Virtual DOM Coverage: Drive an Effective Testing for Dynamic Web Applications

Yunxiao Zou[+*], Zhenyu Chen[+], Yunhui Zheng[*], Xiangyu Zhang[*], and Zebao Gao[+-]

[+]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China

[*]Department of Computer Science, Purdue University, West Lafayette, Indiana 47907, USA

[-]Department of Computer Science, University of Maryland, College Park, Maryland 20742, USA

{mg1032016, zychen}@software.nju.edu.cn, {zheng41, xyzhang}@cs.purdue.edu, gaozebao@cs.umd.edu

ABSTRACT

Test adequacy criteria are fundamental in software testing. Among them, code coverage criterion is widely used due to its simplicity and effectiveness. However, in dynamic web application testing, merely covering server-side script code is inadequate because it neglects client-side execution, which plays an important role in triggering client-server interactions to reach important execution states. Similarly, a criterion aiming at covering the UI elements on client-side pages ignores the server-side execution, leading to insufficiency.

In this paper, we propose Virtual DOM (V-DOM) Coverage, a novel criterion, to drive web application testing. With static analysis, we first aggregate all the DOM objects that may be produced by a piece of server script to construct a V-DOM tree. The tree models execution on both the client- and server-sides such that V-DOM coverage is more effective than existing coverage criteria in web application testing. We conduct an empirical study on five real world dynamic web applications. We find that V-DOM tree can model much more DOM objects than a web crawling based technique. Test selection based on V-DOM tree criterion substantially outperforms the existing code coverage and UI element coverage, by detecting more faults.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Measurement, Verification, Reliability

ISSTA '14, July 21-25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07 ...$15.00.

Keywords

Dynamic Web, PHP, Coverage Criteria, Virtual DOM

1. INTRODUCTION

Web applications are becoming increasingly important in recent years as people more and more tend to interact with Internet to carry out their everyday activities, e.g. browsing, shopping, gaming, working and socializing. Compared to traditional desktop applications, the bar of developing and deploying web applications is relatively lower. To some extent, anyone could rent some space from an Internet Service Provider and post his/her own web applications. On one hand, this substantially facilitates the prosperity of Internet and web applications. On the other hand, it welcomes many web applications that are not sufficiently validated or tested. These applications may cause all sorts of problems, with some of them having financial impact. Web application testing is hence a very important challenge for the health of web application engineering.

Despite its importance, testing web applications poses many new challenges. First, web applications are often highly dynamic. A server-side script could generate a large number of client-side pages that may appear different and carry out different functionalities, depending on the client-side user inputs and the internal states on the server-side. AJAX (Asynchronous JavaScript and XML), a popular client-side technique, allows on-the-fly page updates by interacting with server scripts. Such page updates do not require loading a new page, but rather self-modify part of the current page (e.g. resizing a frame, repositioning a button, replacing part of the page with an image). Such updates also change both the interface and the functionality of the current client page. These dynamic features make web application testing difficult.

Recently, researchers have developed various web application testing techniques that aim to address the challenges. Particularly, S. Artzi et al. proposed a test generation technique for dynamic web applications by combining concrete and symbolic executions [8]. Their tool Apollo can automatically generate test inputs to expose faults in web applications, guided by similarity of code artifacts. G. Wassermann et al. developed a test generation technique to handle dynamic features in web applications [29]. Their technique also leverages concolic execution to maximize test coverage of certain code artifacts in server scripts, such as string values, objects, and arrays.

While the aforementioned techniques have substantially advanced web application testing, they more or less evolve from traditional software testing techniques and inherit some solutions that may become sub-optimal in the context of web application testing due to its dynamic nature. One of such sub-optimal solutions is coverage criterion, which often serves as the stop condition for testing, answering the question whether a test suite is sufficient for a subject program. It is a critical cornerstone for test selection, test prioritization, and test generation. The most popular traditional coverage criterion is code coverage, or more specifically, statement coverage, which measures the percentage of statements that are executed by a given test suite. It is simple, effective and hence widely used in many testing techniques, including many of the aforementioned web application testing techniques. However, we observe that statement coverage is sub-optimal for web application testing. Web applications are usually heterogeneous systems, including HTML pages, server-side scripts (e.g. PHP scripts), client-side scripts (e.g. JavaScript) that may be embedded in server scripts or HTML pages, and even SQL scripts sometimes. It is very difficult to measure code coverage of client-side scripts or HTML pages as they are usually dynamically generated by server scripts. As such, different user interactions may induce different client-side pages and scripts. It is unclear what would be the universal set of client-side code artifacts that an ideal test suite should cover. In contrast, computing code coverage for server scripts is a tractable problem. In fact, many existing web application testing techniques adopt such a criterion [8]. However, it is unfortunately insufficient for web application testing. In particular, one line of server-side script could induce many client-side code artifacts that may exercise different functionalities. A test case covering a server-side line could not guarantee that the corresponding client-side logic is sufficiently tested. We will show a number of such cases that we have found in practice in Section 2.

Another popular coverage criterion is based on Graphic User Interface (GUI) objects [33]. It measures test sufficiency from the perspective of an end-user. It focuses on covering the events that could be triggered by manipulating the GUI elements presented in the front-end. The corresponding testing methods often leverage crawlers to navigate a subject application and automatically explore GUI objects in the interface. This approach works best when the pages are static such that the events to cover could be identified before-hand. However, in web application testing, crawlers usually fail to crawl all the possible interfaces due to the need of having the right sequences of inputs. As such, the universal set of artifacts to cover cannot be properly computed, making measuring test sufficiency difficult.

In this paper, we propose a novel coverage criterion: Virtual Document Object Model (V-DOM) coverage. Web pages are represented as DOM trees internally by a browser. The root of a tree represents the whole document (i.e. the page). Other objects such as frames, text-boxes and buttons are organized as a tree based on their nesting structure. In some sense, we could consider that each page displayed to the client is essentially a DOM tree. A V-DOM tree is a logical aggregation of all the possible DOM trees produced by a server script. Each object in a V-DOM tree may appear in a page in reality but multiple DOM objects in the tree may not appear together in the same page.

To construct the V-DOM tree of a server script, we perform static analysis on the server-side code (e.g., PHP) to identify all the possible DOM objects that could be emitted by the server code to the page returned to the client, and organize them in a tree just like a real DOM tree. It hence denotes all the possible behaviors of the script and can serve as a reasonable universal set of artifacts to cover during testing.

The contributions of the paper are summarized as follows.

1. A novel coverage criterion, V-DOM coverage, is proposed to facilitate effective testing of dynamic web applications.

2. We develop a static analysis technique on server scripts to construct the V-DOM trees for a web application.

3. An empirical study is conducted to evaluate the effectiveness of V-DOM coverage on five real world web applications.

   The results show that our approach can model much more DOM objects than using a crawler, and V-DOM tree coverage is much more effective in driving test selection for bug detection compared to the traditional code coverage the UI element coverage.

