An Automated Input Generation Method for Crawling of Web Applications

Yuki Ishikawa13, Kenji Hisazumi12, Akira Fukuda125
1Graduate School and Faculty of Information Science and Electrical Engineering
Kyushu University, Motooka 774, Nishi-ku, Fukuoka, Japan 819-0395
2System LSI Research Center
Kyushu University, Motooka 774, Nishi-ku, Fukuoka, Japan 819-0395
3Email: y_ishikawa@f.ait.kyushu-u.ac.jp
4Email: nel@f.ait.kyushu-u.ac.jp
5Email: fukuda@f.ait.kyushu-u.ac.jp

Abstract—Even though many tools provide automatic crawling, sometimes web applications are crawled manually in case of testing. The manual crawling takes much time and effort because it requires the construction of appropriate input sequences. To solve this problem, we proposed and developed an automated input generation method that performs manual crawling. This method employs an observe-select-execute cycle. The cycle enables us to automatically generate consecutive inputs for crawling which changes the states significantly while analyzing the target web application. The paper also demonstrates our method by measuring the server-side code coverage and compares with the manual crawling. As a result, our method covered 47.58% on average, which is close to the 50.21% manual coverage.

Keywords: Web applications, Automated input generation, Event driven, Dynamic analysis

1. Introduction

Nowadays, with the rapid growth of the Internet, research and development targeting various web applications have become prevalent. There is works that researchers and developers must manually provide input to the web applications. The most typical example is a testing. During testing, the tester provides input to the web application and checks whether the resulting behavior is correct. Another example is crawling. The crawling mentioned here is to repeat an action normally performed by the user on the web page, such as a transition from a web page occurring in the same web application or between different web applications, by clicking with a mouse or moving the mouse pointer. Crawling is executed to confirm the behavior of various web applications currently used, and using various data that can be acquired by performing the behavior. For these tasks, it is necessary to construct an appropriate input sequence according to each purpose. This takes much time and effort. We are required to accomplish this manually. Therefore, the efficiency of research and development deteriorates severely.

The technology required to solve the above problem is automated input generation and automatic execution. This paper proposes an automated input generation method specialized for crawling. Various development studies related to crawling have been published [1], [2]. These crawling systems repeat page transitions and are used to acquire various data that the user needs. However, automatic crawling with these tools can not observe dynamic changes in a page, and can not acquire various data owing to the changes. Many current web applications include functions to dynamically make changes within a page [3]. Not supporting these functions makes the quality of data that can be acquired lower. To generate such a change, it is still necessary to perform crawling manually. In this paper, to reduce the time for this manual crawling, we construct an automated input generation system specialized for crawling.

One of the problems in constructing automated input generation specialized for crawling is that it seldom know all the information of the target web application for testing. The information is as follows: the structure of the web application, what type of input is received, what kind of behavior can be expected, etc. From this small amount of information, it is necessary to select inputs that can perform the behavior of crawling while specifying the structure of the web application.

We employ an observe-select-execute cycle [4] as the framework of the automated generation system. In the observer stage, the system observes the current state of the target web application and specifies what kind of input is received. In the selector stage, one input is selected from the specified input group. In the executor stage, the system practically executes one selected input on the web application. By repeating this cycle, it is possible to automatically generate serial inputs for crawling which changes the states significantly while analyzing the target web application.

The rest of the paper is organized as follows. Section 2 introduces related works dealing with the automated input generation system. Section 3 describes challenges associ-
2. Related Works

This section describes related works that automatically generate inputs for other languages in Section 2.1, and for web applications in Section 2.2.

2.1 For Other Languages

Dynodroid is an automated input generation system for Android applications. Dynodroid employs an observe-select-execute cycle. In the observer stage, Dynodroid analyzes the state of the phone for two types of events, UI events and system events, and specifies what kind of event is expected. In the selector stage, one event is selected (using a selection algorithm called BiasedRandom) from the expected events specified by the observer. In the executor stage, Dynodroid executes one event selected by selector, and reflects the state of the result executed on the phone. Dynodroid repeats this cycle and enables the automated generation of consecutive inputs. Our system also uses this observe-select-execute cycle and the BiasedRandom algorithm.

