2014 IEEE International Conference on Software Testing, Verification, and Validation

Detecting Concurrency Errors in Client-side JavaScript Web Applications

Shin Hong, Yongbae Park, Moonzoo Kim
Computer Science Department, KAIST
South Korea
hongshin@kaist.ac.kr, yongbae.park@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

***Abstract*—As web technologies have evolved, the complexity of dynamic web applications has increased significantly and web applications suffer concurrency errors due to unexpected orders of interactions among web browsers, users, the network, and so forth. In this paper, we present WAVE (Web Application's Virtual Environment), a testing framework to detect concurrency errors in client-side web applications written in JavaScript. WAVE generates various sequences of operations as test cases for a web application and executes a sequence of operations by dynamically controlling interactions of a target web application with the execution environment. We demonstrate that WAVE is effective and efficient for detecting concurrency errors through experiments on eight examples and five non-trivial real-world web applications.**

I. Introduction

Web technologies including web browsers, JavaScript, and client-server technologies, have evolved quickly, and the complexity of client-side dynamic web applications (a.k.a., Ajax applications) also has increased rapidly. Although JavaScript has a *single thread execution model*, a dynamic web application written in JavaScript can suffer from *concurrency errors* due to unexpected execution orders [6], [13], [18], [23]. This is because execution of a web application involves operations with multiple external entities such as users, the network, and servers that behave *non-deterministically*.

Because mobile web applications that run on different platforms (i.e., various web browsers and operating systems, with various network speeds) are widely used, concurrency errors due to non-deterministic behaviors of these environments can provide serious threats to web applications. In addition, concurrency issues are becoming more serious for web applications because a new web standard, HTML5, allows web applications to exploit concurrent features and provide interactive services by utilizing multi-core CPUs. Thus, it is important to develop a testing framework that can detect concurrency errors in JavaScript applications systematically.

In this paper, we present WAVE (Web Application's Virtual Environment), a testing framework to detect concurrency errors in client-side JavaScript web applications. WAVE considers a concurrent execution of a web application as a sequence of operations, and generates various sequences of operations as test cases. WAVE causes the web application to exercise the generated test cases by controlling the virtual environment of the target web application at runtime. In addition, WAVE prioritizes test cases for faster detection of concurrency errors.

To demonstrate the effectiveness (in terms of how many concurrency errors are detected) and efficiency (in terms of time taken to detect concurrency errors) of WAVE, we have performed a series of experiments on eight benchmark applications and five real-world open-source web applications. In the experiments, WAVE detected concurrency errors in all eight benchmark programs and detected *new* concurrency bugs in the five real-world web applications (these bugs have been reported to the developers of the web applications).

The contributions of this paper are as follows:

- This paper presents a formal concurrent execution model for JavaScript web applications that can clearly describe concurrency problems in those applications.
- The paper proposes a new testing framework that detects concurrency errors in JavaScript applications effectively and efficiently by generating and enforcing various sequences of operations. In addition, WAVE utilizes three different test case prioritization heuristics to detect likely concurrency errors faster.
- The paper presents an empirical evaluation of WAVE targeting eight benchmark programs and five real-world JavaScript web applications. The results demonstrate that WAVE can detect various concurrency errors quickly and also show that WAVE was able to detect five new bugs in real-world applications.

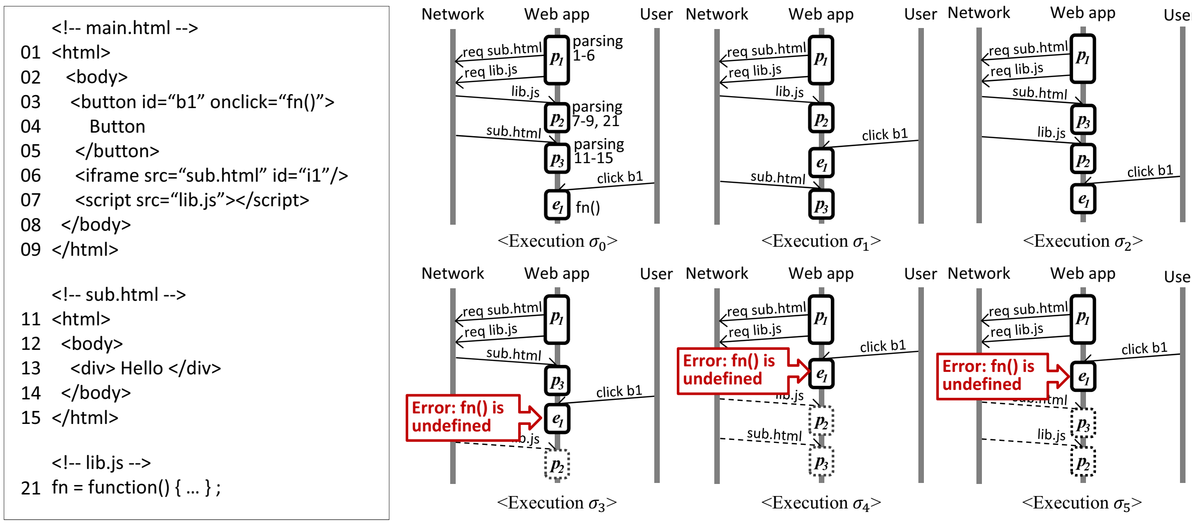
The remainder of this paper is organized as follows. Section II provides background information on JavaScript applications, specifically regarding possible concurrency problems. Section III describes the WAVE framework. Section IV explains the setup for the experiment to demonstrate the effectiveness and efficiency of WAVE and Section V describes the results of the experiments. Section VI discusses observations derived from the experiments. Section VII explains related work, and Section VIII concludes the paper with future work.

II. Background

A. Execution of JavaScript Web Application

A dynamic web application consists of multiple web pages that contain HTML code and JavaScript code. One web page can utilize multiple HTML files through `iframe` elements, external script elements in a page, and dynamic script loading [4]. Based on the W3C standard specifications [1], we can model an execution of a web application as a sequence of *parsing operations* and *event handling operations*.