The rest of paper is organized as follows. We motivate our technique in Section 2. We define V-DOM tree and V-DOM coverage in Section 3. We further explain the details of our approach, including how to generate a V-DOM tree and compute V-DOM coverage during execution, in Section 4. The experimental results are presented and analyzed in Section 5. We discuss the related work in Section 6 and conclude in Section 7.

2. MOTIVATION EXAMPLES

In this section, we use two simplified examples from real world web applications to motivate our approach. These two examples represent two kinds of faults residing in the client-side and the server-side, respectively. We also explain the reason why the code coverage criterion has problems in testing and discovering these faults.

2.1 Faults in the Client-Side

During web application execution, server scripts assemble small pieces of strings to produce the client pages (HTML and JavaScript) on the fly. While these dynamically generated client-side artifacts have to comply with the corresponding syntax and semantics rules, such syntax and semantics constraints are invisible to the server scripts. From their perspective, the client page snippets embedded in their code bodies are just strings. Server scripts will not validate the correctness of these strings.

Therefore, dynamically generated client pages are error-prone. For example, hyper-links denoted by "<a href =...> ... </a>" may fail because the URLs presented or dynamically generated are not correct. If the URL is static (i.e. not changeable by JavaScript based on user inputs or actions), traditional web page crawlers can easily determine if it is a broken link. In contrast, if the URL is dynamically composed on the client-side through JavaScript and some

```
<?php
  ...
1 for($i = 1; $i <= $numpages; $i++)
  {
2   if($i == $_POST["onpage"])
    {
3     print("<a href='JavaScript:  ...
4              document.parents.page2.value = 3;
5              document.parents.submit();'
6            ...  </a>");
    }
  }
  ...
?>
```

Figure 1: Simplified Server-side Script in *Schoolmate*

runtime inputs, it may be difficult to identify the problem if the interactions inducing failure are not exercised during testing.

When we inspect the origin of the hyper-link in the server script, we probably find that it is from an echo or print statement that emits a string to the output page that will be returned to the client. Therefore, it's possible that a test run correctly executes this specific print on the server-side but failed to notice that the hyper-link is broken until the link is clicked in the client interface. In other words, a test case that covers the statement on the server-side may still miss the bug if it does not follow the link on the client-side. The problem becomes prominent when the link could change its form dynamically. The essence is that *server-side code coverage may not take into consideration execution on the client-side so that client-side faults may be missed.*

The server-side script snippet demonstrated in Fig. 1 shows an example of this kind. It's taken from *Schoolmate* [1], an Management Information System (MIS) solution for schools. In *Schoolmate 1.5.4*, there is a navigation bug caused by a hyper-link with embedded JavaScript.

The for loop dynamically generates a table containing data retrieved from databases. The print statement in lines 3-6 adds a button with hyper-link in each row of the table. The result of clicking the button is to execute a piece of JavaScript specified in the href property of the <a> element, which is supposed to send a deletion request and navigate the user away to a web page belonging to the *Parent* module. Such navigation can be achieved by assigning a unique ID to document.parents.page2.value at line 4, which is "3" in the snippet. However, its behavior is buggy because it redirects the client to a page of the *Teacher* module. The reason is that the developers use a wrong module ID "3" instead of "22".

As explained above, although the patch to the bug should be applied to the constant string at line 4 in the server script, the bug is actually a part of the JavaScript code that only gets triggered on the client-side. Therefore, an ideal coverage criterion should allow selecting a test case that exercises the print statement on the server-side and triggers the specific hyper-link on the client-side. However, server-side code coverage can hardly meet the goal because it cannot model the requirement on the client-side. Note that the GUI coverage criterion [33] may not work either as the the print statement is guarded by the conditional check at line 2, meaning that it may not get executed depending on the client-side input. If so, the button is not even present on the client-side interface to have any effect on the GUI coverage.

```
<?php
  ...
1 $tzo = $_COOKIE['tzoffset'];
  ...
2 if ( !empty( $modified_when[$x] ) ) {
3   $modified_when_time = date(..., $modified_when[$x]);
4   $modified_when_date = date(..., $modified_when[$x]);
5 }
  ...
?>
```

(a) Buggy Server-side Script in *PHP Timeclock v1.02*

```
<?php
  ...
1 $tzo = $_COOKIE['tzoffset'];
  ...
2 if ( !empty( $modified_when[$x] ) ) {
3   $modified_when[$x] = $modified_when[$x] + $tzo;
4   $modified_when_time = date(..., $modified_when[$x]);
5   $modified_when_date = date(..., $modified_when[$x]);
6 }
  ...
?>
```

(b) Fixed Server-side Script in *PHP Timeclock v1.03*

```
<script language="JavaScript">
  ...
1 var tzoffset = getthecookie("tzoffset") || timezone;
2 if (...)
3   tzoffset="0";
4 setthecookie("tzoffset", tzoffset);
  ...
</script>
```

(c) JavaScript Snippet Setting Timezone in the Client-side

Figure 2: A Bug in the Server-side of *PHP Timeclock*

2.2 Faults in the Server Side

Sometimes, even faults in server-side scripts can cause problems with the server script code coverage criterion. We use an example from *PHP Timeclock* [2] to demonstrate the scenario. *PHP Timeclock* is a web-based schedule and event tracking system.

The related client-side and server-side code snippets and the fix on the server-side are presented in Fig. 2. Depending on the configuration preferred by the client, *PHP Timeclock* can display the client's local time that may be different from the server's time. This functionality is achieved as follows. Before sending a request, the client's local timezone is saved in the cookie by JavaScript at line 4 of Fig. 2(c). When the server processes the request, it first reads the client's timezone from the cookie at lines 1 of Fig. 2(a) and Fig. 2(b) so that the server is able to understand the time difference and adjust the time to be displayed accordingly.

However, the buggy implementation in Fig. 2(a) forgets to adjust the time although it does load the client's timezone. The bug is fixed by adding the statement at line 3 in Fig. 2(b).

Now consider a test case, in which the configuration is set to display the server time instead of the client's local time. The test case will execute lines 1-5 in Fig. 2(a). However, it fails to disclose the bug even it could achieve full coverage. Note that this is a missing code bug, which is known to be difficult for code coverage criteria. In contrast, a test criterion that tries to cover the various configurations on the client-side would expose the bug (on the server-side). The essence is that *server-side code coverage alone is not sufficient for detecting missing code faults.*

[1] http://sourceforge.net/projects/schoolmate/

[2] http://timeclock.sourceforge.net/

3. VIRTUAL DOM COVERAGE

Through the examples in the previous section, we observe that server-side code coverage is sub-optimal in web application testing. The root cause is that the criterion does not model the execution on the client-side such that faults in client-side pages cannot be sufficiently exposed (e.g., defective DOM objects and parameters as in the first example in Section 2). Code coverage also has difficulty detecting missing code problems, whereas certain client-side inputs could expose them. (e.g., the second example in Section 2).

To develop a test criterion that is capable of modeling execution on both the server- and the client-side, we propose the notation of Virtual DOM (V-DOM) tree. In this section, we will define V-DOM tree and V-DOM coverage.

