# Understanding Asynchronous Interactions in Full-Stack JavaScript

Saba Alimadadi Ali Mesbah Karthik Pattabiraman
University of British Columbia
Vancouver, BC, Canada
{saba, amesbah, karthikp}@ece.ubc.ca

## ABSTRACT

JavaScript has become one of the most popular languages in practice. Developers now use JavaScript not only for the client-side but also for server-side programming, leading to "full-stack" applications written entirely in JavaScript. Understanding such applications is challenging for developers, due to the temporal and implicit relations of asynchronous and event-driven entities spread over the client and server side. We propose a technique for capturing a behavioural model of full-stack JavaScript applications' execution. The model is temporal and context-sensitive to accommodate asynchronous events, as well as the scheduling and execution of lifelines of callbacks. We present a visualization of the model to facilitate program understanding for developers. We implement our approach in a tool, called SAHAND, and evaluate it through a controlled experiment. The results show that SAHAND improves developers' performance in completing program comprehension tasks by increasing their accuracy by a factor of three.

## 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, asynchronicity, full-stack JavaScript

## 1. INTRODUCTION

JavaScript has been selected as the most popular programming language for three consecutive years [42] and it is the most used language on GitHub [23]. JavaScript has been the lingua franca of client-side web development for some years. But platforms such as Node.js [34] have made it possible to use JavaScript for writing code that runs outside of the browser. As such, "full-stack" applications written entirely in JavaScript from client-side to the server-side have also seen an exponential growth recently. Node.js provides a light-weight, non-blocking, fast, and scalable platform for writing network-based applications. It is also more convenient for web developers to use the same language for both front- and back-end development. Despite all the advantages, this approach imposes many challenges on the developers' comprehension of the dynamic execution of a web application. Understanding an application is the necessary first step of almost every software engineering task.

There are three groups of challenges involved in understanding the execution on the client side, the server side, and their interactions. First, JavaScript is a single-threaded language and thus callbacks are often exercised to simulate concurrency. Nested and asynchronous callbacks are used regularly [16] to provide capabilities such as non-blocking I/O and concurrent request handling. This use of callbacks, however, can gravely complicate program comprehension and maintenance — a problem coined as "callback hell" on the web by developers. Second, the Document Object Model (DOM) and custom events, timers and XMLHttpRequest (XHR) objects interact with JavaScript code on the client and server to provide real-time interaction, all of which complicate understanding. Moreover, Node.js deploys the event-loop model for handling and scheduling asynchronous events and callbacks, the improper use of which can lead to unexpected behaviour of the application. Finally, client and server code communicate through XHR messages, and multiple messages (and their responses) can be in transit at a given time. As in any distributed system, there is no guarantee on the order or time of the arrival of requests at the server, and responses at the client. The uncertainty involved in the asynchronous communication makes the execution more intricate and thus more difficult to understand for developers.

Despite the popularity of JavaScript and severity of these challenges, there is currently no technique available that provides a holistic overview of the execution of JavaScript code in full-stack web applications. The existing techniques do not support full-stack JavaScript comprehension [4, 18, 27, 35, 47]. In our earlier work, we proposed a technique, called CLEMATIS [3], for understanding client-side JavaScript. CLEMATIS is, however, only designed for client-side JavaScript, and is agnostic of the server, where most of the program logic is located in full-stack applications.

In this paper, we present a technique called SAHAND, to

help developers gain a holistic view of the dynamic behaviour of full-stack JavaScript applications. Our work makes the following contributions.

- We propose a novel temporal and behavioural model of full-stack JavaScript applications. The model is context-sensitive and creates lifelines of JavaScript execution on both the client and server sides. The model connects both sides through their asynchronous communications, to provide a holistic view of the application behaviour.
- We create a visual interface for displaying the model to the developers, to help them understand the underlying mechanisms of execution. We treat the model as a multi-variate time series, based on which, we create a temporal visualization of the lifelines of JavaScript execution.
- We implement our approach in a tool called SAHAND [39]. The tool is browser-independent and non-intrusive. SAHAND can handle the simulated concurrency of JavaScript through asynchronous execution of callbacks, XHR objects, timers, and events.
- We evaluate our approach through a controlled experiment conducted with 12 participants. Our results show that using SAHAND helps developers perform program comprehension tasks three times more accurately.

## 2. CHALLENGES AND MOTIVATION

To comprehend the behaviour of a full-stack web application, one must understand the full lifecycle of a feature on both the client and server sides. We elaborate on some the challenges involved using the examples illustrated in Figures 1–3. These are simple examples and the challenges are more potent in large and complex applications.

### 2.1 Challenge 1: Server-Side Callbacks

**Receiving requests at the end points.** Various types of HTTP requests are received at the end points on a server. Node.js applications have one or more handlers assigned to each incoming request. Each of the handlers can change the flow of execution, return a response to the client, or pass the execution to the next handler. The ability to register anonymous functions, or arrays of functions, can complicate the process of understanding and maintaining the handling and routing the requests.

*Example.* The example of Figure 1 depicts an end point for receiving a `GET` request (lines 12–18 use Express.js APIs [14]). Three items are registered as handlers of the `/locate` request. First, an anonymous function is registered (lines 12–15), which can return a response to the client conditionally (lines 13–14) and prevent the execution of the remaining handlers. The second assigned handler (line 16) is an array of callback functions `cb1()` and `cb2()` (line 8). An additional function `cb0()` can be pushed to the array at runtime based on dynamic information (lines 9–11). `cb0()` can itself affect the control flow and send a response to the client in a specific scenario (lines 3–4). Finally, another anonymous function is added to the list of request handlers (line 16). Understanding how a request is received and routed in the server depends on understanding the complex control flow of all these handlers. This task becomes more challenging as the number of handlers increases in practice.

**Callback hell.** Functions are first-class citizens in JavaScript. They can be passed as arguments to other functions and be executed later. Callback functions are widely used in JavaScript applications [16]. However, It is not trivial to understand the JavaScript code that deploys callbacks. In many cases callbacks are nested (up to eight levels deep [16]) or are assigned in loops, which negatively impacts the readers' ability to follow the data and the control flow. This problem is know as the *callback hell* by developers [8]. To aggravate the situation, Node.js deploys the event loop model for scheduling and organizing callbacks. The event loop is not visible to the developers, but it determines the asynchronous execution on the server side.

```
1  var cb0 = function (req, res, next) {
2     var region = locateClient(req.body.client)
3     if (region.ASIA) {
4         res.send(customizedRes(req.body.content))
5     }
6     next()
7  }
8  var cbacks = cb1, cb2
9  if (user.isLoggedIn) {
10    cbacks.push(cb0);
11 }
12 app.get('/locate', function(req, res, next) {
13    if (req.header('appStats'))
14        res.send(statCollectionResponse(req.body.↩
              stats))
15    next();
16 }, cbacks, function(req, res) {
17    // do stuff
18 })
```

**Figure 1: Receiving HTTP requests at an end point**

```
1 app.post('/cparse', function(req, res) {
2  customParse(req.body, function(er, list) {
3     list.forEach(function (row, index) {
4        buildScript(row, req.body.format).↩
              extractArgs(row, function (instType) {
5           row.forEach(function (arg, i) {
6              resolveAliases(instType, arguments0);
7  }) }) }) })
8    // send response back
9 })
```

**Figure 2: Callback hell**

*Example.* The code in Figure 2 depicts a simple example of callback hell. Many callback functions are passed as arguments to other functions in a nested manner (lines 2–5). Callbacks can also get assigned in loops. In the case of our example (lines 3–5), the same anonymous function is assigned as a callback for all iterations of a loop.

