

# **Understanding Motifs of Program Behaviour and Change**

by

Saba Alimadadi Jani

B.Sc., University of Tehran, Iran, 2009

M.Sc., Simon Fraser University, Canada, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT  
OF THE REQUIREMENTS FOR THE DEGREE OF

**Doctor of Philosophy**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL  
STUDIES

(Electrical and Computer Engineering)

The University of British Columbia  
(Vancouver)

October 2017

© Saba Alimadadi Jani, 2017

# Abstract

Program comprehension is crucial in software engineering; a necessary step for performing many tasks. However, the implicit and intricate relations between program entities hinder comprehension of program behaviour and change. It is particularly a difficult endeavour to understand dynamic and modern programming languages such as JavaScript, which has grown to be among the most popular languages. Comprehending such applications is challenging due to the temporal and implicit relations of asynchronous, DOM-related and event-driven entities spread over the client and server sides.

The goal of the work presented in this dissertation is to facilitate program comprehension through the following techniques. First, 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 execution through different semantic levels of granularity. Then, we present a DOM-sensitive hybrid change impact analysis technique for JavaScript through a combination of static and dynamic analysis. Our approach incorporates a novel ranking algorithm for indicating the importance of each entity in the impact set. Next, we introduce a method 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 comprehension for developers. Finally, we propose an approach for facilitating comprehension by creating an abstract model of software behaviour. The model encompasses hierarchies of recurring and application-specific motifs. The motifs are abstract patterns extracted from traces through our novel

technique, inspired by bioinformatics algorithms. The motifs provide an overview of the behaviour at a high level, while encapsulating semantically related sequences in execution. We design a visualization that allows developers to observe and interact with inferred motifs.

We implement our techniques in open-source tools and evaluate them through a set of controlled experiments. The results show that our techniques significantly improve developers' performance in comprehending the behaviour and impact of change in software systems.

# Lay Summary

Program comprehension is crucial in software engineering; a necessary step for performing many tasks. Assisting comprehension through analysis of code and program execution traces has been a popular research area. However, the implicit and intricate relations between program entities hinder comprehension of program behaviour and change. It is particularly challenging to understand modern programming languages such as JavaScript, which has grown to be among the most popular languages for both client and server development.

The goal of the work presented in this dissertation is to facilitate program comprehension through semi-automated techniques, using both static and dynamic analysis. Our techniques create behavioural models of program execution through our proposed algorithms, and visualize them for developers in order to improve their performance. We evaluate our techniques through a set of controlled experiments. The results show that our methods significantly improve developers' performance in terms of task completion duration and accuracy.

# Preface

Each research chapter of this dissertation relates to one or more papers, which have been published or are currently under review. I have collaborated with my supervisors, Dr. Ali Mesbah and Dr. Karthik Pattabiraman, for conducting the research projects in all chapters. I was the main contributor for all chapters, including the idea, design, development, and evaluation of the work. I had the collaboration of Sheldon Sequeira, a former Masters' student in our lab, for Chapter 2.

The publications for each chapter are as follows.

- Chapter 2:
  - Understanding JavaScript Event-based Interactions [4]: S. Alimadadi, S. Sequeira, A. Mesbah and K. Pattabiraman, ACM/IEEE International Conference on Software Engineering (ICSE), 2014, 11 pages. (Acceptance Rate: 20%)  
ACM SIGSOFT Distinguished Paper Award
  - Understanding JavaScript Event-based Interactions with CLEMATIS [8]: S. Alimadadi, S. Sequeira, A. Mesbah and K. Pattabiraman, ACM Transactions on Software Engineering and Methodology (TOSEM), 2016, 39 pages.
- Chapter 3:
  - Hybrid DOM-Sensitive Change Impact Analysis for JavaScript [6]: S. Alimadadi, A. Mesbah and K. Pattabiraman, European Conference on Object-Oriented Programming (ECOOP), 2015, 25 pages. (Acceptance Rate: 22.8%)

- Chapter 4:
  - Understanding Asynchronous Interactions in JavaScript Applications [7]: S. Alimadadi, A. Mesbah and K. Pattabiraman, ACM/IEEE International Conference on Software Engineering (ICSE), 2016, 12 pages. (Acceptance Rate: 19%)
- Chapter 5:
  - Inferring Hierarchical Motifs from Execution Traces: S. Alimadadi, A. Mesbah and K. Pattabiraman, 12 pages (submitted).

# Table of Contents

<b>Abstract</b> . . . . .	<b>ii</b>
<b>Lay Summary</b> . . . . .	<b>iv</b>
<b>Preface</b> . . . . .	<b>v</b>
<b>Table of Contents</b> . . . . .	<b>vii</b>
<b>List of Tables</b> . . . . .	<b>xi</b>
<b>List of Figures</b> . . . . .	<b>xii</b>
<b>Acknowledgments</b> . . . . .	<b>xvi</b>
<b>Dedication</b> . . . . .	<b>xvii</b>
<b>1 Introduction</b> . . . . .	<b>1</b>
1.1 Challenges and Motivation . . . . .	2
1.1.1 Understanding Dynamic JavaScript Behaviour . . . . .	2
1.1.2 Understanding Impact of Change in JavaScript . . . . .	5
1.1.3 Understanding Hierarchical Motifs of Behaviour in Execution	6
1.2 Origin of Chapters and Acknowledgements . . . . .	7
<b>2 Understanding JavaScript Event-Based Interactions</b> . . . . .	<b>9</b>
2.1 Challenges and Motivation . . . . .	11
2.1.1 Challenge 1: Event Propagation . . . . .	12

2.1.2	Challenge 2: Asynchronous Events . . . . .	13
2.1.3	Challenge 3: Implications of Events . . . . .	15
2.1.4	Challenge 4: Linking Test Failures to Faults . . . . .	15
2.2	Approach . . . . .	18
2.2.1	JavaScript Transformation and Tracing . . . . .	19
2.2.2	Capturing a Behavioural Model . . . . .	23
2.2.3	Understanding Test Assertion Failures . . . . .	25
2.2.4	Visualizing the Captured Model . . . . .	31
2.2.5	Tool Implementation: Clematis . . . . .	36
2.3	Controlled Experiments . . . . .	37
2.3.1	Experimental Design . . . . .	37
2.3.2	Experimental Procedure . . . . .	39
2.4	Experiment 1: Lab Environment . . . . .	39
2.4.1	Approach . . . . .	39
2.4.2	Results . . . . .	40
2.5	Experiment 2: Industrial . . . . .	42
2.5.1	Approach . . . . .	42
2.5.2	Results . . . . .	44
2.5.3	Qualitative Analysis of Participant Feedback . . . . .	45
2.6	Experiment 3: Test Failure Comprehension . . . . .	49
2.6.1	Approach . . . . .	49
2.6.2	Results . . . . .	50
2.7	Performance Overhead . . . . .	52
2.8	Discussion . . . . .	53
2.8.1	Task Completion Duration . . . . .	53
2.8.2	Task Completion Accuracy . . . . .	55
2.8.3	Consistent Performance . . . . .	57
2.8.4	Threats to Validity . . . . .	57
2.8.5	Limitations . . . . .	58
2.9	Concluding Remarks . . . . .	59
<b>3</b>	<b>Hybrid DOM-Sensitive Change Impact Analysis for JavaScript . . .</b>	<b>60</b>
3.1	Impact Transfer in JavaScript . . . . .	62



3.1.1	Impact through the DOM . . . . .	64
3.1.2	Impact through Event Propagation . . . . .	65
3.1.3	Impact through Asynchronous Callbacks . . . . .	66
3.1.4	JavaScript Dynamism . . . . .	67
3.1.5	Impact Paths . . . . .	68
3.2	Exploratory Study: DOM-related and Event-based Impacts . . . .	68
3.3	Hybrid Analysis . . . . .	70
3.3.1	Static Control-Flow and Partial Data-Flow Analysis . . . .	70
3.3.2	Analyzing the Dynamic Features . . . . .	71
3.3.3	Hybrid Model for Impact Analysis . . . . .	74
3.4	Impact Metrics and Impact Set Ranking . . . . .	77
3.5	Tool Implementation: TOCHAL . . . . .	79
3.6	Evaluation . . . . .	80
3.6.1	Study 1: Comparing Static, Dynamic, and Hybrid Analyses	80
3.6.2	Study 2: Industrial Controlled Experiment . . . . .	84
3.7	Concluding Remarks . . . . .	91
<b>4</b>	<b>Understanding Asynchronous Interactions in Full-Stack JavaScript</b>	<b>92</b>
4.1	Challenges and Motivation . . . . .	94
4.1.1	Challenge 1: Server-Side Callbacks . . . . .	94
4.1.2	Challenge 2: Asynchronous Client Side . . . . .	95
4.1.3	Challenge 3: Network Communication . . . . .	97
4.2	Approach . . . . .	97
4.2.1	Temporal and Context-Sensitive Model . . . . .	97
4.2.2	Client-Side Analysis . . . . .	100
4.2.3	Server-Side Analysis . . . . .	104
4.2.4	Connecting Client and Server . . . . .	106
4.2.5	Visualizing the Model . . . . .	106
4.2.6	Implementation . . . . .	108
4.3	Evaluation . . . . .	109
4.3.1	Experimental Setup . . . . .	109
4.3.2	Results . . . . .	111
4.3.3	Discussion . . . . .	112

4.4	Conclusion . . . . .	115
<b>5</b>	<b>Inferring Hierarchical Motifs from Execution Traces . . . . .</b>	<b>117</b>
5.1	Introduction . . . . .	117
5.2	Challenges and Motivation . . . . .	119
5.3	Overview of the Methodology . . . . .	122
5.4	Algorithm for Inferring Motifs . . . . .	123
5.4.1	Inspiration from Analyzing Biological Sequences . . . . .	124
5.4.2	Forming a Knowledge Base . . . . .	125
5.4.3	Finding Exact Matches . . . . .	125
5.4.4	Allowing Abstraction in Motifs . . . . .	126
5.4.5	Inferring Hierarchies of Motifs . . . . .	129
5.5	Creating and Visualizing the Model . . . . .	130
5.5.1	Creating the Motif Model . . . . .	131
5.5.2	Visualizing the Model . . . . .	133
5.6	Implementation: SABALAN . . . . .	134
5.7	Evaluation . . . . .	134
5.7.1	Motif Characteristics . . . . .	134
5.7.2	Controlled Experiment . . . . .	136
5.7.3	Discussion . . . . .	140
5.7.4	Threats to Validity . . . . .	143
5.8	Concluding Remarks . . . . .	144
<b>6</b>	<b>Related Work . . . . .</b>	<b>145</b>
<b>7</b>	<b>Concluding Remarks . . . . .</b>	<b>152</b>
7.1	Contributions . . . . .	152
7.2	Research Questions Revisited . . . . .	154
7.3	Reflections and Future Directions . . . . .	157
	<b>Bibliography . . . . .</b>	<b>160</b>

