Understanding JavaScript Event-Based Interactions with Clematis

SABA ALIMADADI, SHELDON SEQUEIRA, ALI MESBAH,
and KARTHIK PATTABIRAMAN, University of British Columbia

Web applications have become one of the fastest-growing types of software systems today. Despite their popularity, understanding the behavior of modern web applications is still a challenging endeavor for developers during development and maintenance tasks. The challenges mainly stem from the dynamic, event-driven, and asynchronous nature of the JavaScript language. We propose a generic technique for capturing low-level event-based interactions in a web application and mapping those to a higher-level behavioral model. This model is then transformed into an interactive visualization, representing episodes of triggered causal and temporal events, related JavaScript code executions, and their impact on the dynamic DOM state. Our approach, implemented in a tool called CLEMATIS, allows developers to easily understand the complex dynamic behavior of their application at three different semantic levels of granularity. Furthermore, CLEMATIS helps developers bridge the gap between test cases and program code by localizing the fault related to a test assertion. The results of our industrial controlled experiment show that CLEMATIS is capable of improving the comprehension task accuracy by 157% while reducing the task completion time by 47%. A follow-up experiment reveals that CLEMATIS improves the fault localization accuracy of developers by a factor of two.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms: Design, Algorithms, Experimentation

Additional Key Words and Phrases: Program comprehension, event-based interactions, JavaScript, web applications, fault localization

ACM Reference Format:
Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding JavaScript event-based interactions with Clematis. ACM Trans. Softw. Eng. Methodol. 25, 2, Article 12 (May 2016), 38 pages.
DOI: http://dx.doi.org/10.1145/2876441

1. INTRODUCTION

JavaScript is widely used today to create interactive modern web applications that replace many traditional desktop applications. However, understanding the behavior of web applications is a challenging endeavor for developers [Oney and Myers 2009; Zaidman et al. 2013]. Program comprehension is known to be an essential step in software engineering, consuming up to 50% [Corbi 1989] of the effort in software maintenance and analysis activities.

This work was supported in part by an NSERC Strategic Project Grant and a research gift from Intel Corporation.
Authors' addresses: S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, Department of Electrical and Computer Engineering, University of British Columbia, 2332 Main Mall, V6T1Z4 Vancouver, BC, Canada; emails: {saba, sheldon, amesbah, karthikp}@ece.ubc.ca.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 1049-331X/2016/05-ART12 $15.00
DOI: http://dx.doi.org/10.1145/2876441

First, the weakly typed and highly dynamic nature of JavaScript makes it a particularly difficult language to analyze. Second, JavaScript code is extensively used to seamlessly mutate the Document Object Model (DOM) at runtime. This dynamic interplay between two separate entities, namely, JavaScript and the DOM, can become quite complex to follow [Ocariza et al. 2013]. Third, JavaScript is an event-driven language allowing developers to register various event listeners on DOM nodes. While most events are triggered by user actions, timing events and asynchronous callbacks can be fired with no direct input from the user. To make things even more complex, a single event can propagate on the DOM tree and trigger multiple listeners according to the event *capturing* and *bubbling* properties of the event model [W3C 2000].

Unfortunately, despite its importance and challenges, there is currently not much research dedicated to supporting program comprehension for web applications [Cornelissen et al. 2009]. Popular tools, such as Firebug and Chrome DevTools, are limited in their capabilities to support web developers effectively.

In this article, we present a generic, nonintrusive technique, called CLEMATIS, for supporting web application comprehension. Through a combination of automated JavaScript code instrumentation and transformation, we capture a detailed trace of a web application's behavior during a particular user session. Our technique transforms the trace into an abstract behavioral model, preserving temporal and causal relations within and between involved components. The model is then presented to the developers as an interactive visualization that depicts the creation and flow of triggered events, the corresponding executed JavaScript functions, and the mutated DOM nodes.

We then apply our approach to further aid developers in understanding root causes of test failures. Fault localization has been found to be one of the most difficult phases of debugging [Vessey 1985] and has been an active topic of research in the past [Agrawal et al. 1995; Cleve and Zeller 2005; Jones and Harrold 2005; Zeller 2002]. Although testing of modern web applications has received increasing attention in the recent past [Artzi et al. 2011; Mesbah et al. 2012; Thummalapenta et al. 2013], there has been limited work on what happens after a test reveals an error.

To the best of our knowledge, we are the first to provide a generic technique for capturing low-level event-based interactions in a JavaScript application, and mapping and visualizing those interactions as higher-level behavioral models. This article builds upon our previous work, where we proposed CLEMATIS and evaluated it through two user experiments [Alimadadi et al. 2014a]. In this article, we extend the approach and evaluation of CLEMATIS, as we propose a novel test failures comprehension unit and evaluate its effectiveness through a user experiment. Overall, our work makes the following key contributions:

- We propose a generic technique for capturing and presenting the complex dynamic behavior of web applications. In particular, our technique:
  - captures the consequences of JavaScript and DOM events in terms of the executed JavaScript code, including the functions that are called indirectly through event propagation on the DOM tree;
  - extracts the source-and-target relations for asynchronous events, that is, timing events and XMLHttpRequest requests/callbacks; and
  - identifies and tracks mutations to the DOM resulting from each event.
- We build a novel model for capturing the event-driven interactions as well as an interactive, visual interface supporting the comprehension of the program through three different semantic levels of zooming granularity.
- We implement our technique in a generic open-source tool called CLEMATIS, which (1) does not modify the web browser, (2) is independent of the server technology, and (3) requires no extra effort from the developer to use.

```
1 <BODY>
2   <FIELDSET class="registration">
3     Email: <INPUT type="text" id="email"/>
4     <BUTTON id="submitBtn">Submit</BUTTON>
5     <DIV id="regMsg"></DIV>
6   </FIELDSET>
7 </BODY>
```

Fig. 1. Initial DOM state of the running example.

—We extend CLEMATIS to automatically connect test assertion failures to faulty JavaScript code considering the involved DOM elements.
—We empirically evaluate CLEMATIS through three controlled experiments consisting of 48 users in total. The first two studies investigate the code comprehension capabilities of CLEMATIS. One of these studies is carried out in a lab environment, while the other is carried out in an industrial setting. The results of the industrial experiment show that CLEMATIS can reduce the task completion time by 47% while improving the accuracy by 157%. We evaluate the test failure comprehension unit of CLEMATIS through a third user experiment. The results show that CLEMATIS improves the fault localization accuracy of developers by a factor of two.

2. CHALLENGES AND MOTIVATION

Modern web applications are largely event driven. Their client-side execution is normally initiated in response to a user-action-triggered event, a timing event, or the receipt of an asynchronous callback message from the server. As a result, web developers encounter many program comprehension challenges in their daily development and maintenance activities. We use an example, presented in Figures 1 and 2, to illustrate these challenges.

Furthermore, developers often write test cases that assert the behavior of a web application from an end-user's perspective. However, when such test cases fail, it is difficult to relate the assertion failure to the faulty line of code. The challenges mainly stem from the existing disconnect between front-end test cases that assert the DOM and the application's underlying JavaScript code. We use another example, illustrated in Figure 3, to demonstrate these challenges. A different example was chosen to allow us to focus on challenges in understanding test failures.

Note that these are simple examples and these challenges are much more potent in large and complex web applications.

2.1. Challenge 1: Event Propagation

The DOM event model [W3C 2000] makes it possible for a single event, fired on a particular node, to propagate through the DOM tree hierarchy and indirectly trigger a series of other event handlers attached to other nodes. There are typically two types of this event propagation in web applications: (1) with *bubbling* enabled, an event first triggers the handler of the deepest child element on which the event was fired, and then it *bubbles* up on the DOM tree and triggers the parents' handlers; and (2) when *capturing* is enabled, the event is first *captured* by the parent element and then passed to the event handlers of children, with the deepest child element being the last. Hence, a series of lower-level event handlers, executed during the capturing and bubbling phases, may be triggered by a single user action. The existence or the ordering of these handlers is often inferred manually by the developer, which becomes more challenging as the size of the code/DOM tree increases.

```
1 $(document).ready(function() {
2     $('#submitBtn').click(submissionHandler);
3     $('fieldset.registration').click(function() {
4       setTimeout(clearMsg, 3000);
5 }); });
6 ...
7 function submissionHandler(e) {
8   $('#regMsg').html("Submitted!");
9   var email = $('#email').val();
10  if (isEmailValid(email)) {
11      informServer(email);
12      $('#submitBtn').attr("disabled", true);
13  }
14 }
15 ...
16 function informServer(email) {
17   $.get('/register/', { 'email': email }, function(data) {
18     $('#regMsg').append(data);
19   });
20   return;
21 }
22 ...
23 function clearMsg() {$('#regMsg').fadeOut(2000);}
```

Fig. 2. JavaScript code of the running example.

Example. Consider the sample code shown in Figures 1 and 2. Figure 1 represents the initial DOM structure of the application. It mainly consists of a `fieldset` containing a set of elements for the users to enter their email address to be registered for a service. The JavaScript code in Figure 2 partly handles this submission. When the user clicks the `submit` button, a message appears indicating that the submission was successful. This message is displayed from within the event handler `submissionHandler()` (line 7), which is attached to the button on line 2 of Figure 2. However, after a few seconds, the developer observes that the message unexpectedly starts to fade out. In the case of this simple example, he or she can read the whole code and find out that the click on the `submit` button has bubbled up to its parent element, namely, `fieldset`. Closer inspection reveals that `fieldset`'s anonymous handler function is responsible for changing the value of the same DOM element through a `setTimeout` function (lines 3–5 in Figure 2). In a more complex application, the developer may be unaware of the existence of the parent element, its registered handlers, or the complex event propagation mechanisms such as bubbling and capturing.

2.2. Challenge 2: Asynchronous Events

Web browsers provide a single thread for web application execution. To circumvent this limitation and build rich responsive web applications, developers take advantage of the asynchronous capabilities offered by modern browsers, such as *timeouts* and *XMLHttpRequest* (XHR) calls. Asynchronous programming, however, introduces an extra layer of complexity in the control flow of the application and adversely influences program comprehension.

Timeouts. Events can be registered to fire after a certain amount of time or at certain intervals in JavaScript. These timeouts often have asynchronous callbacks that are executed when triggered. In general, there is no easy way to link the callback of a timeout to its source, which is important to understand the program's flow of execution.

XHR Callbacks. XHR objects are used to exchange data asynchronously with the server, without requiring a page reload. Each XHR goes through three main phases:

open, send, and response. These three phases can be scattered throughout the code. Further, there is no guarantee on the timing and the order of XHR responses from the server. As in the case of timeouts, mapping the functionality triggered by a server response back to its source request is a challenging comprehension task for developers.

Example. Following the running example, the developer may wish to further investigate the unexpected behavior: the message has faded out without a direct action from the developer. The questions that a developer might ask at this point include “What exactly happened here?” and “What was the source of this behavior?” By reviewing the code, he or she can find out that the source of this behavior was the expiration of a timeout that was set in line 4 of Figure 2 by the anonymous handler defined in lines 3 through 5. However, the callback function, defined on line 23 of Figure 2, executes asynchronously and with a delay, long after the execution of the anonymous handler function has terminated. While in this case the timing behavior can be traced by reading the code, this approach is not practical for large applications. A similar problem exists for asynchronous XHR calls. For instance, the anonymous callback function of the request sent in the informServer function (line 17, Figure 2) updates the DOM (line 18).

2.3. Challenge 3: Implications of Events

Another challenge in understanding the flow of web applications lies in understanding the consequences of (in)directly triggered events. Handlers for a (propagated) DOM event, and callback functions of timeouts and XHR requests, are all JavaScript functions. Any of these functions may change the observable state of the application by modifying the DOM. Currently, developers need to read the code and make the connections mentally to see how an event affects the DOM state, which is quite challenging. In addition, there is no easy way of pinpointing the dynamic changes made to the DOM state as a result of event-based interactions. Inferring the implications of events is, therefore, a significant challenge for developers.

Example. After the submitBtn button is clicked in the running example, a confirmation message will appear on-screen and disappear shortly thereafter (lines 8 and 23, Figure 2). Additionally, the attributes of the button are altered to disable it (line 12). It can be difficult to follow such DOM-altering features in an application’s code.

2.4. Challenge 4: Linking Test Failures to Faults

To test their web applications, developers often write test cases that check the application’s behavior from an end-user’s perspective using popular frameworks such as Selenium.[1] Such test cases are agnostic of the JavaScript code and operate by simulating a series of user actions followed by assertions on the application’s runtime DOM. As such, they can detect deviations in the expected behavior as observed on the DOM.

However, when a web application test assertion fails, determining the faulty program code responsible for the failure can be a challenging endeavor. The main challenge here is the implicit link between three different entities, namely, the test assertion, the DOM elements on which the assertion fails (checked elements), and the faulty JavaScript code responsible for modifying those DOM elements. To understand the root cause of the assertion failure, the developer needs to manually infer a mental model of these hidden links, which can be tedious. Further, unlike in traditional (e.g., Java) applications, there is no useful stack trace produced when a web test case fails as the failure is on the DOM, and not on the application’s JavaScript code. This further hinders debugging as the fault usually lies within the application’s code, and not in its representative DOM

[1]http://seleniumhq.org.