2.2 For Web Applications

Kudzu [5] and Jalangi [6] are tools that apply symbolic execution [7] to JavaScript. Symbolic execution is a technique of evaluating a result by treating variables in the program as symbols and simulating the program. Kudzu automatically generates inputs to analyze security vulnerabilities in web applications. It applies symbolic execution to JavaScript programs, guesses what kind of inputs the program accepts, estimate what kind of output can be expected, and execute the input. Jalangi performs symbolic execution for the dynamic analysis of JavaScript programs. Artemis [8] and JSEFT [9] are tools that apply feedback testing to JavaScript applications. These tools employ the process of generating inputs and determine how to generate the next input by feeding back the information gathered during the execution.

To summarize, there are some research studies dealing with automated input generation for web applications. Although Kudzu and Jalangi are only for use with JavaScript, and inappropriate for the entire web application. In addition, these related works mainly aim at testing or program analysis, and are not specialized for crawling.

3. Challenges

3.1 Input in Web Applications

First, for the purpose of automated input generation for web crawling, it is necessary to identify the types of input that web applications can receive. We must take into account both of HTML [10] and JavaScript [11] to identify inputs. This section describes each type of input and selects corresponding inputs based on statistics.

3.1.1 HTML

HTML is a language that describes the document structure of a web application as elements in the form of tags. The document structure described by HTML is called Document Object Model (DOM) [12]. Tags have various roles for each type, and each can be described regarding HTML specifications. Several tags receive user inputs, and the received inputs and resulting behaviors are determined by the specification. The most used tag for crawling is the anchor tag. Since the anchor tag fulfills the role of a link for performing behaviors such as transitions of a page, it must be considered for the purpose of crawling. In addition, there are input tags and button tags described in form tags. The form tag describes an input and transmission form, and receives string inputs by the user, on/off inputs from a button, etc.

There are several tags that receive inputs, but the tag practically used is biased. Figure 1 shows the statistics of HTML tags used in the top 100 sites of the website rankings of traffic (site visitors) published by Alexa. Tags with numbers less than 1000 are omitted. From the figure, it can be seen that anchor tags are most frequently used among tags receiving input. There are button tags which that also receive inputs, but they are fewer in number when compared with anchor tags. Therefore, automated generation dealing with anchor tags can perform sufficient behavior.
behavior of web applications. JavaScript employs an event-driven programming model, which handles operations executed by users and other programs as events and processes them accordingly. In JavaScript, inputs are received as various events, and the corresponding processes are performed. Events include clicks and scrolls performed by the user. The event listener decides what kind of behavior will occur as a result of the events that occur as inputs. Event listeners are described registering in HTML elements. For example, if a click event listener is registered in an element and the user clicks the element, the behavior determined by the event listener occurs.

Event listeners in JavaScript are rich in type. In some cases, event listeners of the web application are newly created, so it is difficult to deal with all types of event listeners when building an automated generation system. Therefore, it is necessary to select the type of the corresponding event listener.

Figure 2 shows the statistics of the event listeners used for the same sites as in Figure 1. The figure shows the most used event listeners in the current web application. Event listeners whose number is 100 or less are omitted.

First, the click event listener is used much more than other event listeners. A click is the most general input to be received from the user. Users can easily speculate on the behaviors that will occur by clicking on the element and by changes in the shape of the mouse cursor.

After the click, the mousedown event listener, mouseover event listener, and mouseleave event listener are the most used event listeners. Mousedown is an event fired when the mouse is pressed, mouseover is an event fired when the mouse moves onto an element on the page, and mouseout is an event fired when the mouse moves off elements on the page. Since it is easy to assume a sequence of actions that mouse moves onto and off an element, mouseover and mouseout are often used as one set. As for mousedown, mouseup events fired when a pressed mouse is released are often used as one set. The figure shows they are used much more than other event listeners not as often as clicks, and event listeners which we should deal with.