# List of Tables

Table 2.1	Adopted and adapted comprehension activities. . . . .	38
Table 2.2	Comprehension tasks used in study 1. . . . .	40
Table 2.3	Comprehension tasks used in study 2. . . . .	42
Table 2.4	Injected faults for the controlled experiment. . . . .	49
Table 3.1	(A) Results of analyzing JavaScript’s DOM-related and dynamic features. (B) Factors in determining impact metrics. . . . .	70
Table 3.2	Impact transfer through different entities. . . . .	74
Table 3.3	Impact Metrics . . . . .	77
Table 3.4	Results of comparison between static, dynamic and TOCHAL (RQ1) (A) Comparison of impact sets (B) Comparison of functions in system dependency graphs . . . . .	82
Table 3.5	Impact analysis tasks used in the controlled experiment. . . . .	86
Table 4.1	Types of vertices in the model graph . . . . .	99
Table 4.2	Interaction Edges . . . . .	100
Table 4.3	Creation and extension of the behavioural graph based on the operations . . . . .	101
Table 4.4	Comprehension tasks of the experiment . . . . .	110
Table 5.1	Characteristics of traces and inferred motifs . . . . .	136
Table 5.2	Comprehension tasks used in the study. . . . .	138

# List of Figures

Figure 2.1	Initial DOM state of the running example. . . . .	12
Figure 2.2	JavaScript code of the running example. . . . .	13
Figure 2.3	Test assertion understanding example (a) JavaScript code, (b) Portion of DOM-based UI, (c) Partial DOM, (d) DOM-based (Selenium) test case, (e) Test case assertion failure. The dotted lines show the links between the different entities that must be inferred. . . . .	16
Figure 2.4	A processing overview of our approach. . . . .	18
Figure 2.5	Instrumented JavaScript function declaration. . . . .	20
Figure 2.6	Instrumented JavaScript return statement. . . . .	21
Figure 2.7	Instrumented JavaScript function calls. . . . .	21
Figure 2.8	Comparison of instrumentation techniques for JavaScript function calls. . . . .	22
Figure 2.9	Relating assertions to DOM accesses for the test case of Figure 2.3d. . . . .	26
Figure 2.10	Example JavaScript code after our selective instrumentation is applied. Slicing criteria: <code>&lt;10, size&gt;</code> . . . . .	30
Figure 2.11	Overview of all captured stories. . . . .	31
Figure 2.12	Top: menu of CLEMATIS. Bottom: overview of a captured story. . . . .	32
Figure 2.13	Three semantic zoom levels in CLEMATIS. Top: overview. Middle: zoomed one level into an episode, while preserving the context of the story. Bottom: drilled down into the selected episode. . . . .	33

Figure 2.14	Visualization for a test case. (a) Overview of the passing test case, (b) Three semantic zoom levels for the failing test case; Top: overview. Middle: second zoom level showing assertion details, while preserving the context. Bottom: summary of failing assertion and the backwards slice. . . . .	35
Figure 2.15	t-test analysis with unequal variances of task completion duration by tool type. Lower values are better. [Study 1] . . . . .	41
Figure 2.16	Box plots of task completion duration data per task for each tool. Lower values are better. [Study 1] . . . . .	41
Figure 2.17	t-test analysis with unequal variances of task completion accuracy by tool type. Higher values are better. [Study 1] . . . . .	41
Figure 2.18	Box plots of task completion accuracy data per task for each tool. Higher values are better. [Study 1] . . . . .	41
Figure 2.19	Notched box plots of task completion duration data per task and in total for the control (gold) and experimental (green) groups (lower values are desired). [Study 2] . . . . .	43
Figure 2.20	Notched box plots of task completion accuracy data per task and in total for the control (gold) and experimental (green) groups (higher values are desired). [Study 2] . . . . .	43
Figure 2.21	Box plots of task completion accuracy data per task and in total for the control (blue) and experimental (cream) groups (higher values are desired). [Study 3] . . . . .	51
Figure 2.22	Box plots of task completion duration data per task and in total for the control (blue) and experimental (cream) groups (lower values are desired). [Study 3] . . . . .	51
Figure 3.1	Motivating example: JavaScript code . . . . .	62
Figure 3.2	Motivating example: HTML/DOM . . . . .	63
Figure 3.3	Impact transfer through DOM elements. . . . .	64
Figure 3.4	Impact transfer through event propagation. . . . .	65
Figure 3.5	Impact transfer through asynchronous callbacks. . . . .	67
Figure 3.6	A static call graph, displaying the dependencies extracted from the running example (Figures 3.1 and 3.2). . . . .	75

Figure 3.7	A sample hybrid graph, including the dynamic and DOM-sensitive dependencies extracted from the running example (Figures 3.1 and 3.2). . . . .	75
Figure 3.8	Task completion accuracy per task and in total for the control (ctrl) and experimental (exp) groups (RQ4.2.1). . . . .	88
Figure 3.9	Task completion duration data per task and in total for the control (ctrl) and experimental (exp) groups (RQ4.2.2). . . . .	88
Figure 4.1	Receiving HTTP requests at an end point . . . . .	95
Figure 4.2	Callback hell . . . . .	96
Figure 4.3	Asynchronous client-side JavaScript . . . . .	96
Figure 4.4	A sample temporal, context-sensitive and asynchronous model of events, lifelines, and interactions. . . . .	99
Figure 4.5	A snapshot of the visualization. . . . .	107
Figure 4.6	Accuracy results. Gold plots display experimental (SAHAND) group, and green plots display the control group. Higher values are better. . . . .	113
Figure 5.1	I: A sample registration form. II: A) Sample execution trace, and B) hierarchy of inferred motifs. III: Dynamic call graph of example . . . . .	120
Figure 5.2	Initial DOM state of the running example. . . . .	121
Figure 5.3	[Partial] JavaScript code of the running example. . . . .	121
Figure 5.4	This figure depicts a DB of traces (A) and a sample query trace (B) of an application, on the left and right side, respectively. Exact matches of length 2 and 3 between the query trace and different DB traces are marked. . . . .	125
Figure 5.5	This figure briefly depicts (A) the process of taking two expanded trace subsets, (B, C) forming a scoring matrix based on similarities between sub-traces, and (D, E) finding a match in manner that maximizes the similarities between sub-traces. The final motif can be seen in (F). . . . .	128

Figure 5.6	Sample model of the running example. The root node (1) is the highest-level inferred motif. Node 2 is a sub-motif of node (1), marked by the hierarchical edge between the two. Node 3 is abstract allowing variations of its child node to occur in the motif. The leaves of (nodes 4–8) are concrete function executions in the trace. . . . .	131
Figure 5.7	A [modified] screenshot of visualization. (A): Query trace. (B): Inferred motifs depicted on the table. (C): Motif hierarchies. (D): All motifs. (E): Code panel displaying selected function/-motif code. . . . .	133
Figure 5.8	Notched box plots of accuracy results. Green plots display experimental (SABALAN) group, and gold plots display the control group. Higher values are better. . . . .	139

# Acknowledgments

Foremost, I wish to express my utmost gratitude to my senior supervisor, Dr. Ali Mesbah. I am indebted to him for his expertise and insight. For continuously supporting me in my research, while steering me in the right direction when I needed. I owe my deepest gratitude to my co-supervisor, Dr. Karthik Pattabiraman, for his inspiration and motivation. For perfectly balancing the roles of a critical reviewer and a kind advisor at the same time. I am thankful to my supervisors for allowing me to discover my path, while always being there for guidance.

I am most grateful to Dr. Ivan Beschastnikh and Dr. Alexandra Fedorova, for their insightful comments and hard questions. My sincere gratitude goes to my thesis committee, Dr. Gail C. Murphy, Dr. Sidney Fels, and Dr. William G.J. Halfond. I would also like to acknowledge all my dear friends and colleagues in Software Analysis and Testing (SALT) Lab at UBC.

I wish to thank my beloved brothers, Siavash and Soroush. I am sincerely grateful to you for your constant love and support. I cannot find words to express my gratitude to my parents, Parviz and Parvin. I am eternally grateful to you for your unconditional and endless love, guidance and support.



To my parents.

# Chapter 1

## Introduction

**Program comprehension** is known to be an essential step in software engineering. Developers spend a considerable amount of time understanding code. About 50% of maintenance effort is spent on comprehension alone [31]. To understand code, developers typically start by *searching* for clues in the code and the environment. Then they navigate the incoming and outgoing dependencies to *relate* pieces of foraged information. Throughout the process, they *collect* information they find relevant for understanding the code on an “as-needed” basis [71]. However, developers often fail in searching and relating information, and lose track of relevant information when using such ad-hoc strategies [117]. Further, they form a mental model of the entities, relations and the intent of the code, which they use throughout development to help them make decisions. Unfortunately, these models are almost always inaccurate [92]. Thus, code understanding is challenging, and there is a need for systematic and automated techniques that facilitate this process [75]. Further, presence of programming languages characteristics such as dynamism, asynchrony and non-determinism in the execution makes the analysis more problematic and burdensome, and renders conventional techniques ineffective. It is particularly a challenging endeavour to understand the complex behaviour of modern programming languages, such as JavaScript.

**JavaScript** is widely used today to create interactive modern web applications that replace many traditional desktop applications. It has been selected as the most popular programming language for five consecutive years [126] and it is the most

used language on GitHub [74]. However, understanding the behaviour of web applications is troublesome for developers [101, 143].

## 1.1 Challenges and Motivation

The goal of this thesis is to investigate support for developers to improve their performance in program comprehension tasks. However, the implicit and intricate relations between program entities hinder comprehension of program behaviour. Comprehending client- and server-side JavaScript applications is particularly challenging due to the temporal and implicit relations of asynchronous, dynamic, and event-driven entities split between the client and server. These relation also affect the way a change propagates in a JavaScript application and impacts the system as a whole. Moreover, size and complexity of execution traces further complicate comprehension of real-world applications.

### 1.1.1 Understanding Dynamic JavaScript Behaviour

Despite their popularity, understanding the behaviour of modern web applications is still a rigorous task for developers during development and maintenance tasks. The challenges mainly stem from the dynamic, event-driven, and asynchronous nature of the JavaScript language.

#### Understanding Event-Based Interactions

To understand JavaScript, developers must understand its weakly-typed, event-driven, and highly-dynamic nature, its interactions with the Document Object Model (DOM), and communications with the server. Unfortunately, despite its importance and challenges, there is currently not much research dedicated to supporting program comprehension for web applications [34]. Popular tools, such as Firebug and Chrome DevTools, are limited in their capabilities to support web developers effectively.

**Research Question 1.** *How can we enhance developers' performance in understanding the event-based interactions in client-side JavaScript?*

To address RQ1, 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 state of the DOM. Our approach allows developers to easily understand the complex dynamic behaviour of their application at three different semantic levels of granularity. Furthermore, it helps developers bridge the gap between test cases and program code by localizing the fault related to a test assertion. This method is discussed in Chapter 2.

### **Understanding Test Failures and Their Root Causes**

To test their web applications, developers often write test cases that check the application’s behaviour from an end-user’s perspective using popular frameworks such as Selenium [121]. Such test cases are agnostic of the JavaScript code and operate by simulating a series of user actions followed by assertions on the application’s runtime DOM. As such, they can detect deviations in the expected behaviour as observed in the DOM.

However, when a web application test assertion fails, determining the faulty program code responsible for the failure can be challenging. Fault localization is one of the most difficult phases of debugging [132], and has been an active topic of research [1, 28, 67, 144]. Although testing of modern web applications has received increasing attention in the recent past [15, 88, 129], there has been limited work on what happens after a test reveals an error.

**Research Question 2.** *How can we enhance developers’ performance in understanding the root-causes of test assertion failures in client-side JavaScript?*

We extend our approach for understanding program behaviour in Chapter 2 to further aid developers in understanding root causes of test failures. Our test-case comprehension strategy automatically connects test assertion failures to faulty JavaScript code considering the involved DOM elements.