### 2.2 Challenge 2: Asynchronous Client Side

There are two asynchronous events typically used in the client side. First, asynchronous XHR messages are used to seamlessly communicate with the server without changing the state. Second, timing events are utilized for performing periodic tasks, or tasks that must take place after a temporal delay. To handle asynchronous events, developers typically use callbacks which are triggered when the event occurs. However, mapping the observed functionality of the event to its original source is a challenging task for developers. This is especially so when the source and the callback are often semantically and temporally separate.

*Example.* The sample code in Figure 3 displays a simplified client-side JavaScript code. The `updateUnits()` function (line 1) posts a set of XHR requests to the server in a loop (lines 2–7). Each of these messages has a callback function that is invoked upon receipt of the server's response. The callback function of all sent is the same anonymous func-

```
1  function updateUnits() {
2    for (var i = 0; i < unit.length; i ++) {
3      (function(i) {
4        $.post(extractUrl(i), function(data) {
5          if (data.requiresAlert())
6            setTimeout(extractMessage(data), ↩
                 msgDelay);
7        });
8  } } })(i);
9  function periodicUpdate() {
10   $.get('/pupdate', function(data) {
11     // do stuff
12   });
13 }
14 setInterval(periodicUpdate, updateCycle);
```

**Figure 3: Asynchronous client-side JavaScript**

tion (lines 4–7). Based on the content of the response data, a timeout may be set that will execute after a certain delay (line 6). In another part of the code, an interval is set that executes the `periodicUpdate()` function at periodic intervals throughout the lifecycle of the application. `periodicUpdate()` in turn sends a get request to the server and continues its execution upon arrival of the response.

## 2.3 Challenge 3: Network Communication

The server and the client communicate through request/response messages. Hence, the role of the network layer needs to be taken into account to obtain a holistic overview of the execution. The requests do not necessarily arrive at the server in the same order as they are sent on the client side. The processing times of different requests can vary on the server side as well. Moreover, after the responses are sent from the server, there is no guarantee on the time and order in which they will arrive at the client. Observing the behaviour of the application as a whole on both client and server sides is non-trivial. However, this is necessary for developers to understand the full functionality of the features throughout their lifespan.

# 3. APPROACH

In this section, we first present the building blocks of our model. We then discuss the different steps of our approach and how they contribute to the generation of the model.

## 3.1 Temporal and Context-Sensitive Model

Our approach creates a custom directed graph of the context-sensitive executions of events and functions during their lifespan. The model is designed to accommodate the temporal nature of function executions and the asynchronous scheduling mechanisms of full-stack JavaScript. The relations of functions and (a)synchronous events are also temporal to reflect the precise dynamic and asynchronous behaviour of the application. We use the notations introduced here to show how our approach creates the model based on dynamic analysis.

**Vertices.** The vertices of the graph can be events or lifelines of function executions:

$$\begin{array}{lcll} V & ::= & LL & \text{lifeline of a function execution} \\ & | & E & \text{(a)synchronous client/server event} \end{array}$$

Function executions are the focal points of the model. Each function can go through four phases in its lifecycle. Hence, a lifeline of the $i_{th}$ execution of function $f$ at time $\tau$ during execution ($LL < f, i, \tau >$) manifests as one of the following phases:

$$\begin{array}{lcll} LL < f, i, \tau > & ::= & Sch(f) & \text{scheduled : as a callback} \\ & | & Act(f) & \text{active: being executed} \\ & | & Ina(f) & \text{inactive: in stack, but another function is active} \\ & | & Ter(f) & \text{terminated: execution has finished} \end{array}$$

To understand the lifeline of each execution, the model must account for all these phases. There can be a maximum of one scheduling phase per function execution, depending on whether it was triggered asynchronously. This means $Sch(f)$ can occur 0 or 1 times in the beginning of a lifeline. Each execution has at least one active phase ($Act(f)$). If the function invokes another function, the callee becomes active, and the caller becomes inactive until the execution of the callee is finished. Hence, after an initial active phase, a lifeline can contain an arbitrary number of $\{Ina(f), Act(f)\}$ pairs, before its execution is finally terminated ($Ter(f)$). However, there are cases where the execution is left unterminated, for instance due to exceptions, or ending the execution before a scheduled callback occurs. In general, the lifeline of function $f$ can be depicted as:

$$LL(f) = [Sch(f)]Act(f)(Ina(f)Act(f)) * [Ter(f)]$$

The other type of nodes included in our model are *events*. The events can be synchronous or asynchronous, and can be triggered on the client or the server code. Capturing the events and extracting their relations with the rest of the entities in the application is crucial for program understanding. Table 1 summarizes the information required for analyzing various types of vertices that is captured by our approach, in addition to the time of event occurrence.

**Table 1: Types of vertices in the model graph**

| Event Type | Node | Client/ Server | Information gathered |
|---|---|---|---|
| DOM event | $V_e$ | client | user input information, DOM element, handler function |
| Custom event | $V_e$ | client | Custom Event type, DOM element, handler function |
| Node.js event | $V_e(s)$ | server | Custom event type, registered function |
| Timeout set | $V_{ll}(t)$ | client& server | Custom ID, delay, callback function, setter function |
| Timeout callback | $V_{ll}(t)$ | client& server | Custom ID, callback function, setter function |
| XHR send | $V_{ll}(x)$ | client& server | Custom ID, sent data, callback function, opening and sending functions |
| XHR callback | $V_{ll}(x)$ | client& server | Custom ID, response data, callback function, opening and sending functions |

**Edges.** The edges of the graph have three primary attributes, namely time, type, and direction.

Function lifelines are temporal entities over a contiguous time period. A lifeline can interact with other lifelines and events at multiple points in time during its lifespan. The edges must preserve the temporal aspects of the interactions, and reflect them in the model.

The type of each edge represents the type of interaction between the two involved graph nodes. Function lifelines can interact with each other and with events through various types of relations, which are summarized in Table 2.

