Long YH, Chen YC, Chen XP *et al.* Test-driven feature extraction of web components. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(2): 389–404 Mar. 2022. DOI 10.1007/s11390-022-0673-4

Test-Driven Feature Extraction of Web Components

Yong-Hao Long[1,2] (龙永浩), Yan-Cheng Chen[1] (陈彦呈), Xiang-Ping Chen[3] (陈湘萍), *Member, CCF*
Xiao-Hong Shi[4] (石晓红), and Fan Zhou[1,*] (周　凡)

[1] *School of Computer Science and Engineering, National Engineering Research Center of Digital Life Sun Yat-sen University, Guangzhou 510006, China*

[2] *School of Design, The Hong Kong Polytechnic University, Hong Kong, China*

[3] *Guangdong Key Laboratory for Big Data Analysis and Simulation of Public Opinion, The School of Communication and Design, Sun Yat-sen University, Guangzhou 510006, China*

[4] *School of Information Technology and Engineering, Guangzhou College of Commerce, Guangzhou 511363, China*

E-mail: {longyh3, chenych28}@mail2.sysu.edu.cn; chenxp8@mail.sysu.edu.cn; shixh@gcc.edu.cn
isszf@mail.sysu.edu.cn

Received May 31, 2020; accepted February 18, 2022.

Abstract With the growing requirements of web applications, web components are developed to package the implementation of commonly-used features for reuse. In some cases, the developer may want to reuse some features which cannot be customized by the component's APIs. He/she has to extract the implementation by hand. It is labor-intensive and error-prone. Considering the widely-used test cases which can be useful to specify the software features, a test-driven approach is proposed to extract the implementation of the desired features in web components. The satisfaction of the user's requirements is transformed into the passing rate of user-specified test cases. In this way, the quality of the extraction result can be evaluated automatically. Meanwhile, a record/replay-based GUI test generation method is proposed to ensure that the extraction result has the correct GUI appearance. To extract the feature implementation, a hierarchical genetic algorithm is proposed to find the code snippet that can pass all the tests and has the approximate smallest size. We compare our method with two existing feature extraction methods. The result shows that our method can extract the correct implementation with the minimum size. A human-subject study is conducted to show the effectiveness and weaknesses of our method in helping users extract the features.

Keywords feature extraction, genetic algorithm, graphical user interface, software reuse

1 Introduction

The web application is one of the fastest-growing and most widespread application domains today, performing a crucial role in daily life. Numerous features are developed to meet the massive user requirements. It makes the development efficiency more and more important. Among the updated features, some of the similar functionalities have already existed in a variety of web applications, and facilitating their reuse offers significant benefits [1].

Web components are proposed to package the implementation of commonly-used features to facilitate web development. Fig.1 shows an example of using a mutiselection component. The user needs to provide some contents to be displayed (lines 1–5), initialize the component object (lines 7–10), and customize the component by using the given configurations (for instance, `multiple="multiple"` in line 1 and `closeOnSelect: false` in line 9). The component will handle the user's inputs and generate a multi-selection widget with predefined appearances and interactions.

A problem arises when the user wants to utilize a

Regular Paper

This work was supported by the Key-Area Research and Development Program of Guangdong Province under Grant No. 2020B010165001, the National Natural Science Foundation of China under Grant No. 61976061, and Guangdong Basic and Applied Basic Research Foundation under Grant No. 2020A1515010973.

*Corresponding Author

©Institute of Computing Technology, Chinese Academy of Sciences 2022

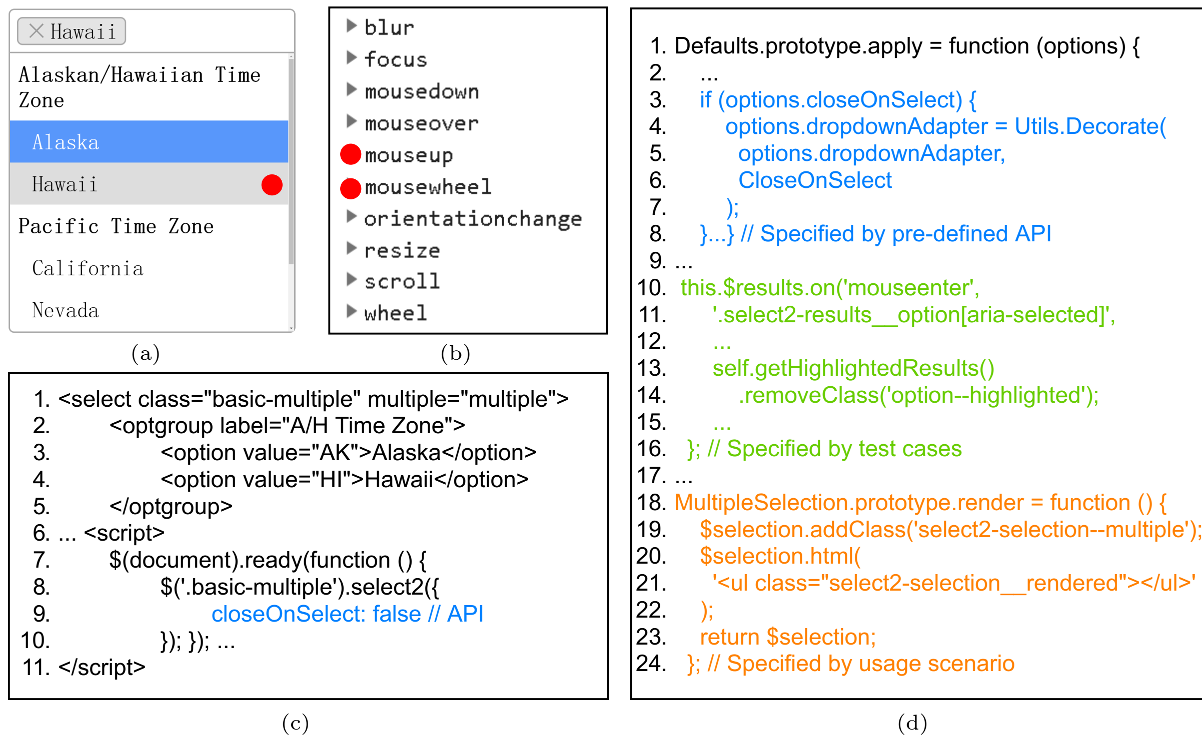


Fig.1. A select box component in (a) implemented by select2①. The desired features are mixed with some default events in (b) and cannot be customized by the component's APIs in (c) but can be specified by the component's test cases combined with the usage scenario as shown in (d).

set of features that cannot be customized by configurations in the component. For instance, the user wants to extract the selecting feature (to update the input area when clicking the "Hawaii" item as shown in the "Usage Scenario") in the component shown in Figs.1(a) and 1(b). These events are default events that are bound to the items when the component is initialized. No APIs are provided by the component to specify the events. In this case, the user has to manually forage and extract the desired features from thousands of lines of code [2]. Extracting the code that corresponds to the feature is complex since the feature may be related to the HTML elements, JS codes, CSS definitions, etc. The dynamic interplay between different entities makes the extracting process labor-intensive and error-prone [3]. The lack of flexible reusing supports makes developers turn to reinvent the features [4].

The feature extraction aims to find a code snippet that implements the required functionalities in a software system [5]. In the domain of web applications, a typical strategy is using the usage scenario to record the user's actions on the component (as shown in Fig.1). Some methods based on software analysis like code instrumentation [6,7] or software slicing [8,9] were used to find the relevant statements to the usage scenario as the extraction result.

Using the usage scenario to specify the features is intuitive and straightforward. However, it may suffer from quality problems. First, the improper scenario will lead to an unsatisfied extraction result. Second, some features are not appearance-relevant and easy to be overlooked in the scenario such as memory management and exception handlers. Considering the test cases, which are widely used in web implementation, especially in the open-source components, can be useful to specify the software features.