3.1 Virtual DOM Tree

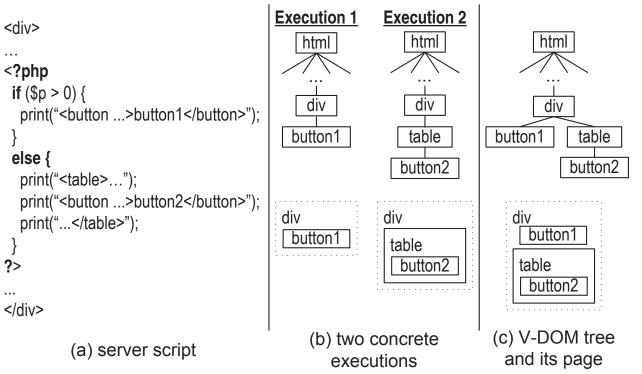
On the client-side, a page is represented as a tree of DOM (Document Object Model) objects. The interface artifacts in the page, such as frames, text-boxes, and buttons, are called the DOM objects. Note that the display of a client-side HTML page is organized in layers such that the corresponding DOM objects form a tree to reflect such layers. For example, a button DOM object is a child of a frame object if the button resides in the frame. The root of a DOM tree represents the entire page.

During web application execution, a piece of server script may generate many different client pages (or DOM trees). These pages are the results of both client-side and server-side executions. We introduce Virtual DOM (V-DOM) tree to denote the universal set of all such pages, which essentially encodes the universe of possible executions on both sides. Intuitively, V-DOM tree is similar to a real DOM tree in that the DOM objects in the tree are concrete and valid. The difference is that a V-DOM tree includes all the possible DOM objects generated by a piece of server script and these DOM objects retain the same relative positions to their containers (e.g. <div>) as in a real DOM tree generated by some execution. In some sense, one can consider a V-DOM tree the aggregation of all possible real DOM trees. The declarative definition is as follows.

Definition 1. *Given a server script P, its V-DOM tree is a tree $\langle N, E\rangle$ satisfying the following conditions.*

- *A node $n \in N$ if and only if in some execution of P, a DOM object n appears in the generated client page;*
- *An edge $n \to m \in E$ if and only if in some execution of P, the DOM object m directly resides in another DOM object n in the generated client page.*

Consider the server script in Fig. 3(a). The PHP script enclosed has two execution paths. Depending on the value of $p, the script may emit a button or a table including a button to the result client page. The results of these two concrete executions are shown in Fig. 3(b). It shows the real DOM trees and their corresponding appearances. The V-DOM tree and the corresponding web page are present in Fig. 3(c). It integrates all the DOM objects appearing in the concrete runs and retains their relative positions in real executions. Please note that *button1* and *button2* can never coexist in the outcome of any concrete execution. In this paper, we also call the HTML page corresponding to a V-DOM tree the *virtual HTML page*. A V-HTML page could be rendered by a browser as a usual HTML page.



```
<div>
...
<?php
 if ($p > 0) {
  print("<button ...>button1</button>");
 }
 else {
  print("<table>…");
  print("<button ...>button2</button>");
  print("...</table>");
 }
?>
...
</div>
```

(a) server script

(b) two concrete executions

(c) V-DOM tree and its page

Figure 3: V-DOM Tree Example

Well-formedness of V-DOM Trees. Theoretically, a DOM object n could be in different containers in different concrete executions of a server script. For instance, in one execution, a button may be emitted in a frame, while in another execution, the same button may be in a table. Although our V-DOM tree construction algorithm (Section 4) could precisely catch such situations, the resulting structure is no longer a tree, but rather a graph, which cannot be properly rendered. While we have not encountered such cases in practice, our solution is to create a clone of the DOM object in each container. We will rename each clone to avoid conflict.

Another possible violation of well-formedness is that loops may be formed. For example, in one execution, a DOM object n may be in another DOM object m whereas in another execution, m is in n. Again, we haven't seen such cases in practice. The solution is similar, cloning the objects to retain a renderable tree shape.

3.2 V-DOM Coverage

Observe that a V-DOM tree essentially models executions on both server- and client-sides. Usually, the purpose of having different execution paths on the server-side is to compose the client page in different manners. These different manners are aggregated and denoted in a V-DOM tree. Hence, covering the DOM objects in the tree implies covering paths on the server-side. On the other hand, any DOM object that a client user could interact with is represented in the V-DOM tree [3], covering objects in a V-DOM tree also means covering all the possible client-side interactions.

If a DOM object is rendered in a test run, it's a *displayed object*. We hence define the *V-DOM display coverage* as the ratio of the displayed objects by a test suite to all the objects in the V-DOM trees. On the client-side, a user can interact with a web application by operating on user controls such as buttons or typing in text-boxes. These actions can trigger events that may potentially change the execution state. We call a DOM object that has event listeners attached an *interactive object*. If a specific event of an interactive object is triggered in a test run, we say the event is covered. Therefore, *V-DOM event coverage* is the ratio of the number of the triggered events to all the trigger-able events in V-DOM trees. It measures the completeness of a test suite in terms of user interactions.

[3] We assume the V-DOM tree construction algorithm is complete at this point. We will discuss factors that affect completeness in later section.

4. MODEL GENERATION AND COVERAGE COMPUTATION

In this section, we explain how to statically construct V-DOM trees and how to measure V-DOM tree coverage at runtime. As shown in Fig. 4, the V-DOM tree construction process consists of three main steps.

1. **Preprocessing**. In this step, the web application is prepared for later static analysis. Particularly, script inclusions are handled by including the contents of the target scripts. The resulting server scripts are then translated to C code.

2. **Control-flow and Data-flow Analysis**. In this step, static control-flow and data-flow analyses are preformed on the translated C code to identify and extract all DOM objects that may be emitted by a script. The output in this step is an analysis record file.

3. **Model Generation and Coverage Computation**. In this step, the tool generates a V-DOM tree based on the program analysis result in the previous step. During test case execution, runtime data related to the coverage criteria are collected.

4.1 Preprocessing

We should not construct V-DOM tree through dynamic analysis as the resulting tree may not be complete. Therefore, we resort to static analysis. However, to the best of our knowledge, there is not a reliable static analysis framework that can directly analyze PHP scripts. Therefore, similar to other PHP static analysis projects [29], we first convert a PHP script into a piece of C code using *phc* [1] in order to leverage existing analysis engines.

The goal of using *phc* is to generate a piece of runnable C code with the same functionality as the PHP script. During the translation, it adds many auxiliary functions and data structures to handle PHP runtime features that are not supported in C. For example, in PHP, the index of an array can be of multiple types but in C it can only be integers. Hence, in the translated C program, such features are supported using hash tables. These additional libraries substantially increase the complexity of the analysis. Fortunately, we are only interested in the control-flow and data-flow of the script and do not need to run the translated program. Therefore, we modify *phc* to simplify the translations for those runtime features for the goal of static analysis. As a result, the translated C program may not be executable but it retains the control-flow and data-flow of the original PHP script.

During the translation, we unroll loops once. The intuition is that unrolling a loop multiple times only introduces multiple instances of a DOM object of the same nature (e.g., the same object type and the same event handlers). Covering these instances does not add much to fault detection capabilities. As a result, the translated C code does not have any loops.

