A Hybrid Coverage Criterion for Dynamic Web Testing

Yunxiao Zou\textsuperscript{1}, Chunrong Fang\textsuperscript{1}, Zhenyu Chen\textsuperscript{1*}, Xiaofang Zhang\textsuperscript{2} and Zhihong Zhao\textsuperscript{1}

\textsuperscript{1}State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
\textsuperscript{2}School of Computer Science and Technology, Soochow University, Suzhou, China
*Email: zychen@software.nju.edu.cn

Abstract

Testing criterion is a fundamental topic of software testing. A criterion is important to evaluate and drive a testing method. Code coverage is widely used in software testing, due to its simple implementation and effectiveness. Dynamic web techniques have been used to improve the usability and user experience of applications. However, it brings some new challenges for testing. Dynamic web applications have richer iterations between client-side and server-side, such that code coverage is difficult to capture these complex iterations for sufficient testing.

In this paper, we present a novel coverage criterion - hybrid coverage. A hybrid coverage criterion which combines statement coverage and HTML element coverage, covers both client-side and server-side features. The experimental result shows that the hybrid coverage can detect 22.2%-48.1% more bugs than statement coverage, 7.9%-57.1% more bugs than element coverage.

Keyword: Dynamic Web, Coverage Criteria, Web Crawler, Test Case Prioritization

1 Introduction

Web applications have been one of the most widely used types of software. And a new technology, acting as the core of the web 2.0, is AJAX (Asynchronous JavaScript and XML) which has been changing the way of developing web applications significantly. It can improve user interactivity and achieve rich user experience like the desktop GUI applications. On the other hand, Web applications should be assured of and maintained with high usability, availability, and reliability to build user loyalty.

Testing is a major way to ensure the quality of software. There are many existing testing methods for different applications, including web applications. In order to evaluate and drive testing methods, some coverage criteria are introduced to calculate the percentage of program elements that the test sets exercise. Statement coverage which is a kind of code coverage is widely used because of its ease of implementation and its low overhead on the execution of the program under test. However, statement coverage is not sufficiently capable to cover rich features of web 2.0 enabled web applications. First, a large number of client-side functionality and content in web applications that are generated dynamically at the server-side code (e.g., PHP). The same code can produce many different web contents via combining different server states, session variables and other data. Statement coverage may be failed to distinguish test cases executing the same pieces of server-side code. Second, the new AJAX technology, which is the combination of JavaScript and DOM tree manipulation with asynchronous server communications enables powerful client-side execution capabilities. Server-side statement coverage is not appropriate to cover these rich features on client-side and interactions between server-side and client-side.

For web applications, the basic unit of displaying contents and receiving user input events is standard HTML elements (in this paper, HTML element may be abbreviated as element). A majority of user interactions with the web applications are interactions with the event-able HTML elements. These front-end elements can be suitable indicators for testing coverage. The structure of the web application user interface (UI) which contains HTML elements can be created by web crawlers. Web crawlers navigate the application under test which can automatically explore its structure and collect the HTML elements in the pages. However, coverage criterion merely based on HTML elements has its own shortages. First, it lacks execution information of server-side scripts, which is an important runtime feature for dynamic web applications. Second, the current web crawlers may miss a part of the contents of web application which may abate the practical usefulness of element coverage.

Intuitively, the coverage based on code covers the runtime features of server-side execution, and the coverage based on element covers the runtime features of client-side execution. Each of them measures one dimension of the web application. A hybrid coverage based on both of them
can measure multiple dimensions of web application features. In this paper, we present a novel hybrid coverage for dynamic web applications. It combines statement coverage and element coverage to represent both server-side and client-side coverage conditions of the test sets to verify whether the web applications are adequately tested. The UI (user interface) structure will be created by web crawler, and the HTML elements and code points exercised by the test sets are collected and are used to calculate the hybrid coverage. The experiment result shows hybrid coverage can drive a more effective testing.

The main contributions of this paper are as follows.

1. A novel coverage criterion: hybrid coverage which counts both the code executed and the HTML elements exercised by the test sets which represents both server-side and client-side conditions.

2. An empirical study is conducted to validate the effectiveness of hybrid coverage. Two web applications are used to compare hybrid coverage with coverage criteria such as statement coverage and element coverage. The result shows that our approach can drive a more effective testing for dynamic web applications.

The rest of the paper is organized as follows. Section 2 defines hybrid coverage and element coverage. Section 3 proposes the detail of our approach, including methods of calculating the element and hybrid coverage. The experiment design and the result analysis are presented in Section 4. We discuss the related work in Section 5. Section 6 are the conclusion and future work.