In this paper, we propose a novel method to automatically extract a code snippet and related resources in a web component that implements the user-specific functionalities. Considering test cases may not cover all the software features in some cases, a GUI test case generation algorithm based on the record/replay strategy is proposed to ensure the elements of interest in the extraction result have the same appearances as those in the original component. An extraction result is thought to contain the required features if it can pass the test cases. As fewer lines of code mean less cost users spending on learning the core concepts of components [10], a

①The select2 component. https://select2.org/, July 2021.

hierarchical genetic algorithm that considers the code structure is proposed to get the approximate minimum size of the extraction result. Several qualitative and quantitative experiments were conducted to show the effectiveness and correctness of our method. The contributions in this paper include the followings.

• An automated GUI test case generating method is proposed based on the record/replay strategy.

• A hierarchical genetic algorithm is proposed to extract an implementation that satisfies the requirements and achieves the approximate minimum lines of code.

• Experimental comparison of our method and existing feature extraction methods shows that our method can correctly extract all the components' features and generate the minimum lines of code.

2 Related Work

Our method aims to extract desired features from a web component. The related studies include feature locating, JavaScript analysis, and web application reuse.

2.1 Feature Extraction

The feature extraction aims to find a code snippet that implements the required functionalities in a software system [11]. Several studies [6, 9, 12] were conducted based on code instrumentation, program slicing, etc.

An intuitive way of feature extraction is recording the executed statements in the scenario as the implementation of the features [6]. Studies like Unravel [7] and Scry [12] show the executed statements in each action in the scenario to help users focus on the code mutating the DOM states. The feature implementation is a subset of the executed statements, and a part of the irrelevant statements is included in the extraction result.

Several feature extraction methods [8, 9] are based on program slicing. These methods record the visited DOM elements and the trigger events in the usage scenario as the slicing start point. Then they apply backward and forward slicing to find the statements in the dependency graph. The feature implementation is thought to be contained in the slicing result. The extraction result might also contain some irrelevant statements which have dependencies on the user scenario but make no contributions to the features, for example, the redundant assignments.

When the developer uses a scenario to specify the features, some features may hardly be presented in the scenario such as memory management or exception handlers. Barr *et al.* [13] used the test cases to specify the features. The code which could pass all the user-specified test cases was thought to contain the feature. Then, a genetic algorithm was developed to find the feature implementation. In our work, we also specify the features by user-specified tests, but we focus on extracting the features in the web components which are cross-language, highly dynamic, and weak-typed.

2.2 JavaScript Analysis

The JavaScript code performs a crucial role in web applications. It is dynamic, weakly-typed, and prototype-based, which makes the JS code hard to be understood and maintained [8]. Considerable studies [14–30] have been proposed to analyze the JS program in recent decades.

Some static analysis methods were proposed to analyze the JS program without actually running it. The static JS program analysis usually parses the source code by an abstract interpreter like TAJS [14], JSAI [15], and λ_{JS} [16]. Combined with the specified rules or mathematical constraints, the analyser is adopted in pointer analysis [17, 18], type inference [14, 19], data flow analysis [20, 21], call graph analysis [22, 23], etc.

The static analysis is effective, but it faces serious limitations in JavaScript because it cannot precisely reason about many dynamic language features. Dynamic analysis was proposed to avoid the challenge of statically approximating behavior [24]. Considerable studies based on dynamic analysis have been proposed in recent decades, in particular, studies like record/replay debugging [25, 26], feature locating [27, 28], and dynamic program slicing [29, 30] have close relations to our work.

Our work has some relations to JavaScript analysis. We propose an automated GUI test case generating method based on the record/replay techniques. In the code extraction, we parse the source code to find the code structure and ensure the extraction process does not break the code structures.

2.3 Web Application Reuse

Developing a satisfying web application requires the developers to have rich knowledge of user experiences, interface design, and programming on multi-languages. Instead of developing from scratch, developers often seek inspiration from examples to reduce the developing cost [31].

Some studies [32, 33] were proposed to provide predefined templates to simplify the development. Users

can choose one template and replace the generic information in the template with their contents. The Bricolage [34] was proposed to enlarge the scale of templates. According to a mapping between the elements in source and reference web pages, the contents in the source elements can be filled into the reference web page, and thus every existing web page can be reused as a template. These studies [32–34] center on providing a static web page for inspiration, while the interactions have to be manually implemented by the users.

In some cases, the users want to reuse part of the features in an example, instead of reusing the whole web page [35]. Several studies were proposed to handle this issue. For example, the web component library [36] allows developers to choose and reuse a component of interest from the library. Similarly, some studies [37–40] were proposed to extract and reuse components from existing web pages. The markup languages CTS [37] and Mavo [38] ask end-users to manipulate their HTML files to build a mapping between the original HTML elements and the example elements, in order to insert the example components' features into their applications.

The component encapsulates the commonly-used functionalities into a class or package, and users can easily reuse the feature by inserting the component into their software. However, the component may contain some features that the user does not want to use, and removing some of the useless features may require the user's knowledge on the implementation of the component. Hence, in this work, we propose a method that helps users reuse the features from the components in finer granularity.

3 Test-Driven Feature Extraction

In this section, we first formulate the feature extraction problem, in which we transfer the requirements of features to passing a set of test cases related to the features. Then we introduce an automated GUI test generating method to guarantee the desired elements in the extraction result have the same appearance as those in the original component. We finally introduce a hierarchical genetic algorithm to extract a code snippet that both passes the given test cases and contains statements as few as possible.

3.1 Problem Formulation

Feature. The feature refers to a specific functionality, defined by requirements, and accessible to developers and users [41]. The feature in this context does not include non-functional requirements such as performance or reusability②. It is implemented by a subset of code and resources of the whole application [9].

In this paper, we focus on extracting the JS code that implements the specified features on the client side. The web component is defined as (C, R) which has two parts: the JS code C and related resources R. The resources include HTML, CSS files, and external resources like background images or videos. Features on canvas are not included in the method.

A component (C, R) contains the specified features F if it satisfies:

$$P(T,(C,R)) = |T|,$$

where T represents the set of test cases for F, and $|T|$ is the number of test cases. $P(T,(C,R))$ means the number of test cases in T that (C, R) can pass. In this way, the user can specify the features by a set of test cases. It has two advantages: 1) the test cases are easy to read since most of them contain descriptions, and 2) we can automatically evaluate the degree of the result's satisfaction to the user's requirements by checking the passing rate of test cases.

In this paper, we focus on extracting the JS code which contributes to the features. The related resources are determined by code instrumentation: any resources that the extraction result requires are kept as the related resources. As a result, we simplify the component's test passing rate as $P(T, C)$.

Irrelevant Statements. For a code snippet C which contains the desired features, there may be some statements that make no contributions to the required feature. A statement s in C is regarded as irrelevant if deleting it does not affect the passing rate of test cases:

$$P(T, C - s) = P(T, C).$$

Feature Extraction. Feature extraction refers to extracting a code snippet and related resources in a web component that implements the user-specific functionalities. The extraction result could run independently and exclude irrelevant statements as many as possible.

Formally, we define the feature extraction as a searching problem: given the source code C of the web component and a set of user-specified test cases T, the feature extraction is finding a code snippet $C^\star$ in C, which can pass all of the given test cases and has the minimum LOC:

$$LOC(C^\star) = \min\{LOC(C') | C' \in C, P(T, C') = |T|\},$$

② Software, and S.E.S. Committee. IEEE Standard for Software and System Test Documentation. IEEE. 2008.

where $LOC(C')$ refers to the lines of code in C'.

We take Fig.2 to illustrate the process. Supposing the required features are specified by the four tests, the colored rectangles represent the feature implementation specified by the tests. The extraction result must pass all of the tests and contains the irrelevant statements as few as possible. To achieve this goal, we look through the original code and find the irrelevant statements hierarchically. At the outmost block, only a function definition exists and it is crucial to keep it to pass the tests. Then we look into the function definition block. The handled statements are the single statements in lines 2–6, 19 and 20, and a compound statement `for` block ranging from line 7 to line 18. Removing the statements in lines 3–5 will not affect the passing rate and they are thought to be irrelevant. The other statements are crucial to the test passing rate. More specifically, the statement in line 2 contributes to passing test #182, and removing other statements will make the result fail to pass tests #184 and #185. After that, we look into the `for` block and find the statements in lines 10 and 11 are also irrelevant. Therefore, the feature extraction result will exclude the statements in lines 3–5, 10 and 11.