4.2 Control-flow and Data-flow Analysis

In the second step, we leverage LLVM [2] to perform control-flow and data-flow analysis on the translated programs. Control-flow analysis is needed to construct the control flow graphs (CFGs) which will be used in V-DOM tree construction. Data-flow analysis is needed to detect data dependences, especially those that are related to the strings emitted by print statements in server scripts. Note that the essence of server script execution is to compose the client page from strings that may come from user inputs or databases. Data dependences would be helpful in figuring out the possible string values at print statements and hence the possible DOM objects.

Algorithm 1 Construct Virtual DOM Tree

Input: CFGs and data dependences of the translated program
Output: V-DOM tree.

```
 1: $vhtml \leftarrow \Phi$
 2: $outputFunction$(the CFG of main(), $\phi$)

 3: procedure OUTPUTFUNCTION(CFG: $G$, Context: $C$)
 4:    for each instruction $i \in G$ in topological order do
 5:       if $isPrint(i)$ then
 6:          for each value $v$ in $computeVar(i.operand)$) do
 7:             $vhtml.append(v)$
 8:          end for
 9:       end if
10:       if $isFunctionInvocation(i)$ then
11:          $C_f \leftarrow \Phi$
12:          for each formal parameter $p$ of $i.callee$ do
13:             $C_f \leftarrow C_f \cup \{p \mapsto$
                      $computeVar$(the actual parameter, $C$)$\}$
14:          end for
15:          $outputFunction$( the CFG of $i.callee, C_f$)
16:       end if
17:    end for
18: end procedure
```

4.3 Model Generation

Alg. 1 describes the V-DOM tree construction algorithm. Its input is the program analysis results obtained in the previous step, including the CFGs and data dependences of the translated program. Its output is a virtual HTML page, which is essentially the V-DOM tree. In the algorithm, the virtual page is initialized to empty at the beginning (line 1). It then invokes the *outputFunction()* method on the *main()* function of the translated program to generate the V-DOM tree. The method takes the CFG of a function in the translated program and an evaluation context, which contains the possible values for the parameters of the function. It traverses each statement in the function in the topological order. If a statement i is a print statement, the algorithm calls *computeVar()* to compute the possible values of the string operand of the print statement and emit such values to the virtual page sequentially (lines 5-8). As such, the corresponding possible DOM objects are arranged next to each other in the same container. Note that the topological order is important, it ensures the emission of the elements in the virtual page must follow the order of some real execution. For example, it ensures an end HTML tag must be emitted after the corresponding start tag. If the statement is a function invocation, it recursively calls *outputFunction()* on the callee function, with a context that contains the possible values for each formal parameter of the callee function (lines 10-16). Particularly, line 13 computes the possible values of a formal parameter, which are acquired through *computeVar()* on the corresponding actual parameter. This essentially makes our analysis context sensitive.

Alg. 2 defines the *computeVar()* method that computes the set of possible values of a variable, given an evaluation context. At line 3, the set of data dependences of the vari-

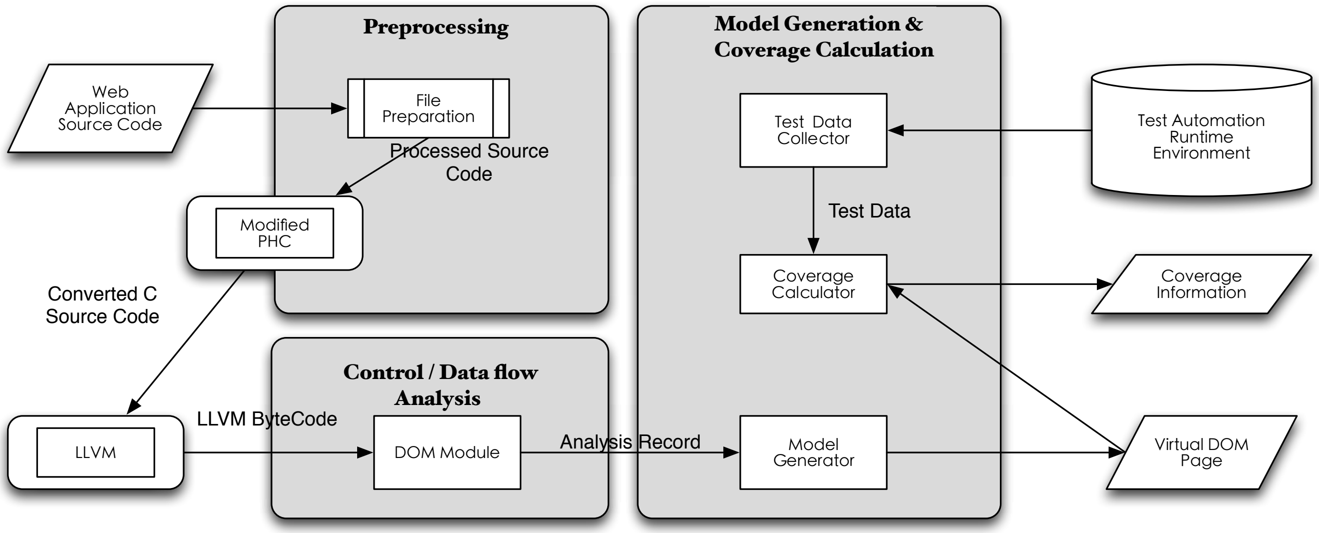


Figure 4: V-DOM Construction Overview. The gray boxes are the three main steps. The rounded white rectangles with a smaller box inside are the tools we used. The white rectangles are the intermediate data or analysis results.