978-0-7695-5185-2/14 $31.00 © 2014 IEEE
DOI 10.1109/ICST.2014.17



```
   <!-- main.html -->
01 <html>
02   <body>
03    <button id="b1" onclick="fn()">
04       Button
05     </button>
06     <iframe src="sub.html" id="i1"/>
07     <script src="lib.js"></script>
08   </body>
09 </html>

   <!-- sub.html -->
11 <html>
12   <body>
13     <div> Hello </div>
14   </body>
15 </html>

   <!-- lib.js -->
21 fn = function() { ... } ;
```

(a) HTML code

(b) Correct (i.e., $\sigma_0, \sigma_1, \sigma_2$) and erroneous (i.e., $\sigma_3, \sigma_4, \sigma_5$) execution scenarios

Fig. 1. A JavaScript web application containing a concurrency bug

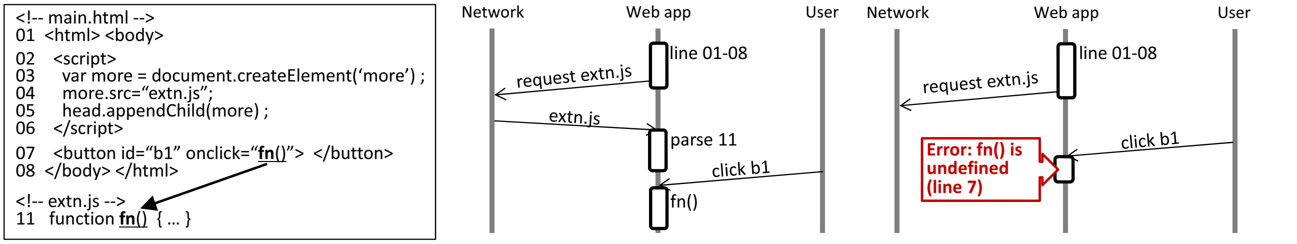
- A parsing operation interprets the HTML code to generate a corresponding *Document Object Model (DOM)* tree to construct a logical presentation of the page. In addition, a parsing operation interprets the JavaScript code of the page to define event handler functions or perform computations.
- An event handling operation occurs as follows. First, a user provides an input to the web page by triggering an event on the DOM tree such as clicking on buttons or pressing keys on text fields. Then, as a response to the event, the web browser invokes a JavaScript function defined as a handler of the event in a web page (i.e., event handling operation) as follows:
  - *Operations to handle network events*
    A web application runs an operation for a `onload` event when a corresponding external file (e.g., an image file) is completely downloaded. A web application runs an operation for a response event (i.e., `onreadystatechange`) when a response from a server is received.
  - *Operations to handle user input events*
    A user input event on the element of a target application (e.g., a mouse click on a button) invokes an operation to execute a corresponding event handler function.
  - *Operations to handle timed events*
    A web application runs an operation to execute a corresponding event handler for periodic timed events, which are registered by using `setTimeout()`/`setInterval()`.

A web browser executes an operation *atomically* and executes only one operation at a time. Note that an execution of a web application within one page (i.e., without changing URLs) can contain multiple event handling operations for multiple events. In addition, an execution of a web application can contain multiple parsing operations because:

- If the HTML/JavaScript code of a web page is contained in multiple files, a web browser should parse each file separately (i.e., repeat a parsing operation multiple times)
- A web browser pauses a parsing operation when it reaches a *waiting point* during parsing, and then invokes a subsequent parsing operation to complete the parsing from the waiting point. Modern web browsers define the following three waiting points [8], [9]:
  - when parsing a `script` element for an external file
  - when parsing an HTML element whose content is longer than a buffer size (e.g., 8 KByte for WebKit)
  - when executing a JavaScript instruction `alert()` that requests a user to click the alert message button.

Unintended scheduling of these operations due to non-deterministic behaviors of the environment of a target application can cause *concurrency errors* (for example, although a developer assumes that `t1.js` is downloaded earlier than `t2.js` which calls a function in `t1.js`, `t2.js` can be downloaded earlier than `t1.js` due to different network speeds).

For example, Figure 1 illustrates a JavaScript application example with a concurrency error. Figure 1(a) shows that the application code consists of `main.html`, `sub.html`, and `lib.js`. `main.html` has a button `b1` (lines 3–5) for which a mouse click event is registered to call `fn()` defined in `lib.js` (line 21) and an `iframe` element (line 6) whose content is in `sub.html`. Figure 1(b) shows six sequences of operations of this application σ_0 to σ_5, each of which consists of three parsing operations p_1, p_2, p_3 and one event handling operation e_1. p_1 parses lines 1–6 of `main.html` and stops to wait for `lib.js` to be downloaded (line 7). p_2 starts parsing lines 7–9 of `main.html` and line 21 of `lib.js` after `lib.js` is downloaded. p_3 begins parsing `sub.html` (line 11–15) after `sub.html` is downloaded. For example,



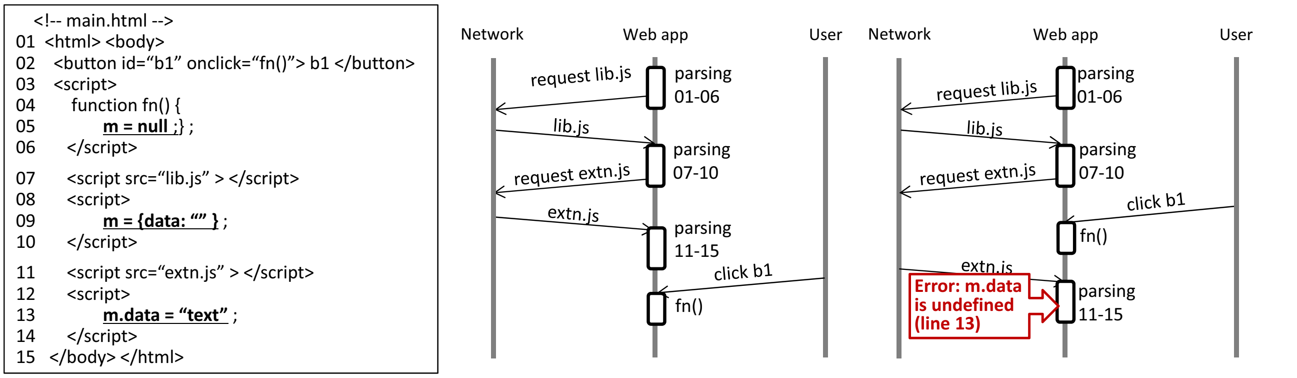
```
<!-- main.html -->
01  <html> <body>
02    <script>
03      var more = document.createElement(‘more’) ;
04      more.src=“extn.js”;
05      head.appendChild(more) ;
06    </script>
07    <button id=“b1” onclick=“fn()”>  </button>
08  </body> </html>

<!-- extn.js -->
11   function fn()  { … }
```

(a) Buggy code example

(b) Correct execution scenario

(c) Erroneous execution scenario

Fig. 2. An order violation example



```
   <!-- main.html -->
01  <html> <body>
02    <button id=“b1” onclick=“fn()”> b1 </button>
03    <script>
04       function fn() {
05            m = null ;} ;
06     </script>
07     <script src=“lib.js” > </script>
08     <script>
09            m = {data: “” } ;
10     </script>
11     <script src=“extn.js” > </script>
12     <script>
13            m.data = “text” ;
14     </script>
15   </body> </html>
```

(a) Buggy code example

(b) Correct execution scenario

(c) Erroneous execution scenario

Fig. 3. An atomicity violation example

σ_0 represents an execution where p_1, p_2, p_3 and e_1 occur in the sequence.