3.2 Augmenting Test Cases

GUI testing is tedious and complex. Most visual features are tested manually. A few of the features are checked by test scripts[42]. Our observation on the components' test cases also supports the point of view: most of the test assertions are checking the status of a function of object in the JS code, instead of the DOM element. To fill this gap, a GUI testing method is proposed to ensure the desired elements in the component will be correctly displayed in the extraction result.

The record/replay[26] method is used to generate a GUI test case automatically. The elements manipulated in the usage scenario are considered as the elements of interest. Users can also point out other elements as the desired elements. If the pointed element is not a leaf, we will ask users to specify the desired leaves in the subtree whose root is this element to filter the uninterest elements.

For a usage scenario that consists of n actions on the original component, our method first records the initial appearances of the desired elements when no action happens. Then, the method records the appearances of the desired elements (named frame) in the component when each of the actions happens. Hence we get $n+1$ frames for the original component. After that, the method generates a test case that triggers the n actions and asserts the frames in the extraction result should be the same as the frames in the original component.

The consistency of the elements' appearances is checked by comparing the visual-related attributes in the source code rather than using the image-based methods in GUI testing[43] for two reasons. The first is our method will remove some elements in the original component, which may change the extraction result's appearances. The second is the image-based methods often consider the number of the same pixels between two snapshots. The color distortion of the snapshots may affect the result.

Four visual aspects that determine the component's appearance are considered in our method: 1) the positions of elements in the HTML file; 2) the required resources such as the background image; 3) the contents of the elements; 4) the inline, internal, and external styles of the elements.

The path from the root <HTML> element to

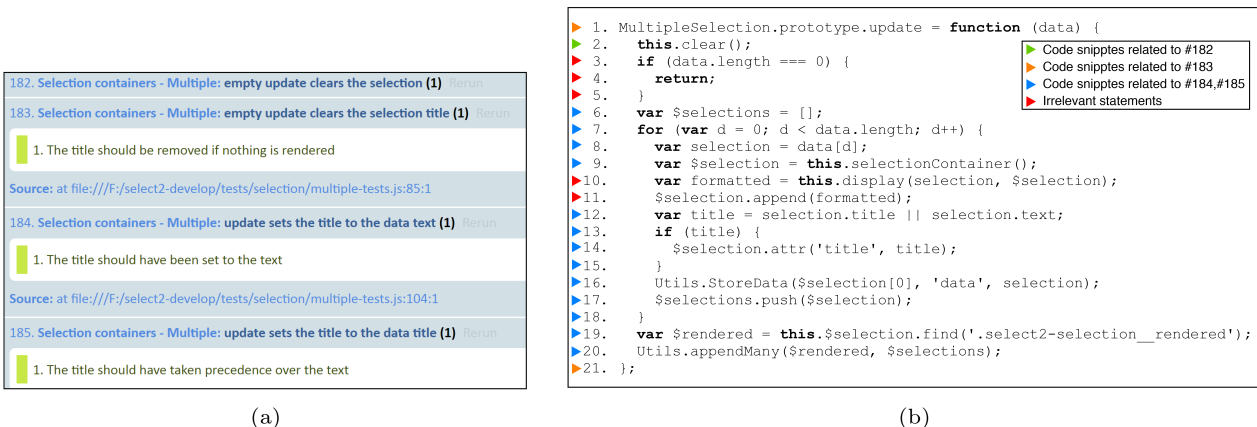


Fig.2. Combining different test cases in (a) can specify different feature implementations in (b).

the target element is recorded as the element's position. The position is used as the element's identifier. The element's resources and contents can be found in HTML source code. The styles are determined by the matching of selectors (including the pseudo-classes, e.g., `a:hover`) in CSS and HTML files. All of the visual properties are recorded as the JSON-format text.

The generated test case triggers the actions in sequence. Once an action is triggered, the desired elements' appearances in the extraction result are recorded. And an assertion is generated to check if the result's appearance text is the same as the text extracted from the original component.

3.3 Hierarchical Genetic Algorithm

Feature extraction is a process of removing irrelevant statements. It is a combination problem since the statements may cooperate to implement a feature. As a result, removing the statements one by one could not achieve the goal. For example, the two cases in line 3 and line 5 in Fig.3 should be removed together to pass the test case. But removing one of them could not pass the given test cases.

```
1 let a = 1
2 ...
3 a += 1 // in somewhere
4 ...
5 a -= 1 // in somewhere else
6 ...
7 assert(ok, a==1) // assertion
```

Fig.3. Statements in line 3 and line 5 are unwanted, and only removing one of them will fail to pass the tests.

Finding the best solution (i.e., satisfying the requirements with the minimum lines of codes) to this combination problem is an NPC problem. We propose a hierarchical genetic algorithm (HGA for short) to find an approximate result that satisfies the user's requirements and contains as few statements as possible. We assume that the required feature is made up of statements that are distributed in some block statements like the functions or objects. As a result, our method starts from the statements in the outmost code blocks, and recursively finds irrelevant statements layer by layer as shown in Algorithm 1.

The feature extraction method is based on the hierarchical genetic algorithm, which detects and removes the irrelevant statements hierarchically. Because removing inner statements may cause the statements in the outer layer to be irrelevant, HGA has to rerun the extraction result. As Algorithm 1 shows, after each running of HGA, we compare the result with the input code to see if the result has LOC decreasing and HGA's running times do not exceed the maximum number R. If so, we will run HGA on the result to find more irrelevant statements.

Algorithm 1. Hierarchical Genetic Algorithm

Input: the original JS code C, a set of test cases T that specify the desired features in C

Output: a result code snippet CS which passes all of T and contains the statements as few as possible

1: **function** HGA($C, count$)
2: $CS \leftarrow C$;
3: $l \leftarrow 0$;
4: **while** $l \leqslant$ the max layer of statements in C **do**;
5: $S_l \leftarrow$ the statements of C in layer l;
6: $res_l \leftarrow$ GA(C, S_l, T); $\triangleright$ Remove the statements in S_l that make no contributions to passing the tests in T
7: **if** $fitness(res_l) \geqslant 1$ **then**
8: $CS \leftarrow res_l$;
9: **end if**
10: $++l$;
11: **end while**
12: **if** LOC(CS) < LOC(C) **and** $count \leqslant R$ **then**
13: $CS \leqslant$ HGA($CS, count + 1$)
14: **end if**
15: **return** CS
16: **end function**

Removing the statements could be regarded as pruning nodes in an abstract syntax tree (AST). Once a statement is deleted, the corresponding nodes in the AST should be removed. Any subtrees whose roots are in the removed nodes should also be deleted. That is, if a compound statement is removed, the inner statements should also be removed. We give each statement a number to indicate its layer in the hierarchical structures of statements in the AST. The statements in layer 0 are defined as the children of the root in the AST. The sta-tement's layer will be increased with the depth of the AST node.

In each layer, a genetic algorithm (GA) is used to find a code snippet with the highest fitness as denoted by res_l in Algorithm 1. If the fitness of res_l is higher than 1, which means some statements are removed and the result passes all of the test cases without errors, the result will be the candidate code and handled in the next layer. Otherwise, the original code will be the candidate in the next layer.

To perform the GA on each layer, we need to encode the source code to a chromosome, define the gene, and specify the crossover, mutation, and selection strategies. Moreover, the individual's fitness should be defined and an appropriate initial population needs to be

constructed to find an approximate best result quickly.

For a code snippet with n lines of code, the algorithm generates an array of n integers to be the chromosome of the code. The element in the chromosome is a gene which represents the choosing of the code. For instance, if the i-th gene in chromosome is 1, the statement in line i will be kept; otherwise, the statement will be removed if the corresponding gene is 0.

The crossover in GA is Uniform Crossover, and the mutation is Flip Mutation. The selection strategy in GA is selecting the top 10% individuals into the next generation first, and we use champion selection for the others. The individuals are ranked by fitness.