Algorithm 2 Static Variable Approximation

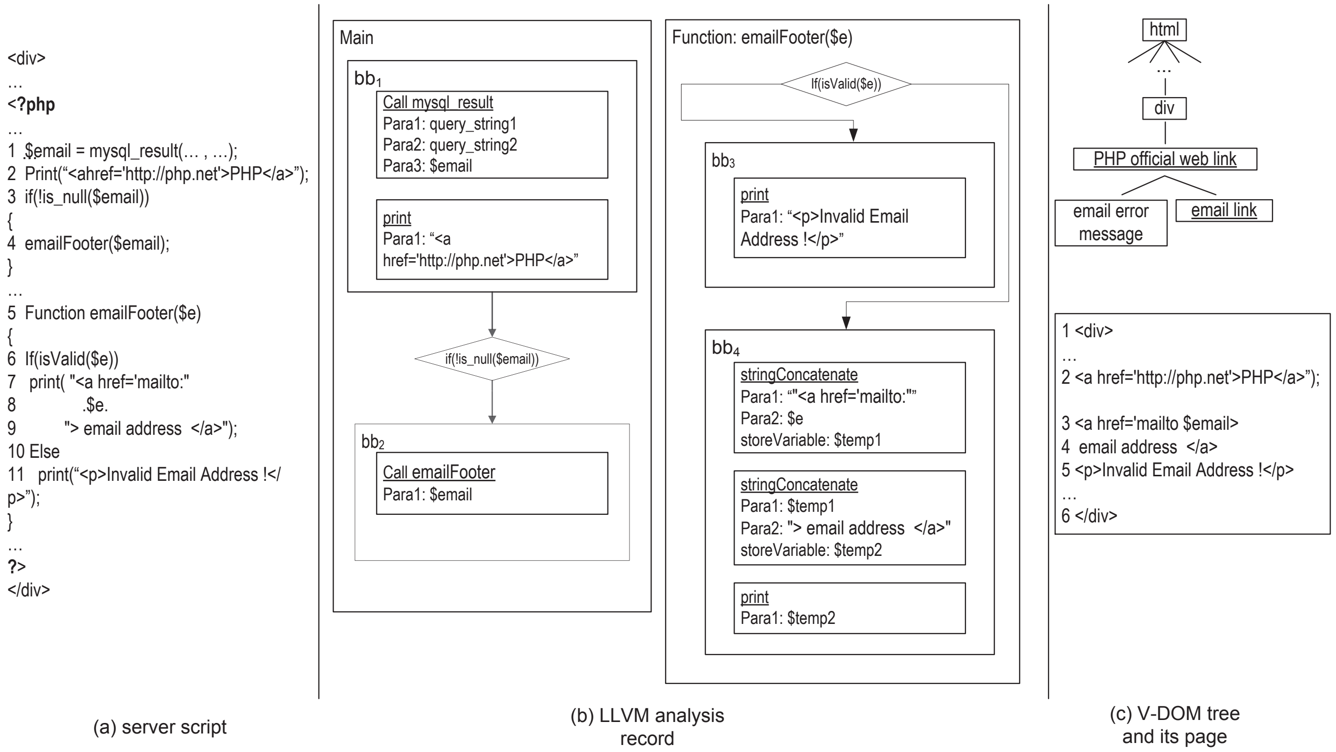
Input: A variable and the evaluation context
Output: A set of values.

```
 1: function COMPUTEVAR(Variable: $x$, Context: $C$ )
 2:   $vSet \leftarrow \Phi$
 3:   $dataDepSet \leftarrow getDirectDependance(x)$
 4:   for each $dep \in dataDepSet$ do
 5:     if $isConstant(dep)$ then
 6:       $vSet \leftarrow vSet \cup \{dep.constantVal\}$
 7:     end if
 8:     if $isCopy(dep)$ then
 9:       $vSet \leftarrow vSet \cup \{computeVar(dep.operand, C)\}$
10:     end if
11:     if $isParameter(dep)$ then
12:       $vSet \leftarrow vSet \cup \{C[dep]\}$
13:     end if
14:     if $isBinOP(dep)$ then
15:       $vS1 \leftarrow compuateVar(dep.operand\_1)$
16:       $vS2 \leftarrow compuateVar(dep.operand\_2)$
17:       if $isApplicable(dep.opType)$ then
18:         for each combination of values ($v1$, $v2$)
                  in $vS1$ and $vS2$ do
19:           $vSet \leftarrow vSet \cup \{eval(dep.opType, v1, v2)\}$
20:         end for
21:       else
22:         for each combination of values ($v1$, $v2$) in
                  $vS1$ and $vS2$ do
23:           $vSet \leftarrow vSet \cup \{dep.opType.toString()+$
                  $v1.toString() + v2.toString()\}$
24:         end for
25:       end if
26:     end if
27:     if $isOtherType(dep)$ then
28:       $vSet \leftarrow vSet \cup \{"\$" + var.Name\}$
29:     end if
30:   end for
31:   return $vSet$
32: end function
```

able is acquired. Note that this was pre-computed using LLVM. A variable used at a statement n may be defined at multiple locations m_1, m_2, and so on, leading to multiple data dependences. The loop in lines 4-30 traverses each data dependence, which essentially denotes a definition of the variable, to compute the possible values. If the definition is an assignment of a constant value, the value is added to the set (lines 5-7); If it is a copy from another variable, method *computeVar()* is recursively invoked on the source variable (lines 8-10); If the variable is a formal parameter, the possible values of the parameter are loaded from the context and added to the value set; If it is defined through a binary operation such as string concatenation and integer addition, depending on if we could execute the operation offline, we may evaluate the operation on each pair of possible operand values (lines 17-20) or simply convert the operation to a string and concatenate it with the operand values to get the symbolic representations of the possible values (lines 22-24). We currently support offline evaluation of arithmetic operations and string concatenation, string length and substring operations if the concrete values of the string operands can be computed. Other unary and ternary operations are similarly supported and hence omitted from the algorithm. For other types of operations, such as acquiring a value posted by the client-side (e.g. `POST[...]`), the possible values are represented by a symbolic variable.

Example. Fig. 5 shows an example of V-DOM construction. The PHP script is a simplified version of the snippet in *Timeclock 1.0.2* to output the footer of a page. It outputs the link to the official website of PHP language. And an email address link is also provided right after the URL link if the address is valid. The email address is retrieved by a database's query (line 1). If a non-null string value is returned by the query, a user-defined function is called to output the email information (lines 3-4). In the function, if the email address is valid, it is printed (lines 7-9). A warning message is emitted otherwise.

In the previous LLVM analysis phase, an analysis log record is generated, which contains CFG and data dependence information. The CFGs of the two functions are showed in figure (b). According to Alg. 1, the CFG of the main function is traversed in the topological order. As such, in the resulting page in figure (c), the first two lines of the client page are emitted as in a real execution. Upon reaching the function invocation at line 4, the algorithm continues to analyze the function *emailFooter()*. In the context, the



```
<div>
…
<?php
…
1 $email = mysql_result(… , …);
2 Print(“<ahref='http://php.net'>PHP</a>”);
3 if(!is_null($email))
{
4 emailFooter($email);
}
…
5 Function emailFooter($e)
{
6 If(isValid($e))
7   print( "<a href='mailto:"
8            .$e.
9         "> email address  </a>");
10 Else
11   print(“<p>Invalid Email Address !</
p>”);
}
…
?>
</div>
```

(a) server script

(b) LLVM analysis record

```
1 <div>
…
2 <a href='http://php.net'>PHP</a>”);

3 <a href='mailto $email>
4  email address  </a>
5 <p>Invalid Email Address !</p>
…
6 </div>
```

(c) V-DOM tree and its page

Figure 5: V-DOM Construction Example

formal parameter e is determined to have a symbolic value “$*email*” as we cannot determine the concrete value returned from a database's query. Inside the function, the true and the false branches are traversed one after the other, leading to the hyper-links at lines 3-5 in (c) being emitted next to each other, both included in the same container. Note that the algorithm evaluates the string concatenation at lines 7-9 in (a) to acquire lines 3-4 in (c).

4.4 Test Data Collection and Coverage Calculation

In this step, we collect DOM objects exercised in concrete executions and compute the coverage of a test suite using the V-DOM tree constructed in the previous step.

We run the test suite on the web application and collect the concrete DOM trees. We remember the attributes and text values (if any) of the events of any interactive DOM objects. We also record all the displayed DOM objects.