2 Hybrid Coverage

In this section, we present a description of HTML element, and the definitions of element coverage and hybrid coverage.

2.1 HTML Element

A HTML element is an individual component of an HTML document which is the basic block for the front-end content of web applications. Each element can have specified attributes, and content. There are three kinds of HTML elements: void elements, raw text elements, and normal elements.

Void elements only have a start tag, which contain any attributes. Raw text elements are constructed with: (1) A start tag marking the beginning of an element, which may incorporate any number of attributes. (2) Some amount of text content, but no elements. (3) An end tag, in which the element name is prefixed with a slash. In some versions of HTML, the end tag is optional for some elements. The end tag is required in XHTML. Normal elements usually have both a start tag and an end tag, and contents. The content can be nested element(s) or a combination of element(s) and text content.

2.2 Element Coverage

An underlying model to represent the structure of web application is essential for element coverage calculation. According to [II], the front-end structure of AJAX-based web application can be represented as state-flow model. In this paper, the UI model of web application is defined as follows:

Definition 1 (web front-end state graph) A web front-end state graph $G$ for a dynamic web application $A$ is a 3-tuple $(i, V, E)$ where:

- $i$ is the initial state (usually the index entry for the web application), after the completion of loading of $A$.
- $V$ is a set of states representing runtime DOM tree conditions, for the web application $A$.
- $E \subseteq V \times V$ is a set of directed edges between vertices. Each edge $(v_1, v_2)$ exists if and only if $v_2$ can be reached from $v_1$ by performing an event on a HTML element $e$ in state $v_1$.

The element set contains all the elements of a web application. In our method, the set is obtained from the web UI model. It is defined as follows: element set $E$ is a set of HTML elements. Each element $e \in E$ exists if and only if there is at least one state $v \in V$ of $G$, the DOM tree structure of $v$ contains $e$. Element coverage is defined as follows:

Definition 2 (element coverage) A set $T$ of test cases satisfies the element coverage criterion if and only if for each element $e \in E$, there is at least one test case $t \in T$ such that $e$ is accessed by $t$, where $E$ denotes the element set of application under test.

In the definition, an element is accessed by a test case means that the element is displayed or receives event(s) on the browser during the execution of the test case.

2.3 Hybrid Coverage

The key idea of hybrid coverage is to combine the runtime client-side and server-side features of the web application. The server-side features can be indicated as the condition of the code execution of server scripts. While the client-side features lie in the twisted states of JavaScript execution and DOM tree condition. The DOM tree condition...
can be represented by its elements, and JavaScript execution is often triggered by element events. The interaction of server-side and client-side can be also represented by the accessed elements.

Based on statement coverage and element coverage, hybrid coverage can be defined as follows:

**Definition 3 (hybrid coverage)** A set $T$ of test cases satisfies the hybrid coverage criterion if and only if for each element $e \in E$, there is at least one test case $t \in T$ such that $e$ is accessed by $t$, and for each statement $s \in S$, there is at least one test case $t \in T$ such that $s$ is executed by $t$, where $E$ denotes the element set, and $S$ denotes the statement set of application under test.

The statement set in the definition contains all the statements of server-side scripts of the application under test.

### 3 Approach

Our hybrid coverage is built on statement coverage and element coverage. In the current state, there are mature tools to calculate statement coverage for web applications. However, to our knowledge, there are no ready-to-use tools to obtain element coverage data for web application under test.

We implement a framework to obtain the element coverage of test case(s). In the framework, a web crawler dynamically navigates on the web application under test to generate a web UI model to represent the front-end structure of the web application. On the other hand, test case(s) are executed and specific test data are collected at runtime. And the element coverage information are calculated from the web UI model and the collected data.

Web crawling is a widely used method to rip the contents of web application, but the advent of web 2.0 technologies brings new challenges to traditional web crawlers. We use Crawljax as the model generation tool for the web application under test, because, to the best of our knowledge, it is the only currently available tool to support AJAX-based web sites.

We utilize the Crawljax to generate the front-end model with the default configuration. The data (e.g., user name and password) needed to access hidden web content are provided to the crawler. The output of the crawling process is a state graph with runtime DOM tree content contained in state vertices.

Having got the web front-end model in the previous step, the framework performs the coverage calculation of the test suite using the web UI model and the test data collected in the test automation runtime environment. For collecting test data, the framework runs the test cases in the test suite on the web application and collects data from the DOM tree in the browser after executing each test command in the test case. Our technique records three attributes (id, class and tag name) and XPATH value of the HTML element(s) which receive(s) event(s) from each test command, and it also records the same kinds of attributes and XPATH for the all the HTML elements displayed in the DOM tree after executing each test command. The data collected are used for calculating the coverage information for each test case.