In the GA process of software transplantation [13], the passing rate of test cases is considered to evaluate the fitness of an individual, while the compiler would remove the individuals failing to compile. In our case, JavaScript is an interpreted programming language and has no "compile error". But during the loading phase, "Script Error" or "Reference Error" may appear. Any individual with such errors will have zero fitness. Meanwhile, if both the two individuals can pass all of the test cases with no errors, we prefer the one with less lines of code. The assumption is based on the learning barriers of unfamiliar codes [10]. From the user's perspective, the extraction result is a demo for using the desired features. Less code means lower learning barrier and less superfluous details that distract from learning core concepts.

The calculation of fitness is defined as (1), where $fitness(i)$ is the fitness of individual i which contains $|chromosome|$ lines of code. $P(T, i)$ stands for the number of tests in T that i can pass, and $LOC(i)$ represents the lines of code in i. ERR_i indicates if the i-th individual has any "Script Error" or "Reference Error", and it is assigned to 1 if so.

$$fitness(i) = \begin{cases} 0, & \text{if } ERR_i = 1, \\ \frac{P(T,i)}{|T|}, & \text{if } \frac{P(T,i)}{|T|} < 1 \text{ and } ERR_i = 0, \\ 2 - \frac{LOC(i)}{|chromosome|}, & \text{if } \frac{P(T,i)}{|T|} = 1 \text{ and } ERR_i = 0. \end{cases} \quad (1)$$

The individual is run on the browser to calculate the fitness. It will cost about 1 second per individual, which means the time cost will be unacceptable if the search space (i.e., the population of GA) is too large. As a result, we generate a set of individuals that have high fitness as the initial population in each layer as shown in Algorithm 2. To ensure that we will get at least one result that can pass all of the tests and has no errors, we put the original code in the current layer into the initial population (IP) as shown in line 3.

Algorithm 2. Initial Population Generation

Input: the handled JS statements S_l in the current layer l, the size n of population, the chosen test cases T

Output: the initial population IP

```
1:  function IPGENERATOR(C, S_l, T)
2:      IP ← ∅;
3:      IP.push(C)
4:      greedyRes ← GETGREEDYRES(C, S_l, T);
5:      IP.push(greedyRes);
6:      for s_i ∈ S_l do
7:          IP.push(S_l − s_i);
8:      end for
9:      if |IP| ⩾ n then
10:         IP ← top-n individuals in IP sorted by fitness;
11:     else
12:         randomIP ← RANDOMCHOOSING(S_l, n − |IP|)
13:         IP.push(randomIP)
14:     end if
15:     return IP
16: end function
17: function GETGREEDYRES(C, S_l, T)
18:     C' ← C;
19:     for s_i ∈ S_l do
20:         if P(T, C' − s_i) = P(T, C) then
21:             C' ← C' − s_i
22:             C' ← GETGREEDYRES(C', S_l, T)
23:         end if
24:     end for
25:     return C'
26: end function
```

Then, we use the greedy strategy (line 4) to find a result by removing the statements individually as the second member of IP. More specifically, for each statement in the current layer, it will be removed if the removing does not affect the result's passing rate; otherwise, it will be kept. Once a statement is removed, the algorithm will repeat checking the irrelevant statements in the result until no statements can be removed (as shown in lines 20–22).

To enlarge the population of IP, we remove the statements in the current layer in sequence and add the results to IP as shown in lines 5–7. If the population is beyond the threshold of IP (supposed to be n), we will sort the individuals by fitness, and choose top n individuals to be the final result as shown in line

9 and line 10. Otherwise, we will randomly select a set of statements in C as individuals to fill up IP as shown in lines 11–13.

The number of generation and the size of population per generation are related to the handled statements (denoted as $|gene|$) in the current layer. We set two upper bounds to limit the searching space. The two properties are defined as:

$$|generation| = \max(5 \times |gene|, 100),$$
$$|population| = \max(10 \times |gene|, 200).$$

3.4 Running Example

We take the code snippet in Fig.2 to illustrate the HGA process. As Fig.4(a) shows, we first transform the code to a chromosome, where each gene is a binary number that represents the choosing of statements. In the first layer, only one statement (i.e., the function declaration of `update`) exists. We do bit flip mutation on the first gene in individual i_{1-1}, and get a new individual i_{1-2}. Note that the first gene in i_{1-1} corresponds to the function definition in line 1 in Fig.2, and the last gene is the end of the definition statement. To keep the code structure, both genes are changed to 0, and the inner statements are set to be 0 consequently. The fitness of i_{1-2} is zero since none of the chosen tests can be passed.

In the second layer, suppose we get three individuals i_{2-1}, i_{2-2} and i_{2-3}, in one generation. The fitness of i_{2-1} is 0.25 since it can only pass test #183. While in the second generation, the uniform crossover happens on i_{2-1} and i_{2-2}. We get i_{2-4} and i_{2-5}. Similarly, the fitness of i_{2-4} is 0.25 since it can only pass test #183. Meanwhile, one gene (in orange) in i_{2-3} is mutated (bit flip) and we get a new individual i_{2-6}.

Suppose the upper bound of the population is 4, when picking the individuals to the next generation, i_{2-6} is picked since its fitness is in the top 10%. And the other three individuals will be selected from the left five individuals by champion selection.

4 Implementation

We run all the tests on Ubuntu 18.04 with 3.0 GHz Intel Core i5 and 8 GB Memory. The test tool and the browser are on the SSD for the I/O speed purposes. We run the tests on headless Chrome and the GPU-disabled mode.

We develop the GA algorithm based on DEAP[3], and the probability of crossover is 0.5 and that of mutation is 0.1. In our implementation, we set the upper bound of HGA's running times to be 3.

In some cases, the component contains more than one JS file. A feature implementation may involve subroutine invocations. In respect of HGA, we use one chromosome to represent the statements in all of the JS files. We build a gene-statement mapping to ease the modifications of the statements in JS files: once a gene is changed, we find the location of the corresponding statement in the mapping and change the statement in the corresponding file.

5 Evaluation

In this section, we investigate the correctness and effectiveness of our method. We address two research questions.

*RQ*1. Compared with existing studies, what are the strength and weakness of our method?

*RQ*2. To what extent can our method facilitate the user in feature extraction?

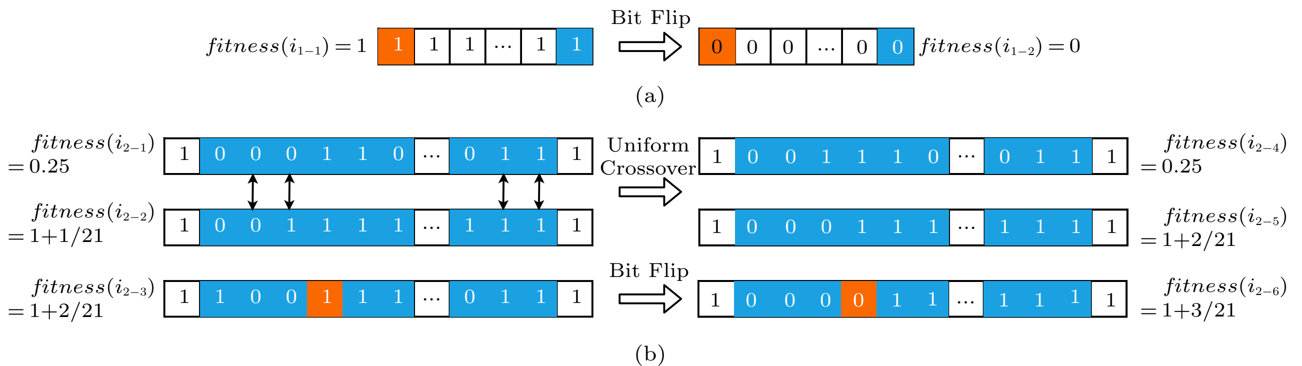


Fig.4. Running example of the code in Fig.2. (a) Individuals in layer 1. (b) Individuals in layer 2.

