Understanding JavaScript Event-Based Interactions

Saba Alimadadi Sheldon Sequeira Ali Mesbah Karthik Pattabiraman
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
{saba, sheldon, amesbah, karthikp}@ece.ubc.ca

ABSTRACT

Web applications have become one of the fastest growing types of software systems today. Despite their popularity, understanding the behaviour of modern web applications is still a challenging endeavour 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 behavioural 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 behaviour of their application at three different semantic levels of granularity. The results of our industrial controlled experiment show that CLEMATIS is capable of improving the task accuracy by 61%, while reducing the task completion time by 47%.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging-Tracing; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Design, Algorithms, Experimentation

Keywords

Program comprehension, event-based interactions, JavaScript

1. INTRODUCTION

JavaScript is widely used today to create interactive modern web applications that replace many traditional desktop applications. However, understanding the behaviour of web applications is a challenging endeavour for developers [19, 24]. Program comprehension is known to be an essential step in software engineering, consuming up to 50% [8] of the effort in software maintenance and analysis activities.

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 [18]. 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 [22].

Unfortunately, despite its importance and challenges, there is currently not much research dedicated to supporting program comprehension for web applications [10]. Popular tools, such as Firebug and Chrome DevTools, are limited in their capabilities to support web developers effectively.

In this paper, we present a generic, non-intrusive 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 behaviour during a particular user session. Our technique transforms the trace into an abstract behavioural 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, within each episode.

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 behavioural models. Our work makes the following key contributions:

- We propose a generic technique for capturing and presenting the complex dynamic behaviour 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.

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

  - Extracts the source-and-target relations for asynchronous events, i.e., timing events and XMLHttpRequest requests/callbacks.
  - 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 [1], 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.
- We empirically evaluate CLEMATIS through two controlled experiments comprising 34 users in total. One of the 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 61%.

2. RELATED WORK

UI Feature Location. Li and Wohlstadter [13] present a tool called Script Insight to locate the implementation of a DOM element in JavaScript code. Similarly, Maras et al. [14, 15] 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 behaviour 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. Related to our work are 'capture and replay' techniques for web applications [5, 6, 16, 17, 21]. 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.

Visualization. There are many tools that use visualization to improve the process of understanding the behaviour of software applications. For instance, Extraviz [9] visualizes 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. [24] 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 behaviour of the web application.

FireCrystal [19] 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 behaviour of the application, as CLEMATIS does. DynaRIA [4] 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 behaviour using a high-level model and visualization based on a semantically partitioned trace.

3. 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 1–2, to illustrate these challenges. Note that this is a simple example and these challenges are much more potent in large and complex web applications.

Challenge 1: Event Propagation. The DOM event model [22] 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 and triggers the parents' handlers. (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.

Example. Consider the sample code shown in Figures 1–2. Figure 1 represents the initial DOM structure of the application. It mainly consists of a `fieldset` containing a set of elements for the user 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, 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.

Challenge 2: Asynchronous Events. Web browsers provide a single thread for web application execution. To cir-

```
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>
```

Figure 1: Initial DOM state of the running example.

cumvent 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 behaviour: 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 behaviour?". By reviewing the code, she can find out that the source of this behaviour was the expiration of a timeout that was set in line 4 of Figure 2 by the anonymous handler defined in lines 3–5. However the callback function, defined on line 22 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 behaviour 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).

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.

```
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 }, function(data) {
18     $('#regMsg').append(data);
19   });
20 }
21 ...
22 function clearMsg() {$('#regMsg').fadeOut(2000);}
```

Figure 2: JavaScript code of the running example.

Example. After the `submitBtn` button is clicked in the running example, a confirmation message will appear on-screen and disappear shortly thereafter (lines 8&22, 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.

4. APPROACH

In this section, we describe our approach for addressing the challenges mentioned in the previous section. The overall process 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 4.1).
- Next, a behavioural 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 4.2).
- Finally, based on the inferred behavioural 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 4.3).

We describe each step further below. Our technical report [3] contains a more elaborate description of the technical details of the approach.

4.1 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 comprising 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 behavioural model (as described in subsection 4.2).