These event listeners are “input by a mouse” considered to be most used by the user for the purpose of crawling. Statistics indicate they account for the majority of used event listeners as the result of statistics shows. Therefore, automated generation dealing with these event listeners can perform sufficient behavior.

3.1.3 Ratio of Anchor Tag to Event Listeners

It can be seen from Figure 1 and 2 that the total number of event listeners is about 10000, while the number of anchor tags is as large as about 23000. Therefore, as a feature of the type of input, there is a bias in the ratio between the anchor tags and the event listeners.

3.2 Requirement Analysis for Crawling

This section analyzes the properties that automated input generation for crawling should satisfy.

Figure 3 shows the requirement analysis for crawling. There are four major requirements for crawling:

- Page transition
- Checking behavior on pages
- Not repeating the same behavior
- Execution on the client side

**Page Transition, Checking Behavior on Pages.** The page transition is the main behavior when crawling. By repeating the page transition, information on new pages is obtained one after another. Checking behavior on pages is also a property that crawling should satisfy. Although it is not done to the extent of testing and debugging in a web application, some degree of checking behavior is
also accomplished during crawling. It is a basic process in crawling to perform some degree of behavior check in the same page while mainly making a page transition.

**Not Repeating the Same Behavior.** In normal crawling, it is rare to repeat the same behavior. This is because the information that can be acquired by repeating the same behavior is not changed, and as a result, there is a high possibility that it will become meaningless.

**Execution on the Client Side.** When crawling is done manually, it is basically accomplished on the client side. In many cases, it is not possible to access the source code of the web application to be crawled, and it is required to be executed on the client side through a browser in order to target all web applications.

### 3.3 Challenges

Possible problems in order that the automated input generation system satisfies the requirements described in Section 3.2 are the following four challenges:

- Responding to dynamic state change
- Specifying of input space and type
- Avoid generating redundant inputs
- Method of generating inputs

**Responding to Dynamic State Change.** The state of the web application is changed dynamically by the page transition or the behavior on the page. The state of the currently open page changes owing to the page transition, and the DOM changes owing to the behavior on the page. This requires a method that can deal with these dynamic state changes.

**Specifying of Input Space and Type.** In order to automatically generate an input that causes a transition of a page or an action on a page, it is necessary to specify the input space and the input type that confirm some output (behavior). Unless this specific problem of the input space and type is solved, there are some cases in which a transition of the page or a dynamic change on the page does not occur even when the input is generated and executed. Therefore, there is a possibility that the effect by the automated generation is deteriorated.

**Avoid Generating Redundant Inputs.** If the input to be generated overlaps multiple times, the behavior obtained as a result of the input will be also redundant. In order to satisfy the property of not repeating the same behavior, it is necessary to minimize the duplication of inputs to be generated.

**Method of Generating Inputs.** In manual crawling, the user performs a behavior such as a page transition by clicking the link using the mouse. However, automated crawling cannot use the mouse. In addition, as mentioned in Section 3.2, inputs must be executed on the client side. There are problems such as how to generate the input, execute it on the client side, and reflect it as behavior in the web application.

---

### 4. Approach

This section describes our solutions to the challenges described in Section 3.

#### 4.1 Overview

We propose an automated generation method that supports dynamic state change, such as page transitions and structural changes of the DOM. Our method refers to the **observe-select-execute** cycle employed in Dynodroid [4] which is an automated input generation system for Android applications. Dynodroid observes which input is related in the current state of the Android terminal (**observer**), selects one event from the inputs obtained as a result of observation (**selector**), and reflects the selected input to the terminal (**executor**). By employing this cycle in our method, even if the web application to be crawled changes dynamically, it can be dealt with.

Figure 4 shows an overview of the automated generation method. The role of the **observer** in our method is to specify the input space and type shown in Section 4.2. Our method observes the state of the current web application from DOM and uses the command line API of the Developer Tools described later. The role of the **selector** is to execute the input selection algorithm described in Section 4.3. Our method selects one input from the list obtained by specifying the input space and type by using the algorithm. The role of the **executor** is to execute the Developer Tools. Our method executes an input on the Developer Tools and reflects the state in Google Chrome that opens the target web application. Details are discussed in Section 4.4.