[3]DEAP evolutionary computation framework. https://github.com/DEAP/deap, July 2021.

5.1 RQ1: Method Comparison

Two existing feature extraction methods were compared with our method: FireCrystal [6] and Firecrow [9]. FireCrystal is based on code instrumentation, and Firecrow is based on program slicing.

5.1.1 *Study Design*

We conducted the feature extraction on two datasets. One dataset (denoted as dataset-1) is proposed in Firecrow [9], which includes nine web pages and 13 required features. One feature was not included in our evaluation since it was not found in the author's dataset④ and the given URL was not accessible. Hence we actually used eight different web pages and 12 required features. The features in this dataset were specified by usage scenarios which were described as Selenium test script.

The features in dataset-1 are GUI-related and we built another dataset (denoted as dataset-2) containing 10 web components with 10 sets of features specified by usage scenarios and test cases. The web components were collected from Github. As shown in Table 1, #TA and #TC represent the number of test assertions and test cases respectively. All the components have more than 100 stars and have relatively complete test cases using QUnit⑤. The required features could not be directly extracted by configurations. The test cases were chosen by two participants, and we accepted the test case for a feature if it was chosen by both of the participants; otherwise, we would ask another participant to decide the test cases.

We did not have FireCrystal's source code. Instead, we implemented it by recording the executed statements in the usage scenario and the test cases. Firecrow was a Firefox plugin, thereby we conducted the experiments on Firefox (v28.0.0.5186).

The extracted result was thought to be correct if the usage scenario could be reproduced on the result correctly, and it could pass the given tests with no errors. If two results were correct, we preferred the result with fewer statements.

The goal of our method was to extract the implementation of features in the component. As a result, we did not extract the library codes like jQuery⑥ or requireJS⑦ although they contributed to the feature.

5.1.2 *Results*

Table 2 and Table 3 describe the extraction results in the two datasets respectively. In dataset-2, we failed to use Firecrow to do the extraction in any of the components. The main reason was that Firecrow did not support handling the JS files written with ECMAScript6.

Table 1. Web Components Used in Dataset-2

ID	URL	Feature Requirement	#TA	#TC
C1	uxsolutions/bootstrap-datepicker	Get the "highlight day of the week" feature from the calendar	5	1
C2	mugifly/jquery-simple-datetimepicker	Extract the "date" from "datetimepicker"	29	9
C3	Mottie/Keyboard	Simple arithmetic operations (+, −, ×, ÷, =) and numbers	13	1
C4	OwlCarousel2 /OwlCarousel2	Extract the "swipe" function which can browse the image by dragging or clicking the buttons	17	8
C5	Prinzhorn/skrollr	Get the "rotate texts", "horizonal or vertical text moving" when the scrollbar moves	64	6
C6	smalot/bootstrap-datetimepicker	Change the "dateTimePicker" to a simple date picker, without changing the format of the date	239	53
C7	igorescobar/jQuery-Mask-Plugin	Get the telephone number formatting function	60	10
C8	select2/select2	Select a set of elements; when selecting the same element or clicking the fork icon of the element, the element should be removed	10	5
C9	jquery-backstretch/jquery-backstretch	Clicking different buttons, change the background to the corresponding image	8	3
C10	jquery-validation/jquery-validation	Get the E-mail and password formatting function	9	1

Note: All of the URLs start with "https://github.com/". #: number of.

④Firecrow. https://github.com/jomaras/Firecrow, July 2021.
⑤QNnit JavaScript testing framework. https://qunitjs.com/, July 2021.
⑥The jQuery JavaScript library. https://jquery.com/, July 2021.
⑦RequireJS. https://requirejs.org/, July 2021.

Table 2. Feature Extraction Results on Dataset-1

ID	LOC_{ori}	$LOC_{\text{FireCrystal}}$	LOC_{FireCrow}	LOC_{ours}
F1	1 642	476	239	35
F2	1 642	474	207	13
F3	1 642	592	345	207
F4	1 174	229	139	19
F5	1 174	228	139	17
F6	260	71	38	26
F7	594	160	130	13
F8	53	47	47	46
F9	10	10	10	10
F10	16	16	16	16
F11	301	161	125	101
F12	518	224	202	158

Table 3. Feature Extraction Results on Dataset-2

ID	LOC_{ori}	FireCrystal		Ours	
		LOC	Correct	LOC	Correct
C1	1 757	980	Y	258	Y
C2	2 143	1 679	Y	578	Y
C3	4 267	2 557	N	802	Y
C4	2 976	2 318	Y	1 368	Y
C5	1 088	736	Y	600	Y
C6	1 772	1 257	Y	610	Y
C7	473	364	Y	161	Y
C8	5 301	3 375	Y	1 304	Y
C9	1 276	688	Y	273	Y
C10	1 404	833	Y	371	Y

All of the tasks in the first dataset were successfully fulfilled by the three methods. In dataset-2, one FireCrystal's result (C3) included some irrelevant buttons and got the wrong result. In general, FireCrystal got the biggest size of results, FireCrow the medium, and our method the minimum.

The reason for FireCrystal having the biggest size of results is that the code instrumentation would involve statements that were executed in the scenario but made no contributions to the features [6]. As shown in Fig. 5, in C3, the features were specified by a test case including 13 assertions checking the calculation of plus, minus, multiplication, and division for the integers and floats. Hence an acceptable result should only contain the basic arithmetic operators and the digits. The developer gave every button a property to define the functionality and these definitions were located in the constructor function. The definitions were executed and thus some irrelevant buttons were kept with the buttons of interest in FireCrystal's result.

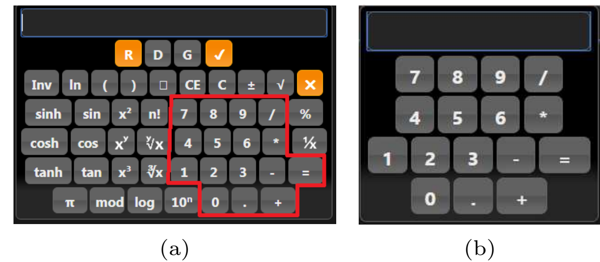


Fig.5. FireCrystal's result in (a) includes some unwanted buttons which are not in the red closure and is different to (b) the correct appearance.

In Firecrow, the method first built a dependency graph of the source code. Then the method did dynamic slicing starting from the triggered events and the user-specified DOM elements. All the dependent statements were treated as the feature implementation, and thus the executed irrelevant statements were removed.

The dynamic slicing was heuristic. It would keep the statements that manifest the features directly or indirectly. From our observation of Firecrow's results, statements that had dependencies on the specified DOM elements were kept as the feature implements. And consequently, some irrelevant statements were involved. For instance, in F4 and F5, some event listeners of the specified DOM elements were kept in Firecrow, while the handlers were removed (as shown in lines 18, 21–25, Fig.6). These events were not triggered in the usage scenario, thereby they were feature-irrelevant. But in the view of program slicing, these events belonged to the DOM object's properties and thus had dependencies on the DOM element. As a consequence, these events were kept. Another majority part of irrelevant statements kept in Firecrow were the maps, since the method did not handle arrays (as shown in lines 3–12, in Fig.6). But in the jQuery code, many parameters in the callbacks were written in a key-value way, and that is another reason why our method reduced irrelevant statements significantly.

Compared with the above methods, our method achieved the minimum size of feature implementation. However, some of the removed statements should be kept to enhance the robustness of the program. For instance, our method removed the variable declarations without assignments. The result could also run correctly, but the related variables became global and would be harmful to the program's robustness.