### **Asynchronous Interactions in Full-Stack JavaScript**

JavaScript has been the lingua franca of client-side web development for some years. But platforms such as Node.js [97] 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 such applications is challenging for developers, due to the temporal and implicit relations of asynchronous and event-driven entities split across the client and server side. JavaScript applications take extensive advantage of asynchronous callbacks [49] to simulate concurrency, which complicates the flow of execution. The impact of asynchrony intensifies due to the communication between the client and server. They interact heavily with the DOM, events, timers, and XHR objects, across both client and server, all of which negatively affect developers’ understanding. The uncertainty involved in the asynchronous communication makes the execution more intricate and thus more difficult to understand for developers.

Despite the popularity of full-stack JavaScript development 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, 10, 61, 84, 101, 143].

**Research Question 3.** *How can we improve developers’ understanding of the temporal and asynchronous behaviour of full-stack JavaScript?*

In Chapter 4, 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. Our goal is to help developers gain

a holistic view of the dynamic behaviour of full-stack JavaScript applications.

### 1.1.2 Understanding Impact of Change in JavaScript

To remain useful, a software program must continually change to adapt to the changing environment [77]. Code change impact analysis (CIA) [13] aims at identifying parts of the program that are potentially affected by a change in the code. Impact analysis has been a popular research area [12, 20, 105, 111, 114]. Most of the research, however, is focused on traditional programming languages and ignores JavaScript and thereby modern web applications.

JavaScript requires a novel approach for effective change impact analysis, because of its unique set of features that make it challenging to comprehend [4] and analyze by traditional code analysis techniques [47]. Each of the parties involved in the execution of a JavaScript application can introduce new and implicit relations in the system that can transfer the impact of change seamlessly. For instance, we have observed that the impact of a code change can be propagated through the DOM, even when there may be no visible connections between JavaScript functions and variables in the JavaScript code [6]. Similarly, events and server interactions can transfer the impact of a change, in addition to the JavaScript code itself.

Traditional impact analysis has been performed either by static analysis or dynamic analysis. However, the aforementioned challenges make it difficult for JavaScript static analysis techniques [47, 66, 125] to carry out impact analysis effectively. None of these techniques can provide support for the DOM-based, dynamic and asynchronous JavaScript features. Further, current dynamic and hybrid analysis techniques [136, 137] ignore the aforementioned challenges, i.e., they overlook the important role of the DOM in their analysis and they do not support the hidden relations that are created through event-handler registration, event propagation, and asynchronous client/server communication.

**Research Question 4.** *How does providing a model of the dependencies in the application improve developers' performance in understanding the change impact in JavaScript applications?*

We first perform an empirical study of change propagation, the results of which show that the DOM-related and dynamic features of JavaScript need to be taken into consideration in the analysis since they affect change impact propagation. We propose a DOM-sensitive hybrid change impact analysis technique for JavaScript through a combination of static and dynamic analysis. The proposed approach incorporates a novel ranking algorithm for indicating the importance of each entity in the impact set. Our approach is explained in Chapter 3.

### 1.1.3 Understanding Hierarchical Motifs of Behaviour in Execution

Assisting comprehension through dynamic analysis of execution traces is a popular research area. Traces are rich sources of information regarding the behaviour of a program, leading to precise analysis techniques. However, these techniques typically do not scale well, due to the size and complexity of execution traces.

Common techniques such as trimming, summarizing, and visualizing traces [75, 92] do not solve the problem for large, complex, and real-world applications and understanding higher-level key points of the semantics of program behaviour remains difficult [33, 141, 142]. The interpretation of prior work from execution patterns, if existent, is different from ours. These papers have predominantly focused on generic and pre-defined design patterns, low-level architectural relations between program artifacts, or visualizations of all details of execution [7, 21, 60, 73].

**Research Question 5.** *How does providing high-level and semantic motifs of a application’s behaviour improve comprehensibility of the application?*

To address this research question, we propose a generic technique in Chapter 5 for facilitating comprehension by creating an abstract model of software behaviour. The model encompasses hierarchies of recurring and application-specific motifs. The motifs are abstract patterns extracted from traces through our novel technique, inspired by bioinformatics algorithms. The motifs provide an overview of the behaviour at a high-level, while encapsulating semantically related sequences in execution. We design a visualization that allows developers to observe and interact with inferred motifs.

## 1.2 Origin of Chapters and Acknowledgements

Chapters 2 – 4 are based on published peer-reviewed papers listed below. Chapter 5 is based on a paper that is currently under submission in a software engineering conference. The author of this thesis is the main contributor of all chapters. Chapter 2 is co-authored by Sheldon Sequeira, a former Masters’ student in SALT lab. All chapters are co-authored by the author’s supervisors: Dr. Ali Mesbah and Dr. Karthik Pattabiraman.

- **Chapter 2.** Addressing RQ1, this chapter was originally published in the ACM/IEEE International Conference on Software Engineering (ICSE), in 2014. This paper received an ACM SIGSOFT Distinguished Paper Award. The extended version of the paper, addressing RQ2, was later published in ACM Transactions on Software Engineering and Methodology (TOSEM), in 2016.
  - Understanding JavaScript Event-based Interactions [4]: S. Alimadadi, S. Sequeira, A. Mesbah and K. Pattabiraman, ACM/IEEE International Conference on Software Engineering (ICSE), 2014, 11 pages. (Acceptance Rate: 20%)
  - Understanding JavaScript Event-based Interactions with CLEMATIS [8]: S. Alimadadi, S. Sequeira, A. Mesbah and K. Pattabiraman, ACM Transactions on Software Engineering and Methodology (TOSEM), 2016, 39 pages.
- **Chapter 3.** This chapter, targeting RQ3, was published in the European Conference on Object-Oriented Programming (ECOOP) in 2015.
  - Hybrid DOM-Sensitive Change Impact Analysis for JavaScript [6]: S. Alimadadi, A. Mesbah and K. Pattabiraman, European Conference on Object-Oriented Programming (ECOOP), 2015, 25 pages. (Acceptance Rate: 22.8%)
- **Chapter 4.** This chapter addresses RQ4 and was published at the ACM/IEEE International Conference on Software Engineering (ICSE) in 2016.
  - Understanding Asynchronous Interactions in JavaScript Applications



[7]: S. Alimadadi, A. Mesbah and K. Pattabiraman, ACM/IEEE International Conference on Software Engineering (ICSE), 2016, 12 pages. (Acceptance Rate: 19%)

- **Chapter 5.** This chapter, addressing RQ5, is currently under submission at a software engineering conference. The original idea of this paper was accepted to the Doctoral Symposium track at the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE) in 2016.
  - Inferring Hierarchical Motifs from Execution Traces: S. Alimadadi, A. Mesbah and K. Pattabiraman, 12 pages, under review.
  - Understanding Behavioural Patterns in JavaScript [3]: S. Alimadadi, ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016, 3 pages.

## Chapter 2

# Understanding JavaScript Event-Based Interactions

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 challenging 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. Furthermore, CLEMATIS helps developers bridge the gap between test cases and program code by localizing the fault related to a test assertion. The results of our industrial controlled experiment show that CLEMATIS is capable of improving the comprehension task accuracy by 157%, while reducing the task completion time by 47%. A follow up experiment reveals that CLEMATIS improves the fault localization accuracy of developers by a factor of two.

Understanding the implicit and intricate relations between program entities in JavaScript applications is challenging for many reasons. 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 [98]. 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 [133].

In this chapter, 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.

We then apply our approach to further aid developers in understanding root causes of test failures. Fault localization is one of the most difficult phases of debugging [132], and is an active topic of research [1, 28, 67, 144]. Although testing of modern web applications has received attention [15, 88, 129], there has been limited work on what happens after a test reveals an error.

To the best of our knowledge, we are the first to provide a generic technique for capturing low-level event-based interactions in a JavaScript application, and mapping and visualizing those interactions as higher-level behavioural models. We extend the approach, as we propose a novel test failures comprehension unit and evaluate its effectiveness through a user experiment. Overall, our work makes the following key contributions:

- We propose a generic technique for capturing and presenting the complex dynamic 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.
  - Extracts the source-and-target relations for asynchronous events, i.e., timing events and XMLHttpRequest requests/callbacks.
  - Identifies and tracks mutations to the DOM caused by each event.
- We build a novel model for capturing the event-driven interactions as well as an interactive, visual interface supporting the comprehension of the program through three different semantic levels of zooming granularity.
  - We implement our technique in a generic open source tool called CLEMATIS, which (1) does not modify the web browser, (2) is independent of the server technology, and (3) requires no extra effort from the developer to use.
  - We extend CLEMATIS to automatically connect test assertion failures to faulty JavaScript code considering the involved DOM elements.
  - We empirically evaluate CLEMATIS through three controlled experiments comprising 48 users in total. The first two studies investigate the code comprehension capabilities of CLEMATIS. One of these studies is carried out in a lab environment, while the other is carried out in an industrial setting. The results of the industrial experiment show that CLEMATIS can reduce the task completion time by 47%, while improving the accuracy by 157%. We evaluate the test failure comprehension unit of CLEMATIS through a third user experiment. The results show that CLEMATIS improves the fault localization accuracy of developers by a factor of two.

## 2.1 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 2.1–2.2, to illustrate these challenges.

Furthermore, developers often write test cases that assert the behaviour of a web

```
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 2.1: Initial DOM state of the running example.**

application from an end-user’s perspective. However, when such test cases fail, it is difficult to relate the assertion failure to the faulty line of code. The challenges mainly stem from the existing disconnect between front-end test cases that assert the DOM and the application’s underlying JavaScript code. We use another example, illustrated in Figure 2.3, to demonstrate these testing challenges.

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

### 2.1.1 Challenge 1: Event Propagation

The DOM event model [133] makes it possible for a single event, fired on a particular DOM node, to propagate through the DOM tree hierarchy and indirectly trigger a series of other event-handlers attached to other nodes. There are typically two types of this event propagation in web applications; (1) with *bubbling* enabled, an event first triggers the handler of the deepest child element on which the event was fired, and then it *bubbles* up on the DOM tree and triggers the parents’ handlers. (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 2.1–2.2. Figure 2.1 represents the initial DOM structure of the application. It mainly consists of a `fieldset` containing a set of elements for the users to enter their email address to be registered for a service. The JavaScript code in Figure 2.2 partly handles this submission.