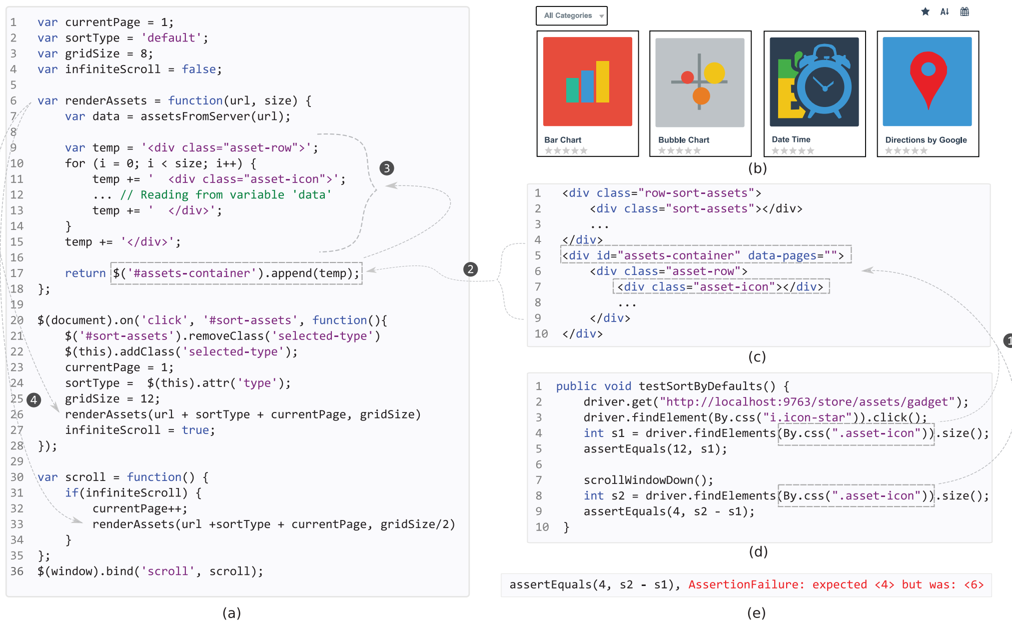

Fig. 3. Test assertion understanding example: (a) JavaScript code, (b) portion of DOM-based UI, (c) partial DOM, (d) DOM-based (Selenium) test case, and (e) test case assertion failure. The dotted lines show the links between the different entities that must be inferred.

state. To the best of our knowledge, there is currently no tool support available to help developers in this test failure understanding and fault localization process.

Example. The example in Figure 3 uses a small code snippet based on the open-source WSO2 eStore application[2] to demonstrate the challenges involved and our solution. The store allows clients to customize and deploy their own digital storefront. A partial DOM representation of the page is shown in Figure 3(c). Figure 3(d) shows a Selenium test case, written by the developers of the application for verifying the application's functionality in regards to "sorting" and "viewing" the existing assets. The JavaScript code responsible for implementing the functionality is shown in Figure 3(a).

After setting the environment, the test case performs a click to sort the *assets*. Then, an assertion is made to check whether the expected *assets* are present on the DOM of the page (line 5 of Figure 3(d)). The second portion of the test case involves scrolling down the webpage and asserting the existence of four additional *assets* on the DOM (lines 7–9).

While the mapping between the test case and related JavaScript code may seem trivial to find for this small example, challenges arise as the JavaScript code base and the DOM increase in size. As a result, it can be difficult to understand the reason for a test case failure or even which features are being tested by a given test case.

When a test case fails, first one needs to *identify the dependencies of the test case*. Based on the fail message in our example (Figure 3(e)), it is almost impossible to determine the cause of failure. Closer examination reveals the dependencies of assertions on variables `s1` and `s2`, which in turn depend on DOM elements with class `asset-icon` (link ❶ in Figure 3).

[2]https://github.com/wso2/product-es.

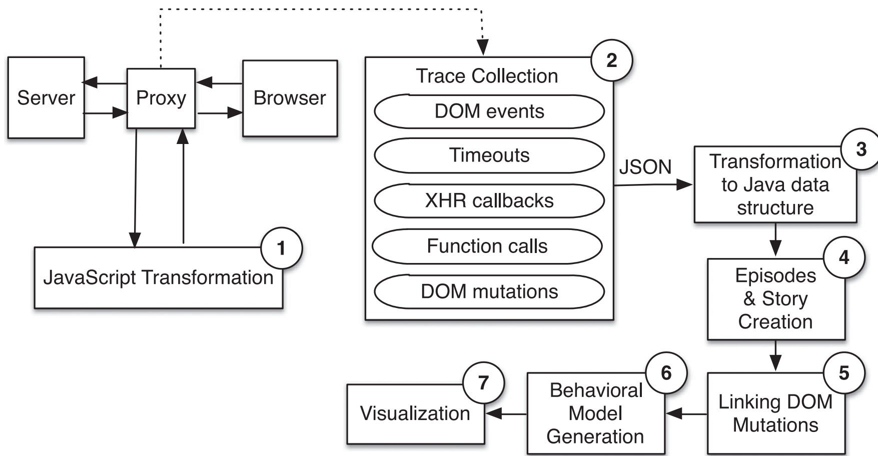

Fig. 4. A processing overview of our approach.

Next, the developer/tester is forced to *find the points of contact* between the DOM elements and the JavaScript code. Finding the JavaScript code responsible for modifying this subset of DOM elements is not easy. In the context of our example, a developer would eventually conclude that line 17 of Figure 3(a) is actually responsible for appending elements to the DOM. Discovering such implicit links (❷ and ❸ in Figure 3) needs tedious examination in smaller programs and may not be feasible in larger applications.

In JavaScript, events can trigger code execution and must be taken into account for finding the source of the fault. The `renderAssets()` function in our example (Figure 3) can be called from within two event handlers (lines 26 and 33, respectively, shown as ❹). While in our example it may be straightforward to link the call to `scrollWindowDown()` (line 7 of Figure 3(d)) to the execution of the event handler scroll (line 30–35 of Figure 3(a)) due to the similarity in naming convention, such a linear mapping is neither possible in all cases nor easily inferable.

Finally, to fully understand an assertion and its possible cause of failure, the data and control dependencies for the DOM-altering statements must be determined and examined by the developer in order to identify all possible points of failure. In the case of eStore, the modification of the DOM within `renderAssets()` depends on the arguments passed into the function (lines 7 and 10). Dotted line 4 shows possible invocations of `renderAssets()`, revealing dependencies on global variables such as *gridSize*. Tracing the dependencies reveals that an update to `gridSize` on line 25 of Figure 3(a) is the root cause of the unusual behavior.

3. APPROACH

In this section, we describe our approach for addressing the challenges mentioned in the previous section. An overview of the overall process is depicted in Figure 4, which consists of the following main steps:

—First, our technique captures a fine-grained trace of all semantically related event-based interactions within a web application's execution, in a particular user session. The collection of this detailed trace is enabled through a series of automated JavaScript transformations (Section 3.1).

—Next, a behavioral model is extracted from the information contained within the trace. The model structures the captured trace and identifies the implicit causal and temporal relationships between various event-based interactions (Section 3.2).
—Then, the model is extended through a combination of selective code instrumentation and dynamic backward slicing to bridge the gap between test cases and program code (Section 3.3).
—Finally, based on the inferred behavioral model, our approach generates an interactive (web-based) user interface, visualizing and connecting all the pieces together. This interactive visualization assists developers during their web application comprehension and maintenance tasks (Section 3.4).

We describe each step further next.

3.1. JavaScript Transformation and Tracing

To automatically trace semantically related event-based interactions and their impact, we transform the JavaScript code on the fly. Our approach generates a trace consisting of multiple trace units. A trace unit contains information acquired through the interception of a particular event-based interaction type, namely, DOM events, timing events, XHR calls and callbacks, function calls, and DOM mutations. The obtained trace is used to build a behavioral model (as described in Section 3).

Interposing on DOM Events. There are two ways event listeners can be bound to a DOM element in JavaScript. The first method is programmatically using the DOM Level 1 (`e.click=handler`) or DOM Level 2 (`e.addEventListener`) methods [W3C 2000] in JavaScript code. To record the occurrence of such events, our technique replaces the default registration of these JavaScript methods such that all event listeners are wrapped within a tracing function that logs the occurring event's time, type, and target.

The second and more traditional way to register an event listener is inline in the HTML code (e.g., `<DIV onclick='handler();'>`). The effect of this inline assignment is semantically the same as the first method. Our technique interposes on inline-registered listeners by removing them from their associated HTML elements, annotating the HTML elements, and re-registering them using the substituted `addEventListener` function. This way we can handle them similarly to the programmatically registered event handlers.

Capturing Timeouts and XHRs. For tracing timeouts, we replace the browser's `setTimeout()` method and the callback function of each timeout with wrapper functions, which allow us to track the instantiation and resolution of each timeout. A timeout callback usually happens later and triggers new behavior, and thus we consider it as a separate component than a `setTimeout()`. We link these together through a `timeout_id` and represent them as a causal connection later. In our model, we distinguish between three different components for the `open`, `send`, and `response` phases of each XHR object. We intercept each component by replacing the `XMLHttpRequest` object of the browser. The new object captures the information about each component while preserving its functionality.

Recording Function Traces. To track the flow of execution within a JavaScript-based application, we instrument three code constructs, namely, *function declarations*, *return statements*, and *function calls*. Each of these code constructs are instrumented differently, as explained next.

Function Declarations: Tracing code is automatically added to each function declaration, allowing us to track the flow of control between developer-defined functions by logging the subroutine's name, arguments, and line number. In case of anonymous functions, the line number and source file of the subroutine are used as supplementary information to identify the executed code.

```
1 function clearMsg() {
2   send(JSON.stringify({messageType: "FUNCTION_ENTER", fnName: "clearMsg",↩
         args: null, ...}));
3   $('#submissionMsg').fadeOut(2000);
4   send(JSON.stringify({messageType: "FUNCTION_EXIT", fnName: "clearMsg", ↩
        ...}));
5 }
```

Fig. 5. Instrumented JavaScript function declaration.

```
1 function informServer(email) {
2   $.get('/register/', { email }, function(data) {
3     $('#regMsg').append(data);
4   });
5   return RSW(null, 5);
6 }
```

Fig. 6. Instrumented JavaScript return statement.

As this communication is done each time a function is executed, argument values are recorded dynamically at the cost of a small overhead. Figure 5 contains the simple `clearMsg()` JavaScript function from the running example shown in Figure 2 (line 22), which has been instrumented to record both the beginning and end of its execution (lines 2 and 4).

Return Statements: Apart from reaching the end of a subroutine, control can be returned back to a calling function through a return statement. There are two reasons for instrumenting return statements: (1) to accurately track nested function calls and (2) to provide users with the line numbers of the executed return statements. Without recording the execution of return statements, it would be difficult to accurately track nested function calls. Furthermore, by recording return values and the line number of each return statement, Clematis is able to provide users with contextual information that can be useful during the debugging process.

Figure 6 illustrates the instrumentation for the return statement of `informServer()`, a function originally shown in the running example (Figure 2, lines 16–21). The wrapper function `RSW` receives the return value of the function and the line number of the return statement and is responsible for recording this information before execution of the application's JavaScript is resumed.

Function Calls: In order to report the source of a function invocation, our approach also instruments function calls. When instrumenting function calls, it is important to preserve both the order and context of each dynamic call. To accurately capture the function call hierarchy, we modify function calls with an inline wrapper function. This allows us to elegantly deal with two challenging scenarios. First, when multiple function calls are executed from within a single line of JavaScript code, it allows us to infer the order of these calls without the need for complex static analysis. Second, inline instrumentation enables us to capture nested function calls. Figure 7 depicts the instrumentation of function calls for two methods from Figure 1, `submissionHandler()` and `clearMsg()`.

Once instrumented using our technique, the function calls to `isEmailValid()` and `informServer()` are wrapped by function `FCW` (lines 4 and 5). The interposing function `FCW()` executes immediately before each of the original function calls and interlaces our function logging with the application's original behavior. Class methods `html()`, `value()`, `attr()`, and `fadeOut()` are also instrumented in a similar way (lines 2, 3, 6, and 10 respectively).

For comparison, an alternative instrumentation technique is shown on lines 8 through 10 of Figure 8. While such a technique might be sufficient for measuring

```
1 function submissionHandler(e) {
2   $('#regMsg')[FCW("html")]("Submitted!");
3   var email = $('#email')[FCW("value")]();
4   if (FCW(isEmailValid)(email)) {
5     FCW(informServer)(email);
6     $('#submitBtn')[FCW("attr")]("disabled", true);
7   }
8 }
9 function clearMsg() {
10   $('#regMsg')[FCW("fadeOut")](2000);
11 }
12 function FCW(fnName) { // Function Call Wrapper
13     send(JSON.stringify({messageType: "FUNCTION_CALL",...,targetFunction:↩
         fnName}));
14     return fnName;
15 }
```

Fig. 7. Instrumented JavaScript function calls.

```
1 Before Instrumentation:
2   getRegistrationDate(getStudentNumber(document.getElementById('username'↩
      ).value));

4 After Clematis Instrumentation:
5   FCW(getRegistrationDate)(FCW(getStudentNumber)(document↩
      FCW("getElementById")('username').value));

7 Alternative Instrumentation:
8   FCW(getRegistrationDate);
9   FCW(getStudentNumber);
10  FCW(getElementById);
11  getRegistrationDate(getStudentNumber(document.getElementById('username'↩
      ).value));
```

Fig. 8. Comparison of instrumentation techniques for JavaScript function calls.

function coverage, it does not capture the order of execution accurately for nested function calls or when multiple function calls are made from a single line. Doing so would require more complex static analysis.

DOM Mutations. Information about DOM mutations can help developers relate the observable changes of an application to the corresponding events and JavaScript code. To capture this important information, we introduce an observer module into the system. This information is interleaved with the logged information about events and functions, enabling us to link DOM changes with the JavaScript code that is responsible for these mutations.

3.2. Capturing a Behavioral Model

We use a graph-based model to capture and represent a web application's event-based interactions. The graph is multiedge and directed. It contains an ordered set of nodes, called *episodes*, linked through edges that preserve the chronological order of event executions.[3] In addition, causal edges between the nodes represent asynchronous events. We describe the components of the graph next.