Briefly speaking, all of the methods supported extracting features specified by usage scenarios or test cases (Firecrow also supports extracting the features specified by the tests as described in [9]). Our method

```
▶ Arrays
▶ Event Handlers

  1. ...
  2. jQuery.extend(jQuery.easing, {
▶ 3.     def: 'easeOutQuad',
▶ 4.     swing: function (x, t, b, c, d) {
▶ 5.         return jQuery.easing[jQuery.easing.def](x, t, b, c, d);
▶ 6.     },
▶ 7.     easeInQuad: function (x, t, b, c, d) {},
▶ 8.     easeOutQuad: function (x, t, b, c, d) {
▶ 9.         return ((-c * (t /= d)) * (t - 2)) + b;
▶10.     },
▶11.     easeInOutQuad: function (x, t, b, c, d) {},
▶12.     easeInCubic: function (x, t, b, c, d) {},
 13.    ...
 14. });
 15. $(document).ready(function () {
 16.     $('a[rel=external]');
 17.     $('#portfolio .index a').each(function (i, link) {
▶18.         $(link).click(function (e) {
 19.         });
 20.     });
▶21.     $('#news li a').click(function (e) {});
▶22.     $('#news-article .close').live('click', function (e) {});
▶23.     $('#news-article .arrow').live('click', function (e) {});
▶24.     $('#footer .interactive').hover(function (){}, function () {
▶25.     });
 26.     ...
 27. });
 28. ...
```

Fig.6. Some redundant statements kept in Firecrow: the arrays and event handlers.

generated the smallest size of results in all of the tasks.

5.2 RQ2: Human-Subject Study

In this study, we checked if our method could help the participants extract the features. We investigated the efforts that the participants spent on using our method and extracting the feature implementation by hand. We transformed the usage scenario to be a GUI test case with a series of assertions. The GUI test case was appended to the chosen test cases as the requirements of features. Moreover, the diversity of the test cases chosen by participants was checked to see if they had a significant difference. Our results were compared with the manually extracted results to see the advantages and drawbacks of our method.

5.2.1 Study Design

We invited five participants to the experiments. They had some experience with programming but were relatively new to JavaScript programming. The subjects were three sets of features in three web components listed in Table 1. Each set of features contains both visual and appearance-irrelevant features.

On the one hand, we asked the participants to choose the feature-relevant tests from the test cases provided by the implementation of the corresponding web component. On the other hand, the participants were asked to use our GUI generating method to interact with the feature-relevant elements to generate a GUI test script.

We recorded the time that participants spent on choosing the test cases and interacting with the components (the two processes were denoted as "feature specification"). The time cost represented the labor cost of using our method in the feature extraction. For comparison, we asked the participants to manually extract the implementation of the features and recorded the extraction time. A retrospective interview was conducted after the participants finished the extraction.

We compared the time of feature specification with the time of manual feature extraction to see if our method can help reduce the human efforts. Then we evaluated the consistency of the test cases chosen by different participants. Moreover, we compared the manually extracting result with our result and asked the participants to explain why they kept or deleted the statements.

5.2.2 Results

Fig. 7 shows the comparison of time cost on feature specification and extracting the implementation by hand. We can see the time cost spent on feature

specification is much less than that spent on manual extraction. Participants reported that in C5 and C6, they could not get the correct entry point of the feature by Firefox Developer tools. Instead, they started foraging from the constructors and guessed the entry point by reading the API documentation and the developers' comments. When the participants found out the core functions related to the features, they struggled to find all of the dependencies that supported the executing of the core functions. In contrast, the participants told that choosing the feature-related test cases was much easier since the test cases provide detailed descriptions and the number of test cases was smaller than the number of statements. And interacting with the feature-relevant elements was also simple to the participants because the feature descriptions were clear and straightforward.

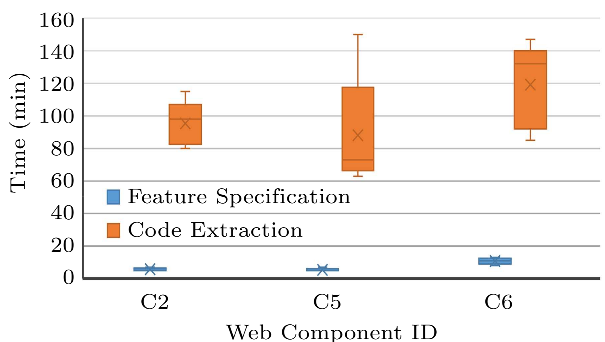


Fig.7. Time cost of feature specification and manually feature extraction by five participants.

We calculated the consistency of the test cases chosen by different participants to check if the chosen test cases were varied by different participants. Table 4 shows that in C2 and C5, most of the tests are the same in five participants' results. That means the participants have a similar understanding of the test cases. The main reason was that the test cases in these components had explicit descriptions and were well structured in modules, as one participant said: "It is easy to know what this test is working for. I clearly know what I want."

Table 4. Number of Consistency Test Assertions Chosen by Participants

Component	#Inconsistency/#Ours				
	P01	P02	P03	P04	P05
C2	0/25	0/25	0/25	2/25	1/25
C5	0/64	−1/64	0/64	0/64	0/64
C6	−2/239	+44/239	+31/239	+12/239	+15/239

Note: The positive number means the participant chose more test assertions than ours. And the negative number means the participant missed some test assertions. "#Ours" represents the number of assertions chosen by our method.

In C6, there were some test assertions that the participants chose but were not included in our tests. We found that most of these assertions were described as testing the format of the date like "tomorrow", "next year", "last month". But in fact, the subject of these test cases was a JS function "`val()`". The test oracles checked if the outputs of this function were as expected in various inputs as shown in Fig.8. The participants were misled by the test description, and they wrongly chose these test cases since they did not check the implementations of these tests.

An essential question is whether the additional/missing test cases will make the extraction result different. The answer is yes. The extracting result will be influenced by the type of additional/missing test cases.

If an additional test case is chosen by users, an extra feature will be included in our result. If a test case is missing, a feature may or may not be lost in the extraction result. If the chosen tests and the usage scenario do not cover the missing feature, the extraction result will lose the feature. Otherwise, the extracting result will be the same as expected. The missing test case needs to be selected by hand. Luckily, the foraging cost is not big.

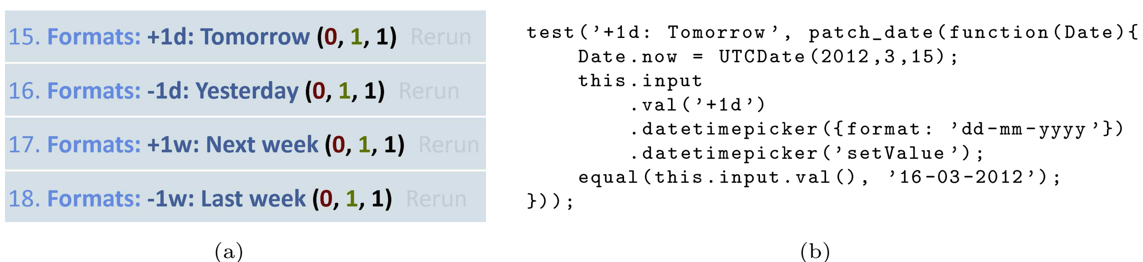


Fig.8. Test cases in C6 wrongly chosen by participants. (a) #15 was described to test the format of the setting of the day "Tomorrow", but in fact, the test assertion was checking if the function `input.val('+1d')` in (b) could correctly run.

We compared the manual extraction results of the five participants with our results as shown in Table 5. Our results contained fewer statements than any of the participants' results. We found that the participants mostly deleted the coarse-grained code snippets, for instance, the class or function definitions, the `if` statements, or the `try-catch` blocks, while they seldom removed the single statements. All of them said they could delete more statements in their results. But they gave up as most of the rest had complex dependencies, and removing the irrelevant statements costs much effort. One participant said in the interview: "Locating the implementation (of the core function) is overwhelming. I think it is enough to delete the irrelevant functions. And I have to admit that there are more statements that could be removed. But it will cost me much more time (to finish the task), because finding the dependencies of variables is very complex and boring."