In the example, σ_3, σ_4 and σ_5 where e_1 precedes p_2 are error executions that fail to run `fn()`, because `fn()` is not defined yet. Note that p_1 always occurs before p_2, p_3, and e_1 since a user can click `b1` only after `b1` is created by p_1. The execution orders between p_2, p_3, and e_1, however, depend on when `lib.js` and `sub.html` are downloaded and when a user clicks `b1`. For example, p_2 occurs before p_3, if `lib.js` is downloaded before `sub.html` (i.e., σ_0, σ_1, and σ_4) or vice versa if `sub.html` is downloaded before `lib.js` (i.e., σ_2, σ_3, and σ_5). As we have seen in this example, although a web browser executes only one operation at a time, a web application can still suffer from concurrency errors.

B. Concurrency Bug Patterns in JavaScript Web Applications

A concurrency error in a JavaScript web application is caused by an illegal execution order of operations. There are two common concurrency bug patterns in JavaScript web applications: *order violations* and *atomicity violations* [25].

An order violation refers to a unintended race condition among two operations that must be executed in one order, but due to a programming mistake are executed in the reverse order. The order violation coincides with the race bug pattern defined in Petrov et al. [17] and Raychev et al. [19]. Figure 2 shows an order violation example. `main.html` dynamically loads the external file (`extn.js`) (lines 3–5) that contains `fn()` function definition (line 11). Thus, invoking `fn()` should not occur before `extn.js` is loaded and `fn()` is defined. Figure 2(b) shows a correct execution scenario where the web page works correctly because `extn.js` is loaded before clicking `b1`. However, clicking button `b1` before `extn.js` is loaded causes an error. Figure 2(c) shows such an erroneous execution scenario.

An atomicity violation refers to a unintended race condition that permits an operation to be scheduled between two operations that should be executed consecutively [25]. Figure 3 describes an example of an atomicity violation. `main.html` creates a button `b1` whose event handler is a function `fn()`. As `main.html` has two external script elements (at line 07 and at line 11), the parsing of `main.html` consists of the following three operations: parsing lines 01-06, parsing lines 07-10, and parsing lines 11-15. The third parsing operation should be executed only when `m.data` is defined by the second parsing operation (line 09). As shown in Figure 3(b), the web page does not produce any error when the second and the third parsing operations are executed consecutively. However, as shown in Figure 3(c), if a user clicks `b1` between the second and the third parsing operations, the execution raises an error because `m` is set to `null` by `fn()`.

III. WAVE Framework

A. Overview

Figure 4 illustrates the Web Application's Virtual Environment (WAVE) framework. First, WAVE monitors an execution of a target web application with a given user input scenario (called the execution σ_0). Then, WAVE generates various feasible operation sequences as test cases by alternating the order of the operations in σ_0 systematically. Finally, WAVE executes the application with these test cases and checks whether these different test cases produce different results (i.e., a concurrency error). WAVE determines that a web application has a concurrency error if, with the same sequence of user input events, the final DOM tree states of the application are different. Since WAVE generates all test cases based on σ_0 that keep the same sequence of user input events as σ_0, the results of the test cases should be the same. We assume that the given execution σ_0 shows the correct behavior. WAVE detects concurrency errors in a target application in the two phases, a *test generation phase* and a *test execution phase*.

In the test generation phase, the *execution recorder* obtains a *monitored execution* σ_0 by monitoring an execution (i.e., a sequence of operations) that contains interactions between a target application and its environment (e.g., user inputs, network operations such as sending and receiving packets, etc.). Then, it constructs an *execution model* that specifies constraints on orders between the operations in σ_0. One example of such a constraint is that generated test cases should not change the sequence of the user input events in σ_0. Another example is that, in Figure 1, p_1 should precede p_2, p_3 and e_1. The *test case generator* automatically generates a set of alternative sequences of the observed operations in σ_0 that satisfy the constraints of the execution model (Section III-C). Then, the test case generator prioritizes these test cases to detect concurrency errors quickly (Section III-D).

In the test execution phase, the *execution scheduler* executes the target application to run the list of test cases in the prioritized order. For each test case, the execution scheduler causes a target application to execute the operations of σ_0 in the order specified in the test case by controlling the interactions between the application and its environment. Finally, the *result checker* reports a concurrency error if one of the following conditions holds: (1) the target application raises an uncaught exception, (2) a web browser does not respond within 100 seconds (i.e., the order of operations in a given test case is infeasible), or (3) the final result page obtained in testing is different from the original result page obtained in σ_0.

B. Implementation

We have implemented WAVE on top of the WebKit browser framework. Figure 5 describes the structure of WebKit and WAVE. WebKit provides the WebKitCoreAPI, a set of event manager modules (GUI event manager, page loader, `XmlHttpRequest` manager, and timed event manager), and the JavaScript engine. WAVE is located between the WebKitCoreAPI and the set of event manager modules as an interface layer to manipulate interactions between the environment and a web application running on a web browser.

The WebKitCoreAPI receives an external event from the environment of a target application (i.e., a user, network, and/or timer) and then invokes the corresponding manager module to generate an operation to handle the event. The event manager modules run sequentially (i.e., once an event manager is invoked by the WebKitCoreAPI, no other event manager can be invoked until the event manager completes its operation). Thus, the order of operations in an execution is determined by a sequence of event manager invocations by the WebKitCoreAPI.