Episode Nodes. An episode is a semantically meaningful part of the application behavior, initiated by a synchronous or an asynchronous event. An event may lead to the execution of JavaScript code and may change the DOM state of the application. An

[3]Because JavaScript is single threaded on all browsers, the events are totally ordered in time.

episode node contains information about the static and dynamic characteristics of the application and consists of three main parts:

(1) *Source:* This is the event that started the episode, and its contextual information. This source event is either a DOM event, a timeout callback, or a response to an XHR request, and often causes a part of the JavaScript code to be executed.
(2) *Trace:* This includes all the functions that are executed either directly or indirectly after the source event occurs. A direct execution corresponds to functions that are called from within an event handler on a DOM element. An indirect execution corresponds to functions that get called due to the bubbling and capturing propagation of DOM events. The trace also includes all (a)synchronous events that were created within the episode. All the invoked functions and initiated events are captured in the trace part, and their original order of execution and dependency relations are preserved.
(3) *Result:* This is a section in each episode summarizing the changes to the DOM state of the application. These changes are caused by the execution of the episode trace and are usually observable by the end-user.

Edges. In our model, edges represent a progression of time and are used to connect episode nodes. Two types of edges are present in the model:

(1) *Temporal:* The temporal edges connect one episode node to another, indicating that an episode succeeded the previous one in time.
(2) *Causal:* These edges are used to connect different components of an asynchronous event (e.g., timeouts and XHRs). A causal edge from episode `s` to `d` indicates episode `s` was caused by episode `d` in the past.

Story. The term “story” refers to an arrangement of episode nodes encapsulating a sequence of interactions with a web application. Different stories can be captured according to different features, goals, or use cases that need investigation.

Algorithm 1 takes the trace collected from a web application as input and outputs a story with episodes and the edges between them. First, the trace units are extracted and sorted based on the timestamp of their occurrence (line 3). Next, the algorithm iteratively forms new episodes and assigns trace units to the source, the trace, and the result fields of individual episodes. If it encounters a trace unit that could be an episode source (i.e., an event handler, a timeout, or an XHR callback), a new episode is created (lines 5 and 6) and added to the list of nodes in the story graph (line 8). The encountered trace unit is added to the episode as its source (line 7). Line 9 shows different types of trace units that could be added to the trace field of the episode. This trace is later processed to form the complete function call hierarchy as well as each function’s relation with the events inside that episode. Next, DOM mutation units that were interleaved with other trace units are organized and linked to their respective episode (lines 11 and 12). An episode terminates semantically when the execution of the JavaScript code related to that episode is finished. The algorithm also waits for a time interval τ to ensure that the execution of *immediate* asynchronous callbacks is completed (line 13). When all of the trace units associated with the source, trace, and result of the episode are assigned and the episode termination criteria are met, a temporal edge is added to connect the recently created episode node to the previous one (line 14). The same process is repeated for all episodes by proceeding to the next episode captured in the trace (line 15). After all episodes have been formed, the linkages between *distant* asynchronous callbacks—those that did not complete *immediately*—are extracted and added to the graph as causal edges (lines 16–18). Finally, the story is created based on the whole graph and returned (lines 19 and 20).

ALGORITHM 1: Story Creation

```
    input: trace
    output: story
    Procedure CreateModel() begin
1       G < V, E > story ← ∅
2       e_curr, e_prev ← ∅
3       Σtu ← ExtractAndSortTraceUnits(trace)
4       foreach tu ∈ Σtu do
5           if e_prev ≡ ∅||e_prev.ended()&&
            tu.type ≡ episodeSource then
6               e_curr ← CreateEpisode()
7               e_curr.source ← SetEpisodeSource(tu)
8               V ← V ∪ e_curr
9           else if (tu.type ≡ FunctionTrace||EventHandler) ||
            (tu.type ≡ XHRCallback||TimeoutCallback
            && ¬episodeEndCriteria) then
10              e_curr.trace ← e_curr.trace ∪ tu
11          else if tu.type ≡ DOMMutation then
12              e_curr.results ← e_curr.results ∪ tu
13          if episodeEndCriteriaSatisfied then
14              E ← E ∪ createTemporalLink(e_prev, e_curr)
15              e_prev ← e_curr
16      timeoutMap<TimeoutSet, TimeoutCallback> ← MapTimeoutTraceUnits(Σtu)
17      XHRMap<XHROpen, XHRSend, XHRCallback> ← MapXHRTraceUnits(Σtu)
18      E ← E ∪ ExtractCausalLinks(timeoutMap, XHRMap)
19      story ← BuildStory(G < V, E >)
20      return story
```

3.3. Understanding Test Assertion Failures

In this section, we extend Clematis to further assist developers in the comprehension process. We add a test case comprehension strategy to Clematis to help developers understand the root cause of a test failure. Our technique automatically links a test assertion failure to the checked DOM elements, and subsequently to the related statements in the JavaScript code. The following subsections describe our strategies for fulfilling the aforementioned requirements of JavaScript test failure comprehension.

Relating Test Assertions to DOM Elements. The DOM acts as the interface between a front-end test case and the JavaScript code. Therefore, the first step to understanding the cause for a test case failure is to determine the DOM dependencies for each test assertion. While this seems simple in theory, in practice, assertions and element accesses are often intertwined within a single test case, convoluting the mapping between the two.

Going back to the test case of our example in Figure 3(d), the first assertion on line 5 is dependent on the DOM elements returned by the access on the previous line. The last assertion on line 9 is more complex as it compares two snapshots of the DOM and therefore has dependencies on two DOM accesses (lines 4 and 8). Figure 9 summarizes the test case's execution and captures the temporal and causal relations between each assertion and DOM access.

To accurately determine the DOM dependencies of each assertion (❶ in Figure 3), we apply dynamic backward slicing to each test case assertion. In addition, we track the runtime properties of those DOM elements accessed by the test case. This runtime information is later used in our analysis of the DOM dependencies of each assertion.

Contextualizing Test Case Assertion. In the second step, our approach aims to (1) help developers understand the context of their assertions by monitoring

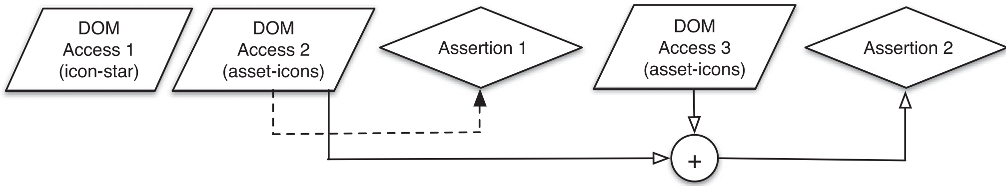

Fig. 9. Relating assertions to DOM accesses for the test case of Figure 3(d).

test-related JavaScript execution, asynchronous events, and DOM mutations and (2) determine the initial link between JavaScript code and the checked DOM elements (❷ in Figure 3).

In order to monitor JavaScript events, we leverage the tracing technique outlined in Section 3.1, which tracks the occurrence of JavaScript events, function invocations, and DOM mutations. We utilize the tracked mutations in order to focus on the segments of JavaScript execution most relevant to the assertions in a test case. As we are only interested in the subset of the DOM relevant to each test case, our approach focuses on the JavaScript code that interacts with this subset.

The previous step yields the set of DOM elements relevant to each assertion. We cross-reference these sets with the timestamped DOM mutations in our execution trace extracted from CLEMATIS to determine the JavaScript functions and events (DOM, timing, or XHR) relevant to each assertion.

Once the relevant events and JavaScript functions have been identified for each assertion, we introduce wrapper functions for the native JavaScript functions used by developers to retrieve DOM elements. Specifically, we redefine methods such as `getElementById` and `getElementsByClassName` to track DOM accesses within the web application itself so that we know exactly where in our collected execution trace the mutation originated. The objects returned by these methods are used by the application later to update the DOM. Therefore, we compute the forward slice of these objects to determine the exact JavaScript lines responsible for updating the DOM. Henceforth, we refer to the references returned by these native methods as *JavaScript DOM accesses*.

We compare the recorded JavaScript DOM accesses with the DOM dependencies of each test case assertion to find the *equivalent* JavaScript DOM accesses within the application's code. Moreover, the *ancestors* of those elements accessed by each assertion are also compared with the recorded JavaScript DOM accesses. This is important because in many cases, a direct link might not exist between them. For instance, in the case of our example (Figure 3(d)), a group of *assets* is compiled and appended to the DOM after a `scroll` event. We compare the properties of those DOM elements accessed by the final assertion (*assets* on lines 4 and 8 of Figure 3(d)), as well as the properties of those elements' ancestors, with the recorded JavaScript DOM accesses and conclude that the *assets* were added to the DOM via the *parent* element *assets container* on line 17 of Figure 3(a) (❷).

Slicing the JavaScript Code. At this point, our approach yields the set of JavaScript statements responsible for updating the DOM dependencies of our test case. However, the set in isolation seldom contains the cause of a test failure. We compute a backward slice for these DOM-mutating statements to find the entire set of statements that perform the DOM mutation.

In our approach, we have opted for dynamic slicing, which enables us to produce thinner slices that are representative of each test execution, thus reducing noise during the debugging process. The slices incorporate data and control dependencies derived from the application. Moreover, by using dynamic analysis, we are able to present the user with valuable runtime information that would not be available through static analysis of JavaScript code.

Selective Instrumentation. An ideal test case would minimize setup by exercising only the relevant JavaScript code related to its assertions. However, developers are often unaware of the complete inner workings of the application under test. As a result, it is possible for a test case to execute JavaScript code that is unrelated to any of its contained assertions. In such a case, instrumenting an entire web application's JavaScript code base would yield a large trace with unnecessary information. This can incur high performance overheads, which may change the web application's behavior. Therefore, instead of instrumenting the entirety of the code for dynamic slicing, our approach intercepts and statically analyzes all JavaScript code sent from the server to the client to determine which statements may influence the asserted DOM elements. Then, this subset of the application's code is instrumented. This approach has two advantages. First, it minimizes the impact our code instrumentation has on the application's performance. Second, selective instrumentation yields a more relevant and concise execution trace, which in turn lowers the processing time required to compute a backward slice.

Our approach first converts the code into an abstract syntax tree (AST). This tree is traversed in search of a node matching the initial slicing criteria. Once found, the function containing the initial definition of the variable in question is also found, henceforth referred to as the *parent closure*. Based on this information, the algorithm searches this parent closure for all references to the variable of interest. This is done in order to find all locations in the JavaScript code where the variable may be updated, or where a new alias may be created for the variable. Moreover, for each variable update pertaining to the variable of interest, we also track the data dependencies for such an operation. Repeating these described steps for each of the detected dependencies allows us to iteratively determine the subset of code statements to efficiently instrument for a given initial slicing criteria.

Once all possible data and control dependencies have been determined through static analysis, each variable and its parent closure are forwarded to our code transformation module, which instruments the application code in order to collect a concise trace. The instrumented code keeps track of all updates and accesses to all relevant data and control dependencies, hereby referred to as *write* and *read* operations, respectively. This trace is later used to extract a dynamic backward slice.

Figure 10 shows an example of our code instrumentation technique's output when applied to the JavaScript code in Figure 3(a) with slicing criteria <10, `size`>. By acting as a control dependency for variable `temp`, `size` determines the number of displayed *assets* for the example. For each relevant write operation, our instrumentation code logs information such as the name of the variable being written to, the line number of the executed statement, and the type of value being assigned to the variable. Moreover, the data dependencies for such a write operation are also logged. Likewise, for each read operation, we record the name of the variable being read, the type of value read, and the line number of the statement. Information about variable type is important when performing alias analysis during the computation of a slice.

Computing a Backward Slice. Once a trace is collected from the selectively instrumented application by running the test case, we run our dynamic slicing algorithm. We use dynamic slicing as it is much more accurate than static slicing at capturing the exact set of dependencies exercised by the test case.

The task of slicing is complicated by the presence of aliases in JavaScript. When computing the slice of a variable that has been assigned a nonprimitive value, we need to consider possible aliases that may refer to the same object in memory. This also occurs in other languages such as C and Java; however, specific to JavaScript is the use of the *dot notation*, which can be used to seamlessly modify objects at runtime.

```
1   var currentPage = 1;
2   var sortType = 'default';
3   var gridSize = _write("gridSize", 8, 3);
4   var infiniteScroll = false;
5
6   var renderAssets = function(url, size) {
7       var data = assetsFromServer(url);
8
9       var temp = '<div class="asset-row">';
10      for (i = 0; i < _read("size", size, 10); i++) {
11          temp += '  <div class="asset-icon">';
12          ... // Reading from variable 'data'
13          temp += '  </div>';
14      }
15      temp += '</div>';
16
17      return $('#assets-container').append(temp);
18  };
19
20  $(document).on('click', '#sort-assets', function(){
21      $('#sort-assets').removeClass('selected-type')
22      $(this).addClass('selected-type');
23      currentPage = 1;
24      sortType =  $(this).attr('type');
25      gridSize = _write("gridSize", 12, 25);
26      renderAssets(url + sortType + currentPage, _readAsArg("gridSize", gridSize, 26));
27      infiniteScroll = true;
28  });
29
30  var scroll = function() {
31      if(infiniteScroll) {
32          currentPage++;
33          renderAssets(url +sortType +currentPage, _readAsArg("gridSize", gridSize, 33)/2);
34      }
35  };
36  $(window).bind('scroll', scroll);
```

Fig. 10. Example JavaScript code after our selective instrumentation is applied. Slicing criteria: <10, `size`>.