Table 5. LOC Comparison Between Manual Results and Our Results

Component	Participant's ID					Our Result
	P01	P02	P03	P04	P05	
C2	896	1 140	876	912	843	578
C5	869	982	757	891	756	600
C6	1 053	1 366	1136	964	1 169	610

Our results were compared with manual results to see the divergent decisions on the statements. The removed code snippets were classified into three categories by their sizes: small block (1–5 consecutively deleted statements), medium block (6–10 consecutively deleted statements), and large block (10+ consecutively deleted statements). We counted the number of blocks and LOC for each kind of blocks. There were 1 001 small blocks (total LOC = 2 077), 158 medium blocks (total LOC = 1 144) and 157 large blocks (total LOC = 2 448). In respect of LOC, the small blocks occupied 36.63% of the deleted statements ($\overline{delLOC}$ = 2.05), 20.19% of the deleted statements were in medium blocks ($\overline{delLOC}$ = 7.25), and 43.18% were in large blocks ($\overline{delLOC}$ = 15.59); while in respect of the number of blocks, 76.24% of the deleted code snippets were small blocks, 11.92% were medium blocks, and 11.84% were large blocks.

We looked through the deleted statements and found some typical statements that were missed by the participants. Most statements in small blocks were variable declarations, branches, function calls, etc. The medium and large blocks were mostly function definitions and class definitions. Our method removed considerable small code blocks, meaning that the participants forgot to remove a lot of scattered small code snippets.

We also found some statements that could be retained as the participants did but were removed from our results. The typical statements are the variable declarations without initializations and the functions for enhancing the compatibility on different browsers. In our results, the variable declaration without the initialization was deleted and made the variable global if no error happened. This would influence the program understanding and software maintenance. In the further, heuristic rules to enhance code quality will be added to solve this problem, which will be discussed in Section 6. The code of adaptation was removed since we only used Firefox and these statements were not executed in the test cases. The latter case happened in the other existing methods since these features were not specified in the scenario or the test cases. Keeping these statements in the extraction result can enhance the robustness of the component. But checking the completeness of the feature specification is beyond the scope of this paper.

In conclusion, our method can extract the implementation of certain features. Users can save significant time with our method. To enhance the implementation quality, users could manually look through the extraction result and decide the statements which should be deleted.

6 Limitations and Discussion

In this section, we discuss the limitations in our method, including the flaky test problem, the quality of test cases, and possible irrelevant code in the extraction result.

We generate a GUI test case to compare the desired elements' appearance-related properties. In some cases, the properties may be dynamically generated by the components and may not be consistent in different runs of tests. The generated test case may be flaky. For instance, a test assertion checks the value of "currentDate" to be "1st Jan.", resulting in that the extracting result can pass the test only on 1st Jan..

Our HGA method only considers the hierarchical structure of the code, and it works in most cases. However, the code structure is more complex. For instance, there may be some irrelevant statements distributed in different layers and had dependencies on each other. In this case, our method will be ineffective since it only considers removing statements in the same layer. These

cases rarely happened in our experiments and are remained to be solved in our future work.

If the case cannot cover all the feature implementation, some features such as the exception handlers or adaptation functions may be removed. Enlarging the test cases is vital but beyond the scope of this paper. Currently, we generate a result that highlighted the removed statements and asked the user to reserve or remove them. Moreover, some irrelevant parameters may also exist in our result even though they are not active since the handlers are removed. Removing these irrelevant parameters can help users better understand the implementation of features. But the process is complex since the parameters have much more complex dependencies than statements.

7 Conclusions

In this paper, a novel code extraction method was proposed to extract the implementation of features specified by the test cases. The feature requirements were transformed to be a set of test cases which cover both the visual and logical functionalities. The results of quantitative and qualitative experiments showed that the proposed method could correctly extract the implementation utilizing the test cases provided by users and generated from user scenarios. Compared with directly extracting features from the source code, specifying the features by test cases could significantly save the human efforts. The proposed method could facilitate the software reuse of web applications.

In the GUI test case generation, we only generated a test case that checks the visual appearance of the result but ignores the appearance-unrelated features such as the memory cost. If these features are not covered by the test cases or usage scenario, they cannot be chosen in our method. We will provide more information on the user's interaction with the component in the future work, and this may help the user find more features of interest.

It is also important to seek to insert the extracted result into users' software automatically. That will help the users reuse the components in a flexible and easy way. In this case, the extraction result has to adapt to the environment. We prepare to seek inspiration from studies on software transplantation [13] to achieve the goal.

References

[1] Krueger C W. Software reuse. *ACM Computing Surveys*, 1992, 24(2): 131-183. DOI: 10.1145/130844.130856.

[2] Chattopadhyay S, Nelson N, Gonzalez Y R, Leon A A, Pandita R, Sarma A. Latent patterns in activities: A field study of how developers manage context. In *Proc. the 41st IEEE/ACM Int. Conference on Software Engineering*, May 2019, pp.373-383. DOI: 10.1109/ICSE.2019.00051.

[3] Gascon-Samson J, Jung K, Goyal S, Rezaiean-Asel A, Pattabiraman K. ThingsMigrate: Platform-independent migration of stateful JavaScript IoT applications. In *Proc. the 32nd European Conference on Object-Oriented Programming*, July 2018, Article No. 18. DOI: 10.4230/LIPIcs.ECOOP.2018.18.

[4] Xu B, An L, Thung F, Khomh F, Lo D. Why reinventing the wheels? An empirical study on library reuse and re-implementation. *Empirical Software Engineering*, 2020, 25(1): 755-789. DOI: 10.1007/s10664-019-09771-0.

[5] Krüger J, Mukelabai M, Gu W, Shen H, Hebig R, Berger T. Where is my feature and what is it about? A case study on recovering feature facets. *Journal of Systems and Software*, 2019, 152: 239-253. DOI: 10.1016/j.jss.2019.01.057.

[6] Oney S, Myers B. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Proc. the 2019 IEEE Symp. Visual Languages and Human-Centric Computing*, Sept. 2009, pp.105-108. DOI: 10.1109/VLHCC.2009.5295287.

[7] Hibschman J, Zhang H. Unravel: Rapid web application reverse engineering via interaction recording, source tracing, and library detection. In *Proc. the 28th Annual ACM Symp. User Interface Software & Technology*, Nov. 2015, pp.270-279. DOI: 10.1145/2807442.2807468.

[8] Alimadadi S, Sequeira S, Mesbah A, Pattabiraman K. Understanding JavaScript event-based interactions. In *Proc. the 36th Int. Conference on Software Engineering*, May 2014, pp.367-377. DOI: 10.1145/2568225.2568268.

[9] Maras J, Stula M, Carlson J, Crnkovic I. Identifying code of individual features in client-side web applications. *IEEE Trans. Software Engineering*, 2013, 39(12): 1680-1697. DOI: 10.1109/TSE.2013.38.

[10] Shaffer D W, Resnick M. "Thick" authenticity: New media and authentic learning. *Journal of Interactive Learning Research*, 1999, 10(2): 195-216.

[11] Razzaq A, Le Gear A, Exton C, Buckley J. An empirical assessment of baseline feature location techniques. *Empirical Software Engineering*, 2020, 25(1): 266-321. DOI: 10.1007/s10664-019-09734-5.

[12] Burg B, Ko A J, Ernst M D. Explaining visual changes in web interfaces. In *Proc. the 28th Annual ACM Symp. User Interface Software & Technology*, Nov. 2015, pp.259-268. DOI: 10.1145/2807442.2807473.

[13] Barr E T, Harman M, Jia Y, Marginean A, Petke J. Automated software transplantation. In *Proc. the 2015 Int. Symp. Software Testing and Analysis*, July 2015, pp.257-269. DOI: 10.1145/2771783.2771796.

[14] Jensen S H, Møller A, Thiemann P. Type analysis for JavaScript. In *Proc. the 16th Int. Symp. Static Analysis*, August 2009, pp.238-255. DOI: 10.1007/978-3-642-03237-0_17.

[15] Kashyap V, Dewey K, Kuefner E A *et al.* JSAI: A static analysis platform for JavaScript. In *Proc. the 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, Nov. 2014, pp.121-132. DOI: 10.1145/2635868.2635904.

[16] Guha A, Saftoiu C, Krishnamurthi S. The essence of JavaScript. In *Proc. the 24th European Conference on Object-Oriented Programming*, June 2010, pp.126-150. DOI: 10.1007/978-3-642-14107-2_7.