```

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

```

**Figure 2.2: JavaScript code of the running example.**

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 2.2. 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 2.2). In a more complex application, the developer may be unaware of the existence of the parent element, its registered handlers, or the complex event propagation mechanisms such as bubbling and capturing.

### 2.1.2 Challenge 2: Asynchronous Events

Web browsers provide a single thread for web application execution. To circumvent this limitation and build rich responsive web applications, developers take advantage

of the asynchronous capabilities offered by modern browsers, such as *timeouts* and *XMLHttpRequest* (XHR) calls. Asynchronous programming, however, introduces an extra layer of complexity in the control flow of the application and adversely influences program comprehension.

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

**XHR Callbacks.** XHR objects are used to exchange data asynchronously with the server, without requiring a page reload. Each XHR goes through three main phases: *open*, *send*, and *response*. These three phases can be scattered throughout the code. Further, there is no guarantee on the timing and the order of XHR responses from the server. As in the case of timeouts, mapping the functionality triggered by a server response back to its source request is a challenging comprehension task for developers.

*Example.* Following the running example, the developer may wish to further investigate the unexpected 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 2.2 by the anonymous handler defined in lines 3–5. However the callback function, defined on line 23 of Figure 2.2, 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 2.2) updates the DOM (line 18).

### 2.1.3 Challenge 3: Implications of Events

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

*Example.* After the `submitBtn` button is clicked in the running example, a confirmation message will appear on-screen and disappear shortly thereafter (lines 8&23, Figure 2.2). Additionally, the attributes of the button are altered to disable it (line 12). It can be difficult to follow such DOM-altering features in an application's code.

### 2.1.4 Challenge 4: Linking Test Failures to Faults

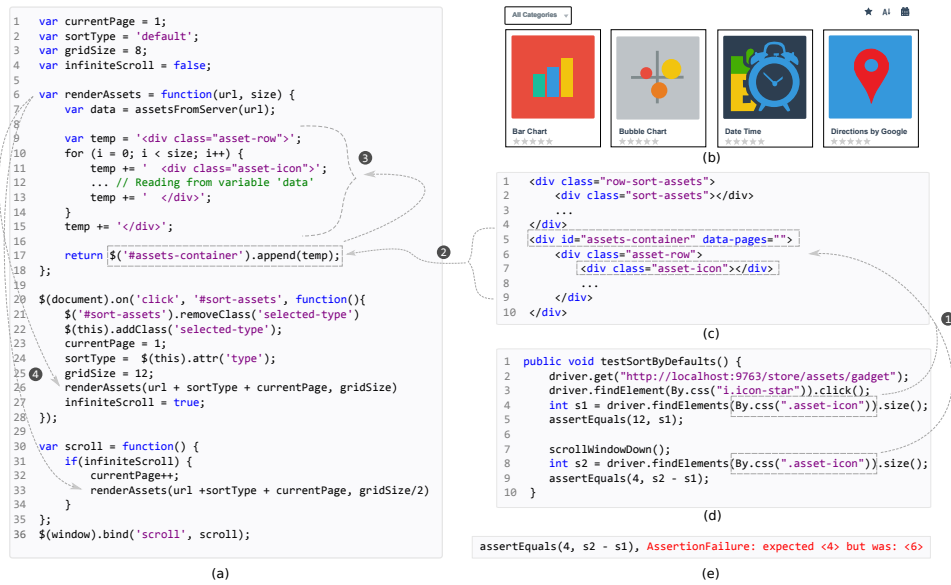
To test their web applications, developers often write test cases that check the application's behaviour from an end-user's perspective using popular frameworks such as Selenium<sup>1</sup>. Such test cases are agnostic of the JavaScript code and operate by simulating a series of user actions followed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

However, when a web application test assertion fails, determining the faulty program code responsible for the failure can be a challenging endeavour. The main challenge here is the implicit link between three different entities, namely, the test assertion, the DOM elements on which the assertion fails (checked elements), and the faulty JavaScript code responsible for modifying those DOM elements. To understand the root cause of the assertion failure, the developer needs to manually infer a mental model of these hidden links, which can be tedious. Further, unlike in traditional (e.g., Java) applications, there is no useful stack trace produced when

---

<sup>1</sup><http://seleniumhq.org>





**Figure 2.3: Test assertion understanding example (a) JavaScript code, (b) Portion of DOM-based UI, (c) Partial DOM, (d) DOM-based (Selenium) test case, (e) Test case assertion failure. The dotted lines show the links between the different entities that must be inferred.**

a web test case fails as the failure is on the DOM, and not on the application’s JavaScript code. This further hinders debugging as the fault usually lies within the application’s code, and not in its representative DOM state. To the best of our knowledge, there is currently no tool support available to help developers in this test failure understanding and fault localization process.

*Example.* The example in Figure 2.3 uses a small code snippet based on the open-source WSO2 eStore application <sup>2</sup> to demonstrate the challenges involved and our solution. The store allows clients to customize and deploy their own digital storefront. A partial DOM representation of the page is shown in Figure 2.3c. Figure 2.3d shows a Selenium test case, written by the developers of the application for verifying the application’s functionality in regards to “sorting” and “viewing” the existing assets. The JavaScript code responsible for implementing the functionality is shown in Figure 2.3a.

<sup>2</sup><https://github.com/wso2/product-es>

After setting the environment, the test case performs a click to sort the *assets*. Then, an assertion is made to check whether the expected *assets* are present on the DOM of the page (line 5 of Figure 2.3d). The second portion of the test case involves scrolling down the webpage and asserting the existence of four additional *assets* on the DOM (lines 7–9).

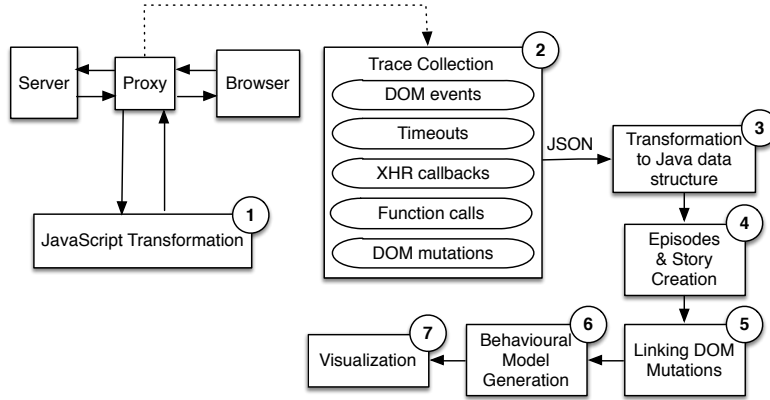
While the mapping between the test case and related JavaScript code may seem trivial to find for this small example, challenges arise as the JavaScript code-base and the DOM increase in size. As a result, it can be difficult to understand the reason for a test case failure or even which features are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

When a test case fails, first one needs to *identify the dependencies of the test case*. Based on the fail message in our example (Figure 2.3e), it is almost impossible to determine the cause of failure. Closer examination reveals the dependencies of assertions on variables `s1` and `s2`, which in turn depend on DOM elements with class `asset-icon` (link ❶ in Figure 2.3).

Next, the developer/tester is forced to *find the points of contact* between the DOM elements and the JavaScript code. Finding the JavaScript code responsible for modifying this subset of DOM elements is not easy. In the context of our example, a developer would eventually conclude that line 17 of Figure 2.3a is actually responsible for appending elements to the DOM. Discovering such implicit links (❷ and ❸ in Figure 2.3) needs tedious examination in smaller programs, and may not be feasible in larger applications.

In JavaScript, events can trigger code execution and must be taken into account for finding the source of the fault. The `renderAssets()` function in our example (Figure 2.3) can be called from within two event handlers (lines 26 and 33, respectively, shown as ❹). While in our example it may be straight-forward to link the call to `scrollWindowDown()` (line 7 of Figure 2.3d) to the execution of event handler `scroll` (line 30–35 of Figure 2.3a) due to the similarity in naming convention, such a linear mapping is neither possible in all cases nor easily inferable.

Finally, to fully understand an assertion and its possible cause of failure, the data and control dependencies for the DOM-altering statements must be determined and examined by the developer in order to identify all possible points of failure. In the case of `eStore`, the modification of the DOM within `renderAssets()`



**Figure 2.4: A processing overview of our approach.**

depends on the arguments passed into the function (lines 7 & 10). Dotted line 4 shows possible invocations of `renderAssets()`, revealing dependencies on global variables such as `gridSize`. Tracing the dependencies reveals that an update to `gridSize` on line 25 of Figure 2.3a is the root cause of the unusual behaviour.

## 2.2 Approach

In this section, we describe our approach for addressing the challenges mentioned in the previous section. An overview of the overall process is depicted in Figure 2.4, which consists of the following main steps:

- First, our technique captures a fine-grained trace of all semantically related event-based interactions within a web application’s execution, in a particular user session. The collection of this detailed trace is enabled through a series of automated JavaScript transformations (Section 2.2.1).
- 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 2.2.2).
- Then, the model is extended through a combination of selective code instrumentation and dynamic backward slicing to bridge the gap between test cases

and program code (Section 2.2.3).

- 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 2.2.4).

We describe each step further below.

### 2.2.1 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 Section 2.2.2).

**Interposing on DOM Events.** There are two ways event listeners can be bound to a DOM element in JavaScript. The first method is programmatically using the DOM Level 1 (`e.click=handler`) or DOM Level 2 (`e.addEventListener`) methods W3C [133] 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

```

1 function clearMsg() {
2   send(JSON.stringify({messageType: "FUNCTION_ENTER", fnName: "clearMsg",
3     "args": null, ...}));
4   $('#submissionMsg').fadeOut(2000);
5   send(JSON.stringify({messageType: "FUNCTION_EXIT", fnName: "clearMsg",
6     "args": null, ...}));
7 }

```

**Figure 2.5: Instrumented JavaScript function declaration.**

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.

As this communication is done each time a function is executed, argument values are recorded dynamically at the cost of a small overhead. Figure 2.5 contains the simple `clearMsg()` JavaScript function from the running example shown in Figure 2.2 (line 22), which has been instrumented to record both the beginning and end of its execution (lines 2 and 4).

*Return Statements:* Apart from reaching the end of a subroutine, control can be returned back to a calling function through a return statement. There are two reasons for instrumenting return statements: (1) to accurately track nested function calls, and (2) to provide users with the line numbers of the executed return statements. Without recording the execution of return statements, it would be difficult to accurately

```

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

```

**Figure 2.6: Instrumented JavaScript return statement.**

```

1 function submissionHandler(e) {
2   $('#regMsg')[FCW("html")]("Submitted!");
3   var email = $('#email')[FCW("value")]();
4   if (FCW(isEmailValid)(email)) {
5     FCW(informServer)(email);
6     $('#submitBtn')[FCW("attr")]("disabled", true);
7   }
8 }
9 function clearMsg() {
10  $('#regMsg')[FCW("fadeOut"])(2000);
11 }
12 function FCW(fnName) { // Function Call Wrapper
13   send(JSON.stringify({messageType: "FUNCTION_CALL", ...,<
14     targetFunction: fnName}));
15   return fnName;
16 }

```

**Figure 2.7: Instrumented JavaScript function calls.**

track nested function calls. Furthermore, by recording return values and the line number of each return statement, CLEMATIS is able to provide users with contextual information that can be useful during the debugging process.

Figure 2.6 illustrates the instrumentation for the return statement of `informServer()`, a function originally shown in the running example (Figure 2.2, lines 16-21). The wrapper function `RSW` receives the return value of the function and the line number of the return statement and is responsible for recording this information before execution of the application’s JavaScript is resumed.

*Function Calls:* In order to report the source of a function invocation, our approach also instruments function calls. When instrumenting function calls, it is important to preserve both the order and context of each dynamic call. To accurately capture the function call hierarchy, we modify function calls with an inline wrapper function. This allows us to elegantly deal with two challenging scenarios. First,

