the authors created the labels

Table 3: Frameworks used by the training-set apps

Application	Framework
Dimeshift	JQuery
Pagekit	UIKit
Petclinic	Angularjs
Retroboard	React
Splittypie	Emberjs

Table 4: Frameworks used by, and labeled clickables for, the evaluation-set apps

Application	Framework	# labels
Addressbook	Javascript	46
Claroline	Javascript (claroline.js)	190
Collabtive	Javascript	21
Jpetstore	Javascript	39
Mantisbt	Javascript	148
Mrbs	Javascript	67
Phoenix	React	404
Ppma	JQuery	42

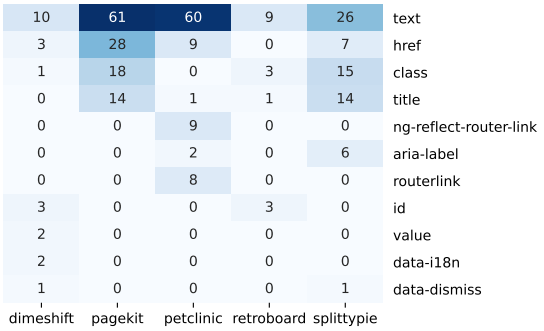

Figure 7: Frequency distribution of attributes for test cases of the training-set apps

independently, and conflicts were resolved by discussion. It took approximately 30 minutes to three hours to label each application. During this process, we classified some clickables as "cannot be labeled" if a human cannot perform the associated actions. For example, a clickable can appear behind a UI modal dialog, in which case a human cannot click it, but because the clickable occurs in a DOM tree, Crawljax can fetch it. There are 18 elements that cannot be manually labeled in 8 applications which accounts for 1.9% of all elements. Edges that have the same XPath and target state are viewed as duplicate clickables and only one is kept (e.g., a clickable in the navigation bar an application that occurs in all states of the application).

The supervised approach requires training data. We use five of the 13 applications as the training data because they have different frameworks (shown in Table 3), and one application is randomly selected when multiple applications have the same framework. The remaining eight applications serve as the evaluation-set apps; Table 4 lists these applications along with the frameworks used and the number of labeled clickables.

Using the training data, we count the number of times a label has an overlap with the attribute of the clickable and rank these attributes based on frequency. Figure 7 visualizes the results. As shown in the figure, `text` is the most relevant attribute for labeling, followed by `href`, `class`, and `title`.

5.2 Evaluation Metrics

We use the following metrics for the evaluation.

5.2.1 Precision, Recall, and F1. Precision, recall, and F1 measures are widely used in evaluation of summarization techniques. For each clickable element, the manually assigned labels are considered as the reference text or the ground truth. These measures are computed by comparing the model output with the reference text:

$$Precision = \frac{|W_{ref} \cap W_{cand}|}{|W_{cand}|} \times 100$$

$$Recall = \frac{|W_{ref} \cap W_{cand}|}{|W_{ref}|} \times 100$$

$$F1 = \frac{2PR}{P+R} \times 100$$

where W_{cand} and W_{ref} denote the words contained in the model output and the reference text, respectively.

5.2.2 Rouge. Rouge [30] is a set of metrics (ROUGE recall, precision, or F1 score) widely used in measuring accuracy of summarization. Rouge-N measures the number of matching "n-gram" between the model output and the reference text. Rouge-L measures the longest common subsequence (LCS) between the model output and the reference text. For Rouge-2, if the manual label contains only one word (e.g., "login"), we use the value of Rouge-1 as Rouge-2 (as a smoothing method) [9].

5.2.3 Edit Similarity. Levenshtein distance [50] is a string metric for measuring the smallest number of edit operations required to transform one string to another. The character-level edit similarity is calculated as (1 − normalized Levenshtein distance) × 100. The higher the edit similarity, the more similar the model output is to the reference text.