The prevalent use of aliases and the dot notation in web applications often complicates the issue of code comprehension. Static analysis techniques often ignore addressing this issue [Feldthaus et al. 2013].

To remedy this issue, we incorporate dynamic analysis in our slicing method. If a reference to an object of interest is saved to a second object's property, possibly through the use of the *dot notation*, the object of interest may also be altered via aliases of the second object. For example, after executing statement `a.b.c = objOfInterest;`, updates to `objOfInterest` may be possible through `a`, `a.b`, or `a.b.c`. To deal with this and other similar scenarios, our slicing algorithm searches through the collected trace and adds the forward slice for each detected alias to the current slice for our variable of interest (e.g., `objOfInterest`).

The line numbers for each of the identified relevant statements in the computed slice are collected and used during the visualization step, as shown in Section 3.4.

3.4. Visualizing the Captured Model

In the final step, our technique produces an interactive visualization of the generated model, which can be used by developers to understand the behavior of the application.

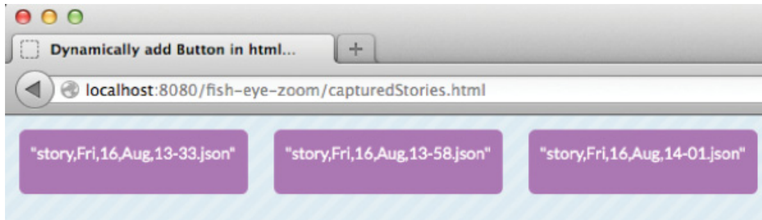

Fig. 11. Overview of all captured stories.

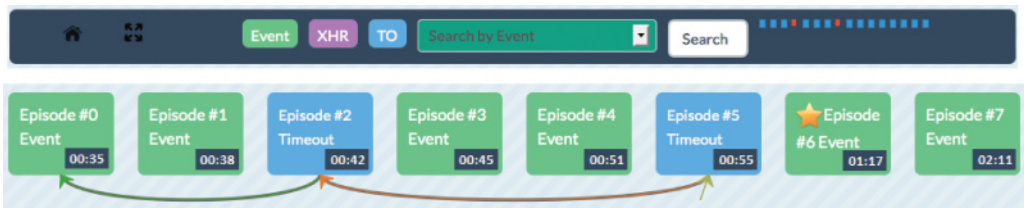

Fig. 12. Top: menu of Clematis. Bottom: overview of a captured story.

The main challenge in the visualization is to provide a way to display the model without overwhelming the developer with the details. To this end, our visualization follows a focus+context [Cockburn et al. 2009] technique that provides the details based on a user's demand. The idea is to start with an overview of the captured story, let the users determine which episode they are interested in, and provide an easy means to drill down to the episode of interest. With integration of focus within the context, developers can semantically zoom in to each episode to gain more details regarding that episode, while preserving the contextual information about the story.

Multiple Sessions, Multiple Stories. The user can capture multiple sessions, which leads to creation of multiple stories. After each story is recorded, it will be added to the list of captured stories. The name of each story is the date and time at which it was captured. Figure 11 shows a screenshot of sample captured stories in the visualization of Clematis. Once the users select their desired story, the browser opens a new page dedicated to that story. The initial view of a story contains a menu bar that helps the user navigate the visualization (Figure 12, top). It also displays an overview of all captured episodes inside the story (Figure 12, bottom).

Story Map, Queries, and Bookmarking. A menu bar is designed for the visualization that contains two main parts: the *story map* and the *query mechanism* (Figure 12, top). The story map represents a general overview of the whole story as a roadmap. Panning and (semantic) zooming are available for all episodes and may cause users to lose the general overview of the story. Hence, based on the user's interaction with the story (e.g., episode selection), the episodes of interest are highlighted on the roadmap to guide the user. The query section enables users to search and filter the information visualized on the screen. Users can filter the episodes displayed on the screen by the episode types (i.e., Event, Timeout, or XHR). They can also search the textual content of the events as well as the actual code. Moreover, they have the option to bookmark one or more episodes while interacting with the target web application. Those episodes are marked with a star in the visualization to help users to narrow the scope and spot related episodes (e.g., episode #6 in Figure 12 is bookmarked). The episodes' timing information is also shown.

Semantic Zoom Levels. The visualization provides three semantic zoom levels.

Zoom Level 0. The first level displays all of the episodes in an abstracted manner, showing only the type and the timestamp of each episode (Figure 12, bottom). The type of each episode is displayed by the text of the episode as well as its background color. The horizontal axis is dedicated to time, and episodes are sorted from left to right according to the time of their occurrence (temporal relations). The causal edges between different sections of each timeout or XHR object are shown by additional edges under the episodes.

Zoom Level 1. When an episode is selected, the view transitions into the second zoom level, which presents an outline of the selected episode, providing more information about the source event as well as a high-level trace (Figure 13, middle). The trace at this level contains only the names of the (1) invoked functions, (2) triggered events, and (3) DOM mutations, caused directly or indirectly by the source event. At this level, the user can view multiple episodes to have a side-by-side comparison.

Zoom Level 2. The final zoom level exhibits all the information embedded in each episode (Figure 13, bottom). Clicking on the "Event" tab will display the type of the event that started the episode (DOM, timeout, or XHR event). The contextual information of the event is displayed based on its type. Choosing the "DOM mutations" tab will list all the changes that were made to the DOM after the execution of this episode. For each DOM element that was added, removed, or modified, an item is added to the list of mutations that identifies the modified element, the type of the change, and additional information about the change. The third and final tab depicts a detailed trace of the episode. The trace at this level includes a customized sequence diagram of the dynamic flow of all the invoked JavaScript functions and events within that episode. When the user clicks on any of the functions or events in the diagram, the JavaScript code of each executed function is displayed and highlighted (Figure 13, bottom).

Inferred Mappings Between Test Failures and Code. The test case comprehension unit extends the interactive visualization to depict the inferred mappings for the test failure. The visualization helps to understand (1) the client-side JavaScript code related to the assertion failure, (2) the test case's relations to DOM changes and JavaScript execution, and/or (3) any deviations in the expected behavior with respect to a previous version where the test passed. Figure 14 depicts an example of the high-level view provided by our visualization for a test case.

In the high-level view, the progress of an executed test case over time is depicted on the horizontal axis, where the earliest assertions are shown on the left-hand side of the high-level view and the most recent JavaScript events and assertions are shown closer to the right-hand side. The top of Figure 14(b) shows the high-level visualization produced by running the same test case from Figure 14(a) on a faulty version of the application. Passing assertions for a test case are represented as gray nodes, and failures are shown in red. In the case of an assertion, causal links relate the assertion to prior events that may have influenced its outcome. These are events that altered portions of the DOM relevant to the assertion. DOM events, timing events, and network-related JavaScript events are visualized alongside the assertions as green, purple, and blue nodes, respectively.

Clicking on a failed assertion node reveals additional details about it (Figure 14(b)). Details include related (1) DOM dependencies, (2) failure messages, and (3) related JavaScript functions. The final zoom level of an assertion node displays all the information captured for the assertion including the captured slice and the line numbers of the failing test case assertions.

When displaying the code slice for an assertion, each line of JavaScript code that may have influenced the assertion's outcome is highlighted in the context of the source

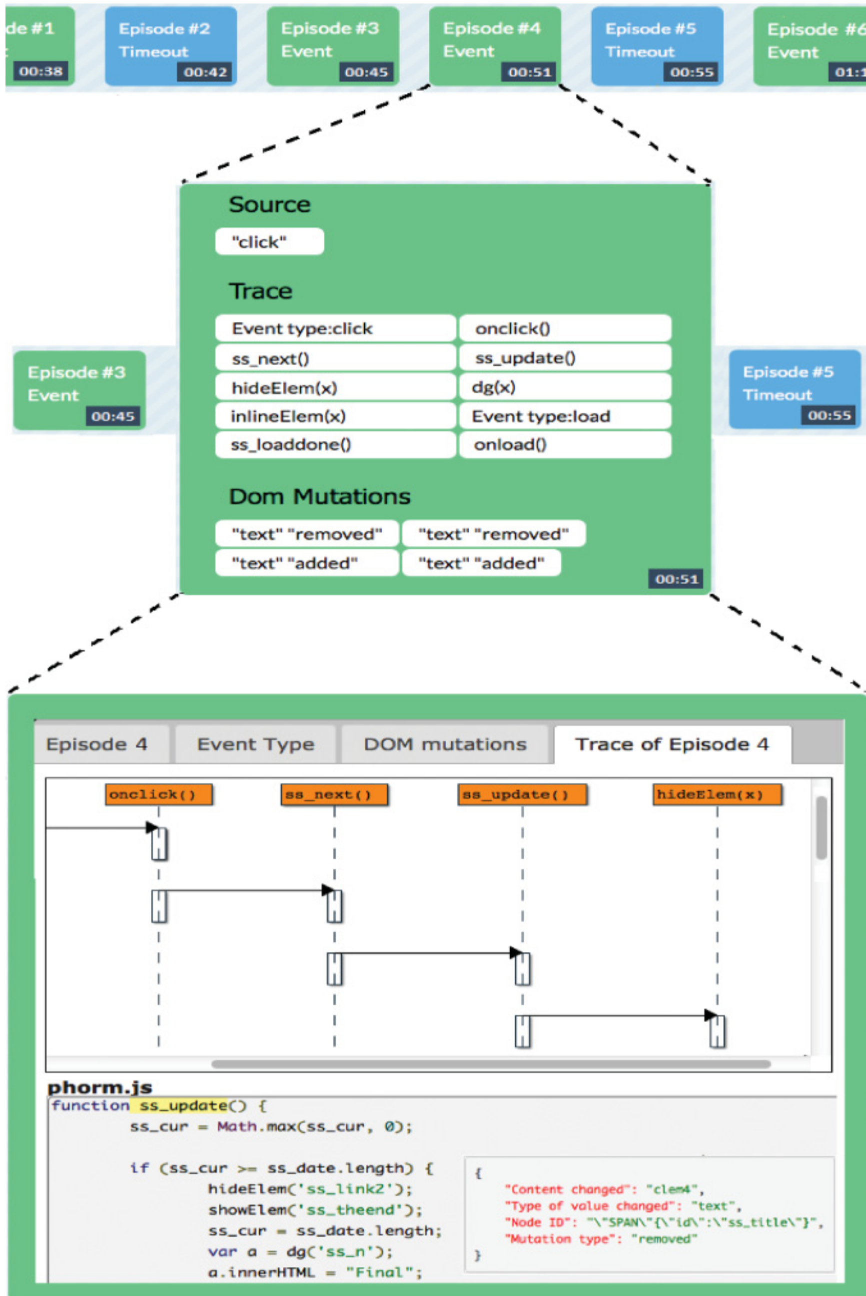

Fig. 13. Three semantic zoom levels in Clematis. Top: overview. Middle: zoomed one level into an episode, while preserving the context of the story. Bottom: drilled down into the selected episode.

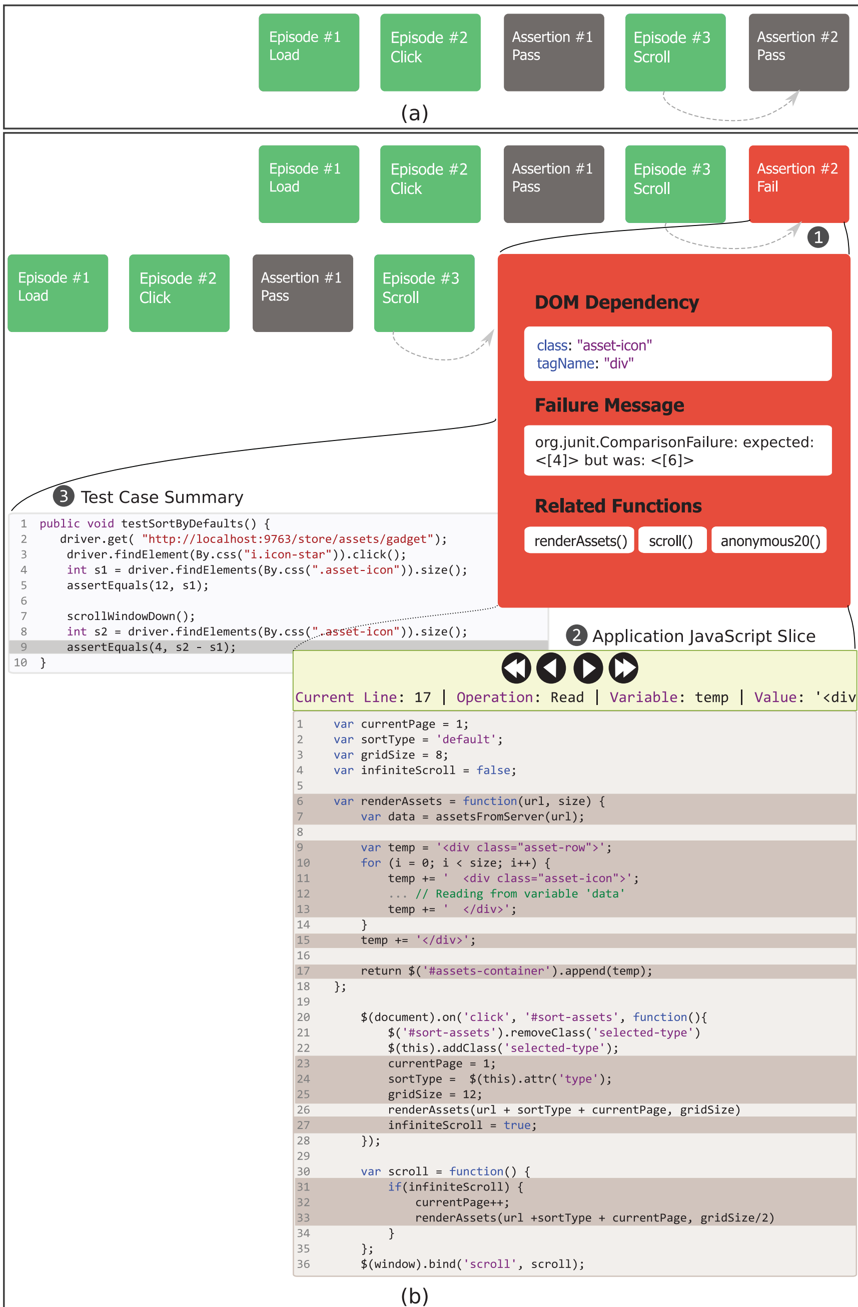

Fig. 14. Visualization for a test case. (a) Overview of the passing test case. (b) Three semantic zoom levels for the failing test case: Top: overview. Middle: second zoom level showing assertion details, while preserving the context. Bottom: summary of failing assertion and the backward slice.

code (Figure 14(b), lower right). The user can further explore the captured slice by stepping through its recorded execution using a provided control panel, shown in green on Figure 14(b). By doing so, the user is able to take a postmortem approach to fault localization whereby the faulty behavior is studied deterministically offline after execution has completed. Further, the user can also examine the captured runtime values of relevant JavaScript variables.