Next, we match the DOM objects collected during testing with those in the corresponding V-DOM trees by their IDs and compute both the display coverage and event coverage.

In theory, if two DOM objects match, they should share the same set of attributes. However, since we use static value approximation in V-DOM tree construction, whole words matching is not always possible. We solve this problem by treating the symbolic variables introduced during V-DOM tree construction as wildcard characters that can match with any strings. Also, if there are multiple matches in the V-DOM tree, the first one is used.

To compute event coverage, we need to identify the set of DOM objects that are interactive and their events. It is not trivial to automatically to achieve this goal [22]. In our design, we detect interactive DOM objects from their tag names. More specifically, DOM objects tagged as “<a>”, “<button>” and “<input=submit>” are considered interactive. In addition, objects with event handlers attached are also treated as interactive. Currently, we only consider those event handlers that are statically registered as an attribute of a DOM object. We do not support runtime handler hooking by JavaScript.

5. EXPERIMENT

In our empirical study, we focus on answering the following research questions:

RQ1: Can V-DOM trees model more DOM objects than a dynamic approach based on a state-of-the-art crawler *crawljax* [22] with reasonable overhead?

RQ2: Does V-DOM tree coverage perform better than server-side code coverage and UI coverage based on dynamic models generated by *crawljax* in fault detection?

We chose five programs for our experiments. They are collected from existing works in web application testing [8].

- aphpkb 0.95.5 is a knowledge base management system.
- faqforge 1.3.2 is a web-based document creation and management tool.
- newspro 1.4.0 is an advanced news management system.
- schoolmate 1.5.4 is a PHP solution for elementary, middle and high schools' administration.
- timeclock 1.0.2 is an effective web-based time clock system which replaces manual sign-in sheets.

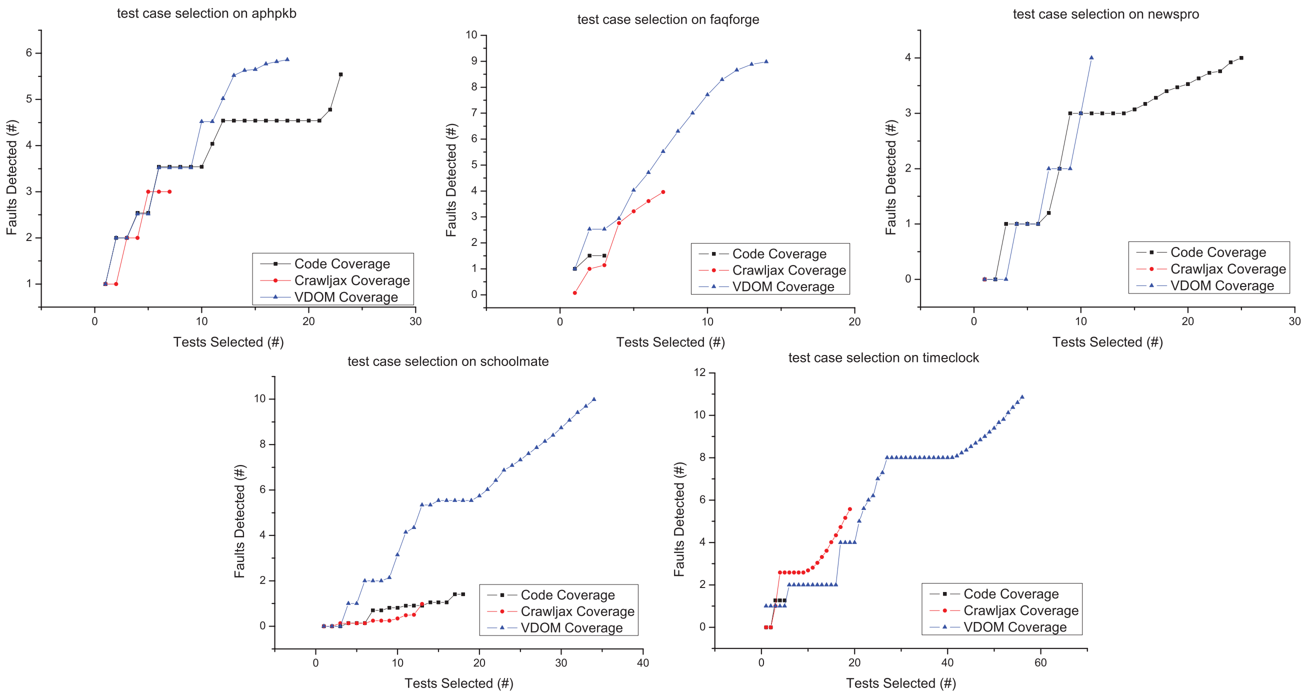


Figure 6: Fault Detection Rate

Table 1: HTML Element Extraction Evaluation

	aphpkb	faqforge	newspro	schoolmate	timeclock
Version	0.95.5	1.3.2	1.4.0	1.5.4	1.0.2
LOC	4283	734	6925	8181	20789
Files	46	19	30	63	62
$\lvert E_V \rvert$	869	439	1395	2191	11511
$\lvert E_C \rvert$	225	167	71	685	769
$\lvert E_V \cap E_C \rvert$	210	167	69	685	750
$\lvert E_V \setminus E_C \rvert$	659	272	1326	1506	10761
$\lvert E_C \setminus E_V \rvert$	15	0	2	0	19
V-DOM Size	227KB	52KB	1322KB	164KB	1212KB
Crawljax Size	745KB	33KB	59KB	439KB	222KB

5.1 V-DOM Tree Models vs. Dynamic Models

In the first experiment, we generate the V-DOM tree models of the subject programs, and compare them with the dynamic UI models generated by a state-of-the-art crawler *crawljax* [22]. To the best of our knowledge, *crawljax* is the first and the only automatic UI model generator for web applications that supports web 2.0 techniques. It works by automatically following hyper-links and providing random values as client inputs. We run *crawljax* on the subject programs with the default configuration. For some web applications, we manually provide some inputs to the tool to access pages (e.g. username and password). The results are presented in Table 1. It shows the sizes of the V-DOM tree model and the *crawljax* model (the last two rows), the number of DOM objects extracted from the V-DOM tree model (row $|E_V|$) and the *crawljax* model (row $|E_C|$) , the number of DOM objects in both models (row $|E_V \cap E_C|$) and the number of DOM objects that are present in one model but not the other (rows $|E_V \setminus E_C|$ and $|E_C \setminus E_V|$).

The results show that the proposed V-DOM tree approach can model far more DOM objects than using a crawler. It suggests that even an advanced crawler like *crawljax* can only crawl a small portion of all the possible client pages because these pages heavily depend on the client-side inputs, the server-side state (e.g. database's state) and the sequence of user interactions. In practice, the search space for all possible inputs and sequences is too large to explore through dynamic executions.