The data collected in the previous step is used for identifying the accessed HTML elements in the web UI model and computing the element coverage. There’s an issue of matching between the HTML elements collected in the previous step and in the web UI model. The tracked attributes and XPATH are used for matching. Each of above properties has to be exactly matched. If there are multiple matches in the model, the first match is selected. After matching the HTML elements between the elements in the collected data and in the web UI model, the set of accessed HTML elements of test case(s) can be obtained.

For statement coverage, we utilize a combination of the Xdebug extension for PHP and PHPUnit. The tool chain can generate test report which contains statement coverage information. Hybrid coverage can be easily computed with element and statement coverage information.

### 4 Experiment

To explore the effectiveness of different coverage criteria, a test case prioritization experiment is conducted. The aim of the experiment to select a part of test cases from the test pool which have rather high fault-detect capability. The selection strategies of test case prioritization are usually on the basis of coverage metrics. Therefore, in this section we evaluate the usefulness of different coverage criteria by comparing the fault-detect capabilities of the selected test case sets which are selected via different coverage-based strategies.

#### 4.1 Experiment Subjects

Due to the considerable manual cost to create web automation test scripts, we chose two PHP web applications as our subject programs, schoolmate and timeclock which have been studied by other researchers. Table I shows some information of these subjects, including version information and lines of code (LOC), along with number of files.

We recruited a team consisting of both senior undergraduate and graduate students to create test cases for subject

---

1. [http://crawljax.com](http://crawljax.com)
2. [http://www.xdebug.org](http://www.xdebug.org)
3. [http://www.phpunit.de](http://www.phpunit.de)
programs. These team members are currently in the research projects focusing on software analysis and testing and are highly experienced in software testing. They created test automation scripts using Selenium ⁶, according to the specification requirements. Each member of this team submitted test cases and a detected bug description documentation, along with a test report. Table I lists some information of the test cases, and the number of detected bugs.

4.2 Experimental Procedure

In this sub-section we present a description in detail of the experiment procedure. We first initialize the test pool, which contains all the test cases shown in Table I. The initial set of selected test cases is empty, and at every round of the procedure, 0 or 1 test case is selected from the test tool. For each selection, the procedure traverses the unselected test cases in the test tool, and calculates the fitness value as the selection metric for each candidate test case. The candidate test case with maximum fitness are selected. If there are multiple qualified candidate tests with same maximum fitness value, we pick up one test case randomly among them. If the coverage of the selected tests reaches the maximum coverage (That is, the coverage of the test pool), the coverage data are cleared and the procedure continues. The procedure will end if the end condition is met.

![Figure 1. Experiment Procedure](image)

The procedure will end if selected test cases detect all bugs. However, in the industrial practices, Re-executing a large number of test cases is too time-consuming in regression scenarios. Therefore, we set the maximum number of selected test cases. The procedure will end if enough test cases are selected, or all bugs are detected. For our experiment, the maximum number of selected test cases is one third the size of the test pool. The experiment procedure is shown in Figure I.

The procedure should be sampled for many times for there exists randomness in the selection phase. The guidance in [9] suggests the sampling frequency should be more than 30. Following this suggestion, the procedure will be performed 100 times for each combination of different subject programs and selection strategies.

4.3 Selection Strategies

The key of our experiment is the fitness function. Test case with maximum fitness value is chosen from the test tool at each round of the procedure. Fitness functions are coverage-based. And the executing cost of test cases is also taken into consideration, like a multi-objective function in [15]. The basic idea is that test case with high contribution of coverage points and low executing cost is selected at each round. The coverage contribution of candidate test case to the selected test case set can be measured by Additional Coverage Set (ACS), which contains coverage points which are in Coverage Set (CS(t)) of candidate test case t but not in the Coverage Set of Coverage Information (CS(Cover_Info(T))) of selected test case set T. the additional coverage set can be computed by the formula ACS(t, Cover_Info(T)) = CS(t) - CS(Cover_Info(T)). We substitute CS(Cover_Info(T)) for CS(T) because the coverage information may be cleared in the experiment procedure. The executing cost of test case is measured by the cost of initializing the test case plus the cost of executing test commands in the test cases. In our measures, the cost of test case initializing is a constant value, we set it with 10; the command-executing cost is approximated by the number of test commands in the test case. The general fitness formula is defined as follows:

\[
\text{Fitness}(t, T) = \frac{|\text{ACS}(t, \text{Cover_Info}(T))|}{\text{Cost}(t)}
\] (1)

We define the fitness formulas for the statement coverage, element coverage, and hybrid coverage, respectively. The formulas are defined according to the general formula in Formula I, with the substitution of the information and calculation of each specific coverage for the information and calculation of the general coverage.