RESTful API. We deployed a RESTful API that provides access to details about captured stories and allows the approach to remain portable and scalable. This architectural decision enables all users, independent of their environments, to take advantage of the behavioral model. By invoking authorized calls to the API, one can represent the model as a custom visualization or use it as a service in the logic of a separate application.

3.5. Tool Implementation: Clematis

We implemented our approach in a tool called CLEMATIS, which is freely available.[4] We use a proxy server to automatically intercept and inspect HTTP responses destined for the client's browser. When a response contains JavaScript code, it is transformed by CLEMATIS. We also use the proxy to inject a JavaScript-based toolbar into the web application, which allows the user to start/stop capturing his or her interactions with the application. We used a proxy since it leads to a nonintrusive instrumentation of the code. A browser plugin would be a suitable alternative. However, unlike browser plugins, a proxy-based approach does not require installing a plugin, is not dependent on the type of the browser, and does not need to be maintained and updated based on browser updates. The trace data collected is periodically transmitted from the browser to the proxy server in JSON format. To observe low-level DOM mutations, we build on and extend the JavaScript Mutation Summary library.[5] The model is automatically visualized as a web-based interactive interface. Our current implementation does not capture the execution of JavaScript code that is evaluated using `eval`. CLEMATIS provides access to details of captured stories through a RESTful API.

4. CONTROLLED EXPERIMENTS

To assess the efficacy of our program comprehension approach, we conducted two controlled experiments, following guidelines by Wohlin et al. [2000], one in a research lab setting and the other in an industrial environment. In addition, to assess the test failure comprehension extension of CLEMATIS, we conduct a third controlled experiment.

Common design elements of all experiments are described in this section. Sections 5 through 7 are dedicated to describing the specific characteristics and results of each experiment, separately.

Our evaluation aims at addressing the following research questions. The first four research questions are designed to evaluate the main code comprehension unit of CLEMATIS. These questions are investigated in the first two experiments (Section 5–6). RQ5, however, assesses the extended test failure comprehension unit of CLEMATIS (Section 7). In order to be able to maintain the duration of experiment sessions reasonably, we decided to evaluate the test comprehension unit separately.

RQ1. Does CLEMATIS decrease the task completion *duration* for common tasks in web application comprehension?

RQ2. Does CLEMATIS increase the task completion *accuracy* for common tasks in web application comprehension?

[4]http://salt.ece.ubc.ca/software/clematis/.

[5]http://code.google.com/p/mutation-summary/.

Table I. Adopted and Adapted Comprehension Activities

Activity	Description
A1	Investigating the functionality of (a part of) the system
A2	Adding to/changing the system's functionality
A3	Investigating the internal structure of an artifact
A4	Investigating the dependencies between two artifacts
A5	Investigating the runtime interaction in the system
A6	Investigating how much an artifact is used
A7	Investigating the asynchronous aspects of JavaScript
A8	Investigating the hidden control flow of event handling

Table II. Comprehension Tasks Used in Study 1

Task	Description	Activity
T1	Locating the implementation of a feature modifying the DOM	A1, A4
T2	Finding the functions called after a DOM event (nested calls)	A1, A4, A5
T3.a	Locating the place to add a new functionality to a function	A2, A3
T3.b	Finding the caller of a function	A4, A5
T4.a	Finding the functions called after a DOM event (nested calls + bubbling)	A1, A4, A5
T4.b	Locating the implementation of a UI behavior	A1, A3, A4
T5.a	Finding the functions called after a DOM event (bubbling + capturing)	A1, A5, A8
T5.b	Finding the changes to DOM resulting from a user action	A4, A5
T6.a	Finding the total number of sent XHRs	A6, A7
T6.b	Finding if there exists an unresponded XHR	A4, A5, A7

RQ3. For what types of tasks is CLEMATIS most effective?
RQ4. What is the performance overhead of using CLEMATIS? Is the overall performance acceptable?
RQ5. Is the test failure comprehension unit helpful in localizing (and repairing) JavaScript faults detected by test cases?

4.1. Experimental Design

The experiments had a "between-subject" design; that is, the subjects were divided into two groups: experimental group using CLEMATIS and control group using other tools. The assignment of participants to groups was done manually, based on the level of their expertise in web development. We used a 5-point Likert scale in a prequestionnaire to collect this information and distributed the level of expertise in a balanced manner between the two groups. None of the participants had any previous experience with CLEMATIS, and all of them volunteered for the study.

Task Design. The subjects were required to perform a set of tasks during the experiment, representing tasks normally used in software comprehension and maintenance efforts. We adapted the activities proposed by Pacione et al. [2004], which cover categories of common tasks in program comprehension, to web applications by replacing two items. The revised activities are shown in Table I. We designed a set of tasks for each experiment to cover these activities. Tables II and III show the tasks for studies 1 and 2 accordingly. Because study 2 was conducted in an industrial setting, participants had limited time. Therefore, we designed fewer tasks for this study compared to study 1. Table IV depicts the tasks used in study 3, which aims the fault localization capabilities of CLEMATIS.

Independent Variable (IV). This is the tool used for performing the tasks and has two levels: CLEMATIS represents one level, and other tools used in the experiment represent the other level (e.g., Chrome developer tools, Firefox developer tools, Firebug).

Dependent Variables (DV). These are (1) task completion *duration*, which is a continuous variable, and (2) *accuracy* of task completion, which is a discrete variable.

Data Analysis. For analyzing the results of each study, we use two types of statistical tests to compare dependent variables across the control and experimental groups. Independent-sample t-tests with unequal variances are used for duration and accuracy in study 1, and for duration in study 2. However, the accuracy data in study 2 was not normally distributed, and hence we use a *Mann-Whitney U* test for the analysis of accuracy in this study. We use the statistical analysis package R[6] for the analysis.

4.2. Experimental Procedure

All experiments consisted of four main phases. First, the subjects were asked to fill out a prequestionnaire regarding their expertise in the fields related to this study.

In the next phase, the participants in the experimental group were given a tutorial on CLEMATIS. They were then given a few minutes to familiarize themselves with the tool after the tutorial.

In the third phase, each subject performed a set of tasks, as outlined in Tables II and III. Each task was given to a participant on a separate sheet of paper, which included instructions for the task and had room for the participant's answer. Once completed, the form was to be returned immediately and the subject was given the next task sheet. This allowed us to measure each task's completion time accurately, to answer RQ1 and RQ3. To address RQ2 and RQ3, the accuracy of each task was later evaluated and marked from 0 to 100 according to a rubric that we had created prior to conducting the experiment. The design of the tasks allowed the accuracy of the results to be quantified numerically. The tasks and sample rubrics are available in our technical report Alimadadi et al. [2014b].

In the final phase, participants filled out a postquestionnaire form providing feedback on their experience with the tool used (e.g., limitations, strength, usability).

5. EXPERIMENT 1: LAB ENVIRONMENT

The first controlled experiment was conducted in a lab setting with students and postdocs at the University of British Columbia (UBC).

5.1. Approach

Experimental Design. For this experiment, both groups used Mozilla Firefox 19.0. The control group used Firebug 1.11.2. We chose Firebug in the control group since it is the de facto tool used for understanding, editing, and debugging modern web applications.[7] Firebug has been used in other similar studies [Zaidman et al. 2013].

Experimental Subjects. We recruited 16 participants for the study, three females and 13 males. The participants were drawn from different educational levels: two undergraduate students, five master's students, eight PhD students, and one postdoctoral fellow, at UBC. The participants represented different areas of software and web engineering and had skills in web development ranging from beginner to professional. The tasks used in this study are enumerated in Table II.

Experimental Object. We decided to use a web-based survey application that was developed in our lab. The application had modest size and complexity, so that it could

[6]http://www.r-project.org.

[7]Firebug has over 3 million active daily users: https://addons.mozilla.org/en-US/firefox/addon/firebug/statistics/usage/.

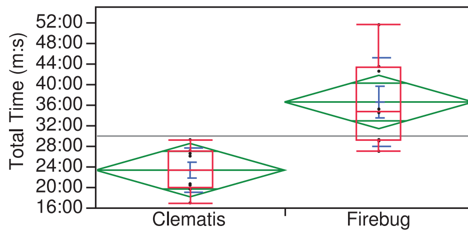

Fig. 15. t-Test analysis with unequal variances of task completion duration by tool type. Lower values are better [Study 1].

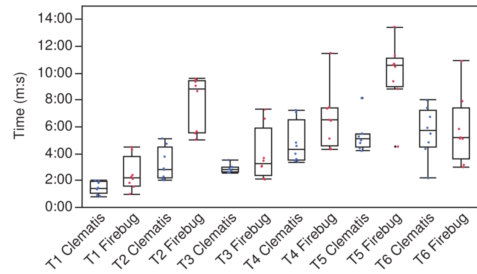

Fig. 16. Box plots of task completion duration data per task for each tool. Lower values are better [Study 1].

be managed within the time frame anticipated for the experiment. Yet it covered the common comprehension activities described in Table I.

Experimental Procedure. We followed the general procedure described in Section 4.2. After filling out the prequestionnaire form, the participants in the control group were given a tutorial on Firebug and had time to familiarize themselves with it, though most of them were already familiar with Firebug.

5.2. Results

Duration. To address RQ1, we measured the amount of time (minutes:seconds) spent on each task by the participants and compared the task durations between Clematis and Firebug using a t-test. According to the results of the test, there was a statistically significant difference (p-value = 0.002) in the durations between Clematis (M = 23:22, SD = 4:24) and Firebug (M = 36:35, SD = 8:35). Figure 15 shows the results of the comparisons.

To investigate whether certain categories of tasks (Table II) benefit more from using Clematis (RQ3), we tested each task separately. The results showed improvements in time for all tasks. The improvements were statistically significant for tasks 2 and 5, and showed a 60% and 46% average time reduction with Clematis, respectively. The mean times of all tasks for Clematis and Firebug are presented in Figure 16. *The results show that on average, participants using* Clematis *require 36% less time than than the control group using Firebug for performing the same tasks.*

Accuracy. The accuracy of answers was calculated in percentages. We compared the accuracy of participants' answers using a t-test. The results were again in favor of Clematis and were statistically significant (p = 0.02): Clematis (M = 83%, SD = 18%) and Firebug (M = 63%, SD = 16%). This comparison of accuracy between tools is depicted in Figure 17. As in the duration case, individual t-tests were then performed for comparing accuracy per task (related to RQ3). Clematis showed an increased average accuracy for all tasks. Further, the difference was statistically significant in favor of Clematis for task 5 and subtasks 4.a and 5.a. *The results show that participants using* Clematis *achieved 22% higher accuracy than participants in the control group*. We plot the average accuracies of all tasks for Clematis and Firebug in Figure 18. We discuss the implications of these results in Section 9.

6. EXPERIMENT 2: INDUSTRIAL

To investigate Clematis's effectiveness in more realistic settings, we conducted a second controlled experiment at a large software company in Vancouver, where we recruited

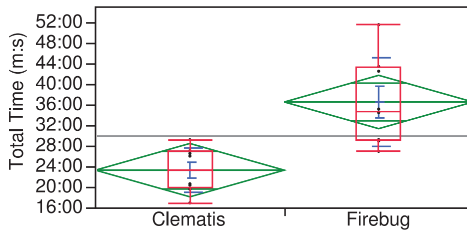

Fig. 17. t-Test analysis with unequal variances of task completion accuracy by tool type. Higher values are better [Study 1].

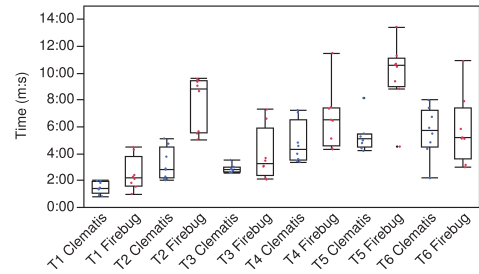

Fig. 18. Box plots of task completion accuracy data per task for each tool. Higher values are better [Study 1].

Table III. Comprehension Tasks Used in Study 2

Task	Description	Activity
T7	Extracting the control flow of an event with delayed effects	A1, A4, A5, A7
T8	Finding the mutations in DOM after an event	A1, A5
T9	Locating the implementation of a malfunctioning feature	A1, A2, A3
T10	Extracting the control flow of an event with event propagation	A1, A5, A8

professional developers as participants and used an open-source web application as the experimental object.