5.3 Baseline Techniques

To the best of our knowledge, there is no existing tool or technique for labeling, summarizing, or generating comments for UI test steps. Therefore, we created two simple internal baseline techniques to evaluate our approach against: the Preprocess approach and the KeyBERT approach.

The Preprocess approach first converts the attributes of a clickable into a bag of words, and then performs tokenization and lemmatization, followed by removal of stop words and non-English words. This simple baseline is intended to maximize the recall by retaining as many words as possible from the attributes of a clickable.

The KeyBERT approach leverages a keyphrase extraction technique with a pretrained model (trained on natural-language documents). It takes a clickable as input and generates for it a label of length one or two. It uses HTML tags as stop words and Maximal Margin Relevance [10], a diversity ranking technique, to create keywords/keyphrases based on cosine similarity. This baseline, thus, enables the investigation of how a keyphrase extraction tool that is trained on natural-language documents (e.g., news) performs on the information associated with DOM clickables. An alternative to KeyBERT possibly could be using programming-specific [8] or domain-specific dictionaries. However, in our current approach, we view each application as a domain-independent application; i.e., labels are computed as general and well-known English terms. Thus, we hypothesize that a pre-trained model trained on news documents could be sufficient for our requirement.

5.4 Results and Analysis

For each generated label, we calculate Rouge-1, Rouge-2, Rouge-L, and edit similarity. To reduce the inaccuracy introduced by manual labels, we remove English stop words, and lemmatize labels with the python NLTK library. For example, "adding an event" and "add event" are considered to be equivalent.

Table 5 presents the average scores by the techniques over all labels. We use the notation r@x, p@x, and f@x to represent the recall, precision, and F1 score of Rouge-x, where $x = 1, 2$, or L, representing 1-gram, 2-gram, and LCS, respectively.

Technique effectiveness. For each column in Table 5, the highest score is highlighted in boldface. We find that the Supervised approach performs better than the baselines and the Unsupervised approach in terms of precision, F1 score, and edit similarity. The Preprocess baseline outperforms other approaches in terms of recall; this is expected because it retains all the words in a clickable except the stop words. Our approaches are *extractive*, which means that they cannot generate words that do not appear in a clickable's attributes. Preprocess, thus, represents the best recall that can be achieved by an approach that generates labels by analyzing the clickable only. Overall, on average, Supervised achieves the best scores on seven of the 10 metrics, whereas Unsupervised ranks second on six of the metrics.

Figure 8 presents as boxplots how the techniques perform on each application under the metrics f@1, f@2, f@L, and edit similarity. For the metric f@1, in terms of the median value, the Unsupervised approach performs better than or the same as the baseline techniques for six applications. The Supervised approach performs better than or the same as other approaches for six applications. For Mrbs, Supervised approach does not perform better than Unsupervised approach because the attribute of clickable contains insufficient information; e.g., consider the clickable

```
<a href="week.php?year=2022&amp;month=05&amp;day=14&amp;area=1&amp;
room=1">
May 08
</a>.
```

The label here represents a date; however, the entire clickable represents the action of viewing the week of of the relevant date. For Mantisbt, Supervised does not perform better than KeyBERT again due to incomplete information in the clickable.

For metric f@2, Supervised and Unsupervised perform better than or the same as the baselines for six applications and three applications, respectively. For the Address application, Supervised and Unsupervised do not achieve good scores because the ground-truth label often contains more than three words.

For metric f@L, Supervised and Unsupervised perform better than or the same as the baselines for seven applications and five applications, respectively.

In terms of edit similarity, Supervised and Unsupervised perform better than or the same as the baselines for six applications and three applications, respectively. For Ppma, Supervised approach has a median score lower than KeyBERT because Supervised approach either generates labels that exactly match the reference text or do not match, while KeyBERT generates more partially matched labels.