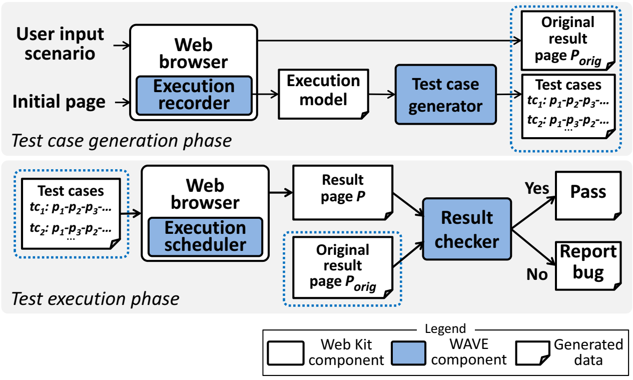


Fig. 4. WAVE Framework

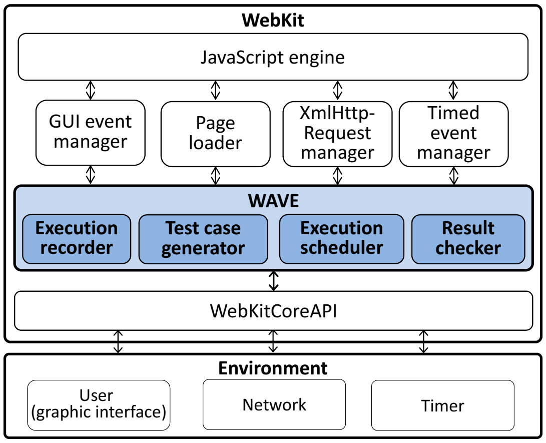


Fig. 5. WebKit and WAVE internal structure

In the test case generation phase, the execution recorder obtains σ_0 by monitoring the event manager invocations with relevant parameters (in addition, it records all information on user events to replay them later in the test execution phase). In the test execution phase, to exercise a generated test case, the execution scheduler intercepts event manager invocations by the WebKitCoreAPI to rearrange the order of operations as in the test case. For this purpose, the execution scheduler maintains a queue that contains all event manager invocations for external events (i.e., user events, network events, and timer events) to execute queued invocations in the order specified in a given test case. In other words, the event scheduler postpones an invocation of an event manager that handles an uncontrollable external event until it can invoke the event managers in the same order of operations in a given test case. For example, in Figure 1, suppose that the execution scheduler tries to execute a target application with σ_2 as a generated test case. And suppose that the application receives `lib.js` first. Then, the execution scheduler keeps an event manger (i.e., page loader) invocation corresponding/reacting to the event of receiving `lib.js` in the queue until it receives `sub.html`. After receiving `sub.html`, the execution scheduler invokes a page loader to execute p_3 first, and then invokes a page loader again to execute p_2 in the order specified in σ_2. In contrast, the execution scheduler replays a user input event anytime according to the generated test cases, since it can replay the user input event.

We have implemented the WAVE modules by adding 226 functions in C/C++ (2689 LOC) to WebKit rev.141241 on Qt5.

C. Execution Model Construction

The execution recorder monitors the runtime behavior of a web application and constructs an *execution model* $(Op, \rightarrow_{hb}, \rightarrow_{ui})$ of that web application based on σ_0. Op is a set of operations in σ_0. $\rightarrow_{hb}$ is a happens-before relation between two operations in Op induced by the web application code; $p \rightarrow_{hb} q$ means that p should happen before q in all executions. $\rightarrow_{ui}$ is a temporal order relation between two user input event operations. $p \rightarrow_{ui} q$ indicates that a user input event operation p precedes the other user input event operation q in σ_0.

$\rightarrow_{hb}$ indicates the constraints on the temporal order between operations, which should be satisfied in all executions. For example, in Figure 1, $p_1 \rightarrow_{hb} p_2$, $p_1 \rightarrow_{hb} p_3$, $p_1 \rightarrow_{hb} e_1$, but $p_2 \not\rightarrow_{hb} p_3$ since p_3 can precede p_2. WAVE generates various executions σ_is as test cases, whose operation orders are different from the monitored execution σ_0, but still satisfy $\rightarrow_{hb}$. The execution recorder determines $p \rightarrow_{hb} q$ by checking whether one of the following conditions on p and q holds in σ_0:

- For two parsing operations p and q:
  - p initiates q. In other words,
    - A file parsed by p contains `<iframe src=x.html>` where `x.html` is parsed by q, or
    - A target file parsed by p contains a script loading of `x.js` that is parsed by q.
  - Or, p and q parse the same file, and p precedes q.
- For a parsing operation p and a user input event operation q:
  - p creates a DOM object(e.g., a button) on which q is defined, or
  - p registers an event handler on a DOM object d and q executes the event handler for a user input event on d.
- For a network event operation q:
  - p creates a DOM object d (e.g., an image file), and q handles an `onload` event on d, or
  - p registers an event handler of an `onload` event on a DOM object d, and q handles the `onload` event, or
  - p sends a request to a server, and q handles the response event to the request.
- For an operation q to run a registered event handler for a timed event:
  - p registers a handler for timed events by `setTimeout()/setInterval()`, and q executes the handler for the timed events.

The event recorder determines $p \rightarrow_{ui} q$ for operations to handle user input events p and q if p precedes q in σ_0. $\rightarrow_{ui}$ represents a user input scenario that should be the same across all test cases generated. In other words, a new sequence of operations σ_i should follow the order of operations to handle user input events in σ_0; σ_i can be invalid otherwise. For example, suppose that p handles a user click on a 'File menu' and q handles a user click on a 'Save file' button in the 'File menu' sub-window. Then, p should always happen before q (i.e., $p \rightarrow_{ui} q$) since the 'Save file' button is created by p.

D. Test Case Generation and Prioritization

The test case generator creates an ordered list of test cases based on the execution model as follows. First, the test case generator generates all valid combinatorial sequences of operations that satisfy a transitive closure of $\rightarrow_{hb}$ and $\rightarrow_{ui}$ (calling the closure relation $\Rightarrow = \{\rightarrow_{hb} \cup \rightarrow_{ui}\}^*$). For the example in Figure 1, WAVE generates a total of 24 (=4!) different sequences by permuting p_1, p_2, p_3 and e_1. However, 18 of these sequences are invalid and discarded because they violate $\Rightarrow$ (for example, a sequence $p_2.p_3.e_1.p_1$ violates $\Rightarrow$ because $p_1 \Rightarrow p_2$).

Thus, we obtain six valid sequences that satisfy $\Rightarrow$ (i.e., σ_1 to σ_6 in Figure 1(b)). Each generated sequence of operations is used as a test case to detect concurrency errors.

Next, the test case generator prioritizes the generated test cases to detect concurrency errors quickly using two heuristics: a *precedence first (PF)* heuristic and an *adjacency first (AF)* heuristic.

Precedence first heuristic: The precedence first heuristic utilizes precedence relations for test cases. The precedence relation for test case σ is a set of *precedence pairs* of two operations in σ defined as follows:

$$Pr(\sigma) = \{(p, q) \mid \sigma[i] = p \wedge \sigma[j] = q \text{ for } 1 \leq i < j \leq |\sigma|\}$$

where $\sigma[i]$ indicates the ith operation in σ. For example, for $\sigma_0 = p_1.p_2.p_3.e_1$ in Figure 1, $Pr(\sigma_0) = \{(p_1, p_2), (p_1, p_3), (p_1, e_1)\ (p_2, p_3), (p_2, e_1), (p_3, e_1)\}$.

The precedence first (PF) heuristic selects a test case σ_{max} first such that $Pr(\sigma_{max})$ has the largest number of *uncovered precedence pairs*. In other words, the PF heuristic selects σ_{max} first such that $|Pr(\sigma_{max}) - \cup_{\sigma_j \in TC_{sel}} Pr(\sigma_j)|$ is the largest, where TC_{sel} is the set of test cases that have been already selected (initially $TC_{sel} = \{\sigma_0\}$).