6.1. Approach

Experimental Design. Similar to the first experiment, the participants were divided into experimental and control groups. The experimental group used Clematis throughout the experiment. Unlike in the previous experiment, members of the control group were free to use the tool of their choice for performing the tasks. The intention was for the participants to use whichever tool they were most comfortable with. Five participants used Google Chrome's developer tools, two used Firefox's developer tools, and three used Firebug.

Experimental Subjects. We recruited 20 developers from a large software company in Vancouver, four females and 16 males. They were 23 to 42 years old and had medium to high expertise in web development.

Task Design. For this experiment, we used fewer but more complex tasks compared to the first experiment. We designed four tasks (Table III) spanning the following categories: following the control flow, understanding event propagation, detecting DOM mutations, locating feature implementation, and determining delayed code execution using timeouts.

Experimental Object. Phormer[8] is an online photo gallery in PHP, JavaScript, CSS, and XHTML. It provides features such as uploading, commenting, rating, and displaying slideshows for users' photos. It contains typical mechanisms such as dynamic DOM mutation, asynchronous calls (XHR and timeouts), and event propagation. Phormer has over 6,000 lines of JavaScript, PHP, and CSS code in total (1,500 lines of JavaScript). It was rated 5.0 stars on SourceForge and had over 38,000 downloads at the time of conducting the experiment.

[8]http://p.horm.org/er/.

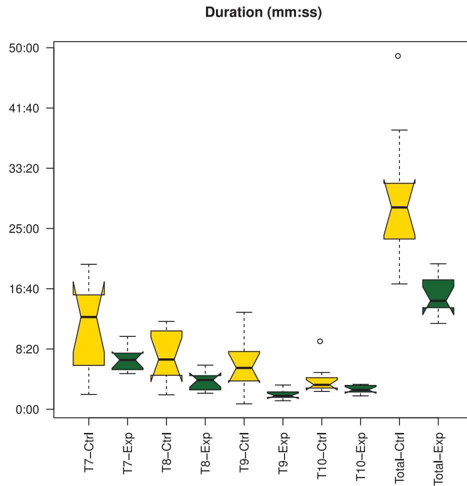

Fig. 19. Notched box plots of task completion duration data per task and in total for the control (green) and experimental (gold) groups (lower values are desired) [Study 2].

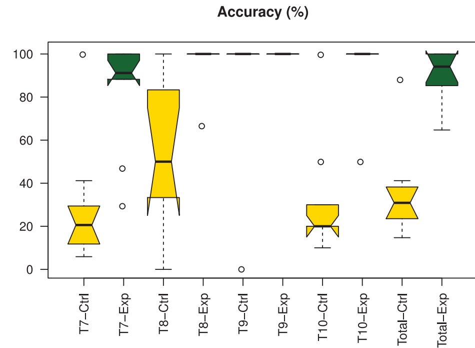

Fig. 20. Notched box plots of task completion accuracy data per task and in total for the control (green) and experimental (gold) groups (higher values are desired) [Study 2].

Experimental Procedure. We followed the same procedure described in Section 4.2, with one difference: the participants in the control group were not given any tutorial regarding the tool they used throughout the experiment, as they were all proficient users of the tool of their choice.

6.2. Results

Box plots of task completion duration and accuracy, per task and in total, for the control (Ctrl) and experimental (Exp) groups are depicted in Figures 19 and 20, respectively.

Duration. Similar to the previous experiment, we ran a set of t-tests for the total task duration as well as for the time spent on individual tasks. The results of the tests showed a statistically significant difference (p-value = 0.0009) between the experimental group using Clematis (M = 15:37, SD = 1:43) and the control group (M = 29:12, SD = 5:59), in terms of total task completion duration. The results showed improvements in duration when using Clematis for all four tasks. We found significant differences in favor of Clematis for tasks T7, T8, and T9. *The results show that developers using* Clematis *took 47% less time on all tasks compared to developers in the control group.*

Accuracy. We used Mann-Whitney U tests for comparing the results of task accuracy between the control and the experimental group, since the data was not normally distributed. For the overall accuracy of the answers, the tests revealed a statistically significant difference with high confidence (p-value = 0.0005) between Clematis (M = 90%, SD = 25%) and other tools (M = 35%, SD = 20%). We then performed the comparison between individual tasks. Again, for all tasks, the experimental group using Clematis performed better on average. We observed statistically significant improvements in the accuracy of developers using Clematis for tasks T7, T8, and T10. *The results show that developers using* Clematis *performed more accurately across all tasks by 157% on average compared to developers in the control group.*

6.3. Qualitative Analysis of Participant Feedback

The industrial participants in our second experiment shared their feedback regarding the tool they used in the experiment session (CLEMATIS for the experimental group and other tools for the control group). They also discussed their opinions about the features an ideal web application comprehension tool should have. We systematically analyzed [Creswell 2012] the qualitative data to find the main features of a web application comprehension tool according to professional web developers. To the best of our knowledge, at the time of this study, there were neither any tools available specifically designed for web application comprehension nor any studies on their desirable characteristics.

6.3.1. Data Collection. The participant selection was based on introductions by the team leads in the company. Our research group had started a research collaboration with the company and the company was willing to spread the word about the experiment and help recruit volunteer participants. The examiner was present at the company starting 2 weeks prior to the experiment and helped the procession of recruiting and, if possible, giving an introduction to the potential participants.

Our overall policy for recruiting participants was random sampling. However, throughout the course of the experiment, we tried to partially apply theoretical sampling by asking participants to recommend other candidates fit for attending the experiment. In general, this did have a noticeable impact on our sampling process since our desirable sample set had to be diverse. A wider range of experience and proficiency was suitable for our purpose, as we wanted to support various groups of web developers by CLEMATIS. Moreover, preserving the overall randomness of sampling was necessary for ensuring the validity of our qualitative analysis. Hence, we examined the background and the experience of our potential candidates and tried to include a more diverse group of participants that still met our original requirements.

In the final phase of the experiment, where we gathered the qualitative data, the participants filled out a postquestionnaire form with open-ended questions. The forms allowed them to focus and provide answers without having the sense of being watched. Next, they were interviewed verbally based on both their answers to the questionnaire and the comments of previous participants. During the interviews, the examiner took notes of the participants' answers as well as their expressions and body language, which could convey more insight into participants' intents.

6.3.2. Extracting the Concepts. After each group of consecutive sessions was completed, we started coding the gathered data based on open coding principles. We read and analyzed comments and interview manuscripts of each participant, and coded every comment based on the participant's intention. At this stage, no part of the data was excluded. The coding only helped us extract the existing concepts within the data. Hence, by performing coding parallel to conducting the experiments, we were able to better direct our following interview sessions. This process enabled us to observe the emerging categories as we proceeded with the experiment. We used this information to guide the interviews toward discovering the new data. Moreover, we simultaneously compared the coded scripts of different participants. This allowed us to investigate the consistencies or differences between the derived concepts.

As we progressed further in conducting the experiment sessions, the core categories of concepts began to emerge from the coded data. We used memos to analyze these categories early in the process, while we were still able to improve the interviews.

Categories started to form during the process of coding the data. We started to recognize the core categories based on the density of the data in each category. We then continued with selective coding of the remaining forms and manuscripts. We intentionally permitted the evolution of multiple core categories (as opposed to one), in

order to account for different aspects of an ideal comprehension tool to get recognized. Multiple categories were integrated to create each core category. The concepts that contributed to building each core category were referred to by a noticeable number of participants. Various subcategories were brought together to form different aspects of a desirable web application comprehension tool according the developers who are interested in using such a tool. Closer to the end of the experiments, only the more relevant categories to the core categories were selected due to selective coding. The maturity of the core categories (described later) was indicated when the newly gathered data did not contribute much to the existing categories.

6.3.3. Guidelines for Web Application Comprehension Tools. The following are the characteristics of a desirable web application comprehension tool, derived from the participants' responses to our postquestionnaire forms and interviews:

—**Integration with debugging.** One of the most prevalent concepts that was discussed by the participants was debugging. All of our participants were using a browser-specific debugger in their everyday tasks. Although these debugging capabilities are not best tuned for web application comprehension, they still play a potent role in the web development process. Almost all developers in the control group used one or more features of a debugger. Many developers in the experimental group requested adding features such as setting breakpoints and step-by-step execution to CLEMATIS. Some of our participants suggested the integration of CLEMATIS with commonly used platforms that support debugging.

—**DOM inspection.** The majority of the participants used the DOM inspection feature of browser development tools extensively. However, the participants in the control group were frustrated by the unavailability of a feature that allows them to easily detect all of changes to the DOM after a certain event. This option was provided for CLEMATIS users, most of whom chose this feature as one of their favorite features. The majority of the participants in the experimental group mentioned CLEMATIS's DOM mutation view is particularly useful and requested a better visualization.

—**JavaScript and DOM interaction.** Many participants in the control group were complaining about the lack of better means of relating the JavaScript code to DOM elements and events. Not using CLEMATIS, there is currently no trivial way of relating DOM events to the respective executed JavaScript code. Moreover, there is no connection between a DOM feature and the JavaScript code responsible for that feature. This can make the common task of feature location rigorous.

—**Call hierarchy.** One of the most popular topics of CLEMATIS users was any concept related to the trace it keeps in each episode. The majority of the participants in the experimental group were pleased by the ease of understanding the customized sequence diagrams. They quickly adopted this feature, and many of the CLEMATIS users were also impressed by the inclusion of asynchronous callbacks and propagated event handlers. On the other hand, most of the participants in the control group expressed dissatisfaction with the lack of features such as call stacks in existing tools.

—**Interactivity and real-timeness.** Many CLEMATIS users mentioned more interaction and better responsiveness of the tool as a key factor in adopting it for their everyday tasks. Intrigued by the ability to capture a story of interactions, they were demanding real-time creation of stories while interacting with the application, and better analysis performance. The industrial tools used by the control group provided much better performance but lacked many of the desired features (other core categories).

—**Sophisticated visualization.** Many participants indicated that visualization techniques and the usability factors can hugely impact their usage of a tool. Most of the

Table IV. Injected Faults for the Controlled Experiment

Fault	Fault Description	Detecting Test Case	Related Task
F1	Altered unary operation related to navigating slideshow	SlideShowTest	T11
F2	Modified string related to photo-rating feature	MainViewTest	T12
F3	Changed number in branch condition for photo-rating feature	MainViewTest	T12
F4	Transformed string/URL related to photo-rating feature	MainViewTest	T12

CLEMATIS users preferred the focus+context technique adopted by CLEMATIS. However, being an academic prototype, CLEMATIS has much room for improvement in terms of interface design and usability. In general, any tool that supports all technical core categories can still be unsuccessful should it fail in delivering the necessary information to users through a visualization.

There were few features that the participants found useful but that were not included in the core categories. Among them was semantic zooming, or presenting the overview first and providing more details on demand. Another popular feature was the extraction of DOM mutations per event. The participants also requested for a number of features to be included in future versions of the tool. These features included filtering and query options for DOM mutations, and the ability to attach notes to bookmarked episodes. Overall, according to two of our industrial participants, CLEMATIS is "Helpful and easy to use" and "Very useful. A lot of potential for this tool!"

7. EXPERIMENT 3: TEST FAILURE COMPREHENSION

We conducted a third controlled experiment to assess the effectiveness of our test failure comprehension extension of CLEMATIS.

7.1. Approach

Experimental Design. Once again, we divided the participants into experimental (CLEMATIS) and control groups.

Experimental Subjects. Twelve participants were recruited for the study at UBC, three females and nine males. The participants were drawn from different education levels at UBC. They all had prior experience in web development and testing, ranging from beginner to professional. Furthermore, six of the participants had worked in industry previously either full time or through internships.

Task Design. For this experiment, we used fewer but more complex tasks compared to the first experiment. To answer RQ5, participants were given two main tasks, each involving the debugging of a test failure in the Phormer application (Table IV). For each task, participants were given a brief description of the failure and a test case capable of detecting the failure. We used a test suite written by a UBC student for the Phormer application. The test suite was written as part of a separate and independent course project, 6 months before the inception of our project presented in this article.

For the first task of this experiment (T11), they were asked to locate an injected fault in Phormer given a failing test case. Participants were asked not to modify the application's JavaScript code during T11.

The second task of this experiment (T12) involved identifying and fixing a regression fault (unrelated to the first one). For this task, participants were asked to locate and repair the fault(s) causing the test failure. As the second failure was caused by three separate faults, participants were allowed to modify the application source code in order to iteratively uncover each fault by rerunning the test case. In addition to the failing test case, participants in both groups were given two versions of Phormer, the

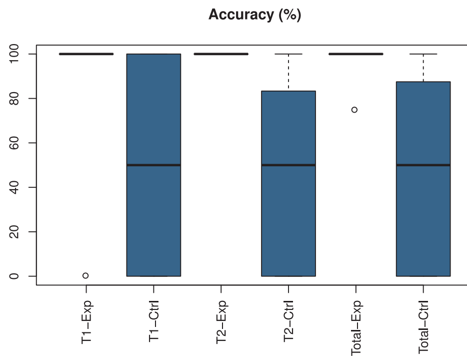

Fig. 21. Box plots of task completion accuracy data per task and in total for the control (blue) and experimental (cream) groups (higher values are desired) [Study 3].