The direction of an edge represents the direction of the

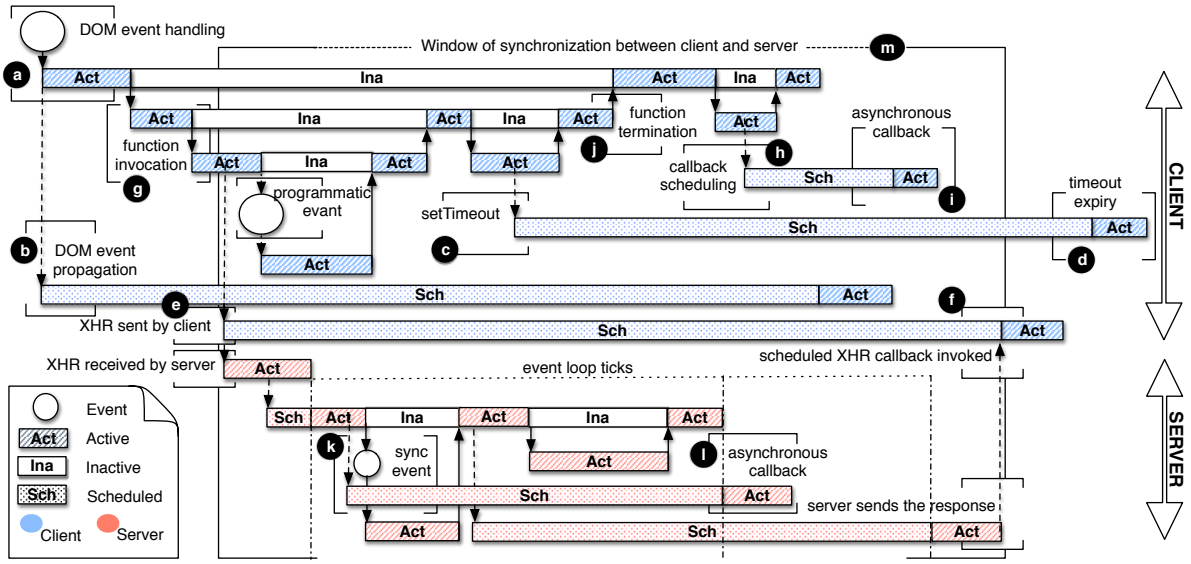


**Figure 4: A sample temporal, context-sensitive and asynchronous model of events, lifelines, and interactions.**

**Table 2: Interaction Edges**

| Edge | Relation | Src | Dst | Sync | Gathered information |
|---|---|---|---|---|---|
| $E_c$ | calls | $LL$ | $LL$ | yes | args, context info |
| $E_t$ | terminates | $LL$ | $LL$ | no | return value |
| $E_{cs}$ | schedules | $LL\|E$ | $LL$ | no | callback type |
| $E_{ss}$ | schedules (s) | $LL$ | $LL$ | no | callback type |
| $E_{ts}$ | timeout set | $LL$ | $LL$ | no | delay |
| $E_{xs}$ | xhr send | $LL$ | $LL$ | both | data |
| $E_t$ | triggers | $LL$ | $E$ | yes | event type |
| $E_e$ | emits | $E$ | $LL$ | yes | event type |

control flow between the involved nodes, which depends on the type of the edge.

Table 3 summarizes the algorithm of creating the model graph based on a selective trace of execution. The rows of the table are the transactions in the trace, and the columns formulate the handling of nodes, edges, and the logic of the algorithm for each transaction. Figure 4 provides a schematic representation of the model. We refer to the algorithm table and the model figure throughout the rest of the section, as we discuss the formation of the model.

## 3.2 Client-Side Analysis

On the client side, each function is either invoked directly by another function, or is triggered by a DOM event, a scheduled callback (including timing events) or a response to a request sent earlier. Next, we discuss how we create the client-side model based on these entities and their relations.

**Events and DOM Interactions.** Our approach captures both DOM and custom client-side events. For each event, we gather information on the involved DOM element, the type of user action or programmatic event, the user input and the invoked handler. Furthermore, our previous study [2] shows that around 14% of the triggered handlers are not invoked directly by an event. These handlers are indirectly called through event propagation mechanisms of JavaScript, where a single event can trigger multiple handlers of the ancestors of the target element [43]. Thus, we capture propagated handlers and their relations with the original events.

Upon invocation of the original handler, we create a node representing the event and add it to the model (Table 3, row 1 & Figure 4, a). The node contains information about the target DOM element and the input data (if applicable). If the call stack at the time of event is empty and the event can be handled immediately, a new lifeline is created for the handler, and is initialized with an *active* phase. However, if the call stack is not empty and the browser thread is executing other JavaScript code, the lifeline will start with a *scheduled* phase, which will terminate and enter an *active* phase as soon as the stack and waiting event queue are empty and the handler can be invoked. For each propagated handler, a new lifeline is created (linked to the same event node of the original event) that is initialized with a *scheduled* phase (Table 3, row 2 & Figure 4, b). The lifeline enters the *active* phase after the execution of the original (preceding) event and its synchronous callers is finished, but before any asynchronous event/callback scheduled in the preceding event handler.

A new edge is created from the event node to each of the newly created handler lifelines. The edge to the original handler's lifeline maintains the user action. The edges to the propagated lifelines (if any) will indicate the occurrence of the propagation as well as the initial user action. We intercept event handling by instrumenting the registration of event listeners in the code. Our tracing technique then retrieves information regarding the element, the event, and the handler(s) once the event occurs.

**Timeouts.** There is often temporal and semantic separation between `setTimeout()` and the callback function. Even in the case of immediate timeouts, the callback is not executed until the JavaScript call stack is empty, and there are no other preceding triggered DOM and asynchronous events that are yet not handled. Hence, a `setTimout`'s delay is merely the *minimum* required time until the timeout expires.

We intercept all timeouts by replacing the browser's `setTimeout()` similar to our previous work [3].

Each timeout must be set within the current *active* phase of a lifeline. Upon setting a timeout, we create a new lifeline, representing the callback function execution, that is

**Table 3: Creation and extension of the behavioural graph based on the operations**

$\sqcup_{ll}$: Stack of function lifelines. $\bigcirc$: Node.js event loop. $\Pi_{fe}$: List of fired DOM events. $\Pi_{ue}$: List of unhanded DOM events. The time $\tau$ and the side (server/client) are included in all transactions.