The Supervised approach has the benefit that the generated label maintains the original order of a phrase. However, it is not as flexible as the Unsupervised approach in combining verbs and

Table 5: Effectiveness scores achieved by the techniques on average. We use the notation r@x, p@x, and f@x to represent the recall, precision, and F1 score of Rouge-x, where $x = 1, 2,$ or L, representing 1-gram, 2-gram, and LCS, respectively

Method	p@1	r@1	f@1	p@2	r@2	f@2	p@L	r@L	f@L	edit-sim
Preprocess	38.31	**75.83**	46.18	16.15	**60.70**	24.15	38.10	**75.27**	45.89	44.35
KeyBERT	72.78	50.22	56.54	26.80	21.69	23.08	66.67	45.65	51.36	60.02
Supervised	**83.38**	60.64	**66.40**	**49.16**	44.85	**46.03**	**83.01**	60.27	**66.03**	**73.27**
Unsupervised	72.37	58.12	59.77	37.29	35.56	35.19	70.07	55.24	57.25	66.97

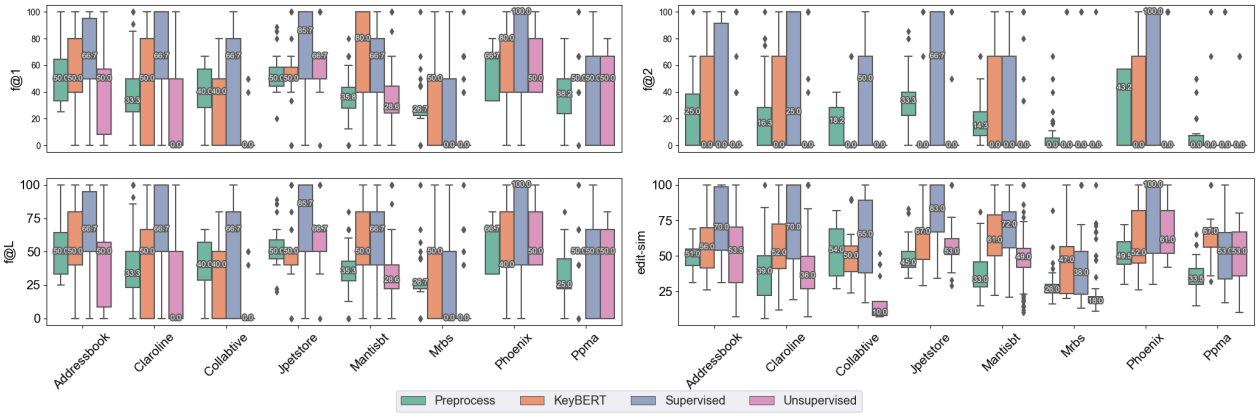

Figure 8: F1 scores of Rouge-1(f@1), Rouge-2(f@2), and Rouge-L(f@L), and edit-similarity scores for each application

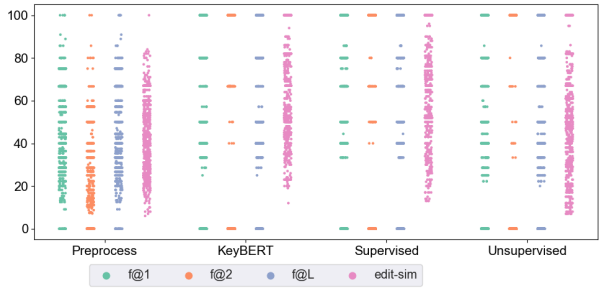

Figure 9: F1 score of Rouge-1(f@1), F1 score of Rouge-2(f@2), F1 score of Rouge-L(f@L), and edit-similarity scores