```

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

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

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

```

**Figure 2.8: Comparison of instrumentation techniques for JavaScript function calls.**

when multiple function calls are executed from within a single line of JavaScript code, it allows us to infer the order of these calls without the need for complex static analysis. Second, inline instrumentation enables us to capture nested function calls. Figure 2.7 depicts the instrumentation of function calls for two methods from Figure 2.1, `submissionHandler()` and `clearMsg()`.

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

For comparison, an alternative instrumentation technique is shown on lines 8 – 10 of figure 2.8. While such a technique might be sufficient for measuring function coverage, it does not capture the order of execution accurately for nested function calls or when multiple function calls are made from a single line. Doing so would require more complex static analysis.

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

and functions, enabling us to link DOM changes with the JavaScript code that is responsible for these mutations.

### 2.2.2 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.<sup>3</sup> 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

---

<sup>3</sup>Because JavaScript is single-threaded on all browsers, the events are totally ordered in time.



episode nodes. Two types of edges are present in the model:

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

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

Algorithm 1 takes the trace collected from a web application as input and outputs a story with episodes and the edges between them. First, the trace units are extracted and sorted based on the timestamp of their occurrence (line 3). Next, the algorithm iteratively forms new episodes and assigns trace units to the source, 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  $\tau$  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

---

**Algorithm 1** Story Creation

---

**input** : *trace***output** : *story***Procedure** CREATEMODEL() **begin**

```
1   $G < V, E > story \leftarrow \emptyset$ 
2   $e_{curr}, e_{prev} \leftarrow \emptyset$ 
3   $\Sigma tu \leftarrow \text{EXTRACTANDSORTTRACEUNITS}(trace)$ 
4  foreach  $tu \in \Sigma tu$  do
5      if  $e_{prev} \equiv \emptyset \parallel e_{prev}.ended() \&\&$ 
         $tu.type \equiv episodeSource$  then
6           $e_{curr} \leftarrow \text{CREATEEPISODE}()$ 
7           $e_{curr}.source \leftarrow \text{SETEPISODESOURCE}(tu)$ 
8           $V \leftarrow V \cup e_{curr}$ 
9      else if  $(tu.type \equiv FunctionTrace \parallel EventHandler) \parallel$ 
         $(tu.type \equiv XHRCallback \parallel TimeoutCallback$ 
         $\&\& \neg episodeEndCriteria)$  then
10          $e_{curr}.trace \leftarrow e_{curr}.trace \cup tu$ 
11     else if  $tu.type \equiv DOMMutation$  then
12          $e_{curr}.results \leftarrow e_{curr}.results \cup tu$ 
13     if  $episodeEndCriteriaSatisfied$  then
14          $E \leftarrow E \cup \text{CREATETEMPORALLINK}(e_{prev}, e_{curr})$ 
15          $e_{prev} \leftarrow e_{curr}$ 
16      $timeoutMap < TimeoutSet, TimeoutCallback > \leftarrow \text{MAPTIMEOUTTRACEUNITS}(\Sigma tu)$ 
17      $XHRMap < XHROpen, XHRSend, XHRCallback > \leftarrow \text{MAPXHRTRACEUNITS}(\Sigma tu)$ 
18      $E \leftarrow E \cup \text{EXTRACTCAUSALLINKS}(TIMEOUTMAP, XHRMAP)$ 
19      $story \leftarrow \text{BUILDSTORY}(G < V, E >)$ 
20     return  $story$ 
```

---

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).

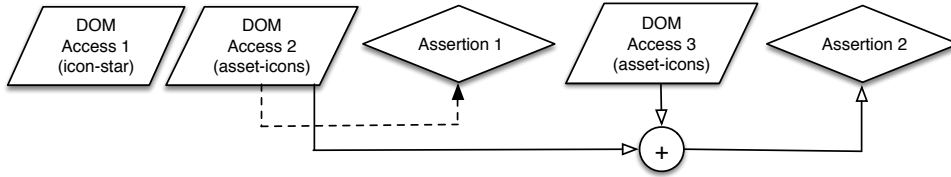
### 2.2.3 Understanding Test Assertion Failures

In this section, we extend CLEMATIS to further assist developers in the comprehension process. We add a test case comprehension strategy to CLEMATIS, to help developers understand the root cause of a test failure. Our technique automatically links a test assertion failure to the checked DOM elements, and subsequently to the

related statements in the JavaScript code. The following subsections describe our strategies for fulfilling the aforementioned requirements of JavaScript test failure comprehension.

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

Going back to the test case of our example in Figure 2.3d, the first assertion on Line 5 is dependent on the DOM elements returned by the access on the previous line. The last assertion on Line 9 is more complex as it compares two snapshots of the DOM and therefore has dependencies on 2 DOM accesses (Lines 4 and 8). Figure 2.9 summarizes the test case’s execution and captures the temporal and causal relations between each assertion and DOM access.



**Figure 2.9: Relating assertions to DOM accesses for the test case of Figure 2.3d.**

To accurately determine the DOM dependencies of each assertion (❶ in Figure 2.3), we apply dynamic backward slicing to each test case assertion. In addition, we track the runtime properties of those DOM elements accessed by the test case. This runtime information is later used in our analysis of the DOM dependencies of each assertion.

**Contextualizing Test Case Assertion.** In the second step, our approach aims to (1) help developers understand the context of their assertions by monitoring test-related JavaScript execution, asynchronous events, and DOM mutations; (2) determine the initial link between JavaScript code and the checked DOM elements (❷ in Figure 2.3).

In order to monitor JavaScript events, we leverage the tracing technique outlined in Section 2.2.1, which tracks the occurrence of JavaScript events, function invocations, and DOM mutations. We utilize the tracked mutations in order to focus on the segments of JavaScript execution most relevant to the assertions in a test case. As we are only interested in the subset of the DOM relevant to each test case, our approach focuses on the JavaScript code that interacts with this subset.

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

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

We compare the recorded JavaScript DOM accesses with the DOM dependencies of each test case assertion to find the *equivalent* JavaScript DOM accesses within the application’s code. Moreover, the *ancestors* of those elements accessed by each assertion are also compared with the recorded JavaScript DOM accesses. This is important because in many cases a direct link might not exist between them. For instance, in the case of our example (Figure 2.3d), a group of *assets* are compiled and appended to the DOM after a `scroll` event. We compare the properties of those DOM elements accessed by the final assertion (*assets* on Lines 4 and 8 of Figure 2.3d), as well as the properties of those elements’ ancestors, with the recorded JavaScript DOM accesses and conclude that the *assets* were added to the DOM via the *parent* element *assets container* on Line 17 of Figure 2.3a (❷).

**Slicing the JavaScript Code.** At this point, our approach yields the set of JavaScript statements responsible for updating the DOM dependencies of our test case.

However, the set in isolation seldom contains the cause of a test failure. We compute a backwards slice for these DOM-mutating statements to find the entire set of statements that perform the DOM mutation.

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

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

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

of the detected dependencies allows us to iteratively determine the subset of code statements to efficiently instrument for a given initial slicing criteria.

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

Figure 2.10 shows an example of our code instrumentation technique’s output when applied to the JavaScript code in Figure 2.3a with slicing criteria `<10, size>`. By acting as a control dependency for variable `temp`, `size` determines the number of displayed *assets* for the example. For each relevant write operation, our instrumentation code logs information such as the name of the variable being written to, the line number of the executed statement, and the type of value being assigned to the variable. Moreover, the data dependencies for such a write operation are also logged. Likewise, for each read operation we record the name of the variable being read, the type of value read, and the line number of the statement. Information about variable type is important when performing alias analysis during the computation of a slice.

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

The task of slicing is complicated by the presence of aliases in JavaScript. When computing the slice of a variable that has been assigned a non-primitive value, we need to consider possible aliases that may refer to the same object in memory. This also occurs in other languages such as C and Java, however, specific to JavaScript is the use of the *dot notation*, which can be used to seamlessly modify objects at runtime. The prevalent use of aliases and the dot notation in web applications often complicates the issue of code comprehension. Static analysis techniques often ignore addressing this issue [47].

To remedy this issue, we incorporate dynamic analysis in our slicing method. If