Algorithm 1 describes how the precedence first heuristic prioritizes test cases. The algorithm receives a set of test cases Σ and a monitored operation sequence σ_0 as input and returns a list of prioritized test cases TC. After the initialization of C and σ_{max} by using σ_0 (lines 1–2), the algorithm repeatedly selects a test case $\sigma_{max} \in \Sigma$ that covers a largest number of uncovered precedence pairs (line 4) until Σ becomes empty (lines 3–8). A selected test case σ_{max} is moved from Σ to TC

Algorithm 1: Precedence first prioritization heuristic

```
Input: A set of valid test cases $\Sigma$ and a monitored
       operation sequence $\sigma_0$
Output: A list of prioritized test cases $TC$ (initially $\emptyset$)
1 $C = Pr(\sigma_0)$ // already covered precedence pairs
2 $\sigma_{max} = \sigma_0$// a test case that covers the largest number of
  uncovered precedence pairs
3 while $\Sigma \neq \emptyset$ do
4     $\sigma_{max} = \sigma \in \Sigma$ such that
      $\forall \sigma' \in \Sigma . |Pr(\sigma) \setminus C| \geq |Pr(\sigma') \setminus C|$
5     $\Sigma = \Sigma \setminus \{\sigma_{max}\}$
6     $TC = append(TC, \sigma_{max})$
7     $C = C \cup Pr(\sigma_{max})$
8 end
9 return $TC$
```

(lines 5–6), and then C is updated to include the precedence pairs of σ_{max} (line 7)

For example, suppose that WAVE generates three test cases $\sigma_1 = p_1.p_2.e_1.p_3$, $\sigma_2 = p_1.p_3.p_2.e_1$, and $\sigma_5 = p_1.e_1.p_3.p_2$ from the monitored execution $\sigma_0 = p_1.p_2.p_3.e_1$ in Figure 1, whose precedence relations are as follows: $Pr(\sigma_1) = \{(p_1,p_2),(p_1,e_1),(p_1,p_3),(p_2,e_1),(p_2,p_3),(e_1,p_3)\}$, $Pr(\sigma_2) = \{(p_1,p_3),(p_1,p_2),(p_1,e_1),(p_3,p_2),(p_3,e_1),(p_2,e_1)\}$, and $Pr(\sigma_3) = \{(p_1,p_3),(p_1,e_1),(p_1,p_2),(p_3,e_1),(p_3,p_2),(e_1,p_2)\}$. The PF heuristic selects σ_5 first to test, because $|Pr(\sigma_5) - Pr(\sigma_0)| = 3 > |Pr(\sigma_2) - Pr(\sigma_0)| = 2 > |Pr(\sigma_1) - Pr(\sigma_0)| = 1$. Note that σ_5 contains a concurrency error (see Figure 1(b)).

The conjecture behind this strategy is as follows. A lack of synchronization between two operations p and q often causes order violations (see Section II-B) that allow unintended orders between p and q. As test cases cover more precedence pairs, these test cases have higher probabilities of exercising unintended orders of operations and triggering concurrency errors.

Adjacency first heuristic: The adjacency first (AF) heuristic prioritizes test cases with respect to the adjacency relations of those test cases. The adjacency relation of σ is a set of pairs of two operations that are executed *consecutively* in σ. We define the adjacency relation as follows:

$$\begin{aligned} Ad(\sigma) \quad &= \quad \{(p,q) \mid \sigma[i] = p \text{ and } \sigma[j] = q \\ & \qquad \text{for } 1 \leq i < |\sigma| \text{ and } j = i+1\} \end{aligned}$$

where $\sigma[i]$ indicates the i-th operation in σ. For example, for $\sigma_0 = p_1.p_2.p_3.e_1$, $Ad(\sigma) = \{(p_1,p_2), (p_2,p_3), (p_3,e_1)\}$. The adjacency first (AF) heuristic first schedules a test case that has the largest number of uncovered adjacent pairs. The AF algorithm is the same as PF except that the adjacency relation is used instead of the precedence relation at line 4.

The conjecture behind the AF heuristic is that atomicity violations (see Section II-B) are caused by unexpected operation executions involving operations that should be executed adjacently. We expect that the test cases covering more adjacency pairs are more likely to detect concurrency errors.

TABLE I
STUDY OBJECT

Type	Name (abbreviation)	Size (LOC)	Num. user event op.	Num. total op.	Bug pattern
Real world app	AjaXplorer (AX)	1217	6	9	atom
	Feng Office (FO)	13530	10	12	atom
	Gallery3 (GL)	26212	1	4	order
	TYPO3 (TP)	8951	1	8	order
	WordPress (WP)	634	1	3	order
Bench-mark app	Benchmark 1 (B1)	44	2	8	order
	Benchmark 2 (B2)	65	2	9	order
	Benchmark 3 (B3)	88	1	8	order
	Benchmark 4 (B4)	83	3	11	order
	Benchmark 5 (B5)	69	3	8	atom
	Benchmark 6 (B6)	34	2	5	atom
	Benchmark 7 (B7)	44	3	7	atom
	Benchmark 8 (B8)	71	2	8	atom

IV. EXPERIMENT SETUP

A. Research Questions

In this study, we evaluate the effectiveness of WAVE in terms of the number of concurrency errors it detects in target JavaScript web applications and its efficiency in terms of the testing time it requires to detect a concurrency error in a target application. For this purpose, we design experiments to answer the following research questions:

- RQ1 (Effectiveness): How many concurrency errors are detected by WAVE with the two different test case prioritization algorithms (PF and AF) for a given number of test generations, compared to random testing techniques?
- RQ2 (Efficiency): How much test generation time is spent by WAVE in order to detect a concurrency error with the two different test case prioritization algorithms (PF and AF), compared to random testing techniques?

B. Study Object

Table I summarizes the 13 target applications used in this study. The first and the second columns describe the type of each study object and its name, respectively. The third column shows the sizes of the tested modules of the applications including HTML code and JavaScript code. The fourth column represents the number of user input event operations, and the fifth the total numbers of operations in each test case (i.e., the numbers in σ_0 since the operations of test cases are taken from σ_0). The sixth column represents the bug pattern related to each target application: 'order' means the order violation pattern and 'atom' the atomicity violation pattern (Section II-B).

We used eight benchmark applications and five non-trivial real-world web applications as target applications in the study. Each of the following eight benchmark applications has a concurrency bug originating from an actual bug report by developers or an example in the related work [9], [13], [17]. [1]

- Benchmark 1: Two network event operations are executed in a unexpected order

[1] Code and descriptions for the eight benchmark applications are available at http://swtv.kaist.ac.kr/data/webapp-race.

- Benchmark 2: Dynamic script loading is completed later than expected
- Benchmark 3: A page in an `iframe` is parsed in a unexpected order
- Benchmark 4: Timer events are executed in a unexpected order
- Benchmark 5: An external script stops parsing unexpectedly
- Benchmark 6: A long HTML content makes parsing stop unexpectedly
- Benchmark 7: An `alert()` stops parsing unexpectedly
- Benchmark 8: An unexpected operation is executed between an `XmlHttpRequest` request and the response event handler operation

For each application, WAVE generated 359.3 test cases from a monitored execution σ_0, each of which contains 8.0 operations (including 2.3 user input event operations) on average.