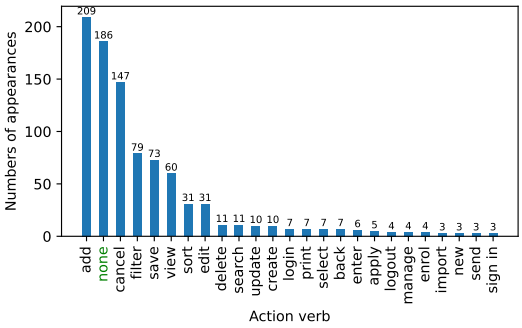

Figure 10: The distribution of action verbs

nouns, especially when the input is less meaningful and the complete phrase is split amongst multiple attributes. The Unsupervised approach is limited by POS tagging, which is designed for natural language instead of DOM: e.g., in some cases, POS tagging on DOM attributes fails to identify verbs correctly. The performance of Supervised and Unsupervised is also related to the number of words in the ground-truth labels: on longer labels (e.g., containing four or five words), both techniques become less effective.

Figure 9 presents a stripplot illustrating the distribution of f@1, f@2, f@L, and edit similarity for each approach. The overall distribution of the Supervised approach is the best, as it has more data points with high scores. The scores of Preprocess and Unsupervised have a wider distribution, whereas the scores of KeyBERT and Supervised are more centralized.

Inaccuracies related to action verbs. To better understand the effectiveness of the Unsupervised approach in extracting verb phrases, we manually labeled the action verbs from the clickable elements. For the action verbs that appeared more than twice, their frequency of occurrence is depicted in Figure 10. The value "none" represents the instances where a meaningful verb phrase could not be extracted solely from a clickable element; i.e., without incorporating additional information from the target state.

The percentages of correct action verbs extracted by Preprocess, KeyBERT, Supervised, and Unsupervised are 98.05%, 77.6%, 72.9%, and 73.02%, respectively. We analyzed the verbs that the Unsupervised approach failed to identify. Both VerbNet and PCFG cannot identify "sort" and "edit" as verbs, whereas "enabled" is identified as verb by PCFG. Therefore, we expect that improving the parts-of-speech recognition would increase the accuracy of the Unsupervised approach.

Statistical analysis. To understand whether the differences between techniques are statistically significant, we performed paired t-tests for each pair of techniques to determine whether there are differences between their mean values. We consider parametric tests over non-parametric tests for the following reasons: they perform well with skewed and non-normal distributions, they perform well even if the spread of each group is different, and they have more statistical power than non-parametric tests.

The null hypothesis for our test is that two approaches are not significantly different (true mean difference is zero). We ran the tests on the metric scores mentioned in Table 5, with the following results:

approach1	approach2	t-value	p-value
Supervised	Preprocess	49.58	0.00
Supervised	KeyBERT	36.20	0.00
Unsupervised	Preprocess	28.30	0.00
Unsupervised	KeyBERT	14.80	0.00

With confidence level $\alpha = 0.05$, each pair has p-value less than α, so we can reject the null hypothesis, indicating that both the Supervised and Unsupervised approaches are significantly better than the baseline techniques.

5.5 Discussion

We next discuss a few key findings from our evaluation and implications for further research on development of techniques for generating natural-language descriptions of UI test steps.

Importance of clickable elements. The success of the Supervised approach reveals that, frequently, the clickable element of a UI test step contains attributes that carry meaningful information about its function. Some attributes, such as `text` and `href`, tend to contain more relevant information than other attributes. This is evident by the high average precision, recall, and F1 score that the Supervised approach achieves: 83.01, 60.27, and 66.03, respectively.

Differencing the source and target states. Intuitively, analyzing differences between the source and target states of a clickable and focusing on the state transformation triggered by the clickable could provide useful information about semantics of the action performed that is relevant for label computation. We investigated this aspect via tree differencing [14] between the source and target states and incorporating the tree differences into label computation. Unfortunately, we found that this caused a significant reduction in the precision. We attribute this phenomena to too much noise being added beyond the semantic differences that appear in the result. For example, consider the following clickable element:

```
<a href="edit.php">add new</a>
```

It has the context

```
<li class="all"> <a href="edit.php">add new</a> </li>
```

We extract the following visible text from diffing the source and target states: "`h1 [ + ] | | | | | | Preferences input label textarea Edit / add address book entry Address: name rows quickadd submit Next quickadd Next address 20`". The meaningful label is "`add address book entry Address`". It exists in the diff, but along with a lot of other text that adds noise for meaningful label extraction.