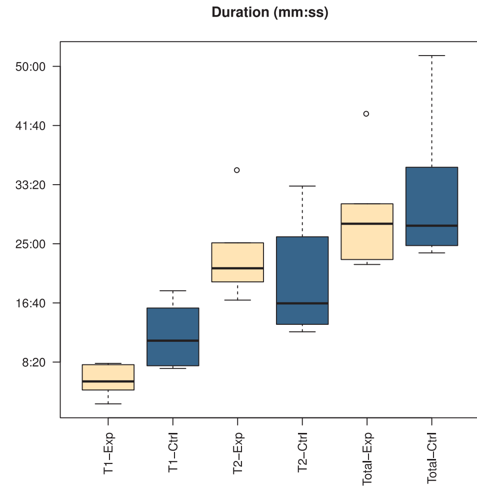

Fig. 22. Box plots of task completion duration data per task and in total for the control (blue) and experimental (cream) groups (lower values are desired) [Study 3].

faulty version and the original fault-free one. The intention here was to simulate a regression testing environment.

The injected faults are based on common mistakes JavaScript developers make in practice, as identified by Mirshokraie et al. [2015].

Experimental Object. Similar to the previous experiment, we used Phormer as the experimental object.

Experimental Procedure. The procedure was similar to what we described in Section 4.2. A maximum of 1.5 hours was allocated for the study: 10 minutes were designated for an introduction, 15 minutes were allotted for users to familiarize themselves with the tool being used, 20 minutes were allocated for task 11, another 30 minutes were set aside for task 12, and 15 minutes were used for completing the questionnaire at the end of the study.

7.2. Results

Figure 21 and Figure 22 depict box plots of task completion accuracy and duration, per task and in total, for both the experimental group (Exp) and the control group (Ctrl).

Accuracy. The accuracy of participant answers was calculated to answer RQ5. Overall, the group using CLEMATIS (M = 95.83, SD = 10.21) performed much more accurately than the control group (M = 47.92, SD = 45.01). The results show a statistically significant improvement for the experimental group (p-value = 0.032). Comparing the results for the two tasks separately, the experimental group performed better on both tasks on average. *The results show that participants using* CLEMATIS *performed more accurately across both tasks by a factor of two, on average, compared to those participants in the control group.*

Duration. To further answer RQ5, we measured the amount of time (minutes:seconds) spent by participants on each task and in total. According to the results of the tests, there was a statistically significant difference in the duration of T11 for CLEMATIS (M =

5:42, SD = 2:10) and the control group (M = 12:03, SD = 4:29), p-value = 0.016. Comparison of the duration data gathered for T12 yielded no significant difference between CLEMATIS (M = 23:23, SD = 6:31) and the control group (M = 19:46, SD = 8:05), p-value > 0.05. Those participants in the control group who answered task 2 correctly required a mean duration of 25:21 to complete the task, which is a longer time than the mean duration of the experimental group. The results revealed no significant difference between the CLEMATIS group (M = 29:05, SD = 7:42) and the control group (M = 31:49, SD = 10:37) with regard to the total time spent (p-value > 0.05). *The results show that developers using* CLEMATIS *took 54% less time to localize a detected fault. The results are inconclusive regarding fault repair time.*

8. PERFORMANCE OVERHEAD

With respect to RQ4, there are three sources of potential performance overhead: (1) instrumentation overhead, (2) execution overhead, and (3) dynamic analysis overhead. The first pertains to the overhead incurred due to the instrumentation code added by CLEMATIS, while the second pertains to the overhead of processing the trace and constructing the model. The third type of overhead is caused by dynamic slicing and can only occur when the test failure comprehension unit is activated. We do not measure the overhead of visualization as this is dependent on the user task performed.

We measure the first two types of overhead when the test comprehension unit is deactivated. Then we activate the test unit and measure the additional overhead. Phormer, the experimental object in study 2, is used to collect performance measurements over 10 one-minute trials of user interaction with the application. We also activate the test comprehension unit and execute each of the two test cases from experiment 3 with selective instrumentation both enabled and disabled. The two tests were run 10 times each. The results are as follows:

Instrumentation Overhead. *Code Comprehension.* Average delays of 15.04 and 1.80 seconds were experienced for the pre- and postprocessing phases with CLEMATIS, respectively. A 219.30ms additional delay was noticed for each page. On average, each captured episode occupies 11.88KB within our trace.

Test Comprehension. Average delays of 1.29 and 1.83 seconds were introduced by the selective and nonselective instrumentation algorithms, respectively, on top of the 407ms required to create a new browser instance. Moreover, the average trace produced by executing the selectively instrumented application was 37KB in size. Executing a completely instrumented application resulted in an average trace size of 125KB. Thus, the selective instrumentation approach is able to reduce trace size by 70% on average, while also reducing instrumentation time by 41%.

Execution Overhead. *Code Comprehension.* For processing 1 minute of activity with Phormer, CLEMATIS experienced an increase of 250.8ms, 6.1ms, and 11.6ms for DOM events, timeouts, and XHRs, respectively. Based on our experiments, there was no noticeable delay for end-users when interacting with a given web application through CLEMATIS.

Test Comprehension. The actual execution of each test case required an additional 246ms for the selectively instrumented application. Instrumenting the entire application without static analysis resulted in each test case taking 465ms longer to execute. Based on these measurements, our selective instrumentation approach lowers the execution overhead associated with CLEMATIS by 47%.

Dynamic Analysis Overhead. It took CLEMATIS 585ms on average to compute each JavaScript slice when utilizing selective instrumentation. Nonselective instrumentation lengthened the required dynamic analysis time to 750ms. By analyzing a more

concise execution trace, CLEMATIS was able to lower the slice computation time by 22%. Thus, we see that CLEMATIS incurs low performance overhead in all three components, mainly due to its selective instrumentation capabilities.

9. DISCUSSION

9.1. Task Completion Duration

Task completion duration is a measure of task performance. Therefore, CLEMATIS improves web developers' performance by significantly decreasing the overall time required to perform a set of code comprehension tasks (RQ1).

Dynamic Control Flow. Capturing and bubbling mechanisms are pervasive in JavaScript-based web applications and can severely impede a developer in understanding the dynamic behavior of an application. These mechanisms also complicate the control flow of an application, as described in Section 2. Our results show that CLEMATIS significantly reduces the time required for completing tasks that involve a combination of nested function calls, event propagation, and delayed function calls due to timeouts within a web application (T2, T5.a, and T7). Hence, CLEMATIS makes it more intuitive to comprehend and navigate the dynamic flow of the application (RQ3).

One case that needs further investigation is T10. This task mainly involves following the control flow when most of the executed functions are invoked through event propagation. The results of this task indicate that although using CLEMATIS caused an average of 32% reduction in task completion duration, the difference was not statistically significant. However, closer inspection of the results reveals that the answers given using CLEMATIS for T10 are 68% more accurate on average. This huge difference shows that many of the developers in the control group were unaware of occurrences of event propagation in the application and terminated the task early. Hence, they scored significantly lower than the experimental group in task accuracy and still spent more time to find the (inaccurate) answers.

Feature Location. Locating features, finding the appropriate place to add a new functionality, and altering existing behavior are a part of comprehension, maintenance, and debugging activities in all software tools, not only in web applications. The results of study 1 suggested that CLEMATIS did reduce the average time spent on the tasks involving these activities (T1, T3, T4.b), but these reductions were not statistically significant. These tasks mostly dealt with static characteristics of the code and did not involve any of the features specific to JavaScript-based web applications. Study 2, however, involved more complicated tasks in more realistic settings. T9 represented the feature location activity in this study, and the results showed that using CLEMATIS improved the average time spent on this task by 68%. Thus, we see that CLEMATIS speeds up the process of locating a feature or a malfunctioning part of the web application (RQ3).

State of the DOM. The final category of comprehension activities investigated in this work is the implications of events on the state of the DOM. Results of Study 1 displayed a significant difference in duration of the task involving finding DOM mutations in favor of CLEMATIS (T5). The results of Study 2 further confirmed the findings of Study 1 by reducing the duration in almost half (T8). Thus, CLEMATIS aids understanding the behavior of web applications by extracting the mutated elements of the DOM, visualizing contextual information about the mutations, and linking the mutations back to the corresponding JavaScript code (RQ3).

Test Failure Comprehension. The average recorded task duration for T11 was significantly lower for the experimental group. The participants in the control group often used breakpoints to step through the application's execution while running the provided

test case. When unsure of the application's execution, these developers would restart the application and re-execute the test case, extending their task duration. Instead of following a similar approach, those developers using CLEMATIS were able to rewind and replay the application's execution multiple times offline, after only executing the test case once. The trace collected by CLEMATIS during this initial test case execution was used to deterministically replay the execution while avoiding the overhead associated with rerunning the test case.

While task duration was significantly improved by CLEMATIS for T11, the average measured task duration was in fact longer for CLEMATIS in T12. However, the participants using CLEMATIS performed much more accurately on T12, suggesting that the task is complex and the main advantage of using CLEMATIS is accurate completion of the task. Studying the accuracy results for T12 reveals that many of the participants in the control group failed at correcting the faults, and instead simply addressed the failure directly. This may explain the reason for no observable improvement in task duration for T12, as hiding the failure often requires less effort than repairing the actual fault.

9.2. Task Completion Accuracy

Task completion accuracy is another metric for measuring developers' performance. According to the results of both experiments, CLEMATIS increases the accuracy of developers' actions significantly (RQ2). The effect is most visible when the task involves *event propagation* (RQ3). The outcome of Study 1 shows that CLEMATIS addresses Challenge 1 (described in Section 2) in terms of both time and accuracy (T5.a). Study 2 further indicates that CLEMATIS helps developers to be more accurate when faced with tasks involving event propagation and control flow detection in JavaScript applications (67% and 68% improvement for T7 and T10, respectively).

For the remaining tasks of Study 1, the accuracy was somewhat, though not significantly, improved. We believe this is because of the simplistic design of the experimental object used in Study 1, as well as the relative simplicity of the tasks. This led us toward the design of Study 2 with professional developers as participants and a third-party web application as the experiment object in the evaluation of CLEMATIS. According to the results of Study 2, CLEMATIS significantly improves the accuracy of completion of tasks (T8) that require finding the implications of executed code in terms of *DOM state changes* (RQ3). This is related to Challenge 3 as described in Section 2.

For the feature location task (T9), the accuracy results were on average slightly better with CLEMATIS. However, the experimental group spent 68% less time on the task compared to the control group. This is surprising as this task is common across all applications and programming languages, and we anticipated that the results for the control group would be comparable with those of the experimental group.

Test Failure Comprehension. The results from both experimental tasks suggest that CLEMATIS is capable of significantly improving the fault localization and repair capabilities of developers (RQ5). Many participants in the control group failed to correctly localize the fault, illustrating the difficulty in tracing dependencies in a dynamic language such as JavaScript. Although users in the control group had access to breakpoints, many of them had difficulty stepping through the application's execution at runtime due to the existence of asynchronous events such as timeouts, which caused nondeterministic behavior in the application when triggered in the presence of breakpoints.

Many of the participants in the control group fixed the *failure* instead of the actual *fault*; they altered the application's JavaScript code such that the provided test case would pass, yet the faults still remained unfixed. The JavaScript code related to task 2 contained multiple statements that accessed the DOM dependency of the failing test

case assertion. Participants who simply corrected the failure had trouble identifying which of these statements was related to the fault, and as a result would alter the wrong portion of the code. On the other hand, those participants using CLEMATIS were able to reason about these DOM-altering statements using the provided links and slices.

9.3. Consistent Performance

Looking at Figures 19 and 20, it can be observed that using CLEMATIS not only improves both the duration and accuracy of individual and total tasks but also helps developers to perform in a much more consistent manner. The high variance in the results of the control group shows that individual differences of developers (or tools in Study 2) influence their performance. However, the low variance in all the tasks for the experimental group shows that CLEMATIS helped *all* developers in the study to perform consistently better by making it easier to understand the internal flow and dependency of event-based interactions.

9.4. Threats to Validity

Internal Threats. The first threat is that different levels of expertise in each subject group could affect the results. We mitigated this threat by manually assigning the subjects to experimental and control groups such that the level of expertise was balanced between the two groups. The second threat is that the tasks in the experiment were biased toward CLEMATIS. We eliminated this threat by adopting the tasks from a well-known framework of common code comprehension tasks [Pacione et al. 2004]. A third threat arises from the investigators' bias toward CLEMATIS when rating the accuracy of subjects' answers. We addressed this concern by developing an answer key for all the tasks before conducting the experiments. A similar concern arises regarding the task completion duration measurements. We mitigated this threat by presenting each task to subjects on a separate sheet of paper and asking them to return it upon completion. The duration of each task was calculated from the point a subject received the task until he or she returned the paper to the investigators, thus eliminating our bias in measuring the time (and the subjects' bias in reporting the time). Finally, we avoided an inconsequential benchmark by choosing a tool for the control group in Study 1 that was stable and widely deployed, namely, Firebug. In Study 2, the developers in the control group were given the freedom to choose any tool they preferred (and had experience with).

External Threats. An external threat to validity is that the tasks used in the experiment may not be representative of general code comprehension activities. As mentioned earlier, we used the Pacione framework and thus these tasks are generalizable. A similar threat arises with the representativeness of the participants. To address this threat, we used both professional web developers and students/postdocs with previous web development experience.

Reproducibility. As for replicating our experiments, CLEMATIS, the experimental object Phormer, and the details of our experimental design (e.g., tasks and questionnaires) are all available, making our results reproducible.

9.5. Limitations

The contributions of this work were essential basic steps toward an interactive approach for understanding event-based interactions in client-side JavaScript. However, our approach entails many limitations and has much room left for future improvements.

JavaScript is a highly dynamic language. There are many cases that occur in JavaScript applications and that are not currently supported by CLEMATIS. As an example, CLEMATIS does not instrument JavaScript code that is maintained in strings and is executed using `eval()`. Also, should an exception occur and change the normal means

of function execution, the resulting model may be affected. However, these are among features of JavaScript that can be handled in the near future using the current design.