#### 4.2 Observer

The **observer** specifies the input space and type of the current web application. First, it is necessary to consider specifically the input space in the web application. As described in Section 3.1.1, the document structure (DOM) of the web application is described as HTML elements. This element can be individually specified, and specifying the input space in the web application can be rephrased as specifying the element that can receive the input. It is necessary to make a list of the input space and type in the **observer**. This section describes the approach for the HTML tag (Section 4.2.1) and event listener (Section 4.2.2).
4.2.1 HTML Tag

As mentioned in Section 3.1.1, an anchor tag is necessary for the purpose of crawling. Therefore, all anchor tags must be specified in the DOM. To solve this problem, our method employs the HTML API, `getElementsByTagName(name)`. This function takes the name of the tag and returns all elements of the specified tag name in the DOM. By passing anchor tag in this function, all elements of the anchor tag are given. Regarding the type of input, inputs that the HTML tag can receive are observed by writing with JavaScript. Several APIs available only in the Developer Tools, it is possible to debug the behavior in that element. The `getEventListeners(element)` in them. This function takes an element and returns all of the event listeners registered in that element. The `observer` applies this function to all elements on the current page to identify the space and type of the event listener.

4.2.2 Event Listener

Event listeners are difficult to specify the input space in the DOM like HTML tags. This is because of the existence of various libraries in the current JavaScript, each of which has a complicated form. To solve this problem, our method employs “Developer Tools” that are functions of Google’s web browser “Google Chrome.” In the command line of the Developer Tools, it is possible to debug the behavior by writing with JavaScript. Several APIs available only in the Developer Tools are provided [13]. There is a function `getEventListeners(element)` in them. This function takes an element and returns all of the event listeners registered in that element. The `observer` applies this function to all elements on the current page to identify the space and type of the event listener.

4.3 Selector

The `selector` selects an input from the list of inputs the `observer` makes. In order to avoid generating redundant inputs, the `selector` employs a selection algorithm. This selection algorithm is called once per cycle. Our method refers to the BiasedRandom algorithm employed in Dynodroid [4].

4.3.1 BiasedRandom Algorithm

Algorithm 1 shows an input selection algorithm using BiasedRandom. In this algorithm, the selection number of each input is recorded as a map, and inputs that are not much selected based on the number of selections are selected. G(i, u) is the number of times an input i of the page whose URL is u was selected. Since the global map G exists for each page, it gets the URL to use the map of the current page.

In lines 3-7, the algorithm adds inputs that are in the input list I and not in G(u). The initial setting function used for adding is `init_score` on lines 22-30. The initial selection times of the anchor tag is 1, and the event listener are 0. The reason for this will be described in Section 4.3.2.

L is a local map used only in one cycle. The initial selection number is 0. In lines 9-21, algorithm selects an input using a comparison between L and G. In line 10, an input i_s is chosen at random from I. In lines 12-16, if i_s satisfies L(i_s) = G(i_s, u), it is selected as an input by adding the selection number of i_s. In lines 17-19, selection of i_s is avoided because the input has been selected frequently in previous cycles. Instead, the algorithm increases the selection number of i_s in the local map and the chance of selecting i_s in the current cycle. This process is repeated until an input is selected. As a result, it is difficult to select inputs having a large number of selections in past cycles, and those with a small number of selections are more likely to be selected.

Algorithm 1 Input selection algorithm using BiasedRandom.