```

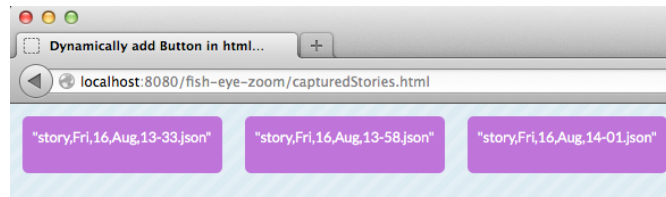
1  var currentPage = 1;
2  var sortType = 'default';
3  var gridSize = _write("gridSize", 8, 3);
4  var infiniteScroll = false;
5
6  var renderAssets = function(url, size) {
7      var data = assetsFromServer(url);
8
9      var temp = '<div class="asset-row">';
10     for (i = 0; i < _read("size", size,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

```

**Figure 2.10: Example JavaScript code after our selective instrumentation is applied. Slicing criteria: <10, size>**

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

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



**Figure 2.11: Overview of all captured stories.**

Section 2.2.4.

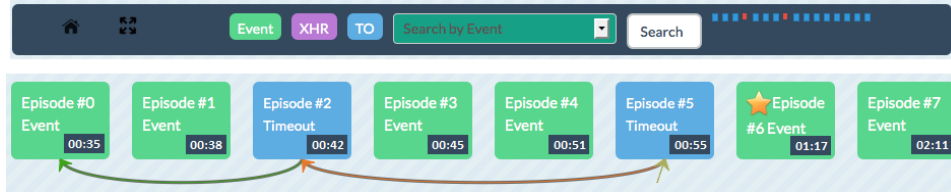
### 2.2.4 Visualizing the Captured Model

In the final step, our technique produces an interactive visualization of the generated model, which can be used by developers to understand the 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 [29] 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.

**Multiple Sessions, Multiple Stories.** The user can capture multiple sessions that leads to creation of multiple stories. After each story is recorded, it will be added to the list of captured stories. The name of each story is the date and time at which it was captured. Figure 2.11 shows a screenshot of sample captured stories in the visualization of Clematis. Once the users select their desired story, the browser opens a new page dedicated to that story. The initial view of a story contains a menu bar that helps the user navigate the visualization (Figure 2.12, top). It also displays an overview of all captured episodes inside the story (Figure 2.12, bottom).

**Story Map, Queries, and Bookmarking.** A menu bar is designed for the visualization that contains two main parts: the *story map* and the *query mechanism* (Figure 2.12, top). The story map represents a general overview of the whole story as a roadmap. Panning and (semantic) zooming are available for all episodes and





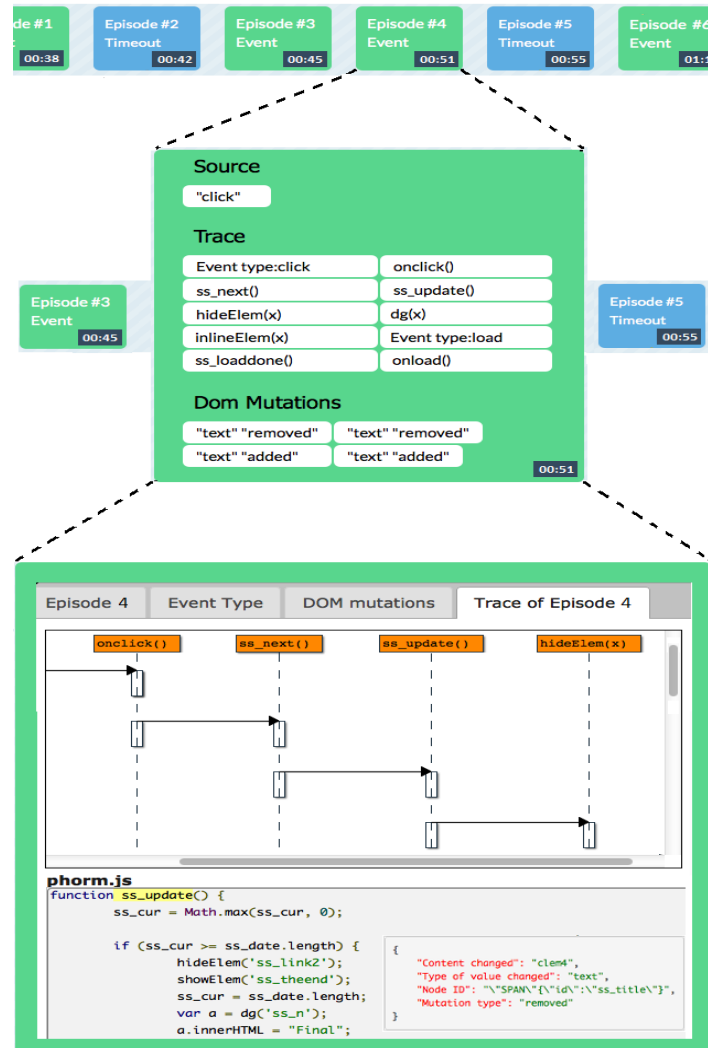
**Figure 2.12: Top: menu of CLEMATIS. Bottom: overview of a captured story.**

may cause users to lose the general overview of the story. Hence, based on the user’s interaction with the story (e.g., episode selection), the episodes of interest are highlighted on the roadmap to guide the user. The query section enables users to search and filter the information visualized on the screen. Users can filter the episodes displayed on the screen by the episode types (i.e., Event, Timeout, or XHR). They can also search the textual content of the events as well as the actual code. Moreover, they have the option to bookmark one or more episodes while interacting with the target web application. Those episodes are marked with a star in the visualization to help users to narrow the scope and spot related episodes (e.g., episode #6 in Figure 2.12 is bookmarked). The episodes’ timing information is also shown.

**Semantic Zoom Levels.** The visualization provides 3 semantic zoom levels.

*Zoom Level 0.* The first level displays all of the episodes in an abstracted manner, showing only the type and the timestamp of each episode (Figure 2.12, bottom). The type of each episode is displayed by the text of the episode as well as its background color. The horizontal axis is dedicated to time and episodes are sorted from left to right according to the time of their occurrence (temporal relations). The causal edges between different sections of each timeout or XHR object are shown by additional edges under the episodes.

*Zoom Level 1.* When an episode is selected, the view transitions into the second zoom level, which presents an outline of the selected episode, providing more information about the source event as well as a high-level trace (Figure 2.13, middle). The trace at this level contains only the names of the (1) invoked functions, (2) triggered events, and (3) DOM mutations, caused directly or indirectly by the source event. At this level, the user can view multiple episodes to have a side-by-side



**Figure 2.13: Three semantic zoom levels in CLEMATIS. Top: overview. Middle: zoomed one level into an episode, while preserving the context of the story. Bottom: drilled down into the selected episode.**

comparison.

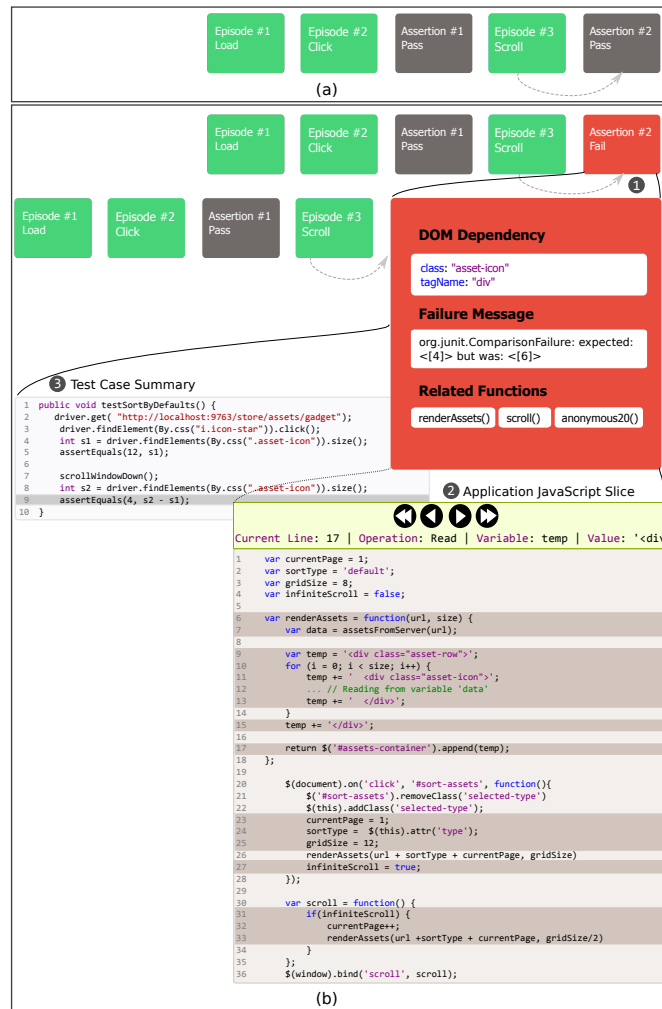
*Zoom Level 2.* The final zoom level exhibits all the information embedded in each episode (Figure 2.13, bottom). Clicking on the “Event” tab will display the type of the event that started the episode (DOM, timeout or XHR event). The

contextual information of the event are displayed based on its type. Choosing the “DOM mutations” tab will list all the changes that were made to the DOM after the execution of this episode. For each DOM element that was added, removed or modified, an item is added to the list of mutations that identifies the modified element, the type of the change and additional information about the change. The third and final tab depicts a detailed trace of the episode. The trace at this level includes a customized sequence diagram of the dynamic flow of all the invoked JavaScript functions and events within that episode. When the user clicks on any of the functions or events in the diagram, the JavaScript code of each executed function is displayed and highlighted (Figure 2.13, bottom).

**Inferred Mappings between Test Failures and Code.** The test case comprehension unit extends the interactive visualization to depict the inferred mappings for the test failure. The visualization helps to understand (1) the client-side JavaScript code related to the assertion failure, (2) the test case’s relations to DOM changes and JavaScript execution, and/or (3) any deviations in the expected behaviour with respect to a previous version where the test passed. Figure 2.14 depicts an example of the high-level view provided by our visualization for a test case.

In the high-level view, the progress of an executed test case over time is depicted on the horizontal axis where the earliest assertions are shown on the left-hand side of the high-level view and the most recent JavaScript events and assertions are shown closer to the right-hand side. The top of Figure 2.14b shows the high-level visualization produced by running the same test case from Figure 2.14a on a faulty version of the application. Passing assertions for a test case are represented as grey nodes, and failures are shown in red. In the case of an assertion, causal links relate the assertion to prior events that may have influenced its outcome. These are events that altered portions of the DOM relevant to the assertion. DOM events, timing events, and network-related JavaScript events are visualized alongside the assertions as green, purple and blue nodes, respectively.

Clicking on a failed assertion node reveals additional details about it (Figure 2.14b). Details include related (1) DOM dependencies, (2) failure messages, and (3) related JavaScript functions. The final zoom level of an assertion node displays all the information captured for the assertion including the captured slice, and the line numbers of the failing test case assertions.



**Figure 2.14: Visualization for a test case. (a) Overview of the passing test case, (b) Three semantic zoom levels for the failing test case; Top: overview. Middle: second zoom level showing assertion details, while preserving the context. Bottom: summary of failing assertion and the backwards slice.**

When displaying the code slice for an assertion, each line of JavaScript code that may have influenced the assertion's outcome is highlighted in the context of the source code (Figure 2.14b, lower-right). The user can further explore the captured slice by stepping through its recorded execution using a provided control

panel, shown in green on Figure 2.14b. By doing so, the user is able to take a post-mortem approach to fault localization whereby the faulty behaviour is studied deterministically offline after execution has completed. Further, the user can also examine the captured runtime values of relevant JavaScript variables.

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

### 2.2.5 Tool Implementation: Clematis

We implemented our approach in a tool called CLEMATIS, which is freely available<sup>4</sup>. 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. We used a proxy since it leads to a non-intrusive instrumentation of the code. A browser plugin would be a suitable alternative. However, unlike browser plugins, a proxy-based approach does not require installing a plugin, is not dependent on the type of the browser, and does not need to be maintained and updated based on browser updates. The trace data collected is periodically transmitted from the browser to the proxy server in JSON format. To observe low-level DOM mutations, we build on and extend the JavaScript Mutation Summary library<sup>5</sup>. 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.

---

<sup>4</sup><http://salt.ece.ubc.ca/software/clematis/>

<sup>5</sup><http://code.google.com/p/mutation-summary/>

## 2.3 Controlled Experiments

To assess the efficacy of our program comprehension approach, we conducted two controlled experiments, following guidelines by Wohlin et al. [139], one in a research lab setting and the other in an industrial environment. In addition, to assess the test failure comprehension extension of CLEMATIS, we conduct a third controlled experiment.

Common design elements of all experiments are described in this section. Sections 2.4–2.6 are dedicated to describing the specific characteristics and results of each experiment, separately.

Our evaluation aims at addressing the following research questions. The first four research questions are designed to evaluate the main code comprehension unit of CLEMATIS. These questions are investigated in the first two experiments (Section 2.4–2.5). RQ1.5, however, assesses the extended test failure comprehension unit of CLEMATIS (Section 2.6). In order to be able to maintain the duration of experiment sessions reasonable, we decided to evaluate the test comprehension unit separately.

**RQ1.1** Does CLEMATIS decrease the task completion *duration* for common tasks in web application comprehension?

**RQ1.2** Does CLEMATIS increase the task completion *accuracy* for common tasks in web application comprehension?

**RQ1.3** For what types of tasks is CLEMATIS most effective?

**RQ1.4** What is the performance overhead of using CLEMATIS? Is the overall performance acceptable?

**RQ1.5** Is the test failure comprehension unit helpful in localizing (and repairing) JavaScript faults detected by test cases?

### 2.3.1 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

**Table 2.1: Adopted and adapted comprehension activities.**

Activity	Description
A1	Investigating the functionality of (a part of) the system
A2	Adding to / changing the system's functionality
A3	Investigating the internal structure of an artifact
A4	Investigating the dependencies between two artifacts
A5	Investigating the 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

in a balanced manner between the two groups. None of the participants had any previous experience with CLEMATIS and all of them volunteered for the study.

**Task Design.** The subjects were required to perform a set of tasks during the experiment, representing tasks normally used in software comprehension and maintenance efforts. We adapted the activities proposed by Pacione et al. [102], which cover categories of common tasks in program comprehension, to web applications by replacing two items. The revised activities are shown in Table 2.1. We designed a set of tasks for each experiment to cover these activities. Tables 2.2 and 2.3 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. Table 2.4 depicts the tasks used in study 3, which aims the fault localization capabilities of CLEMATIS.

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

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

**Data Analysis.** For analyzing the results of each study, we use two types of statistical tests to compare dependent variables across the control and experimental groups. Independent-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<sup>6</sup> for the analysis.

### 2.3.2 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 2.2 and 2.3. 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 RQ1.1 and RQ1.3. To address RQ1.2 and RQ1.3, the accuracy of each task was later evaluated and marked from 0 to 100 according to a rubric that we had created prior to conducting the experiment. The design of the tasks allowed the accuracy of the results to be quantified numerically. The tasks and sample rubrics are available in our technical report Alimadadi et al. [5].

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).

## 2.4 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).

### 2.4.1 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

---

<sup>6</sup><http://www.r-project.org>



**Table 2.2: 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 un-responded XHR	A4, A5, A7

it is the de facto tool used for understanding, editing, and debugging modern web applications.<sup>7</sup> Firebug has been used in other similar studies Zaidman et al. [143].

**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 2.2.

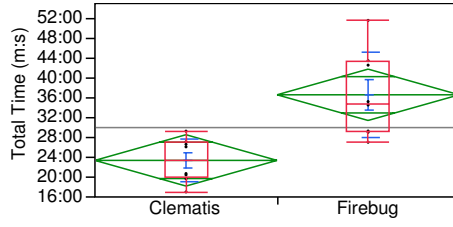
**Experimental Object.** We decided to use a web-based survey application that was developed in our lab. The application had modest size and complexity, so that it could be managed within the time frame anticipated for the experiment. Yet it covered the common comprehension activities described in Table 2.1.

**Experimental Procedure.** We followed the general procedure described in section 2.3.2. 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.

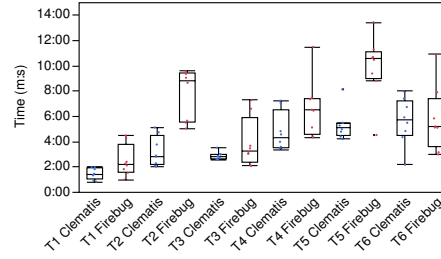
## 2.4.2 Results

**Duration.** To address RQ1.1, we measured the amount of time (minutes:seconds) spent on each task by the participants, and compared the task durations between

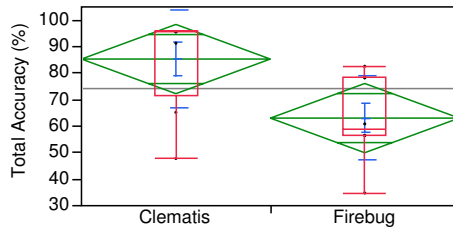
<sup>7</sup>Firebug has over 3 million active daily users: <https://addons.mozilla.org/en-US/firefox/addon/firebug/statistics/usage/>



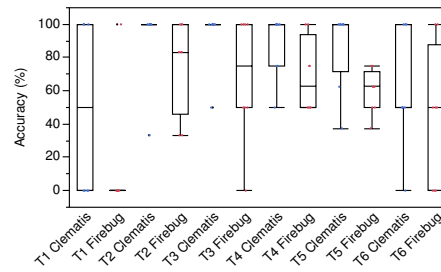
**Figure 2.15: t-test analysis with unequal variances of task completion duration by tool type. Lower values are better. [Study 1]**



**Figure 2.16: Box plots of task completion duration data per task for each tool. Lower values are better. [Study 1]**



**Figure 2.17: t-test analysis with unequal variances of task completion accuracy by tool type. Higher values are better. [Study 1]**



**Figure 2.18: Box plots of task completion accuracy data per task for each tool. Higher values are better. [Study 1]**

CLEMATIS and Firebug using a t-test. According to the results of the test, there was a statistically significant difference ( $p\text{-value}=0.002$ ) in the durations between CLEMATIS ( $M=23:22$ ,  $SD=4:24$ ) and Firebug ( $M=36:35$ ,  $SD=8:35$ ). Figure 2.15 shows the results of the comparisons.

To investigate whether certain categories of tasks (Table 2.2) benefit more from using CLEMATIS (RQ1.3), we tested each task separately. The results showed improvements in time for all tasks. The improvements were statistically significant for tasks 2 and 5, and showed a 60% and 46% average time reduction with CLEMATIS, respectively. The mean times of all tasks for CLEMATIS and Firebug are presented

**Table 2.3: 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

in Figure 2.16. *The results show that on average, participants using CLEMATIS require 36% less time than the control group using Firebug, for performing the same tasks.*

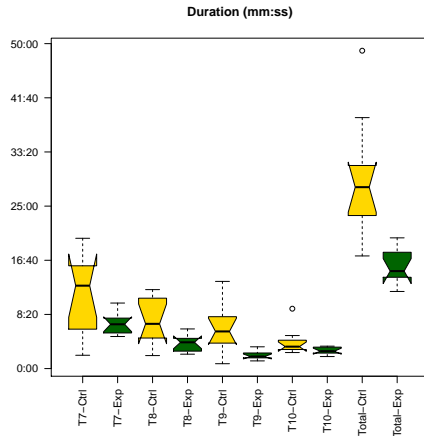
**Accuracy.** The accuracy of answers was calculated in percentages. We compared the accuracy of participants’ answers using a t-test. The results were again in favour of CLEMATIS and were statistically significant ( $p=0.02$ ): CLEMATIS ( $M=83\%$ ,  $SD=18\%$ ) and Firebug ( $M=63\%$ ,  $SD=16\%$ ). This comparison of accuracy between tools is depicted in Figure 2.17. As in the duration case, individual t-tests were then performed for comparing accuracy per task (related to RQ1.3). 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 plot the average accuracies of all tasks for CLEMATIS and Firebug in Figure 2.18. We discuss the implications of these results in Section 2.8.

## 2.5 Experiment 2: Industrial

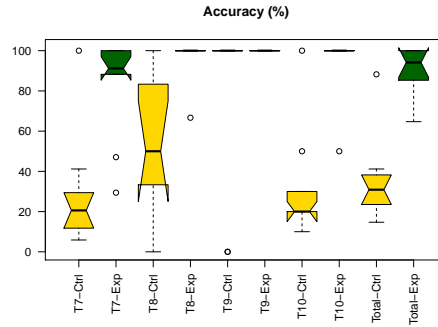
To investigate CLEMATIS’s effectiveness in more realistic settings, we conducted a second controlled experiment at a large software company in Vancouver, where we recruited professional developers as participants and used an open-source web application as the experimental object.