Interposing on DOM Events. There are two ways event listeners can be bound to a DOM element in JavaScript. The first method is programatically using the DOM Level 1 `e.click=handler` or DOM Level 2 `e.addEventListener` methods [22] 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 behaviour, 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 below.

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.

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.

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.

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.

4.2 Capturing a Behavioural Model

We use a graph-based model to capture and represent a web application's event-based interactions. The graph is multi-edge and directed. It contains an ordered set of nodes, called *episodes*, linked through edges that preserve the chronological order of event executions.[1] In addition, causal edges between the nodes represent asynchronous events. We describe the components of the graph below.

Episode Nodes. An episode is a semantically meaningful part of the application behaviour, 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 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:

- *Temporal:* The temporal edges connect one episode node to another, indicating that an episode succeeded the previous one in time.
- *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 `d` was caused by episode `s` in the past.

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

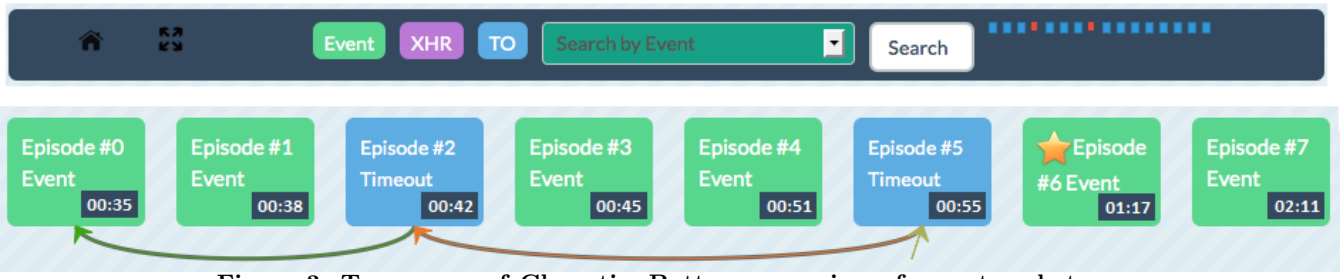

Figure 3: Top: menu of Clematis. Bottom: overview of a captured story.