There is also room left for research in determining the episode-ending criteria. For terminating an episode, the current approach ensures that the call stack is empty and there are no immediate asynchronous timing events in the event loop. If these conditions are valid and there is inactivity in JavaScript execution for a certain amount of time, the algorithm terminates the episode. We determined the minimum required inactivity time by choosing the best results from a set of empirical examinations. Further investigation on this temporal threshold, as well as other criteria that can define the boundaries of episodes, may lead to interesting findings.

Finally, the resulting model can still be overwhelming for users. Large-scale enterprise applications often have customized event frameworks and communicate with their servers constantly. CLEMATIS's semantic zooming can help mitigate this issue, but to a limit. Proposing abstraction and categorization techniques for CLEMATIS's visualization can be applied to further assist the comprehension process.

10. RELATED WORK

According to a literature survey by Cornelissen et al. [2009], despite their unique and challenging characteristics, web applications have rarely been targeted in program comprehension research. Previous research has approached this issue through different perspectives.

Program Analysis. EventRacer is a tool for facilitating dynamic race detection for event-driven applications [Raychev et al. 2013]. Compliant with its goal, EventRacer traces only the events and not other dynamic and asynchronous features of JavaScript. Moreover, unlike CLEMATIS, this approach requires using an instrumented browser. Wei and Ryder [2013] use both static and dynamic analysis to perform a points-to analysis of JavaScript. However, they do not take into account the DOM-based and asynchronous interactions of JavaScript. Ghezzi et al. [2014] extract behavioral models from a different perspective. They focus on users' navigation preferences in user-intensive software. Their approach, called BEAR, depends on server logs to capture user interactions. Unlike CLEMATIS, BEAR only focuses on direct user interactions in order to fulfill its purpose, which is classifying the behavior of users.

UI Feature Location. Li and Wohlstadter [2009] present a tool called Script Insight to locate the implementation of a DOM element in JavaScript code. Similarly, Maras et al. [2012, 2013] propose a technique for deriving the implementation of a UI feature on the client side. While similar to our work at a high level, in these approaches the user needs to select a visible DOM element and its relevant behavior in order to investigate its functionality. This manual effort can easily frustrate the user in large applications. Further, these techniques are not concerned with capturing event-based interactions. Finally, the model they derive and present to the user contains low-level information and noise, which can adversely influence program comprehension.

Capture and Replay. Extensive reliance on user interactions is an important characteristic of modern web applications. Capture and replay tools are used in the literature to address this issue [Cornelissen et al. 2009]. Montoto et al. [2009] propose a set of techniques for generating a navigation sequence for Ajax-based websites and executing the recorded trace. Mugshot [Mickens et al. 2010] is a system that employs a server-side web proxy to capture events in interactive web applications. It injects code into a target web application in order to record sources of nondeterminism such as DOM events and interrupts. The recorded information is used by Mugshot to dispatch synthetic events to a web browser in order to replay the execution trace. WaRR [Andrica and Candea 2011]

is another system for capturing and replaying events. Capturing is accomplished by altering a user's web browser in order to record keystrokes and mouse clicks. In the event of a failure, end-users of a web application may send a record of their keystrokes to the developer for debugging purposes. Jalangi [Sen et al. 2013] is another record/replay tool that supports dynamic analysis by shadow execution on shadow values. Burg et al. [2013] integrate their capture/replay tool with debugging tools.

The goal in most of these techniques is to find a deterministic way of replaying the same set of user events for debugging purposes. Instead of simply replaying recorded events, our approach aims at detecting causal and temporal event-based interactions and linking them to their impact on JavaScript code execution and DOM mutations. Moreover, our approach does not require manual user effort, a modified server, or a special browser.

Visualization. There are many tools that use visualization to improve the process of understanding the behavior of software applications. Matthijssen et al. [2010] conduct a user study for investigating the strategies that web developers use for code comprehension. Extraviz [Cornelissen et al. 2011] is a visualization tool that represents the dynamic traces of Java applications to assist with program comprehension tasks. However, their approach does not concern itself with building a model of the web application, while ours does.

Zaidman et al. [2013] propose a Firefox add-on called FireDetective, which captures and visualizes a trace of execution on both the client and the server side. Their goal is to make it easier for developers to understand the link between client and server components, which is different from our approach, which aims to make it easier for developers to understand the client-side behavior of the web application.

FireCrystal [Oney and Myers 2009] is another Firefox extension that stores the trace of a web application in the browser. It then visualizes the events and changes to the DOM in a timeline. FireCrystal records the execution trace selectively, similar to our work. But unlike CLEMATIS, FireCrystal does not capture the details about the execution of JavaScript code or asynchronous events. Another limitation of FireCrystal is that it does not link the triggering of events with the dynamic behavior of the application, as CLEMATIS does. DynaRIA [Amalfitano et al. 2014] focuses on investigating the structural and quality aspect of the code. While DynaRIA captures a trace of the web application, CLEMATIS facilitates the process of comprehending the dynamic behavior using a high-level model and visualization based on a semantically partitioned trace.

Fault Localization and Debugging. Delta debugging [Zeller 2002] is a technique whereby the code change responsible for a failure is systematically deduced by narrowing the state differences between a passing and a failing run. Other fault localization techniques have been proposed that compare different characteristics of passing and failing runs for a program [Agrawal et al. 1995; Cleve and Zeller 2005; Groce and Visser 2003; Pytlik et al. 2003]. Our approach is different in that it focuses on a single web application test assertion at a time and does not require a passing test per se to operate.

There is limited research geared toward web application fault localization in the literature [Artzi et al. 2010; Ocariza et al. 2012]. Google has recently provided some support for debugging asynchronous JavaScript in Chrome DevTools [Chen 2014]. Our work is different from previous techniques since it aims at making the implicit links between test failures and faulty JavaScript code more explicit to enhance debugging. In addition, calculating and displaying the JavaScript code slice for a test assertion poses new challenges not faced by previous techniques. This stems from the disconnect between a test assertion failure, the DOM, and the JavaScript code interacting with the DOM.

Program Slicing. Originally proposed by Weiser [1981], program slicing techniques can be classified into two categories, namely, static and dynamic slicing [Korel and Laski 1988]. WALA [Sridharan et al. 2007] performs JavaScript slicing by inferring a call graph through static analysis. Since JavaScript is such a dynamic language, WALA yields conservative results that may not be reflective of an application's actual execution. It also ignores the JavaScript-DOM interactions completely. Although not used for slicing purposes, others [Necula et al. 2005; Yong and Horwitz 2005] have utilized static analysis to reduce the execution overhead incurred from code instrumentation. Our approach determines JavaScript slices through a selective code instrumentation algorithm.

11. CONCLUDING REMARKS

Modern web applications are highly dynamic and interactive, and offer a rich experience for end-users. This interactivity is made possible by the intricate interactions between user events, JavaScript code, and the DOM. However, web developers face numerous challenges when trying to understand these interactions. In this article, we proposed a portable and fully automated technique for relating low-level interactions in JavaScript-based web applications to high-level behavior. We proposed a behavioral model to capture these event interactions and their temporal and causal relations. We also proposed a strategy for helping developers understand the root causes of failing test cases. We presented a novel interactive visualization mechanism based on focus+context techniques for presenting these complex event interactions in a more comprehensible format to web developers. Our approach is implemented in a code comprehension tool called CLEMATIS. The evaluation of CLEMATIS points to the efficacy of the approach in reducing the overall time and increasing the accuracy of developer actions, compared to state-of-the-art web development tools. The greatest improvement was seen for tasks involving control flow detection, and especially event propagation, showing the power of our approach.

As part of future work, we plan to improve the interactive visualization and extend the details captured in each story to allow the programmer to gain better insight into the application. Another direction we will pursue is in debugging, where we will improve CLEMATIS's fault localization unit to further help developers detect and localize faulty behavior of JavaScript applications.

ACKNOWLEDGMENTS

We are grateful to all participants of the controlled experiments for their time and commitment. We especially thank SAP Labs Vancouver for all their help and support.

REFERENCES

Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'95)*. IEEE, 143–151.

Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014a. Understanding Javascript event-based interactions. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*. ACM, 367–377.

Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014b. *Understanding JavaScript Event-Based Interactions*. Technical Report UBC-SALT-2014-001. University of British Columbia. http://salt.ece.ubc.ca/publications/docs/UBC-SALT-2014-001.pdf.

Domenico Amalfitano, AnnaRita Fasolino, Armando Polcaro, and Porfirio Tramontana. 2014. The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis. *Innovations in Systems and Software Engineering* 10, 1 (2014), 41–57.

Silviu Andrica and George Candea. 2011. WaRR: A tool for high-fidelity web application record and replay. In *Proceedings of the International Conference on Dependable Systems & Networks (DSN'11)*. IEEE Computer Society, 403–410.

Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of Javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, 571–580.

Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Practical fault localization for dynamic web applications. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*. ACM, 265–274.

Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the Symposium on User Interface Software and Technology (UIST'13)*. ACM, 473–484.

Pearl Chen. 2014. Debugging Asynchronous JavaScript with Chrome DevTools. Retrieved June 18, 2015, from http://www.html5rocks.com/en/tutorials/developertools/async-call-stack/.

Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM, New York, NY, 342–351.

Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. 2009. A review of overview+detail, zooming, and focus+context interfaces. *Computer Surveys* 41, 1, Article 2 (2009), 31 pages.

Thomas A. Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 28, 2 (1989), 294–306.

Bas Cornelissen, Andy Zaidman, and Arie van Deursen. 2011. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering* 37, 3 (2011), 341–355.

Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

John W. Creswell. 2012. *Qualitative Inquiry and Research Design: Choosing Among Five Approaches* (2nd ed.). Sage, Thousand Oaks, CA.

Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of International Conference on Software Engineering (ICSE'13)*. IEEE Computer Society, 752–761.

Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. 2014. Mining behavior models from user-intensive web applications. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 277–287.

Alex Groce and Willem Visser. 2003. What went wrong: Explaining counterexamples. In *Workshop on Model Checking of Software*. Springer, Berlin, 121–135.

James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. ACM, 273–282.

Bogdan Korel and Janusz W. Laski. 1988. Dynamic program slicing. *Information Processing Letters* 29, 3 (1988), 155–163.

Peng Li and Eric Wohlstadter. 2009. Script insight: Using models to explore JavaScript code from the browser view. In *Proceedings of the 9th International Conference on Web Engineering (ICWE'09)*. Springer-Verlag, 260–274.

Josip Maras, Jan Carlson, and Ivica Crnkovi. 2012. Extracting client-side web application code. In *Proceedings of the International Conference on World Wide Web (WWW'12)*. ACM, 819–828.

Josip Maras, Maja Stula, and Jan Carlson. 2013. Generating feature usage scenarios in client-side web applications. In *Proceeding of the International Conference on Web Engineering (ICWE'13)*. Springer, 186–200.

Nick Matthijssen, Andy Zaidman, M.-A. Storey, Ian Bull, and Arie Van Deursen. 2010. Connecting traces: Understanding client-server interactions in Ajax applications. In *Proceedings of the International Conference on Program Comprehension (ICPC'10)*. IEEE Computer Society, 216–225.

Ali Mesbah, Arie van Deursen, and Danny Roest. 2012. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)* 38, 1 (2012), 35–53.

James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, 159–174.

Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering (TSE)* 41 (2015), 19 pages. http://salt.ece.ubc.ca/publications/docs/mutandis-tse.pdf.

Paula Montoto, Alberto Pan, Juan Raposo, Fernando Bellas, and Javier López. 2009. Automating navigation sequences in AJAX websites. In *Proceedings of the International Conference on Web Engineering (ICWE'09)*. Springer, 166–180.

George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems* 27, 3 (May 2005), 477–526. DOI:http://doi.acm.org/10.1145/1065887.1065892

Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2013. An empirical study of client-side javascript bugs. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*. IEEE Computer Society, 55–64.

Frolin Ocariza Jr , Karthik Pattabiraman, and Ali Mesbah. 2012. AutoFLox: An automatic fault localizer for client-side Javascript. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'12)*. IEEE Computer Society, 31–40.

Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, 105–108.

Michael J. Pacione, Marc Roper, and Murray Wood. 2004. A novel software visualisation model to support software comprehension. In *Proceedings of the Working Conference on Reverse Engineering*. IEEE Computer Society, 70–79.

Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. 2003. Automated fault localization using potential invariants. In *International Workshop on Automated and Algorithmic Debugging*. Ghent, Belgium, 273–276.

Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*. ACM, 151–166.

Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A selective record-replay and dynamic analysis framework for Javascript. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, 488–498.

Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin slicing. *SIGPLAN Notices* 42, 6 (June 2007), 112–122. http://doi.acm.org/10.1145/1273442.1250748.

Suresh Thummalapenta, K. Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. 2013. Guided test generation for web applications. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*. IEEE, 162–171.

Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494.

W3C. 2000. Document Object Model (DOM) Level 2 Events Specification. Retrieved from http://www.w3.org/TR/DOM-Level-2-Events/.

Shiyi Wei and Barbara G. Ryder. 2013. Practical blended taint analysis for Javascript. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'13)*. ACM, 336–346.

Mark Weiser. 1981. Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE'81)*. IEEE, 439–449.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering: An Introduction*. Kluwer.

SuanHsi Yong and Susan Horwitz. 2005. Using static analysis to reduce dynamic analysis overhead. *Formal Methods in System Design* 27, 3 (2005), 313–334. DOI:http://dx.doi.org/10.1007/s10703-005-3401-0

Andy Zaidman, Nick Matthijssen, Margaret-Anne Storey, and Arie van Deursen. 2013. Understanding Ajax applications by connecting client and server-side execution traces. *Empirical Software Engineering* 18, 2 (2013), 181–218.

Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE-10)*. ACM, 1–10.

Received February 2015; revised June 2015; accepted November 2015