### 2.5.1 Approach

**Experimental Design.** Similar to the first experiment, the participants were divided into experimental and control groups. The experimental group used CLEMATIS



**Figure 2.19: Notched box plots of task completion duration data per task and in total for the control (gold) and experimental (green) groups (lower values are desired). [Study 2]**



**Figure 2.20: Notched box plots of task completion accuracy data per task and in total for the control (gold) and experimental (green) groups (higher values are desired). [Study 2]**

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 2.3) 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<sup>8</sup> is an online photo gallery in PHP, JavaScript, CSS and XHTML. It provides features such as uploading, commenting, rating, and displaying slideshows for users' photos. It contains typical mechanisms such as dynamic DOM mutation, asynchronous calls (XHR and timeouts), and event propagation. Phormer has over 6,000 lines of JavaScript, PHP and CSS code in total (1500 lines of JavaScript). 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 2.3.2, 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.

### 2.5.2 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 2.19 and 2.20, respectively.

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

---

<sup>8</sup><http://p.horm.org/er/>

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 157% on average, compared to developers in the control group.*

### **2.5.3 Qualitative Analysis of Participant Feedback**

The industrial participants in our second experiment shared their feedback regarding the tool they used in the experiment session (CLEMATIS for the experimental group and other tools for the control group). They also discussed their opinions about the features an ideal web application comprehension tool should have. We systematically analyzed [36] the qualitative data to find the main features of a web application comprehension tool according to professional web developers. To the best of our knowledge, at the time conducting this study, there were neither any tools available specifically designed for web application comprehension, nor any studies on their desirable characteristics.

#### **Data Collection**

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

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

and tried to include a more diverse group of participants that still met our original requirements.

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

### **Extracting the Concepts**

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

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

Categories started to form during the process of coding the data. We started to recognize the core categories based on the density of the data in each category. We then continued with selective coding of the remaining forms and manuscripts. We intentionally permitted the evolution of multiple core categories (as opposed to one), in order to account for different aspects of an ideal comprehension tool to get recognized. Multiple categories were integrated to create each core category.

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

### **Guidelines for Web Application Comprehension Tools**

The following are the characteristics of a desirable web application comprehension tool, derived from the participants' responses to our post-questionnaire forms and interviews.

- **Integration with debugging.**

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

- **DOM inspection.**

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



- **JavaScript and DOM interaction.**

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

- **Call hierarchy.**

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

- **Interactivity and realtimeness.**

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

- **Sophisticated visualization.**

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

**Table 2.4: Injected faults for the controlled experiment.**

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

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

## 2.6 Experiment 3: Test Failure Comprehension

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

### 2.6.1 Approach

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

**Experimental Subjects.** 12 participants were recruited for the study at the University of British Columbia (UBC), three females and nine males. The participants were drawn from different education levels at UBC. They all had prior experience in web development and testing, ranging from beginner to professional. Furthermore, six of the participants had worked in industry previously either full-time or through internships

**Task Design.** For this experiment, we used fewer but more complex tasks compared to the first experiment. To answer RQ1.5, participants were given two main tasks,

each involving the debugging of a test failure in the Phormer application (Table 2.4). For each task, participants were given a brief description of the failure and a test case capable of detecting the failure. We used a test suite written by a UBC student for the Phormer application. The test suite was written as part of a separate and independent course project, six months before the inception of our project presented in this paper.

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

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

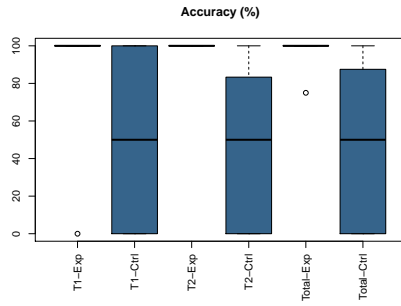
The injected faults are based on common mistakes JavaScript developers make in practice, as identified by Mirshokraie et al. [90].

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

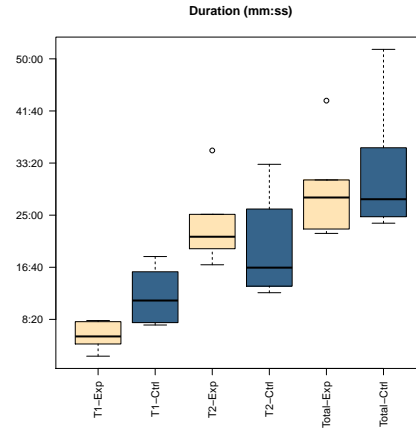
**Experimental Procedure.** The procedure was similar to what we described in 2.3.2. A maximum of 1.5 hours was allocated for the study: 10 minutes were designated for an introduction, 15 minutes were allotted for users to familiarize themselves with the tool being used, 20 minutes were allocated for task 11, another 30 minutes were set aside for task 12, and 15 minutes were used for completing the questionnaire at the end of the study.

## 2.6.2 Results

Figure 2.21 and Figure 2.22 depict box plots of task completion accuracy and duration, per task and in total, for both the experimental group (Exp) and the control group (Ctrl).



**Figure 2.21: Box plots of task completion accuracy data per task and in total for the control (blue) and experimental (cream) groups (higher values are desired).** [Study 3]



**Figure 2.22: Box plots of task completion duration data per task and in total for the control (blue) and experimental (cream) groups (lower values are desired).** [Study 3]

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

**Duration.** To further answer RQ5, we measured the amount of time (minutes:seconds) spent by participants on each task and in total. According to the results of the tests, there was a statistically significant difference in the duration of T11 for CLEMATIS ( $M = 5:42$ ,  $SD = 2:10$ ) and the control group ( $M = 12:03$ ,  $SD = 4:29$ );  $p\text{-value} = 0.016$ . Comparison of the duration data gathered for T12 yielded no significant difference between CLEMATIS ( $M = 23:23$ ,  $SD = 6:31$ ) and the control group ( $M$

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

## 2.7 Performance Overhead

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

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

**Instrumentation overhead.** *Code comprehension.* Average delays of 15.04 and 1.80 seconds were experienced for 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.

*Test comprehension* Average delays of 1.29 and 1.83 seconds were introduced by the selective and non-selective instrumentation algorithms, respectively, on top of the 407 ms required to create a new browser instance. Moreover, the average

trace produced by executing the selectively instrumented application was 37 KB in size. Executing a completely instrumented application resulted in an average trace size of 125 KB. Thus, the selective instrumentation approach is able to reduce trace size by 70% on average, while also reducing instrumentation time by 41%.

**Execution overhead.** *Code comprehension.* For processing 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.

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

**Dynamic analysis overhead.** It took CLEMATIS 585 ms on average to compute each JavaScript slice when utilizing selective instrumentation. Non-selective instrumentation lengthened the required dynamic analysis time to 750 ms. By analyzing a more concise execution trace, CLEMATIS was able to lower the slice computation time by 22%. Thus, we see that CLEMATIS incurs low performance overhead in all three components, mainly due to its selective instrumentation capabilities.

## 2.8 Discussion

### 2.8.1 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 (RQ1.1).

**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 2.1. 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 (RQ1.3).

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 (RQ1.3).

**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 (RQ1.3).

**Test Failure Comprehension.** The average recorded task duration for T11 was significantly lower for the experimental group. The participants in the control group often used breakpoints to step through the application’s execution while running the provided test case. When unsure of the application’s execution, these developers would restart the application and re-execute the test case, extending their task duration. Instead of following a similar approach, those developers using CLEMATIS were able to rewind and replay the application’s execution multiple times offline, after only executing the test case once. The trace collected by CLEMATIS during this initial test case execution was used to deterministically replay the execution while avoiding the overhead associated with re-running the test case.

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

### 2.8.2 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 (RQ1.2). The effect is most visible when the task involves *event propagation* (RQ1.3). The outcome of Study 1 shows that CLEMATIS addresses Challenge 1 (described in Section 2.1) 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* (RQ1.3). This is related to Challenge 3 as described in Section 2.1.

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

**Test Failure Comprehension.** The results from both experimental tasks suggest that CLEMATIS is capable of significantly improving the fault localization and repair capabilities of developers (RQ1.5). many participants in the control group failed to correctly localize the fault, illustrating the difficulty in tracing dependencies in a dynamic language such as JavaScript. Although users in the control group had access to breakpoints, many of them had difficulty stepping through the application’s execution at runtime due to the existence of asynchronous events such as timeouts, which caused non-deterministic behaviour in the application when triggered in the presence of breakpoints.

Many of the participants in the control group fixed the *failure* instead of the actual *fault*; they altered the application’s JavaScript code such that the provided test case would pass, yet the faults still remained unfixed. The JavaScript code related to task 2 contained multiple statements that accessed the DOM dependency of the failing test case assertion. Participants who simply corrected the failure had trouble identifying which of these statements was related to the fault, and as a result would alter the wrong portion of the code. On the other hand, those participants using CLEMATIS were able to reason about these DOM altering statements using the provided links and slices.

### 2.8.3 Consistent Performance

Looking at Figures 2.19 and 2.20, 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.

### 2.8.4 Threats to Validity

**Internal Threats.** The first threat is that different levels of expertise in each subject group could affect the results. We mitigated this threat by manually assigning the subjects to experimental and control groups such that the level of expertise was balanced between the two groups. The second threat is that the tasks in the experiment were biased towards CLEMATIS. We eliminated this threat by adopting the tasks from a well-known framework of common code comprehension tasks Pacione et al. [102]. 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, the experimental object Phormer, and the details of our experimental design (e.g., tasks and questionnaires) are all available making our results reproducible.

### 2.8.5 Limitations

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

JavaScript is a highly dynamic language. There are many cases that occur in JavaScript applications and are not currently supported by CLEMATIS. As an example, CLEMATIS does not instrument JavaScript code that is maintained in strings and is executed using `eval()`. Also, should an exception occur and change the normal means of function execution, the resulting model may be affected. However, these are among features of JavaScript that can be handled in near future using the current design.

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

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

process.

## 2.9 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 also proposed a strategy for helping developers understand the root causes of failing test cases. We presented a novel interactive visualization mechanism based on focus+context techniques, for presenting these complex event interactions in a more comprehensible format to web developers. Our approach is implemented in a code comprehension tool, called CLEMATIS. The evaluation of CLEMATIS points to the efficacy of the approach in reducing the overall time and increasing the accuracy of developer actions, compared to state-of-the-art web development tools. The greatest improvement was seen for tasks involving control flow detection, and especially event propagation, showing the power of our approach.

## Chapter 3

# Hybrid DOM-Sensitive Change Impact Analysis for JavaScript

In Chapter 2, we discussed understanding the interactions of episodes of JavaScript execution. However, software constantly changes to adapt to the changing environment. Understanding and analyzing the impact of change has been a popular research trend. However, performing change impact analysis on JavaScript applications is challenging due to features such as the seamless interactions with the DOM, event-driven and dynamic function calls, and asynchronous client/server communication.

The first feature is the interplay between the JavaScript code and the Document Object Model (DOM) at runtime. The DOM is a standard object model representing HTML at runtime. DOM APIs are used in JavaScript for dynamically accessing, traversing, and updating the content, the structure, and the style of HTML pages. We have observed that the impact of a code change can be propagated through the DOM, even when there may be no visible connections between JavaScript functions and variables in the JavaScript code. The second feature pertains to the highly dynamic [116] and event-driven [133] nature of JavaScript code. For instance, a single fired event can dynamically propagate on the DOM tree [133] and trigger multiple listeners indirectly. These implicit relations between triggered functions are not directly visible in the JavaScript code. Finally, XMLHttpRequest (XHR) objects used for asynchronous communication in JavaScript can transfer the impact

of a change between two parts of the program that are not explicitly connected through the code. For instance, a server response can dynamically generate and execute JavaScript code on the client-side through callbacks.

In this chapter, we propose a hybrid analysis method for change impact analysis of JavaScript-based web applications that combines the advantages of both static and dynamic analysis techniques to obtain a more complete impact set. Our analysis is DOM-sensitive and aware of the event-driven, dynamic and asynchronous entities, and their relations in JavaScript. It creates a novel graph-based representation capturing these relations, which is used for detecting the impact set of a given code change. The main contributions of our work are as follows.

- A formalization of factors and challenges involved in change impact analysis for JavaScript.
- An exploratory study to investigate the existence and role of impact paths that require the analysis of DOM-related and event-based features in JavaScript. The results show that these features exist in real-world applications and cannot be ignored by JavaScript change impact analysis.
- A DOM-sensitive event-aware hybrid change impact analysis technique for JavaScript. The approach creates a novel hybrid model for identifying the impact set of a change in a JavaScript application.
- A set of metrics for ranking the inferred impact set to facilitate the finding and understanding of the desired change impact by developers.
- An implementation of our approach in a tool called TOCHAL (TOol for CHange impact AnaLysis). TOCHAL is open source and available for download [130].
- An empirical evaluation of TOCHAL through a comparison with traditional pure static and dynamic analysis approaches, as well as a controlled experiment to assess the usefulness of TOCHAL in an industrial setting.

Our results show that event-driven and dynamic interactions between JavaScript code and the DOM are prominent in real applications, can affect change propagation, and thus should be part of a JavaScript impact analysis technique. We also find that a hybrid of both static and analysis techniques is necessary for a more complete