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
```

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, 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 5–6) 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 11–12). 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 19–20).

4.3 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 behaviour of the application. 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 [7] 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 into each episode to gain more details regarding that episode, while preserving the contextual information about the story.

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 3, top). The story map represents a general overview of the whole story as a roadmap. Based on a user's interaction with the story (e.g., episode selection), the episodes of interest are highlighted on the roadmap. 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 3 is bookmarked). The episodes' timing information is also shown.

Semantic Zoom Levels. The visualization provides 3 semantic zoom levels. The first level displays all of the

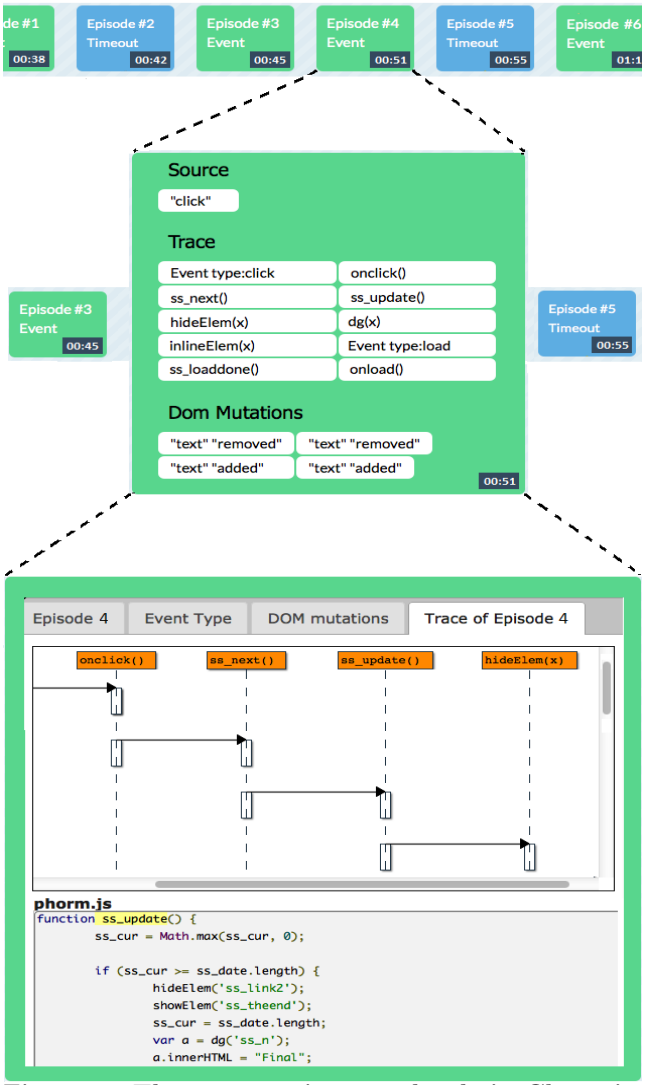

Figure 4: Three semantic zoom levels in Clematis. Top: overview. Middle: zoomed one level into an episode, while preserving the context of the story. Bottom: drilled down into the selected episode.

episodes in an abstracted manner, showing only the type and the timestamp of each episode (Figure 3, bottom).

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 4, 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. The user can view multiple episodes to have a side-by-side comparison.

The final zoom level exhibits all the information embedded in each episode, i.e., detailed information about the source event, the DOM mutations caused by the episode, and the low-level trace. The trace of an episode at this level includes a sequence diagram of the dynamic flow of all the invoked JavaScript functions and events within that episode. Upon request, the JavaScript code of each executed function is displayed and highlighted (Figure 4, bottom).

Table 1: 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 run-time interaction in the system
A6	Investigating how much an artifact is used
A7	Investigating the asynchronous aspects of JavaScript
A8	Investigate the hidden control flow of event handling

4.4 Tool Implementation: Clematis

We implemented our approach in a tool called Clematis, which is freely available [1]. 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 their interactions with the application. 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 [12]. 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. More details may be found in our technical report [3].

5. CONTROLLED EXPERIMENTS

To assess the efficacy of our approach, we conducted two controlled experiments [23], one in a research lab setting and the other in an industrial environment. Common design elements of both experiments are described in this section. The next two sections (6–7) are each dedicated to describing the specific characteristics and results of each experiment separately. Details of the tasks and questionnaires used for both experiments can be found in our technical report [3].

Our evaluation aims at addressing the following research questions:

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?

RQ3 For what types of tasks is Clematis most effective?

RQ4 What is the performance overhead of using Clematis? Is the overall performance acceptable?

5.1 Experimental Design

The experiments had a "between-subject" design; i.e., 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 pre-questionnaire 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 ef-

Table 2: 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

forts. We adapted the activities proposed by Pacione et al. [20], which cover categories of common tasks in program comprehension, to web applications by replacing two items. The revised activities are shown in Table 1. We designed a set of tasks for each experiment to cover these activities. Tables 2 and 3 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.

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-samples 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 [11] for the analysis.

5.2 Experimental Procedure

All experiments consisted of four main phases. First, the subjects were asked to fill a pre-questionnaire 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 2 and 3. 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 the rubric are available in our technical report [3].

In the final phase, participants filled out a post-questionnaire form providing feedback on their experience with the tool used (e.g., limitations, strength, usability). We categorized the qualitative data and present it in Section 9.4.

Table 3: 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

6. 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).

6.1 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.[2] Firebug has been used in other similar studies [24].