In addition, we tested the following five popular real-world web applications. AjaXplorer [2] is an open-source file server that provides a web interface. Feng Office [3] is a project management tool, and Gallery3 [4] is a photo sharing application. TYPO3 [5] and WordPress [6] are open-source content management systems. To generate the initial execution σ_0 for each program, we utilized standard use case scenarios. For example, for AjaXplorer, we used a scenario involving uploading a file to a server. The test case generator created 4.0 test cases per real-world application, each of which contains 7.2 operations (including 3.9 user input event operations) on average.

C. Testing Setup

We applied WAVE with the PF heuristics and WAVE with the AF heuristics to the eight benchmark programs and the five real-world web applications. In addition, to demonstrate the effectiveness and the efficiency of the PF and AF heuristics, we also applied WAVE with a random (RD) heuristics (i.e., prioritizing test cases randomly) as a baseline prioritization heuristics. We performed 30 testing runs per application and per test generation technique. We performed 30 testing runs to alleviate the random effects of the tie breaking of PF and AF as well as the random effects of RD. For the eight benchmark applications, each testing run executed the first 30 test cases of highest priority according to the three heuristics. For the five real-world applications, each testing run executed all generated test cases in the prioritized order since the number of generated test cases is relatively small (i.e., 4.0).

In addition, using the random network delay (RND) techniques, we performed 30 testing runs per application and per test generation technique (i.e., RND-50, RND-500, and RND-1000) and each testing run executed a target program 30 times with various random delays in TCP network operations. RND-x techniques inject a random delay between 0 and x milliseconds into network operations of an execution that contains the same sequence of user input events as σ_0. RND techniques are recommended for testing Ajax web applications in practice [10]. We apply RND-50, RND-500, and RND-1000 on Firefox 20.0.1 (FX) and Internet Explorer 10 (IE). We use these two web browsers because different web browsers may exhibit different temporal behaviors, which can influence the effectiveness and the efficiency of the RND techniques. For the RND techniques, we used WANem 2.3 [7] to generate random packet delays and Selenium 2.32.0 [8] to record and replay user actions. The machine used for running all experiments had a 3.4 GHz quad-core CPU and 2GB RAM.

V. Experiment Results

Using WAVE, we detected new concurrency errors in all five real-world web applications (we reported these five bugs to the application developers and three were confirmed by the developers; we have not received responses for the other two bug reports). Section V-A uses Feng Office as an example. Then, we explain the experiment results regarding the effectiveness and the efficiency of WAVE.

A. Example:Concurrency Bug in Feng Office

Figure 6 describes a concurrency error in Feng Office. Feng Office provides a workspace, a directory shared among users, where users can upload files and browse the list of files. Feng Office allows a user to conduct other activities such as browsing other workspaces while a file is being uploaded to a current workspace. WAVE detected that Feng Office fails to upload `file1.txt` with the following user input scenario where the server has two workspaces `ws1` and `ws2`.

1) A user selects `ws1` as a current workspace.
2) A user uploads a file `file1.txt` to `ws1`.
3) A user changes a current workspace from `ws1` to `ws2`.
4) A user selects `ws1` again.
5) A user browses the file lists of `ws1`.

Figure 6(a) and Figure 6(b) show a correct execution scenario and an erroneous one, respectively. The key difference between these two scenarios is that the correct execution scenario changes a workspace *after* `file1.txt` is completely uploaded, but the incorrect execution scenario changes the workspace *before* `file1.txt` is completely uploaded. Figure 6(c) shows simplified code of the event handler `cb()` (lines 2∼6) for the response network event that handles the completion of a file transmission at the server. `cb()` accesses the DOM object named `genid + 'addfile'` (line 3) to request the server to submit the transmitted file to the workspace (line 5). However, in Figure 6(c), since the current workspace is changed from `ws1` to `ws2`, the DOM object of `ws1` is deleted and an exception saying "`Error: genid is not defined`" is raised. As a result, although `file1.txt` is fully transmitted to the server, `ws1` does *not*

[2] http://ajaxplorer.info

[3] www.fengoffice.com

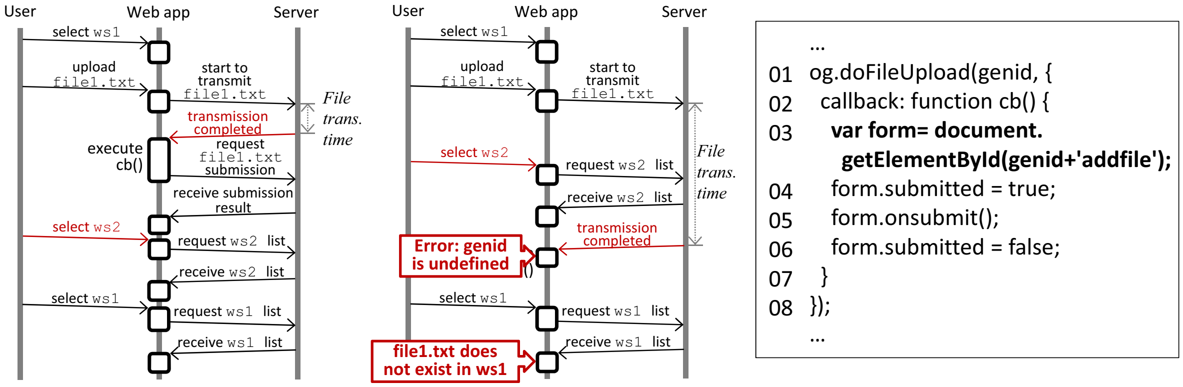
[4] http://galleryproject.org

[5] http://typo3.org/

[6] http://wordpress.org

[7] http://wanem.sourceforge.net/

[8] http://code.google.com/p/selenium/



```
...
og.doFileUpload(genid, {
  callback: function cb() {
    var form= document.
      getElementById(genid+'addfile');
    form.submitted = true;
    form.onsubmit();
    form.submitted = false;
  }
});
...
```

(a) Correct execution scenario

(b) Erroneous scenario

(c) Buggy web application code

Fig. 6. Concurrency bug in Feng Office

contain file1.txt, which results in data loss. This bug is classified as an atomicity violation because the error is caused by executing the operation to change a workspace between the two enclosing operations, one that starts the file1.txt transmission and one that completes the transmission. Note that this error is likely to occur when a file transmission takes a long time due to a large file size, long network delay, or slow server speed, etc. Since a target application cannot control these environmental factors, the detected concurrency bug is a realistic threat.

We detected other concurrency errors in the other four real-world applications. AjaXplorer contains a concurrency bug such that the response event handler operation updates the DOM object unexpectedly between the two user input event operations that manipulate the same DOM object (atomicity violation). In TYPO3, a user input event operation invokes an event handler function before its definition because an external script element stops parsing before the event handler function definition (order violation). Gallery3 and WordPress contain similar bugs – a user input event operation can invoke a JavaScript function defined in an external file before the external file is parsed (order violation). The bug reports for AjaXplorer, Feng Office and Gallery3 were confirmed as real faults by the developers. Thus, to detect such concurrency errors, developers should test their applications systematically and include various sequences of operations involving multiple entities such as users, networks, and servers.