[17] Madsen M, Livshits B, Fanning M. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proc. the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering*, August 2013, pp.499-509. DOI: 10.1145/2491411.2491417.

[18] Guarnieri S, Pistoia M, Tripp O, Dolby J, Teilhet S, Berg R. Saving the world wide web from vulnerable JavaScript. In *Proc. the 2011 Int. Symp. Software Testing and Analysis*, July 2011, pp.177-187. DOI: 10.1145/2001420.2001442.

[19] Malik R S, Patra J, Pradel M. NL2Type: Inferring JavaScript function types from natural language information. In *Proc. the 41st IEEE/ACM Int. Conference on Software Engineering*, May 2019, pp.304-315. DOI: 10.1109/ICSE.2019.00045

[20] Jensen S H, Madsen M, Møller A. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. the 19th ACM SIGSOFT Symp. and the 13th European Conference on Foundations of Software Engineering*, Sept. 2011, pp.59-69. DOI: 10.1145/2025113.2025125.

[21] Kristensen E K, Møller A. Reasonably-most-general clients for JavaScript library analysis. In *Proc. the 41st Int. Conference on Software Engineering*, May 2019, pp.83-93. DOI: 10.1109/ICSE.2019.00026.

[22] Madsen M, Tip F, Lhoták O. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices*, 2015, 50(10): 505-519. DOI: 10.1145/2858965.2814272.

[23] Park C, Ryu S. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *Proc. the 29th European Conference on Object-Oriented Programming*, July 2015, pp.735-756. DOI: 10.4230/LIPIcs.ECOOP.2015.735.

[24] Andreasen E, Gong L, Møller A *et al.* A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys*, 2017, 50(5): Article No. 66. DOI: 10.1145/3106739.

[25] Sen K, Kalasapur S, Brutch T, Gibbs S. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proc. the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, August 2013, pp.488-498. DOI: 10.1145/2491411.2491447.

[26] Burg B, Bailey R, Ko A J, Ernst M D. Interactive record/replay for web application debugging. In *Proc. the 26th ACM Symp. User Interface Software and Technology*, Oct. 2013, pp.473-484. DOI: 10.1145/2501988.2502050.

[27] Mahajan S, Halfond W G. Finding HTML presentation failures using image comparison techniques. In *Proc. the 29th ACM/IEEE Int. Conference on Automated Software Engineering*, Sept. 2014, pp.91-96. DOI: 10.1145/2642937.2642966.

[28] Ocariza F S, Pattabiraman K, Mesbah A. Detecting inconsistencies in JavaScript MVC applications. In *Proc. the 37th Int. Conference on Software Engineering*, May 2015, pp.325-335. DOI: 10.1109/ICSE.2015.52.

[29] Wang J, Dou W, Gao C, Wei J. JSTrace: Fast reproducing web application errors. *Journal of Systems and Software*, 2018, 137: 448-462. DOI: 10.1016/j.jss.2017.06.038.

[30] Li P, Wohlstadter E. Script InSight: Using models to explore JavaScript code from the browser view. In *Proc. the 9th Int. Conference on Web Engineering*, June 2009, pp.260-274. DOI: 10.1007/978-3-642-02818-2_21.

[31] Dow S P, Glassco A, Kass J, Schwarz M, Schwartz D L, Klemmer S R. Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Trans. Computer-Human Interaction*, 2010, 17(4): Article No. 18. DOI: 10.1145/1879831.1879836.

[32] Gibson D, Punera K, Tomkins A. The volume and evolution of web page templates. In *Proc. the 14th Int. Conference on World Wide Web*, May 2005, pp.830-839. DOI: 10.1145/1062745.1062763.

[33] Lee B, Srivastava S, Kumar R, Brafman R, Klemmer S R. Designing with interactive example galleries. In *Proc. the SIGCHI Conference on Human Factors in Computing Systems*, Apr. 2010, pp.2257-2266. DOI: 10.1145/1753326.1753667.

[34] Kumar R, Talton J O, Ahmad S, Klemmer S R. Bricolage: Example-based retargeting for web design. In *Proc. the SIGCHI Conference on Human Factors in Computing Systems*, May 2011, pp.2197-2206. DOI: 10.1145/1978942.1979262.

[35] Swearngin A, Dontcheva M, Li W, Brandt J, Dixon M, Ko A J. Rewire: Interface design assistance from examples. In *Proc. the 2018 CHI Conference on Human Factors in Computing Systems*, Apr. 2018, Article No. 504. DOI: 10.1145/3173574.3174078.

[36] Wang Z, Cheng B, Jin Y, Feng Y, Chen J. EasyApp: A widget-based cross-platform mobile development environment for end-users. In *Proc. the 23rd Annual Int. Conference on Mobile Computing and Networking*, Oct. 2017, pp.591-593. DOI: 10.1145/3117811.3131242.

[37] Benson E O, Karger D R. Cascading tree sheets and recombinant HTML: Better encapsulation and retargeting of web content. In *Proc. the 22nd Int. Conference on World Wide Web*, May 2013, pp.107-118. DOI: 10.1145/2488388.2488399.

[38] Verou L, Zhang A X, Karger D R. Mavo: Creating interactive data-driven web applications by authoring HTML. In *Proc. the 29th Annual Symp. User Interface Software and Technology*, Oct. 2016, pp.483-496. DOI: 10.1145/2984511.2984551.

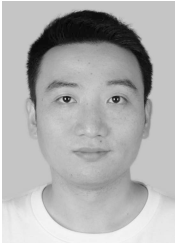
[39] Liu X, Huang G, Zhao Q, Mei H, Brain B M. iMashup: A mashup-based framework for service composition. *Science China Information Sciences*, 2014, 57(1): 1-20. DOI: 10.1007/s11432-013-4782-0.

[40] Huang G, Liu X, Ma Y, Lu X, Zhang Y, Xiong Y. Programming situational mobile web applications with cloud-mobile convergence: An Internetware-oriented approach. *IEEE Trans. Services Computing*, 2016, 12(1): 6-19. DOI: 10.1109/TSC.2016.2587260.

[41] Eisenbarth T, Koschke R, Simon D. Locating features in source code. *IEEE Trans. Software Engineering*, 2003, 29(3): 210-224. DOI: 10.1109/TSE.2003.1183929.

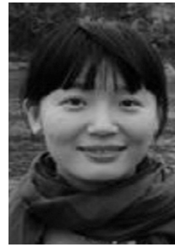
[42] Mahajan S, Alameer A, McMinn P, Halfond W G. Automated repair of layout cross browser issues using search-based techniques. In *Proc. the 26th ACM SIGSOFT Int. Symp. Software Testing and Analysis*, July 2017, pp.249-260. DOI: 10.1145/3092703.3092726.

[43] Chang T H, Yeh T, Miller R C. GUI testing using computer vision. In *Proc. the 28th International Conference on Human Factors in Computing Systems*, Apr. 2010, pp.1535-1544. DOI: 10.1145/1753326.1753555.



Yong-Hao Long is currently a postdoc fellow in The Hong Kong Polytechnic University, Hong Kong. He received his Ph.D. degree in computer science at Sun Yat-sen University, Guangzhou, in 2021. His research interests include software reuse and GUI testing.

Yan-Cheng Chen received his M.S. degree in software engineering from Sun Yat-sen University, Guangzhou, in 2020. His research interests include software testing and verification.



Xiang-Ping Chen is currently an associate professor in the Sun Yat-sen University, Guangzhou. She received her Ph.D. degree in software engineering from the Peking University, Beijing, in 2010. Her research interests include software engineering and mining software repositories.

Xiao-Hong Shi is currently a lecturer in Guangzhou College of Commerce, Guangzhou. She received her Ph.D. degree in mathematics from Guangzhou University, Guangzhou, in 2021. Her research interests include software engineering and multimedia processing.

Fan Zhou is currently a professor in Sun Yat-sen University, Guangzhou. He received his Ph.D. degree in computer science from Sun Yat-sen University, Guangzhou, in 2007. His research interests include computer graphics, computer aided design and image processing.