Experimental Subjects. We recruited 16 participants for the study, 3 females and 13 males. The participants were drawn from different educational levels: 2 undergraduate students, 5 Master's students, 8 Ph.D. students, and 1 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 2.

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 be managed within the time frame anticipated for the experiment. Yet it covered the common comprehension activities described in Table 1.

Experimental Procedure. We followed the general procedure described in section 5.2. After filling the pre-questionnaire 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.

6.2 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).

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

To investigate whether certain categories of tasks (Table 2) 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 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 favour of CLEMATIS and were statistically significant (p=0.02): CLEMATIS (M=83%, SD=18%) and Firebug (M=63%, SD=16%). 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 favour 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 discuss the implications of these results in Section 9.

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

7.1 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 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. 5 participants used Google Chrome's developer tools, 2 used Firefox's developer tools, and 3 used Firebug.

Experimental Subjects. We recruited 20 developers from a large software company in Vancouver, 4 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 4 tasks (Table 3) spanning the categories: following the control flow, understanding event propagation, detecting DOM mutations, locating feature implementation, and determining delayed code execution using timeouts.

Experimental Object. Phormer [2] 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 around 6,000 lines of JavaScript, PHP and CSS code in total. It was rated 5.0 star on SourceForge and had over 38,000 downloads at the time of conducting the experiment.

Experimental Procedure. We followed the same procedure described in 5.2, 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.

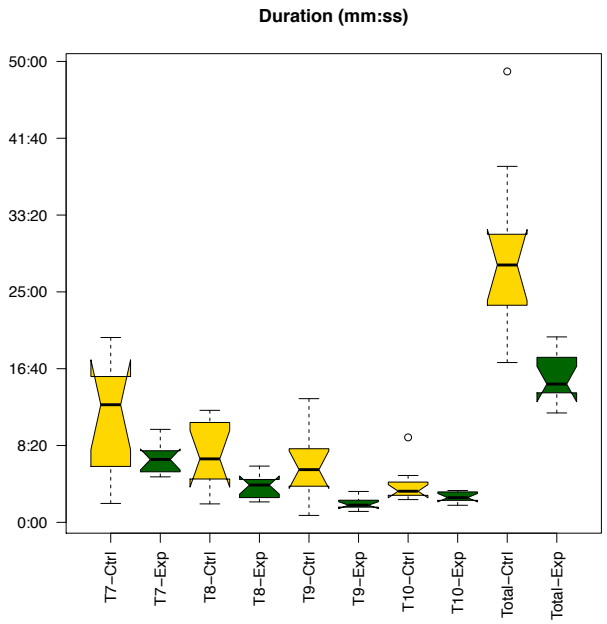

Figure 5: Notched box plots of task completion duration data per task and in total for the control and experimental groups (lower values are desired).

7.2 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 5 and 6, 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 favour 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 statistical 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 61% on average, compared to developers in the control group.*

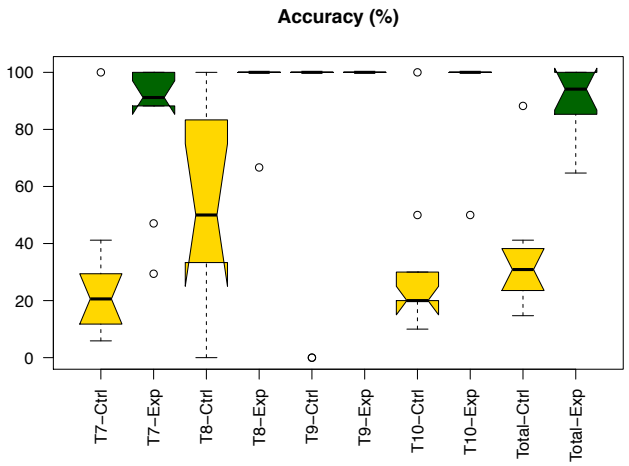

Figure 6: Notched box plots of task completion accuracy data per task and in total for the control and experimental groups (higher values are desired).

8. PERFORMANCE OVERHEAD

With respect to RQ4, there are two sources of performance overhead: (1) instrumentation overhead, and (2) execution overhead. The former pertains to the overhead incurred due to the instrumentation code added by CLEMATIS, while the latter pertains to the overhead of processing the trace and constructing the model. We do not measure the overhead of visualization as this is dependent on the user task performed.

Phormer, the experimental object in study 2, was used to collect performance measurements over 10 one-minute trials. The results were as follows:

Instrumentation overhead. Average delays of 15.04 and 1.80 seconds were experienced for pre and post processing phases with CLEMATIS respectively. And a 219.30 ms additional delay was noticed for each page. On average, each captured episode occupies 11.88 KB within our trace.

Execution overhead. For processing one minute of activity with Phormer, CLEMATIS experienced an increase of 250.8 ms, 6.1 ms and 11.6 ms 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.

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 behaviour of an application. These mechanisms also complicate the control flow of an application, as described in Section 3. 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 in 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 behaviour 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 favour 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 behaviour 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).

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 3) 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 towards 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 3.

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.

9.3 Consistent Performance

Looking at Figures 5 and 6, it can be observed that using CLEMATIS not only improves both duration and accuracy of individual and total tasks, but it 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 Participants' Feedback