B. RQ1: Regarding Effectiveness

Table II shows the error detection ability of the three RND techniques and WAVE. The first column shows the abbreviated object names. The second to seventh columns show the error detection results of RND-50, RND-500, and RND-1000 with firefox (FX) and internet explorer (IE). The eighth to tenth columns show the error detection results of WAVE with the three different prioritization strategies – WAVE with the random (RD), precedence first (PF), and adjacency first (AF) test case prioritization heuristics, respectively. Each cell represents the ratio of testing runs that detected a concurrency error out of the total 30 testing runs per object and per technique.

TABLE II
Error detection ability of the techniques

	RND-50		RND-500		RND-1000		WAVE		
	FX	IE	FX	IE	FX	IE	RD	PF	AF
AX	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
FO	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00
GL	1.00	0.00	1.00	0.00	1.00	0.07	1.00	1.00	1.00
TP	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00
WP	0.00	0.00	0.80	0.00	1.00	0.00	1.00	1.00	1.00
B1	0.13	0.07	0.80	0.33	0.93	0.40	1.00	1.00	1.00
B2	0.00	0.00	0.00	0.00	0.13	0.00	1.00	1.00	1.00
B3	0.73	0.80	0.87	0.73	0.73	0.67	1.00	1.00	1.00
B4	0.00	0.00	0.00	0.07	0.00	0.00	0.90	1.00	0.87
B5	0.00	0.00	0.07	0.00	0.00	0.00	1.00	1.00	1.00
B6	0.60	0.00	0.93	0.00	1.00	0.00	1.00	1.00	1.00
B7	0.00	0.13	1.00	0.00	1.00	0.00	1.00	1.00	1.00
B8	0.07	0.07	0.20	0.80	0.80	1.00	1.00	1.00	1.00
Avg	0.27	0.16	0.51	0.23	0.58	0.24	0.99	1.00	0.99

Note that WAVE with PF detected all concurrency errors in every testing run for all 13 target objects (i.e., all cells in the eighth column are 1.00), but the RND techniques did not. For example, for WordPress (WP), all three RND techniques using IE could not detect a concurrency error in any testing run (i.e., 0.00), but RND-500 and RND-1000 using FX detected a concurrency error at the ratio of 0.80 and 1.00, respectively (this difference between the RND techniques using FX and IE was due to the limitation of Selenium on IE, which does not allow user input event operations until a main page is completely parsed). Thus, we can confirm that WAVE with PF is highly effective for detecting concurrency errors.

Among the three test case prioritization heuristics, PF shows the highest error detection ability; WAVE with RD and AF failed to detect a concurrency error of Benchmark 4 in 3 and 4 out of 30 testing runs, respectively. We conjecture that WAVE with AF could not detect the bug in Benchmark 4 completely because the bug is an order violation, not an atomicity violation for which the AF heuristic was developed.

C. Regarding Efficiency

Table III shows the efficiency results of the RND techniques and the WAVE techniques. The results of WAVE with RD, PF,

TABLE III
TIME TO DETECT THE FIRST ERROR(IN SECONDS)

	RND-50		RND-500		RND-1000		WAVE		
	FX	IE	FX	IE	FX	IE	RD	PF	AF
AX	186.8	35.3	284.9	58.5	56.5	34.6	38.0	27.0	41.3
FO	-	-	-	-	-	-	108.8	22.9	46.4
GL	15.1	-	4.4	-	7.5	101.7	4.0	4.2	4.0
TP	-	-	-	-	-	-	11.2	3.2	8.2
WP	-	-	48.5	-	5.7	-	8.5	8.5	8.4
B1	100.1	108.5	64.1	346.2	103.4	196.5	6.5	1.6	5.1
B2	-	-	-	-	141.3	-	8.3	1.7	9.8
B3	43.0	32.1	61.6	110.7	136.1	166.4	4.0	1.4	5.6
B4	-	-	-	330.6	-	-	196.2	13.2	245.8
B5	-	-	216.8	-	-	-	8.4	1.7	12.9
B6	50.3	-	28.3	-	14.8	-	2.8	1.6	1.6
B7	-	155.0	11.5	-	6.8	-	4.1	1.5	4.0
B8	99.1	104.8	131.3	109.7	97.9	55.9	4.6	4.9	2.6
Avg	82.4	87.1	94.6	191.2	63.3	111.0	31.2	7.2	30.4

and AF are shown in the eighth to tenth columns of the table, respectively. Each cell represents the average time (in seconds) taken to detect the first concurrency error per target object and per technique over 30 testing runs. A '-' denotes the case in which no testing run detected a concurrency error. The result shows that WAVE with PF detected a concurrency error in 7.2 seconds on average over 13 target objects, which is 9 (63.3/7.2) times faster than RND-1000, the fastest RND technique (i.e., 63.3 seconds on average). WAVE with AF and RD spent 30.4 and 31.2 seconds on average, respectively. Considering that the average time of the RND techniques does *not* include the '-' cases where the RND techniques fail to detect any concurrency errors, WAVE is far more efficient than the RND techniques. Thus, we can confirm that WAVE with PF is highly efficient for detecting concurrency errors.

VI. DISCUSSION

A. Comparison with Race Bug Detector

We have compared WAVE with *EventRacer* [19], the latest race bug detection technique for JavaScript applications. EventRacer reports possible race bugs each of which consists of pairs of code elements based on a runtime trace of a target JavaScript web application. It reports a race bug with a risk level (roughly speaking, a higher risk level indicates a higher probability that the reported bug is real). We have applied EventRacer to the 13 study objects with the same user input scenario used for our experiments. Table IV summarizes the results. The second column shows whether or not EventRacer detected the bug that was detected by WAVE. The third column presents the risk level of the reported bugs. The fourth column presents the number of alarms for each study object. The table shows that EventRacer reported many false alarms and failed to detect two out of the 13 race bugs that WAVE detected.

TABLE IV
BUG DETECTION RESULT BY EVENTRACER

Program	The bug detected?	Risk level	Num. total alarms
AjaXplorer	Yes	High	3604
Feng Office	No	-	254
Gallery3	Yes	Low	228
TYPO3	No	-	1501
WordPress	Yes	Low	158
Benchmark 1	Yes	High	6
Benchmark 2	Yes	High	4
Benchmark 3	Yes	Low	2
Benchmark 4	Yes	Low	5
Benchmark 5	Yes	High	3
Benchmark 6	Yes	High	3
Benchmark 7	Yes	Low	7
Benchmark 8	Yes	High	2

EventRacer generated many false alarms. After manual review of the 32 bug reports on the eight benchmark applications, we found that 24 out of the 32 bug reports were false alarms. [9] The main reason for the high ratio of false alarms is that EventRacer assumes that every two script elements in a single HTML file can be parsed separately (i.e., by multiple parsing operations), but this assumption is only partially true (Section II-A). In contrast, WAVE utilizes an execution model to model parsing operations of modern web browsers in a realistic way (Section III-C). Also, WAVE is a testing framework working on real execution scenarios, which can prevent false alarms.