| Row | Operation Type | Node | Edge | Instructions |
|---|---|---|---|---|
| 1 | Original DOM event <ev,el> | $e := newV_e(ev, el)$<br>$ll := newV_{ll}(ev \rightarrow$ handler$)$<br>$\sqcup_{ll} \leftarrow \sqcup_{ll} \cup ll$<br>$\Pi_{fe} \leftarrow \Pi_{fe} \cup e$ | $d = newE_e($src : $e$, dst : $ll$, action$)$ | if(JS active)<br>ll.init(Phase.Sch)<br>$\Pi_{ue} \leftarrow \Pi_{ue} \cup e$<br>O.W.<br>ll.init(Phase.Act) |
| 2 | Propagated DOM events ($\Sigma pe$) | $e_p := \Pi_{fe} \rightarrow$ head<br>$\forall e_i \in \Sigma pe$<br>$ll_i := newV_{ll}(pe \rightarrow$ handler$)$<br>$\sqcup_{ll} \leftarrow \sqcup_{ll} \cup ll$ | $\forall e_i \in \Sigma pe$<br>$d := newE_e($src : $e_p$, dst : $ll_i, e_p \rightarrow action)$ | $\forall e_i \in \Sigma pe$<br>$ll_i.init(Phase.Sch)$<br>$\Pi_{fe} \leftarrow \Pi_{fe} \cup (e_i)$ |
| 3 | Timeout set | $to-id := newuniqueTOID()$<br>$ll_c := \sqcup_{ll} \rightarrow$ head<br>$ll := newV_{ll}(to-id, delay)$ | $d := newE_{ts}($src : $ll_c$, dst : $ll$, $delay)$ | ll.init(Phase.Sch)<br>$if(serverside)$<br>$\bigcirc \leftarrow \bigcirc \cup <$ TO$, ll >$ |
| 4 | Timeout callback | $ll := \sqcup_{ll}.get($TO $\rightarrow to-id)$ | | $ll.$end$(phase.Sch)$<br>$ll.$start$(phase.Act)$<br>$if(serverside)$<br>$\bigcirc.$pop$(ll \rightarrow to-id)$ |
| 5 | XHR send | $xhr-id := newuniqueXHRID()$<br>$ll_c := \sqcup_{ll} \rightarrow$ head<br>$ll := newV_{ll}(xhr-id, url, method)$ | $d := newE_{xs}($src : $ll_c$, dst : $ll$, $data)$ | ll.init(Phase.Sch)<br>$if(serverside)$<br>$\bigcirc \leftarrow \bigcirc \cup <$ XHR$, ll >$ |
| 6 | XHR callback | $ll := \sqcup_{ll}.get($XHR $\rightarrow xhr-id)$ | | $ll.$end$(phase.Sch)$<br>$ll.$start$(phase.Act)$<br>$if(serverside)$<br>$\bigcirc.$pop$(ll \rightarrow xhr-id)$ |
| 7 | Server events | $ll_c := \sqcup_{ll} \rightarrow$ head<br>$e := newV_e(ev)$<br>$ll := newV_{ll}(ev \rightarrow$ handler$)$<br>$\sqcup_{ll} \leftarrow \sqcup_{ll} \cup ll$ | $d = newE_e($src : $ll_c$, dst : $e$, $e \rightarrow$ type$)$<br>$d = newE_e($src : $e$, dst : $ll)$ | $ll.init(Phase.Act)$<br>$ll_c.init(Phase.Ina)$ |
| 8 | Callback scheduling | $ll_c := \sqcup_{ll} \rightarrow$ head<br>$ll := newV_{ll}(callback)$ | $d = newE_{cs}($src : $ll_c$, dst : $ll)$ | $ll.init(Phase.Sch)$<br>$if(serverside)$<br>$\bigcirc \leftarrow \bigcirc \cup <$ CB$, ll >$ |
| 9 | Callback invokation | $ll := \bigcirc \rightarrow$head | | $ll.end(Phase.Sch)$<br>$ll.start(Phase.Act)$<br>$if(serverside)$<br>$\bigcirc.pop(<$ CB$, ll >)$ |
| 10 | Function invokation | $ll_c := \sqcup_{ll} \rightarrow$ head<br>$ll := newV_{ll}(function)$ | $d = newE_c($src : $ll_c$, dst : $ll)$ | $ll.start(Phase.Act)$<br>$ll_c.start(Phase.Ina)$ |
| 11 | Function termination | $ll_c := \sqcup_{ll} \rightarrow$ pop<br>$ll_p := \sqcup_{ll} \rightarrow$ head | $d = newE_t($src : $ll_c$, dst : $ll_p)$ | $ll_c.start(Phase.Ter)$<br>$ll_p.start(Phase.Act)$ |

initialized with a *scheduled* phase in the beginning. An edge is created from current *active* lifeline to the newly created *scheduled* lifeline (Table 3, row 3 & Figure 4, c). The new edge includes the data regarding the details of the timeout (delay and passed arguments). The lifeline proceeds to an *active* phase when the timeout expires and the callback is executed (Table 3, row 4 & Figure 4, d).

**XHRs.** The server is treated as a blackbox at this stage. Our technique captures the information regarding sending the request (e.g., method, data) and the means of receiving the response (e.g., response data, callback) and how it is handled on the client side (sync or async).

When the *active* lifeline sends a request, we create a new node, initialized with a *scheduled* phases (Table 3, row 5 & Figure 4, e). A new edge connects the current *active* lifeline to the new *scheduled* one. The new edge encapsulates information regarding the request (type of the request, sync/async, url, possible sent data). When the response is received, the captured information is completed with the response data (Table 3, row 6 & Figure 4, f).

**Function executions.** Our analysis of function executions is similar to creating a dynamic call graph that is temporal and context sensitive. Our method accumulates a trace of function executions initiated by regular function calls, as well as the function executions caused by any of the mechanism discussed above.

The lifeline node representing the lifecycle of a function execution preserves the temporal states of the function and their respective edges represent their relations with the rest of the application. Lifelines and their edges map to particular executions of functions and maintain the information regarding the context of that execution (e.g., caller information, dynamic arguments, return values).

There are three possible cases of function invocation, each of which is handled differently. First, when a function is invoked without passing any callbacks, a new lifeline node is created (Table 3, row 10 & Figure 4, g). The new lifeline is initialized with an *active* phase, and the execution continues from there. Meanwhile, an *inactive* phase is added to the caller lifeline, which finishes and enters the *active* phase when the callee returns. Second, when a function is invoked with a callback, but the callback is not immediately (synchronously) executed, a new lifeline is added for the callee. The lifeline is initialized with a *scheduled* phase and is not marked as active yet (Table 3, row 8 & Figure 4, h). Finally, when a function is invoked with a callback function, and the passed callback function is executed, our method retrieves the existing lifeline where the callback is already scheduled, and transitions it to an *active* phase (Table 3, row 9 & Figure 4, i). Synchronous callback invocations are treated as regular function calls.

Every time a new lifeline is created, it is added to a stack of lifelines ($\sqcup_{ll}$). When the execution of a function lifeline terminates after an *active* phase, the lifeline enters the *ter-*

*minated* phase, and is popped (Table 3, row 11 & Figure 4, j).

Our technique instruments all JavaScript functions in order to gather a detailed execution trace dynamically. JavaScript functions can have different return statements in different intra-procedural execution paths. Hence, our method instruments all existing return statements individually. Should a path terminate without a return statement, we inject a different logging function for marking the termination of the function. Function invocations are wrapped within our trace functions. All arguments are examined and if they are functions, additional instrumentation is added to distinguish potential callback scheduling. The analysis recursively checks the subprogram and if the potential callback is eventually invoked, the actual callback invocation are annotated through additional tracing code. Further, to distinguish between multiple invocations of the same function, we maintain its contextual information in the caller function, and update it per execution of the callee. We pass the updated state to the callee through our instrumentation, where it is used to customize the collected trace for that specific execution.

## 3.3 Server-Side Analysis

Our approach tracks the incoming requests from their arrival at the endpoints of the server. The endpoint layer typically contains minimal logic, but can highly affect the flow of execution (e.g., routing to different handlers, sending the response back). The essence of this part of the analysis is similar to the client side. However, the focus at this stage is on challenges specific to server-side JavaScript development, such as the callback hell and the server-side events. Before discussing our analysis of the server-side behaviour of a JavaScript application, we need to describe the role of the event loop on the server.