We did find cases where the tree diff contains labeling-relevant information that is available in neither the clickable nor its context in the source state. Thus, further investigation of the usefulness of state differencing for labeling would be worthwhile.

The Unsupervised approach is competitive. The results on the Unsupervised approach are quite promising, considering that (1) it does not require training data and prior familiarity with the AUT, and (2) its performance according to our evaluation is only slightly less than that of the Supervised approach. Beyond straightforward label generation, it can be useful also for clarifying domain-specific words. By associating specific verbs and nouns, it can assist with human comprehension of the semantics of an action and its result. Further exploration of improvements in the Unsupervised approach could, thus, be a fruitful research direction.

Usefulness of analyzing the context of clickables. We further experimented with disabling context analysis for the Unsupervised approach. This causes average p@1, r@1, f@1, p@2, r@2, f@2, p@L, r@L, f@L, and edit-sim values over the eight applications to reduce by 18.14%, 36.99%, 31.02%, 73.02%, 70.16%, 71.10%, 16.08%, 34.54%, 28.72%, and 21.44%, respectively. This reduction indicates that context analysis is important and contributes to the effectiveness of the Unsupervised approach; especially when both POS tagging and VerbNet cannot find a verb by analyzing only the clickable.

For the Supervised approach, however, disabling context analysis has a very small effect: average p@1, r@1, f@1, p@2, r@2, f@2, p@L, r@L, f@L, and edit-sim values over the eight applications to reduce by 0.08%, -0.29%, 0.03%, 0.04%, -0.20%, 0.09%, -0.26%, 0.05% and 0.26%, respectively.

5.6 Threats to Validity

Like any empirical study, there are threats to the validity of our results; we discuss here the most significant among those.

In terms of external validity, because our evaluation is based on a corpus of 13 web applications, our results may not generalize to other applications. However, we note that these web applications have been used in prior empirical evaluations of web application testing techniques, and also that there is considerable variation in applications in terms of the frameworks used (Tables 3 and 4). Our results are based on test cases generated using Crawljax (Table 2) and, thus, may vary with other test cases (generated manually or automatically) that exercise parts of the applications state space not covered by the tests used in the study. Our evaluation was done with specific tool configurations (e.g., KeyBERT parameters) and our results may vary with other configuration settings.

As for threats to internal validity, there may be bugs in our implementation or errors in the manual ground-truth construction, which may affect our results. We mitigated the threat of implementation bugs by testing CrawLabel thoroughly and checking a subset of the computed results by hand. To reduce the chances of errors in manual label computation, one author collected each clickable with its DOM, context, screenshot of source state and screenshot of target state, then two authors each labeled the clickables. The two authors resolved labeling conflicts through discussion.

Finally, with respect to conclusion validity, in measuring the accuracy of computed labels against ground-truth labels, we did not consider synonym matches. This may lower our evaluation scores when the model output does not match the reference text, but still makes sense to a human by accurately capturing the action performed in a test step.

6 RELATED WORK

Test-Case Comprehension. Several works investigate the practical challenges associated with using automatically generated test cases. The user studies they conduct also discuss the question of understandability and its impact on developers and on the testing process. A study by Fraser et al. with 97 subjects (participating

either in a user study or in its replication) reveals that automatically generated unit test cases have a negative effect on the ability to capture intended class behavior, which can be attributed to the struggle to understand the generated tests [16]. This observation however is less conclusive in the replication study than in the original one. A subsequent study with 31 students provides no evidence that software quality changes with automated unit test generation, and concludes that automatically generated unit tests must be easy to read and understand for effective use by developers [40]. A user study by Panichella et al. with 30 subjects shows that with the addition of test case summaries, improved test case understanding helps developers find twice as many bugs in automatically generated unit test cases than without the summaries [38].