<table>
<thead>
<tr>
<th>Require:</th>
<th>List I of inputs, URL u</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ensure:</td>
<td>An input in I</td>
</tr>
<tr>
<td>1:</td>
<td>i := an input</td>
</tr>
<tr>
<td>2:</td>
<td>G := global map</td>
</tr>
<tr>
<td>3:</td>
<td>for each input in I do</td>
</tr>
<tr>
<td>4:</td>
<td>if G(i, u) is not existed then</td>
</tr>
<tr>
<td>5:</td>
<td>G(i, u) := init_score(i)</td>
</tr>
<tr>
<td>6:</td>
<td>end if</td>
</tr>
<tr>
<td>7:</td>
<td>end for</td>
</tr>
<tr>
<td>8:</td>
<td>L := init local map</td>
</tr>
<tr>
<td>9:</td>
<td>while true do</td>
</tr>
<tr>
<td>10:</td>
<td>i_s := an input selected at random from I</td>
</tr>
<tr>
<td>11:</td>
<td>//Choose an input at random</td>
</tr>
<tr>
<td>12:</td>
<td>if L(i_s) = G(i_s, u) then</td>
</tr>
<tr>
<td>13:</td>
<td>G(i_s, u) := G(i_s, u) + 1</td>
</tr>
<tr>
<td>14:</td>
<td>//Select i_s this time, but decrease chance of</td>
</tr>
<tr>
<td>15:</td>
<td>//selecting i_s after this cycle.</td>
</tr>
<tr>
<td>16:</td>
<td>return i_s</td>
</tr>
<tr>
<td>17:</td>
<td>else</td>
</tr>
<tr>
<td>18:</td>
<td>L(i_s) := L(i_s) + 1</td>
</tr>
<tr>
<td>19:</td>
<td>//Increase chance of selecting i_s in this cycle.</td>
</tr>
<tr>
<td>20:</td>
<td>end if</td>
</tr>
<tr>
<td>21:</td>
<td>end while</td>
</tr>
<tr>
<td>22:</td>
<td>procedure init_score(i)</td>
</tr>
<tr>
<td>23:</td>
<td>if i = anchor tag then</td>
</tr>
<tr>
<td>24:</td>
<td>return 1</td>
</tr>
<tr>
<td>25:</td>
<td>else</td>
</tr>
<tr>
<td>26:</td>
<td>return 0</td>
</tr>
<tr>
<td>27:</td>
<td>end if</td>
</tr>
<tr>
<td>28:</td>
<td>end procedure</td>
</tr>
</tbody>
</table>

4.3.2 Preventing Bias of Input Types

As mentioned in Section 3.1.3, the ratio of anchor tags is much larger than that of the event listeners. As a result, it is highly probable that the anchor tag is removed when an input is randomly chosen from the list, and it is assumed that the anchor tag is easily selected as an input.

Since anchor tags are more likely to be selected, it is conceivable that the number of page transitions becomes
very large. However, since crawling requires not only simple page transitions but also various behaviors on the same page, it is necessary to make a certain number of dynamic changes by the event listeners. Therefore, by setting the initial selection number of the anchor tag to 1 and the event listener to 0, the anchor tag is never actually selected as an input unless it has been randomly chosen at least once. As a result, this intentionally makes anchor tags difficult to select.

### 4.4 Executor

The **executor** executes an input selected by the **selector**. This section describes our method of generating inputs for two types of input: JavaScript and HTML.

In JavaScript, actions generated by the user during manual input generation are handled as events. Therefore, the **executor** generates this event directly.

```javascript
function mouseevent(type, element){
    var event = document.createEvent("MouseEvents");
    event.initMouseEvnt(type);
    element.dispatchEvent(event);
}
```

Listing 1: Generating events

Listing 1 shows the JavaScript code for generating events. Since all events handled in our method are generated by the mouse, the function name is `mouseevent`. This function takes the type of mouseevent and the element to be executed. By passing "Mouseevents" in `createEvent`, an event is generated on line 2. By passing the type of mouseevent in `initMouseEvent`, the event is initialized as the specified mouseevent on line 3. By passing the event in `dispatchEvent` on the element, the input is executed on line 4.

Regarding HTML, our method deals with the anchor tag, and the received input is a click. Therefore, by passing the click and the target anchor tag in `mouseevent`, it is possible to go to the link specified by the anchor tag.

By executing these command in the Developer Tools, the **executor** can generate a selected input on the client side.