**Event loop.** The event loop consists of a *queue* of asynchronous events waiting to be executed at each tick of the loop when the stack (of synchronous functions) becomes empty.

The stack, the event loop, and the mechanisms of pushing/popping events in/from the loop determine the order and time of asynchronous events execution. Hence, we need to consider them in our analysis. For example, there are three ways of scheduling an immediate callback in a Node.js application, namely Immediate `setTimeout()` (a timeout with 0 delay), `setImmediate()` and `process.nextTick()` However, the order and time of execution of the callbacks using each method differs based on the contents of the event loop. `process.nextTick()` pushes the callback to the front of the event loop queue regardless of the contents of the queue. `setImmediate()` enters the callback into the queue after the I/O operations, but before timing callbacks. `setTimeout(0)` pushes the callback to the end of the queue (after all existing callbacks). Hence, even though the delay is set to 0, it may be executed with more delay in practice. This shows the importance of reflecting the exact dynamic execution of asynchronous JavaScript in helping developers understand the behaviour of the application.

**Callbacks.** We capture all callback invocations (synchronous or asynchronous), their relations with the events in the loop that triggered them, and the consequences of their executions. When a callback is scheduled, a new lifeline node is created in the server-side of the model for the callback function, which starts with a *scheduled* phase. The respective asynchronous event is added to the list of events in the loop. Later when the event is popped and the callback is invoked, the lifeline is retrieved, the *scheduled* phase is terminated and the *active* phase starts. This part of the analysis is similar to that of the client side, although we consider the event loop and the respective scheduling methods (Table 3, rows 8–9 & Figure 4, k).

**Events.** There is no DOM on the server side and hence there are no user events. However, developers can take advantage of Node.js events to trigger custom events and invoke their handlers using `EventEmitter`s. A major difference between `EventEmitter`s and client-side events is that the former are synchronous in nature and thus do not occur in the event loop. Although these events can be emitted in asynchronous functions, the invocation of handlers is different from asynchronous handlers and thus has to be analyzed differently. In our model, for each emitted event a new event node is created. An edge connects the current *active* lifeline to the event. The current lifeline enters an *inactive* phase. A new lifeline in the *active* phase is created, which is connected to the new event node through an edge. When the execution of the handler finishes, the *inactive* phase of the original lifeline will finish and it will be *active* again (Table 3, row 7 & Figure 4, l).

## 3.4 Connecting Client and Server

In a typical web application, execution starts on the client side with an event, which can trigger an asynchronous request to the server. This entails code execution on the server and sending the response back to the client, which will complete the lifecycle of interaction when the execution terminates on the client side. However, JavaScript execution can continue on the client side even while the asynchronous request is being handled on the server. The synchronization of the client and server side executions of a full-stack feature occurs in our model when the two ends communicate through XHR objects (Table 3, rows 5–6 & Figure 4, m).

We create temporal models for both client and server sides. Due to the network layer in the middle, each side initially treats the other side as a black box. The connections between the two sides are made by marking and tracking the XHR objects. Because the client and the server may have different clocks, we cannot use the timestamps produced by their respective clocks for synchronization. Hence, we track all communications between the client and the server. This way, our approach can find windows of synchronization between the two sides, which start by a request arriving at the server and end when the response is sent back to the client. While this approach only provides a relative sense of time globally, in practice, this is sufficient for the purposes of our approach, since it is accurate for each specific full-stack interaction.

## 3.5 Visualizing the Model

In the last step of the approach, we create a visual interface based on our inferred temporal model. The visualization shows the temporal characteristics of the lifelines, events, and their relations, to facilitate understanding of execution patterns. There are three major criteria that need to be considered in creating a visualization for temporal data [1].

**1) Time.** There are two types of *temporal primitives*. *Time points* are specific lines on the time axis. *Time intervals* constitute ranges on the time axis. our visualization uses time points to represent events and event loop ticks, and

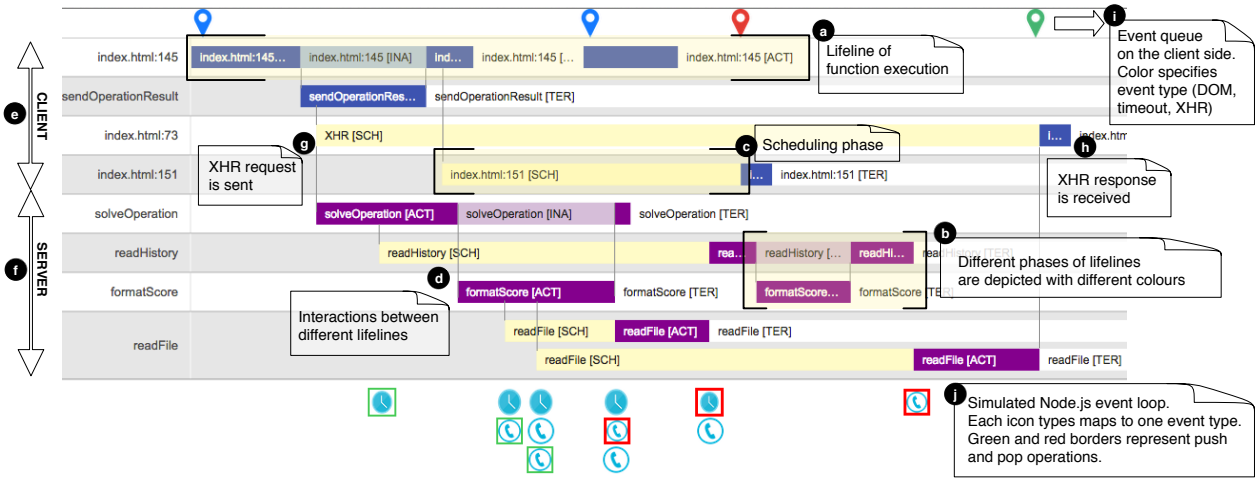


**Figure 5: A snapshot of the visualization.**

time intervals, to depict function lifelines and the phases of their lifespans. The time axis can follow one of the common *structures of time*: *linear*, *cyclic* or *branching*. The structure of our time axis is a mixture of subsets of both linear and branching structures. As a linear structure, it follows the natural perception of time, where time passes from past to future, and the temporal primitives are ordered (as opposed to a cyclic perception of time). Moreover, similar to the branching structure, multiple edges can exit a single temporal primitive node. But unlike branching, the outgoing edges actually occur at different timestamps and do not represent alternatives.

**2) Data.** Data is the second criterion of time-series visualization and can be examined from different aspects. The *frame of reference* for our data is *abstract*, since it does not encompass a *spacial* layout. The data is *multivariate* since each node contains a set of information (variables) accumulated for the event or lifeline it represents.

**3) Representation.** The final criterion is the representation of the time-relevant data. This can be of two kinds: *static* or *animated*. We deploy a static approach, meaning that our visualization makes all the information available on screen on demand, and hence the viewers can concentrate on the data itself and make comparisons on different parts of the model. We collect multiple variables for each node. Presenting them all to the viewers can be overwhelming and obstruct the overview of the whole model. We utilize basic interaction to allow users to view information on demand by clicking on any of the events.