We also observe that there are a small number of DOM objects that are not obtained by the V-DOM tree model but by the dynamic model. For example, in program *aphpkb*, 15 DOM objects are missed by the V-DOM tree model. However, the portion of missing objects is very little: 1.7% for aphpkb, 0.14% for newspro, 0.16% for timeclock, and none for the rest. Further inspection shows that 27 missing objects are due to the mis-transformation from PHP source code to C source code, and from C code to LLVM intermediate byte-code; 9 missing objects are due to the symbolic approximation of certain operations. Recall in Algorithm 2, we convert an operation that is not supported by our current system to its symbolic name and concatenate it to operands to approximate the result of the operation.

5.2 Cost of V-DOM Tree Construction

In the second experiment, we evaluate the runtime performance of V-DOM tree construction and compare it with the cost of running *crawljax*. The results are shown in Table 2. The experiment was performed on a machine with Intel Core i3 3.07GHz CPU and 4GB RAM. Observe that V-DOM tree construction is more expensive due to the various analysis performed. The current implementation has not been optimized. We believe that efficiency could be substantially improved by optimizing the computation of possible variable values. Furthermore, we argue that the performance overhead is only a one-time cost.

5.3 Fault Detection Effectiveness Comparison

In the third experiment, we compare the fault detection capabilities of V-DOM tree coverage, server-side code cover-

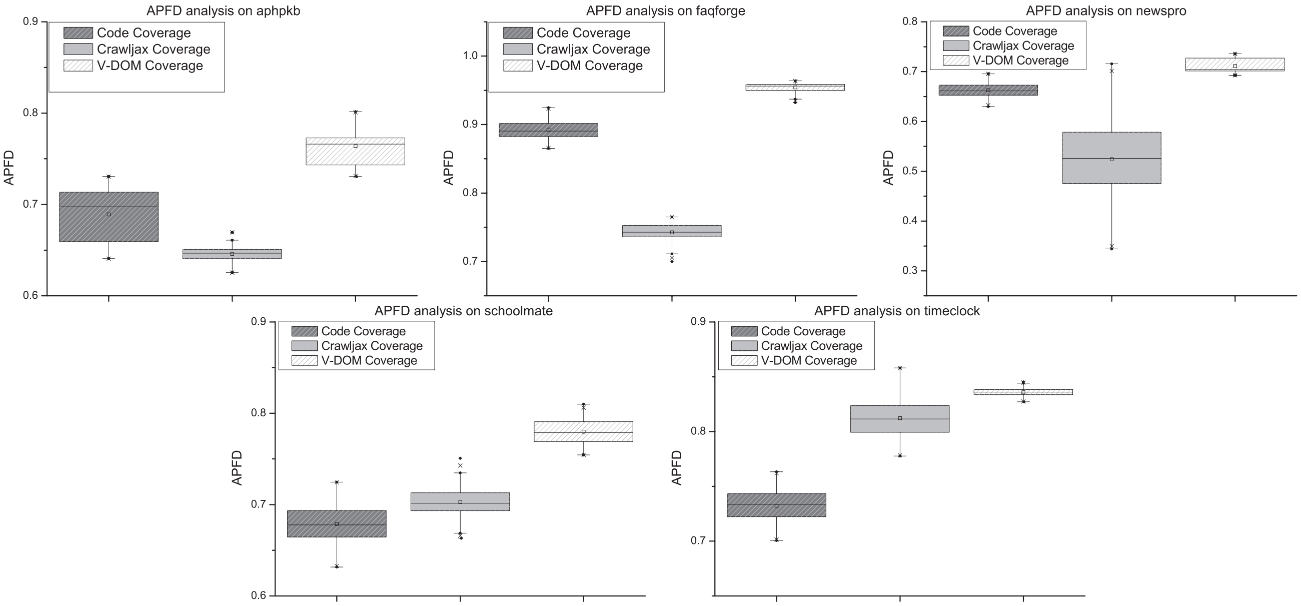


Figure 7: APFD Analysis

Table 2: Runtime Performance (in seconds)

		aphpkb	faqforge	newspro	schoolmate	timeclock
V-DOM	t_1	26.3	32.4	35.2	59.7	66.2
	t_2	163.9	65.8	454.2	3512.6	2491.9
	t_3	5.3	2.3	10.6	24.1	96.7
	t_t	195.5	100.5	500.0	3596.4	2654.8
Crawljax	t_t	61.5	24.2	25.4	61.5	31.0

t_1, t_2, and t_3 are the times of three steps of V-DOM construction, respectively. t_t is the total time.

age, and UI element coverage (based on the dynamic models generated by *crawljax*). We perform test selection using the three coverage criteria on five subject programs. Please note that not all faults could be detected by a subset of tests in test selection.

Seven graduate students and one senior undergraduate student conducted manual testing of the subject programs based on their specifications. The manual tests are then recorded as test scripts using *Selenium* [3]. The information of faults and tests used in our experiment is shown in Table 3. Column 2 shows the number of faults identified by the human testers. All of them are real faults. Columns 3-5 show the number of failing tests, passing tests, and total tests, respectively.

Table 3: Faults and Tests

program	faults	tests		
		fail	pass	total
aphpkb	10	10	49	59
faqforge	11	45	125	170
newspro	7	7	43	50
schoolmate	23	44	151	195
timeclock	11	11	144	155

We then performed test selection on the test suite for each program as follows. First, the coverage of each criterion is computed for each Selenium test case. A greedy algorithm is used to select test cases. The algorithm always tries to select a test case from the remaining (unselected) test cases to maximize the coverage of the selected subset for each criterion. If there are multiple such candidates, one is chosen randomly. After each step of selection, we compute and compare the numbers of faults that are detected by the selected suite for each criterion. The process ends when the selected tests reach the maximum coverage points. That is, the number of tests selected by different coverage criterion may be different. To ensure the results are not biased due to the non-determinism in the greedy algorithm, we repeated the selection process 100 times for each subject program for each coverage criterion.

Fig. 6 shows how the average number of detected faults grows as the number of selected test cases grows. Tests selected by V-DOM coverage can detect faults more fast than tests selected by other two coverage criteria. For all of five programs, V-DOM coverage outperforms UI coverage (Crawljax) significantly. For faqforge, schoolmate and timeclock, V-DOM coverage outperforms code coverage significantly. For aphpkb and newspro, tests selected by V-DOM can detect more faults, despite code coverage can use more tests to achieve the same goal. The results indicate that the fault detection rates of V-DOM are higher than the ones of code coverage and UI coverage.

In order to further investigate the effectiveness of V-DOM coverage, we introduce Average Percentage of Faults Detected (APFD) metric, which is commonly used in test case prioritization [4]. APFD measures the rate of fault detection per percentage of test execution. The APFD scores are calculated by taking the average of the percentage of faults detected during the execution of tests. The APFD scores range from 0 to 1, where higher APFD values imply faster fault detection rates. For fair comparison, we prioritize tests with three coverage criteria in the same test suite. That is, the number of tests for each coverage criterion is the same. We repeat 100 times and draw box-plots for each program, as shown in Fig. 7. The experimental results show that the APFD scores of V-DOM are much higher than the APFD scores of other two coverage criteria. This indicates that V-DOM is a promising coverage criterion for dynamic web testing.