### 5. Evaluation

This section describes an evaluation of the automated generation method. We used code coverage measurement of server-side programs for evaluation. The code coverage is the ratio executed in the program test. The higher the code coverage, the closer the test covers the behavior of the target program and the higher the reliability. During crawling, code coverage is also an indicator of how much of the crawling operation is performed. There are several types of code coverage. We used line coverage in this evaluation. Line coverage shows the ratio of executed lines for executable lines in the program.

<table>
<thead>
<tr>
<th>CMS</th>
<th>initial</th>
<th>manual</th>
<th>automated</th>
</tr>
</thead>
<tbody>
<tr>
<td>baserCMS</td>
<td>27.11</td>
<td>50.31</td>
<td>49.39</td>
</tr>
<tr>
<td>concrete5</td>
<td>29.73</td>
<td>33.68</td>
<td>32.84</td>
</tr>
<tr>
<td>Drupal</td>
<td>49.23</td>
<td>68.19</td>
<td>66.38</td>
</tr>
<tr>
<td>EC-CUBE</td>
<td>43.10</td>
<td>50.89</td>
<td>49.46</td>
</tr>
<tr>
<td>Joomla!</td>
<td>38.19</td>
<td>50.70</td>
<td>40.48</td>
</tr>
<tr>
<td>MODX</td>
<td>32.97</td>
<td>48.06</td>
<td>47.33</td>
</tr>
<tr>
<td>PrestaShop</td>
<td>34.33</td>
<td>39.90</td>
<td>41.83</td>
</tr>
<tr>
<td>WordPress</td>
<td>52.71</td>
<td>59.96</td>
<td>52.89</td>
</tr>
</tbody>
</table>

**Average** 29.84 50.21 47.58

### 5.1 Environment

The target for code coverage measurement is a server-side PHP program. PHP is a language that dynamically generates files such as HTML and JavaScript, and can generate pages in web applications. In other words, the code coverage of the PHP program is directly related to the rate at which page transitions and dynamic state changes are made.

We used open-source content management system (CMSs) for the server-side web application. Open-source CMSs usually have templates that are already available when installing them on the server, and we used this template. The target web applications are eight CMSs.

As a method of measuring the code coverage, we used Xdebug [14], the code coverage measurement function installed in PHP’s debugging extension module. It monitors the PHP file of the target CMS and measures the code coverage of the executed file.

As a subject of comparison of code coverage, manual crawling by a human was executed. By comparing the code coverage by manual crawling and automated generation, we evaluated how often the automated generation executes the input expected for manual crawling. Both manual and automated generation were executed on the client side using Google Chrome. First, both opened the URL of the top page of the CMS as the initial state. Then, manual crawling was executed for a sufficient time, as many page transitions as possible were completed, and behavior other than a page transition was produced at least once in the same page. Automated generation was given 100 as the number of inputs. Finally, we measured the coverage at the time when each generation was completed.

### 5.2 Evaluation Result

Table 1 shows the result, which indicates that the average of manual crawling is 50.21% and that of automated generation is 47.58%. The difference is 2.63%, which indicates that the quality of the automated generation is very close to that of manual crawling.
5.3 Discussion
Although the result indicates that the coverages for manual and automated generation are close, they can cover only 50% of the lines. This result occurs because the PHP program includes the description of not only page generation and dynamic changes of the state, but also the operations of various databases. In addition, there are relatively many lines that the crawling cannot execute. Regarding the comparison with the initial state, both manual and automated generation cover 20% more lines than the initial state. This indicates that both execute the crawling sufficiently.

6. Conclusion and Future Works
We presented an automated input generation method for web crawling. Our method employs an observe-select-execute cycle, and automated generation is repeated with the sequence of specifying the input space and type, selecting a non-redundant input, and executing an input. We applied our method to eight open-source CMSs, and compared them with manual crawling by code coverage measurement on the server side. As a result of this evaluation, the average coverage of the automated generation system is 47.58%, which is very close to the average coverage of manual crawling (50.21%). This shows the usefulness of our system.

As challenges for the future, we plan to deal with all types of input. Regarding the event listener statistics shown in Section 4.3, we can deal with higher used event listeners, but we cannot deal with other event listeners. Dealing with all types of input increases the quality of the automated generation.

References