Lifeline visualization has been extensively used for displaying histories in domains such as medical records [37]. We incorporate custom lifeline visualization in the interface of our behavioural model.

**Visualization example.** Figure 5 displays a sample snapshot of the interface. The main frame of the visualization depicts our lifelines. Each lifeline represents a particular context-sensitive execution of a function (a). Different phases of a lifeline are depicted as rectangles with different colours on the lifeline (b). If a lifeline represents an asynchronous callback, it will start with a scheduling phase (c). Lines between caller/scheduler lifelines and their respective callee/scheduled lifelines display the edges between the function executions (d).

Once an XHR is sent to the server, an edge connects the the scheduled callback to the handler on the server. However, due to the potential network delays, the handler execution may start later than when the request is sent (g). The request is then dispatched and handled, until the response is sent back to the client (h). In addition to the main panel, there are two smaller panels to represent the client-side events and the server event loop. The first row on the client panel (i) represents the DOM events, and timeout and XHR callbacks that occur on the client side. The colour and label of each cell on this row depict the type of each event. The server's event loop is depicted at the bottom of the server panel (j). Every time a user-defined callback is scheduled, a timeout is set, or an XHR is sent, an event is pushed to the event loop (marked with a green border). When it is a callback's turn to be executed, the corresponding event is popped from the loop (marked with a red border), while the remaining events (if any) can still be observed in the loop. Finally, the horizontal axis below both panels represents the time.

## 3.6 Implementation

We implemented our approach in a tool called SAHAND. We instrument JavaScript code on the server side at startup, using a proxy server built with Node.js and Express.js, and on the client-side code on the fly. We create an AST of the code using Esprima [12], instrument the AST using Estraverse [13], and serialize the AST back into JavaScript code with Escodegen [11]. The visualization is built on top of the timeline view of Google chart tool [17]. SAHAND is publicly available [39].

# 4. EVALUATION

We conducted a comparative controlled experiment [46] to investigate the effects of using SAHAND on the performance of developers when understanding full-stack web applications. Our experimental dataset is available online [39].

## 4.1 Experimental Setup

The participants in our study are asked to perform three comprehension tasks on a full-stack JavaScript application.

**Experimental subjects.** We recruited 12 participants for

**Table 4: Comprehension tasks of the experiment**

| Task | Description |
|---|---|
| T1 | Understanding full-cycle implementation of submitting a correct answer on the client side. |
| T2.a | Understanding time-triggered feature of terminating game rounds managed by the server. |
| T2.b | Detecting a potential for an event-race condition during client-server communications. |
| T3.a | Understanding the purpose of a new feature involving nested callbacks. |
| T3.b | Understanding the asynchronous execution of a function involved in nested callbacks. |

the experiment, 11 males and one female, aged between 23 and 33. All of the participants are graduate students at UBC who regularly program with JavaScript. None of the participants had used SAHAND prior to the experiments.

**Experimental object.** We use Math-Race [25] as our experimental object. It is an open-source, online game that allows multiple players to compete over solving simple mathematical problems. During timed cycles of the game, they players can answer questions, keep the history of their scores, and enter the game's hall of fame if they achieve high scores. We chose this application because it is a full-stack JavaScript application built on Node.js. It is also relatively small (about 200-300 LOC of JavaScript on each of the client and server sides), and hence it is feasible for our participants to understand its main features during the limited time of the experiment (about 75 minutes). Although it is a small application, it employs many advanced features such as asynchronous events and callbacks. Our participants had never seen Math-Race before the experiment.

**Experimental design.** The experiment had a between-subject design. We divided the participants into two groups. The experimental group used SAHAND for performing a set of comprehension tasks. The participants in the control group were allowed to use any existing web development tool. They all selected Google Chrome's Developer Tools [10], one of the most popular client-side development tools, as they all self-reported as experts in it. We also provided the control group with JetBrain's WebStorm [44], a popular JavaScript IDE, for working with the server-side code of the experimental object. In contrast, the experimental group were only allowed to view the code in addition to SAHAND's visualization, and not permitted to use an IDE or debugger. We limited their access to other tools because we wanted to gain a better control of SAHAND's impact on understanding.

*Task Design.* We designed a set of tasks that represented common comprehension activities performed in normal development proposed by Pacione et al. [36]. Each of our tasks covers multiple activities, and also involves elements specific to JavaScript comprehension. The tasks are summarized in Table 4.

*Variables.* We wanted to measure the performance of developers in performing program comprehension tasks. The dependent variables (DV) should quantify developers' performance. Our design involves two interval dependent variables, task completion *duration* and *accuracy*. We also considered two nominal independent variables (IV). The first IV is the tool (set of tools) used for the experiment, and has two levels. One level is SAHAND, and the other is the set of Chrome's DevTools and WebStorm. The second IV is the expertise level of participants. We wanted to investigate the effects of expertise of the participants on how they comprehend web applications. We classified participants into two groups, namely experts and novices, based on their responses to a pre-questionnaire form (described below).

**Experimental procedure.** This consists of four parts. In the *first part*, the participants completed a pre-questionnaire form where they ranked their expertise in web application development using a 5-point Likert scale, and prior experience with software development in general. We used a combination of their self-reported expertise and experience to assign an expertise score to each participant. The expertise score was used to assign the participant to either the experimental or control group. We manually balanced the distribution of expertise in both groups. We also used the expertise score to assess whether the expertise of participants affects their program comprehension performance. In the *second part* of the experiment, we presented a short tutorial on SAHAND for the experimental group. However, we did not present any tutorial to the control group as they identified themselves as expert in Chrome Developer Tools. Both groups were given a few minutes to familiarize themselves with the settings of the application, the object application, and the tools. In the *third part*, the participants performed the tasks (Table 4). We presented each task to the participants on a separate sheet of paper, and measured the time from when they started the task until they returned the answer. This setup ensured that the time-tracking process was not biased towards either the examiner or the participant. We measured the accuracy of the answers later, based on a grading rubric that we had finalized prior to conducting the study. The accuracy of the tasks could be quantified with a grade between 0 and 100 per task. The tasks and their rubrics, along with the rest of documentations of the study are available online [39]. In the *fourth part*, when the participants finished all of the tasks, they were given a post-questionnaire form. The form asked about their experience with the tool used in the experiment, and its pros and cons. We also solicited participants' opinions on the features they thought would be useful for a web application comprehension tool.

## 4.2 Results

We were interested in observing the effects of tool and expertise on task completion duration and accuracy. Both variables are conceptually dependent although we did not observe a correlation between them in our experiments. Imagine a case where a participant finishes the tasks early thinking she has found the correct answer, but the answer is incorrect or incomplete. In this case, the fast completion of a task is not an improvement, since the purpose of the question is not fulfilled and the participant has not *performed* better. Because of this relationship, we performed a multivariate analysis, where we examined the pair of both duration and accuracy as the dependent variable.