6. RELATED WORK

Our technique is most relevant to existing web application modeling techniques. Ricca and Tonella [25] developed a high level UML-based representation for web applications. In [27] Tonella et al. extended the model to include server pages. In their method, multiple entities are created for a single server-side script, one for each possible client page generated by the script. However, there may be too many possible client pages to enumerate. In contrast, our approach generates a V-DOM tree for each server script. Liu et al. [20] and Kung et al. [19] proposed multiple models, each targeting on representing a single tier in web applications. They also suggested that data flow analysis could be performed at multiple levels. Though the models are able to represent the interactions between different components of a web application, it is not clear if the models have been implemented and experimentally evaluated. Di Lucca et al. [21] developed a UML-based web applicaion model and a set of tools for evaluation and automation of web application testing. They consider individual pages of an application as components to be tested at the unit level.

String analysis is a form of static program analysis which is to infer the possible values of string expressions [26, 13, 16, 17, 18, 23, 28]. Christensen et al. [13] generate context-free grammars with non-terminals representing string expressions in Java programs to approximate the possible values. Minamide et al. [23] presented an analysis that approximates the string output of a program with a context-free grammar, and used it for cross-site scripting vulnerability detection and validation. Tateishi et al. [26] encoded programs in Monadic Second-Order Logic to check if a string satisfies a given property. Compared to these techniques, our analysis focuses on composing a whole (virtual) client HTML page by concatenating the string values of all output statements in a server script. The composition is driven by a topological traversal of program statements. Furthermore, our analysis is context sensitive and uses symbolic approximation for unsupported operations.

The structures of web pages and their contents can be extracted dynamically by web crawlers [10, 11, 14, 12, 9, 22]. Web 2.0 techniques such as JavaScript and dynamic DOM manipulation post challenges to web crawling due to their dynamic features [14, 22]. Even with advanced web crawlers that support web 2.0 features, a large number of DOM objects are usually missing in the generated models due to the difficulty in covering the whole dynamic input space, as shown by our experience with *crawljax*. Alshahwan et al. [5] presented crawlability metrics to quantify properties of web applications that affect crawling.

Our technique is also related to various test coverage criteria. Structure coverage, such as branch coverage and MC/DC, is one of the most commonly used criteria to guide test generation and selection [7]. In some cases, it can also be used to evaluate the quality of test sets. Data-flow coverage [15] is another kind of popular coverage criterion that measures the portion of variable definitions, uses, and definition-use relations that are covered. Very thorough discussion about coverage criteria could be found in [24, 30, 6].

7. CONCLUSIONS

In this paper, we present a novel coverage criterion for web application testing based on DOM objects. The technique statically analyzes a web application to generate Virtual DOM trees that model all the possible DOM objects that could be generated in some execution of the web application. V-DOM trees model server-side execution by including all the objects that could be generated by different execution paths of server scripts. They model client-side execution by including all the displayable DOM objects and their event handlers. As such, a good coverage of V-DOM tree implies a good coverage of executions on both client- and server-sides. Our experimental results show that V-DOM trees contain far more DOM objects than the dynamic "crawled" models of web applications, and test selection driven by V-DOM tree coverage is substantially more effective than the existing code coverage and UI element coverage.

ACKNOWLEDGMENTS

This work was supported by National Basic Research Program of China (973 Program 2014CB340702), National Natural Science Foundation of China (Grant No. 61170067, 61373013).

8. REFERENCES

[1] Phc: open source PHP compiler. http://www.phpcompiler.org/.

[2] The LLVM Project: a collection of modular and reusable compiler and toolchain technologies. http://llvm.org/

[3] Selenium: web browser automation. http://www.seleniumhq.org/.

[4] S. Elbaum, G. Rothermel, S. Kanduri, and A. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.

[5] N. Alshahwan, M. Harman, A. Marchetto, and P. Tonella. Improving web application testing using testability measures. In *WSE'09*, pages 49–58, 2009.

[6] C. Fang, Z. Chen, and B. Xu. Comparing logic coverage criteria on test case prioritization. *Science China Information Sciences*, 55(12):2826–2840, 2012.

[7] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[8] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, and A. Paradkar. Finding bugs in dynamic web applications In *ISSTA'08*, pages 261–272, 2008.

[9] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[10] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998.

[11] M. Burner. Crawling towards eternity: Building an archive of the World Wide Web. *Web Techniques Magazine*, 2(5), 1997.

[12] J. Cho and H. Garcia-Molina. Parallel crawlers. *Technical report*, 2001.

[13] A. Christensen, A. Feldthaus, and A. Moller. Precise analysis of string expressions. In *SAS'03*, pages 1–18, 2003.

[14] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. Ajax crawl: making AJAX applications searchable. In *ICDE'09*, pages 78–89, 2009.

[15] P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.

[16] E. Geay, M. Pistoia, B. Ryder, and J. Dolby. Modular string-sensitive permission analysis with demand-driven preciion. In *ICSE'09*, pages 177–187, 2009.

[17] P. Hooimeijier and W. Weimer. A decision procedure for subset constraints over regular languages. In *PLDI'09*, pages 188–198, 2009.

[18] A. Kiezun, V. Ganesh, P. Guo, P. Hooimeijier, and M. Ernst. A solver for string constraints. In *ISSTA'09*, pages 105–116, 2009.

[19] D. Kung, C. Liu, and P. Hsia. An object-oriented web test model for testing web applications. In *APAQS'00*, pages 111–120, 2000.

[20] C. Liu, D. Kung, and P. Hsia. Object-based data flow testing of web applications. In *APAQS'00*, pages 7–16, 2000.

[21] G. Lucca, A. Fasolino, F. Faralli, and U. Carlini. Testing web applications. In *ICSM'2002*, pages 310–319, 2002.

[22] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling AJAX by inferring user interface state changes. In *ICWE'08*, pages 122–134, 2008.

[23] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW'05*, pages 432–441, 2005.

[24] A. Namin and J. Andrews. The influence of size and coverage on test suite effectiveness. In *ISSTA'09*, pages 57–68, 2009.

[25] F. Ricca and P. Tonella. Analysis and testing of web applications. In *ICSE'01*, pages 25–34, 2010.

[26] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. In *ISSTA'11*, pages 166–176, 2011.

[27] P. Tonella, F. Ricca, E. Pianta, and G. C. Evaluation methods for web application clustering. In *WSE'03*, pages 33–40, 2003.

[28] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI'07*, pages 32–41, 2007.

[29] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura and Z. Su. Dynamic test input generation for web applications. In *ISSTA'08*, pages 249-260, 2008.

[30] Y. Wei, M. Oriol, and B. Meyer. Is coverage a good measure of testing effectiveness? Technical report, ETH Zurich, 2010.

[31] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. in *ICSE'12*, pages 277-287, 2012.

[32] Y. Zou, C. Fang, Z. Chen, X. Zhang, and Z. Zhao. A hybrid coverage criterion for dynamic web testing. in *SEKE'13*, 2013.

[33] A. Memon, M. Soffa, and M. Pollack. Coverage criteria for GUI testing. in *FSE'01*, pages 256–267, 2001.