Test-Case and Code Summarization. Summarization techniques have been applied to unit-level test cases for the purpose of test case documentation. In [26], a technique named UnitTestScribe is proposed to automatically generate natural language documentation of unit test cases, using static analysis, natural language processing, backward slicing, and code summarization. In [38], dynamic information is leveraged by applying code summarization at the single test case level to the lines of code it covers.

In [13], a technique for creating meaningful unit test case names is suggested, by summarizing the datatypes targeted by the test case for the inputs and output of the method under test.

More generally, there exists a large body of work on automated code summarization [36, 51] that employs various techniques, such as static and dynamic program analysis, natural-language processing, and information retrieval, to generate descriptive comments for source code, with an end goal of assisting with program comprehension and software maintenance.

To the best of our knowledge, summarization techniques have not been applied to-date to end-to-end tests that exercise the application under test via its UI.

Webpage Summarization. There are two types of approaches for summarizing texts in webpages: extraction-based summarization [35] and abstraction-based summarization [34]. For extractive summarization, key-sentences or key-phrases are extracted from the original content. For abstractive summarization, the summary is a semantic representation of the original content. This paraphrasing involves both natural-language processing and a deep understanding of the domain of the original content. These approaches usually summarize whole pages or segments, whereas we deal with clickables, focusing on smaller regions.

Mukherjee et al. [37] propose structural and semantic analysis to partition HTML documents, and annotate them with semantic labels using an ontology encoding domain knowledge. In contrast, our approaches do not require domain specific knowledge in advance. Shah et al. [43] propose D-rank, an unsupervised keyword extraction technique for a single web page, based solely on its text and structural information. D-rank extracts words from common web page locations such as headings and hyperlinks, and ranks them according to their position and frequencies. Xiong et al. [47] propose a neural keyphrase extraction technique for web page with both text and visual features.

Reference [39] surveys a large body of work on keyphrase extraction from documents (not limited to DOM-type ones) in order to concisely summarize their content. The survey separately discusses unsupervised and supervised techniques, and classifies them based on the approach type (e.g., statistics-based, graph-based ranking, topic-based, language model-based, classification-based, and deep learning). The survey also discusses types of features and several technique extensions, such as ensemble models via stacking, incorporating information from similar documents, and incorporating semantics via knowledge graphs.

Natural-Language Locators. Prior research on creating change-resilient test scripts for UI test cases has proposed the usage of natural-language locators. Yandrapally et al. [49] generate textual clues from the DOM hierarchy to disambiguate target UI elements instead of conventional locators that use XPaths. The textual clues extracted do not necessarily relate to the semantics of the target element but rather act as visual locators through which the target element can be identified unambiguously. In contrast, our technique extracts semantically meaningful labels for test steps and assisting with test comprehension. Kirinuki et al. [24] propose a domain-specific language for testers to specify actions on the web pages and automatically identify web elements based on the specified string. They vectorize the specified target string and the web-element attributes; and compare the vectors to compute similarity in order to identify the intended target element. Our technique instead analyzes only the DOM tree to automatically generate semantically meaningful labels for web elements and does not rely on any human input.

7 SUMMARY AND FUTURE WORK

In this work, we presented an approach for generating natural-language labels for UI test cases and its implementation in a tool called CrawLabel. CrawLabel applies NLP to selected DOM attributes and uses two techniques for selecting the DOM attributes: a supervised ranking-based technique and an unsupervised technique based on probabilistic context-free grammar learning. Both techniques achieved relatively high precision, recall, and F1 score. The unsupervised technique proved competitive though the supervised one performed slightly better. Our results also show that, in most cases, analyzing only the clickable element for a test step is sufficient to generate a meaningful label.

We implemented the approach in the context of automatically generated UI test cases for web applications (leveraging the Crawljax tool), but it is applicable more generally to developer-written test cases as well as to UI test cases for other types of applications (e.g., mobile apps).