We performed a set of multivariate analysis of variance (MANOVA) tests to investigate the effects of tool and expertise on the integration of duration and accuracy. Using MANOVA entails two advantages for our analysis. First, it can reveal differences that are not discovered by ANOVA. Second, it can prevent type I errors that may occur when multiple independent ANOVA tests are conducted. We performed the MANOVA tests on the total duration and accuracy (combining all tasks). Next, we ran a MANOVA test on each individual task. If the results of a MANOVA test

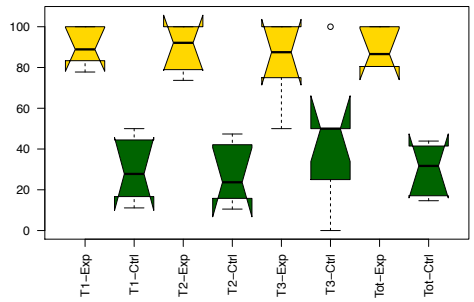


**Figure 6: Accuracy results. Gold plots display experimental (Sahand) group, and green plots display the control group. Higher values are better.**

were significant, we examined the univariate tests (ANOVA) to see if the significance in performance improvement was due to the duration, accuracy, or both.

Examining the total results, we found a significant main effect of tool ($p < .0001$) on the group of accuracy and duration, but no significant main effect of expertise ($p > .05$). For individual tasks too, we found a significant main effect of tool (T1:$p < .001$, T2:$p < .001$, T3:$p < .05$), but not of expertise. We then examined the univariate tests (ANOVA) for each significant result, to find which dependent variable(s) contributed to the significance. From the results, we found that there is a statistically significant difference ($p<.00001$) in **accuracy** between the group using SAHAND (M=89%, SD=10%) and the control group (M=30%, SD=11%). However, we did not find a statistically significant difference for duration ($p > .05$) between the group using SAHAND (M=32:06, SD=5:43) and the control group (M=33:49, SD=6:37).

The above results suggest that that task completion accuracy was the determining factor in the significance of the results of the multivariate tests. The accuracy results are shown in Figure 6. *We find that* SAHAND *helped developers perform comprehension tasks* ***three*** *times more accurately, in about the same amount of time used by the control group.*

## 4.3 Discussion

**"Fast Is Fine, but Accuracy Is Everything".** Using SAHAND significantly improved the accuracy of each individual task in the experiment. The large difference between the means of two groups, and the high confidence of the test results emphasize the impact of the challenges of understanding full-stack JavaScript, even for a simple application as our experimental object. Tasks T1 and T2 were seeking developers' understanding of two of the basic features of the application, whose implementation was divided between both ends of the application. The tasks also involved understanding features such as event propagation on the client-side, and asynchronous time management on the server side. Task T3 required understanding the execution of a nested callback code segment, which can create implicit and intricate connections in the application.

Manual analysis of the answers of the control group showed that they all had an incomplete and sometimes incorrect vision of the full-stack execution of the features. Their mental model of the application's behaviour missed both entities and connections, on both client and server and their interactions. They gained significantly lower accuracy scores, while spending about the same time as the experimental group. On the other hand, SAHAND users were able to see all the involved entities and their relations. The model allowed them to extract the information usually hidden in the application, and finish the tasks much more accurately.

**"It Will Get Better ... in Time".** The results did not show a statistically significant difference of task completion duration between experimental and control groups. Manual investigation of the control group's answers showed that almost all of them had incomplete (and not necessarily wrong) answers for most of the questions. Therefore, it is possible that these participants spent the whole time on a small portion of the answer compared to the experimental group. This means that overall, as the multivariate tests found, SAHAND users performed better than the control group as they used approximately the same amount of time for providing significantly more accurate answers.

Further, none of the participants in the experimental group had seen SAHAND before the experiment. We observed that SAHAND users looked more often at the source code and spent more time analyzing and interpreting the interface at the beginning of the session. However, near the end of task T1, they would shift almost all of their attention on the model while solving the problems. We believe this is due to two main reasons: (1) the users required a short learning phase for performing a real task (although they had a tutorial in the beginning), (2) only after multiple comparisons between the interface and the actual code, were the users able to trust SAHAND as a fair representation of the behaviour. We believe that developers will get faster using SAHAND once these barriers are overcome. Examining the average time spent on each task, we observed that SAHAND users finished T1 only 8% faster than the control group in the beginning of the session. However, by the end of the session, SAHAND users finished T3 32% faster on average. This result strengthens our intuition that by adopting the tool for a longer period of time, users will become much faster in performing the tasks.

**User Feedback.** According to the post-questionnaire forms, all SAHAND users found the tool useful. They particularly liked the overview it provided of the whole interaction. They found the unified client/server view most useful. The participants also found it easy to infer function relations from the model, and liked the abstraction and filtering of details in the visualization. However, some of them mentioned that the context-sensitive depiction of functions can become overwhelming in large interaction sessions. They requested interface features such as direct links to the code, showing connections to the DOM, and integration with a debugger. These are interesting directions for future work.

**Threats to Validity.** The first internal threat is the examiner's bias in measuring the time. We addressed this threat by enforcing a mutual supervision on timekeeping by the examiner and the participant. The start and end time of each task were marked by the exchange of sheets of paper containing the question and the answer of that task between the examiner and the participant. The same threat arises from examiner's bias while marking the accuracy of the tasks. We mitigated this risk by devising the rubrics of each task before conducting the experiments. The rubrics were later used to mark the accuracy of the answers. Another threat is the impact of the expertise level of the participants on their performance in the experiment. We eliminated this threat

by determining the expertise level of participants through a pre-questionnaire form before conducting the experiments. We used this information to rank participants into multiple bins based on their expertise levels, and then used random sampling to assign the members of each bin to one of the control and experimental groups. The tools used by the control group can introduce another threat. We avoided this threat by letting the participants choose the browser development kit for client-side analysis (all chose Chrome). For the server side, we provided them with WebStorm, a popular enterprise IDE for web development. We resolved the bias of the experiment tasks by designing the tasks based on a framework of common comprehension tasks [36]. Using this framework, we also eliminate a potential external threat arising from the representativeness of the tasks. The second external threat is the representativeness of the participants. We addressed this threat by recruiting graduate students who regularly performed (and researched) JavaScript development. Many of the participants had professional development experience during or prior to the time of this work. However, our participants were not full-time professional developers and this could still threaten the validity of our experiment. Finally, to ensure the reproducibility of the experiment, we used an open-source experimental object, and made our tool, the tasks, questionnaires, and the rubrics public [39].

# 5. RELATED WORK

**JavaScript Analysis.** There are numerous static analysis techniques proposed for JavaScript analysis in different domains [15, 20, 21, 26, 32, 41]. We did not choose a static approach, since many event-driven, dynamic and asynchronous features of JavaScript are not well supported statically. Dynamic and hybrid JavaScript analysis techniques have attempted to solve the shortcomings of static analysis [2, 3, 33, 45]. However, existing techniques focus on the client-side and do not consider the server.

Magnus et al. recently proposed a technique to build an event-based call graph for Node.js applications [27]. There are two differences between their work and ours. First, SAHAND considers functions in the graph as temporal and context-sensitive nodes, which can interact with each other and with events throughout different phases of their lifecycle. Second, SAHAND accounts for various means of asynchronous scheduling. It integrates client information, client-server interactions, and asynchronous server execution and creates a behavioural model. It is through this model that SAHAND can provide a holistic and temporal overview of full-stack execution.