4.4 Result And Analysis

An effective strategy can detect large number of faults with small number of test cases. That means an effective
Table 1. Subject Programs

<table>
<thead>
<tr>
<th>program</th>
<th>bugs</th>
<th>tests</th>
<th>Version</th>
<th>LOC</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>schoolmate</td>
<td>23</td>
<td>151</td>
<td>1.5.4</td>
<td>8181</td>
<td>63</td>
</tr>
<tr>
<td>timeclock</td>
<td>13</td>
<td>142</td>
<td>1.0.2</td>
<td>20789</td>
<td>62</td>
</tr>
</tbody>
</table>

strategy has high fault-detect capability with low cost. We measure this feature with fault-detect rate for each combination of subject programs and selection strategies. For each sample of test prioritization, we record the fault-detect rate, which means the number of detected bugs at any given number of select test cases. Then we calculate the average rate of all 100 samples to get the final results. Figure 2 and 3 illustrate the fault-detect rate of three strategies on two subject programs, schoolmate and timeclock.

Figure 2. Fault-detect Rate on Schoolmate

From the figures, We can see that the hybrid-coverage-based strategy can detect more bugs than the statement-coverage-based and element-coverage-based strategies. And whether element coverage performs better than statement coverage is dependent on the subject programs. On schoolmate, the hybrid coverage can detect 22.2% more bugs than statement coverage, 57.1% more bugs than element coverage. On timeclock, the hybrid coverage can detect 48.1% more bugs than statement coverage, and 7.9% more bugs than element coverage.

4.5 Threats to Validity

One threat to external validity is mainly due to the number of our subject programs. Because of manual work of creating the test cases, two well-studied subject programs are chosen as the experiment material. These two applications may not cover the wide spectrum of possible types of web applications.

Threats to internal validity is the possible incompleteness of the web UI model we created for element coverage calculation. Due to the complexity of web application and practical ability of web crawler, some content may not be crawled in the crawling process [1], which may affect the performance of element coverage.

5 Related Work

Many evaluation methods have been proposed for different application scenarios. Structure coverage, including branch coverage, MC/DC, etc, is one of the most commonly used criteria to guide test generation and selection [2]. In some cases, it also can be used to evaluate the effectiveness of test sets. Besides structure coverage, data-flow coverage [8] is another kind of coverage criterion, which includes all definition coverage, all use coverage, etc. However, there are many disputes about coverage criteria [12,17]. For GUI programs, Memon et al. [10] presented a set of coverage criteria based on user input events.

There are some additional coverage criteria for web applications. Most of them are page-based coverage criteria. Sampath et al. [13] presented a dynamic-generated page-level coverage criteria for web applications. In their
method, the coverage criteria are based on URLs (uniform resource locator). But AJAX-based web sites are not based on multi-page paradigm in which every page has one unique URL [11]. For instance, one of our subject programs - schoolmate only consists of only one page for the whole web application. The presented coverage criteria in [14] may not be effective on these AJAX-based web applications. Ricca and Tonella [13] developed an UML-based model for web applications. in [16] Tonella et al. extended the model to include sever pages; in their method, multiple entities are grouped for a single server-side page outcome using static analysis. Generating each possible outcome for every server pages can quickly lead to a model of impractical size. Di Lucca et al. [9] also developed a UML-based test model and tools for the evaluation and automation of testing web applications. They consider that pages of the web application as components should be tested at the unit level, but the method to generate the test model seems not well presented.

The structure of web site can be extracted dynamically by web crawlers [4–7, 11]. Web 2.0 techniques such as JavaScript and dynamic DOM tree manipulation bring challenges to crawling [4, 11]. Even for the web crawler which supports web 2.0 features, some HTML elements are missed in the generated model in our evaluation. Alshahwan et al. [11] presented crawlability metrics to quantify properties of web application that affect crawling.

6 Conclusion and Future Work

In this paper, we present novel coverage criteria for web application testing. Element coverage is used for measuring the set of HTML elements accessed by the test case(s). Based on this element coverage and the traditional statement coverage, we present a hybrid coverage to cover both client-side and server-side features. An experiment evaluates the usefulness of the hybrid coverage criterion.

Our future work is to build a web UI model with high completeness of elements. Element coverage is an useful complement to the traditional code coverage. But the possible incompleteness of the underlying model may affect the performance of element coverage. Static analysis seems another way to build the underlying model.

References