In the future, we plan to perform a more comprehensive technique evaluation by conducting a user study that will investigate the impact of our generated test-step labels on developer tasks, such as test-case comprehension and failure diagnosis, and what skills and level of domain expertise are assumed/required to benefit from this technique. There are also several potential directions for improving and extending our labeling technique, such as better analysis of source contexts of clickable elements and resulting target states, computation of more descriptive labels for test steps, and generation of summaries for entire test methods, by combining the labels of individual test steps.

REFERENCES

[1] Silviu Andrica and George Candea. 2011. WaRR: A tool for high-fidelity web application record and replay. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 403–410. https://doi.org/10.1109/DSN.2011.5958253

[2] artifact 2022. *Crawljax*. https://github.com/crawljax/crawljax

[3] artifact 2022. *KeyBERT*. https://github.com/MaartenGr/KeyBERT

[4] artifact 2022. *Our experiment infrastructure, data, and results*. https://github.com/sweetStreet/AST-2022-submission/

[5] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. 2019. Web test dependency detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 154–164.

[6] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-Based Web Test Generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 142–153. https://doi.org/10.1145/3338906.3338970

[7] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-based web test generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 142–153.

[8] Dave Binkley, Dawn Lawrie, Lori Pollock, Emily Hill, and K Vijay-Shanker. 2013. A dataset for evaluating identifier splitters. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 401–404.

[9] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz Josef Och, and Jeffrey Dean. 2007. Large Language Models in Machine Translation. In *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*, Jason Eisner (Ed.). ACL, 858–867. https://aclanthology.org/D07-1090/

[10] Jaime Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 335–336.

[11] Boris Chidlovskii and Jérôme Fuselier. 2005. A Probabilistic Learning Method for XML Annotation of Documents.. In *IJCAI*. Citeseer, 1016–1021.

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

[13] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. Generating Unit Tests with Descriptive Names or: Would You Name Your Children Thing1 and Thing2?. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 57–67. https://doi.org/10.1145/3092703.3092727

[14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[15] Robot Framework. 2020. *Robot Framework Introduction*. Technical Report. Retrieved 18-9-2020, 2020, from https://robotframework. org/# introduction.

[16] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Trans. Softw. Eng. Methodol.*, Article 23 (Sept. 2015). https://doi.org/10.1145/2699688

[17] Mark Grechanik, Qing Xie, and Chen Fu. 2009. Maintaining and evolving GUI-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*. 408–418. https://doi.org/10.1109/ICSE.2009.5070540

[18] Maarten Grootendorst. 2020. KeyBERT: Minimal keyword extraction with BERT. https://doi.org/10.5281/zenodo.4461265

[19] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. EXSYST: Search-based GUI testing. In *2012 34th International Conference on Software Engineering (ICSE)*. 1423–1426. https://doi.org/10.1109/ICSE.2012.6227232

[20] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. 2016. Why do record/replay tests of web applications break?. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 180–190.

[21] Mark Johnson. 2010. PCFGs, Topic Models, Adaptor Grammars and Learning Topical Collocations and the Structure of Proper Names. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. 1148–1157.

[22] Anup K Kalia, Raghav Batta, Jin Xiao, Maja Vukovic, et al. [n. d.]. Ensemble of Unsupervised Parametric and Non-Parametric Techniques to Discover Change Actions. ([n. d.]).