**Asynchronous Events.** Different approaches target asynchrony in different domains, such as comprehension, debugging and testing. Frameworks such as Arrows [22] have been proposed to help developers understand and avoid asynchronous errors. Zheng et al. [48] used static analysis to find asynchronous bugs in web applications. WAVE [19] is a testing platform for finding concurrency errors on the client side. Libraries and features such as Async.js [9] and Promises [38] have been adopted to "tame" the asynchronous JavaScript issue. Despite being very useful and promising, Async.js is not native to JavaScript. Both Async.js and Promises require the current and future code to follow specific design and syntactic guidelines, which impede their wide adoption.

**Feature Location, Record & Replay, and Tracing.** Many papers have focused on *locating the implementation* of UI- and interaction-based features [7, 24, 28, 29] in web applications. However, they only retrieve the client-side implementation of a feature, and they require a constant manual effort for selecting the elements or features under investigation. FireDetective [47] is a Firefox add-on that captures the client-server interactions to facilitate comprehension. Although its purpose is similar to SAHAND, it only supports partial Java execution on the server side. Further, it does not support a higher level model or a temporal visualization of the trace.

*Record and replay* techniques aid the understanding and debugging tasks of web applications [5, 6, 30, 31]. The goal of these techniques, however, is to provide a deterministic replay of UI events without capturing their consequences. Unlike SAHAND, they don't collect detailed traces of the execution and only support client-sider JavaScript. Jalangi is a multi-purpose framework for record-replay and dynamic analysis of JavaScript [40]. Unlike SAHAND, it neither extracts full-stack interactions, nor does it provide a high-level visualization of the model as its goals are different from ours.

*Tracing* techniques such as FireCrystal [35] and DynaRIA [4] collect traces of JavaScript execution selectively. In our prior work, we introduced CLEMATIS [3], a tool for code comprehension, which creates a behavioural model of the captured trace. CLEMATIS focuses entirely on the client-side code and does not take into account server-side features such as the routing and handling the requests and responses. Moreover, it does not target callbacks, their scheduling, transmutations, and interactions. Overall, CLEMATIS introduces a more primitive type of model that does not support the temporal nature of execution. Unravel [18] is a more recent tool for supporting developer learning. Similar to our work, these tools provide a high-level abstraction and visualization of the trace. However, all these techniques only focus on the client-side JavaScript. SAHAND, on the other hand, traces, models and connects both client and server side traces with a focus on asynchronous JavaScript execution.

# 6. CONCLUSION

Full-stack JavaScript development is becoming increasingly important; yet there is relatively little support for programmers in this space. This paper introduced SAHAND, a novel technique for aiding developers' comprehension of full-stack JavaScript applications by creating a behavioural model of the application. The model is temporal and context sensitive, and is extracted from a selectively recorded trace of the application. We proposed a temporal visualization interface for the model to facilitate developers' understanding of the behavioural model. The implementation of the approach is available as an open-source Node.js application [39]. We investigated the effectiveness of SAHAND by conducting a user experiment. We found that SAHAND improves developers' performance in completing program comprehension tasks by increasing their accuracy by three times, without a significant change in task completion duration.

# 7. ACKNOWLEDGMENTS

This work was supported in part by an NSERC Strategic Project Grant and a research gift from Intel Corporation. We are grateful to anonymous reviewers of ICSE'16 and all participants of our controlled experiment.

# 8. REFERENCES


[1] W. Aigner, S. Miksch, W. Müller, H. Schumann, and C. Tominski. Visualizing time-oriented data - a systematic view. *Computers & Graphics*, 31(3):401–409, 2007.

[2] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Hybrid DOM-sensitive change impact analysis for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 321–345. LIPIcs, 2015.

[3] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014.

[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*, 10(1):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] B. Burg, A. J. Ko, and M. D. Ernst. Explaining visual changes in web interfaces. In *Proceedings of the ACM User Interface Software and Technology Symposium (UIST)*, pages 259–268. ACM, 2015.

[8] Callback hell, a guide to writing asynchronous JavaScript programs. `http://callbackhell.com`, 2015.

[9] Caolan McMahon. Async.js. https://github.com/caolan/async, 2015.

[10] Chrome devtools. `https://developers.google.com/web/tools/chrome-devtools/`.

[11] Escodegen. `https://github.com/estools/escodegen`.

[12] Esprima. `http://esprima.org/`.

[13] Estraverse. `https://github.com/estools/estraverse`.

[14] Express. `http://expressjs.com/`.

[15] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

[16] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 247–256. IEEE Computer Society, 2015.

[17] Google chart tools. `https://developers.google.com/chart/`.

[18] J. Hibschman and H. Zhang. Unravel: Rapid web application reverse engineering via interaction recording, source tracing, and library detection. In *Proceedings of ACM User Interface Software and Technology Symposium (UIST)*, pages 270–279. ACM, 2015.

[19] S. Hong, Y. Park, and M. Kim. Detecting concurrency errors in client-side java script web applications. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 61–70. IEEE, 2014.

[20] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 59–69. ACM, 2011.

[21] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. Jsai: A static analysis platform for JavaScript. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 121–132. ACM, 2014.

[22] Y. P. Khoo, M. Hicks, J. S. Foster, and V. Sazawal. Directing JavaScript with arrows. *ACM SIGPLAN Notices*, 44(12):49–58, 2009.

[23] A. La. Language trends on GitHub. `https://github.com/blog/2047-language-trends-on-github`, 2015.

[24] J. Lo, E. Wohlstadter, and A. Mesbah. Imagen: Runtime migration of browser sessions for JavaScript web applications. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 815–825. ACM, 2013.

[25] I. Loire. Math-race. `https://github.com/iloire/math-race`.

[26] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 499–509. ACM, 2013.

[27] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven node.js JavaScript applications. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 505–519. ACM, 2015.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software*

*Engineering*, pages 518–529. ACM, 2014.

[33] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 791–802. ACM, 2014.

[34] Node.js. `https://nodejs.org/`.

[35] 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.

[36] 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 (WCRE)*, pages 70–79. IEEE Computer Society, IEEE, 2004.

[37] C. Plaisant, B. Milash, A. Rose, S. Widoff, and B. Shneiderman. Lifelines: Visualizing personal histories. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 221–227. ACM, 1996.

[38] Promises/A+. `https://promisesaplus.com`, 2015.

[39] Sahand: tool implementation and dataset. `http://salt.ece.ubc.ca/software/sahand/`, 2015.

[40] 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, pages 488–498. ACM, 2013.

[41] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 435–458. Springer, 2012.

[42] Stack Overflow. Developer survey. `http://stackoverflow.com/research/developer-survey-2015`, 2015.

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

[44] WebStorm. `https://www.jetbrains.com/webstorm/`.

[45] S. Wei and B. G. Ryder. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 1–26. Springer, 2014.

[46] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[47] 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.

[48] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th international conference on World Wide Web (WWW)*, pages 805–814. ACM, 2011.