We analyzed the qualitative data obtained through the post-questionnaire forms. Overall the feedback was very positive. The main features that the participants found most useful were the (1) semantic zooming: presenting the overview first and providing more details on demand, (2) visualizing the hierarchy of functions and events in the customized sequence diagram, (3) linking the visualization back to JavaScript code, and (4) extracting 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 (1) filtering and query options for DOM mutations, (2) ability to attach notes to bookmarked episodes, and (3) integrating CLEMATIS with debugging techniques such as breakpoints. 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!"*.

9.5 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 towards CLEMATIS. We eliminated this threat by adopting the tasks from a well-known framework of common code comprehension tasks [20]. A third threat arises from the investigators' bias towards 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 they 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 above, we used the Pacione's 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/post-docs with previous web development experience.

Reproducibility. As for replicating our experiments, CLEMATIS [1], the experimental object Phormer [2], and the details of our experimental design (e.g., tasks and questionnaires) [3], are all available making our results reproducible.

10. 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 paper, we proposed a portable and fully-automated technique for relating low-level interactions in JavaScript-based web applications to high level behaviour. We proposed a behavioural model to capture these event interactions, and their temporal and causal relations. 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 a better insight into the application. Another direction we will pursue is in debugging, where CLEMATIS can potentially help developers to better detect and localize faulty behaviour of JavaScript applications.

11. ACKNOWLEDGMENTS

This work was supported in part by an NSERC Strategic Project Grant and a research gift from Intel Corporation.

We thank Mohammed Ali for his contributions in improving the visualization of CLEMATIS. 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.

12. REFERENCES

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

[2] Phormer PHP photo gallery. http://p.horm.org/er/.

[3] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. Technical Report UBC-SALT-2014-001, University of British Columbia, 2014. http://salt.ece.ubc.ca/publications/docs/UBC-SALT-2014-001.pdf.

[4] D. Amalfitano, A. Fasolino, A. Polcaro, and P. Tramontana. The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis. *Innovations in Systems and Software Engineering*, pages 41–57, 2014.

[5] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *Proceedings of the International Conference on Dependable Systems & Networks (DSN)*, pages 403–410. IEEE Computer Society, 2011.

[6] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*, pages 473–484. ACM, 2013.

[7] A. Cockburn, A. Karlson, and B. B. Bederson. A review of overview+detail, zooming, and focus+context interfaces. *ACM Computing Surveys*, 41(1):2:1–2:31, 2009.

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

[9] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355, 2011.

[10] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[11] R. Gentleman and R. Ihaka. The R project for statistical computing. http://www.r-project.org.

[12] Google. Mutation Summary Library. http://code.google.com/p/mutation-summary/.

[13] P. Li and E. Wohlstadter. Script InSight: Using models to explore JavaScript code from the browser view. In *Proceedings of the 9th International Conference on Web Engineering (ICWE)*, pages 260–274. Springer-Verlag, 2009.

[14] J. Maras, J. Carlson, and I. Crnkovi. Extracting client-side web application code. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 819–828. ACM, 2012.

[15] J. Maras, M. Stula, and J. Carlson. Generating feature usage scenarios in client-side web applications. In *Proceeding of the International Conference on Web Engineering (ICWE)*, pages 186–200. Springer, 2013.

[16] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for Javascript applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 159–174. USENIX Association, 2010.

[17] P. Montoto, A. Pan, J. Raposo, F. Bellas, and J. López. Automating navigation sequences in Ajax websites. In *Proceedings of the International Conference on Web Engineering (ICWE)*, pages 166–180. Springer, 2009.

[18] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 55–64. IEEE Computer Society, 2013.

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

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

[21] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498. ACM, 2013.

[22] W3C. Document Object Model (DOM) level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events/, 13 November 2000.

[23] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer, 2000.

[24] A. Zaidman, N. Matthijssen, M.-A. Storey, and A. van Deursen. Understanding Ajax applications by connecting client and server-side execution traces. *Empirical Software Engineering*, 18(2):181–218, 2013.