In addition, the bug detection capability of EventRacer was lower than WAVE. EventRacer failed to detect the atomicity violation bug in Feng Office and the order violation bug in TYPO3. Furthermore, the five bugs (i.e., the bugs in Gallery3, WordPress, B3, B4, and B7) were reported as "low" risk level bugs. We guess that EventRacer does not handle complex constructs of JavaScript in the target applications, which weakens the bug detection capability of EventRacer.

B. Lessons Learned for Web Application Developers

Through our study, we have learned the following lessons:

1. Prepare to handle various network delays to download external files: As we have seen in the Feng Office example, unexpectedly long network delays in transmitting files can raise concurrency errors due to unexpected sequences of operations. Since network delays are non-deterministic (particularly in mobile networks), various sequences of operations should be systematically tested.

2. Be careful when you change the location of JavaScript code: Web application developers often recommend moving JavaScript code to the bottom of the web page to shorten the time needed to display the page because parsing JavaScript code is a time-consuming task [22]. However, changing the location of JavaScript code can alter the order between parsing operations and other operations (e.g., user input event operations); thus, introducing concurrency bugs (for example, such a change may allow references to undefined JavaScript functions, as described in Figure 1). Thus, relocation of JavaScript code should be carefully planned.

3. Do not overlook the potential harmfulness of concurrency errors: We have learned that concurrency bugs in the real-world JavaScript applications can cause serious damage such as loss of user-provided data, invalid data updates to servers, and transitions to invalid web pages, etc. Therefore, developers

[9] We could not review the alarms on the five real world applications due to a large number of the alarms and the high complexity of the target applications.

should be careful to avoid concurrency bugs by using an automated testing framework like WAVE.

VII. Related Work

Automated Test Generation Techniques for Web Applications

Conventional automated testing techniques for web applications focus on generating user input event sequences and user input values, but not on testing diverse concurrent behaviors. Artemis [2], Marchetto et al. [11], and Crawljax [12] generate a set of user input event sequences as test cases for web applications and utilize feedback from the executions of the generated test cases (e.g., code coverage, state-flow graphs) to create next test cases with which to explore untested program behaviors. Apollo [3] and Kudzu [20] generate test input values by using symbolic execution techniques to cover large portions of a target application's code.

Concurrency Bug Detection Technique for Web Applications

There are bug detection techniques that identify suspicious code elements in a target web application that may cause concurrency errors through static analysis or dynamic analysis. Zheng et al. [25] propose a static analysis technique to detect possible atomicity violations in asynchronous communications (i.e., `XmlHttpRequest` operations) of web applications. WebRacer [17] uses a dynamic analysis technique to check whether two accesses to a DOM object (or a JavaScript function/variable) can occur in non-deterministic order, which is considered a concurrency bug. EventRacer [19] enhances WebRacer by improving computation efficiency and adding heuristics to prioritize bug reports.

As discussed in Section VI-A, bug detection techniques can generate spurious false alarms because these techniques do *not* actually check the results of target application executions. WAVE generates and tests various execution scenarios of a web application involving parsing operations, user actions, and network operations without generating false alarms.

Concurrent Test Generation for Multithreaded Program

There are techniques to generate various thread schedulings to detect concurrency bug for multi-threaded programs [7], [5], [14], [15], [16], [21], [24]. These techniques generate concurrent executions to achieve certain sequences of operations by controlling the execution orders of running threads. CalFuzzer [7], [15], [21], CTrigger [16], and Narayanasamy et al. [14] control thread scheduling to trigger suspicious interleaving scenarios. To test diverse behaviors of multi-threaded programs, Hong et al. [5] and Maple [24] generate concurrent executions to achieve high concurrent coverage. WAVE is similar to these techniques since Hong et al.'s technique, Maple, and WAVE generate various executions by controlling the execution order of threads/operations. However, WAVE utilizes a different concurrent execution model and different test generation targets specific for JavaScript applications.

VIII. Conclusion and Future Work

We have presented a testing framework, WAVE, that can find concurrency errors in JavaScript web applications. We demonstrate the effectiveness and the efficiency of WAVE in concurrency error detection through experiments on eight benchmark applications and five real-world applications. WAVE detected more concurrency errors nine times faster than the random network delay techniques. Furthermore, WAVE detected new concurrency errors in all five real-world web applications and the bug reports were highly valued by the developers of the applications. As future work, we will extend WAVE to test more concurrency features supported by HTML5 and other dynamic features of web applications.

Acknowledgement

We appreciate Dr. Young-Joo Moon, Prof. In Young Ko, and Prof. Sukyoung Ryu for valuable comments. This work was supported by the MKE, Korea and Microsoft under IT/SW Creative Research Program supervised by the NIPA (NIPA-2012-H0503-12-1006), the NRF Mid-career Researcher Program funded by the MSIP, Korea (2012R1A2A2A01046172), and the IT R&D Program of MKE/KEIT, Korea (10041752).

References

[1] Document Object Model (DOM) Level 2 Events Specification. http://www.w3.org/TR/DOM-Level-2-Events.

[2] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *ICSE*, 2011.

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA*, 2008.

[4] D. Flanagan. *JavaScript: The Definitive Guide, 6th ed.* O'Reilly, 2011.

[5] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *ISSTA*, 2012.

[6] J. Ide, R. Bodik, and D. Kimelman. Concurrency concerns in rich internet applications. In *Exploit. Concur. Effic. Correc. Works.*, 2009.

[7] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, 2009.

[8] I. Kantor. Events and timing in-depth, 2011. http://javascript.info/tutorial/events-and-timing-depth.

[9] O. J. Kjaer. Timing and synchronization in javascript, 2007. http://dev.opera.com/articles/view/timing-and-synchronization-in-javascript/.

[10] D. Lapper. Delaying a page load for Ajax testing. http://www.davelapper.com/blag/delaying-a-page-load-for-ajax-testing/.

[11] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *ICST*, 2008.

[12] A. Mesbah, A. V. Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. on the Web*, 6(1), Feb 2012.

[13] Mozilla Developer Network. Avoiding intermittent oranges. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Avoiding_intermittent_oranges.

[14] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007.

[15] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, 2008.

[16] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[17] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *PLDI*, 2012.

[18] T. Powell. *Ajax: The Complete Reference*. McGraw-Hill, Inc., 2008.

[19] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *OOPSLA*, 2013.

[20] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *SP*, 2010.

[21] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[22] S. Souders. *High Performance Web Sites*. OReilly, 2007.

[23] S. Souders. *Even Faster Web Sites*. OReilly, 2009.

[24] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *OOPSLA*, 2012.

[25] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *WWW*, 2011.