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

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

[25] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2016. Approaches and Tools for Automated End-to-End Web Testing. Advances in Computers, Vol. 101. 193–237. https://doi.org/10.1016/bs.adcom.2015.11.007

[26] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Nicholas A. Kraft. 2016. Automatically Documenting Unit Test Cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 341–352. https://doi.org/10.1109/ICST.2016.30

[27] library 2022. *NLTK grammar library*. https://www.nltk.org/api/nltk.grammar.html

[28] library 2022. *NLTK VerbNet library*. https://www.nltk.org/_modules/nltk/corpus/reader/verbnet.html

[29] library 2022. *Standford CoreNLP Toolkit*. https://stanfordnlp.github.io/stanza/client_properties.html

[30] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

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

[32] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. 2008. State-Based Testing of Ajax Web Applications. In *2008 1st International Conference on Software Testing, Verification, and Validation*. 121–130. https://doi.org/10.1109/ICST.2008.22

[33] Ali Mesbah, Arie van Deursen, and Danny Roest. 2012. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Transactions on Software Engineering* 38, 1 (2012), 35–53. https://doi.org/10.1109/TSE.2011.28

[34] N Moratanch and S Chitrakala. 2016. A survey on abstractive text summarization. In *2016 International Conference on Circuit, power and computing technologies (ICCPCT)*. IEEE, 1–7.

[35] N Moratanch and S Chitrakala. 2017. A survey on extractive text summarization. In *2017 international conference on computer, communication and signal processing (ICCCSP)*. IEEE, 1–6.

[36] Laura Moreno and Andrian Marcus. 2018. Automatic Software Summarization: The State of the Art. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 530–531. https://doi.org/10.1145/3183440.3183464

[37] Saikat Mukherjee, Guizhen Yang, and IV Ramakrishnan. 2003. Automatic annotation of content-rich html documents: Structural and semantic analysis. In *International Semantic Web Conference*. Springer, 533–549.

[38] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. 2016. The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation. In *Proceedings of the 38th International Conference on Software Engineering*. 547–558. https://doi.org/10.1145/2884781.2884847

[39] Eirini Papagiannopoulou and Grigorios Tsoumakas. 2020. A review of keyphrase extraction. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 10, 2 (2020), e1339.

[40] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2015. Automated unit test generation during software development: a controlled experiment and think-aloud observations. In *Proceedings of the 24th International Symposium on Software Testing and Analysis*. ACM, 338–349.

[41] Karin Kipper Schuler. 2005. *VerbNet: A broad-coverage, comprehensive verb lexicon*. University of Pennsylvania.

[42] selenium 2022. *Selenium WebDriver*. https://www.selenium.dev/documentation/webdriver/

[43] Himat Shah, Mohammad Rezaei, and Pasi Fränti. 2019. DOM-based keyword extraction from web pages. In *Proceedings of the International Conference on Artificial Intelligence, Information Processing and Cloud Computing*. 1–6.

[44] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2017. APOGEN: Automatic Page Object Generator for Web Testing. *Software Quality Journal* 25, 3 (sep 2017), 1007–1039. https://doi.org/10.1007/s11219-016-9331-9

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

[46] Suresh Thummalapenta, K. Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. 2013. Guided Test Generation for Web Applications. In *Proceedings of the 2013 International Conference on Software Engineering*. 162–171.

[47] Lee Xiong, Chuan Hu, Chenyan Xiong, Daniel Campos, and Arnold Overwijk. 2019. Open Domain Web Keyphrase Extraction Beyond Language Modeling. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China,

5175–5184. https://doi.org/10.18653/v1/D19-1521

[48] Rahulkrishna Yandrapally and Ali Mesbah. 2021. Fragment-Based Test Generation For Web Apps. *CoRR* abs/2110.14043 (2021). arXiv:2110.14043 https://arxiv.org/abs/2110.14043

[49] Rahulkrishna Yandrapally, Suresh Thummalapenta, Saurabh Sinha, and Satish Chandra. 2014. Robust Test Automation Using Contextual Clues. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 304–314. https://doi.org/10.1145/2610384.2610390

[50] Li Yujian and Liu Bo. 2007. A normalized Levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence* 29, 6 (2007), 1091–1095.

[51] Yuxiang Zhu and Minxue Pan. 2019. Automatic Code Summarization: A Systematic Literature Review. *CoRR* abs/1909.04352 (2019). arXiv:1909.04352 http://arxiv.org/abs/1909.04352