```

1  function checkPrice() {
2      var itemName = extractName($('#item231'));
3      var cadPrice = $('#price_ca').innerText;
4      $.ajax({
5          url : "prices/latest.php",
6          type : "POST",
7          data : itemName,
8          success : eval(getAction() + "Item")
9      });
10     confirmPrice();
11 }
12 function updateItem(xhr) {
13     var updatedInfo = getUpdatedPrice(xhr.responseText);
14     suggestItem.apply(this, updatedInfo);
15 }
16 function suggestItem() {
17     if (arguments.length > 2) {
18         displaySuggestion(arguments1);
19     }
20 }
21 function calculateTax() {
22     $(".price").each(function(index) {
23         $(this).text(addTaxToPrice($(this).text()));
24     });
25 }
26 $("#price-ca").bind("click", checkPrice);
27 $("#prices").bind("click", calculateTax);

```

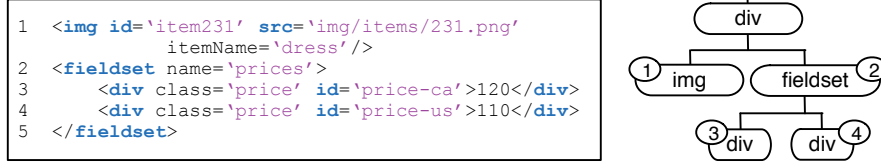
**Figure 3.1: Motivating example: JavaScript code**

analysis. And finally, TOCHAL can improve developers' performance in terms of both impact analysis task completion time (by 78%) and accuracy (by 223%).

### 3.1 Impact Transfer in JavaScript

Many unique features of JavaScript applications require special attention during impact analysis. These features include (but are not limited to) DOM interactions, dynamic event-driven execution of functions, and asynchronous communication with the server. The impact can be transferred through these entities, without direct visible relations in JavaScript. Throughout the rest of the paper, we use the term *indirect* impact to refer to change impact transferred through such features. Impact transferred directly through JavaScript code, e.g., through function calls, is referred to as *direct* impact.

**Relevant Entities.** Let  $f$  be a JavaScript function,  $d$  a DOM element, and  $x$  an XHR



**Figure 3.2: Motivating example: HTML/DOM**

object. If  $F$ ,  $D$ , and  $X$  are sets representing each of those entities, respectively, then the set of all relevant entities is defined as  $E : \sum \varepsilon \leftarrow F \cup D \cup X$

Change impact can propagate between these JavaScript entities through a series of *read* and *write* operations.

**Read/Write Operations.** Let  $\varepsilon_1$  and  $\varepsilon_2$  be arbitrary entities in  $E$ . Suppose  $\varepsilon_1$  writes to  $\varepsilon_2$  at time  $\tau$ . Then the relation is represented as  $\varepsilon_1 W_\tau \varepsilon_2$ . If  $\varepsilon_1$  is read by  $\varepsilon_2$  at time  $\tau$ , then the relation is represented as  $\varepsilon_1 R_\tau \varepsilon_2$ .

The semantics of the relations between entities are drawn from actual JavaScript execution mechanisms (e.g., function  $f$  reads from a DOM element  $d$ ). However, for each  $W/R$  relation between two entities, there is a conceptual  $R/W$  relation between the same two entities in the opposite direction (e.g.,  $d$  writes to  $f$ ). Definition 3.1 formalizes the notion of impact transmission between two entities.

**Impact.** Let  $\varepsilon_1$  and  $\varepsilon_2 \in E$ . If the value and/or the behaviour of  $\varepsilon_2$  depends on the value and/or the behaviour of  $\varepsilon_1$ , then  $\varepsilon_1$  is said to have an impact on  $\varepsilon_2$ , represented as  $\varepsilon_1 \rightarrow \varepsilon_2$ .

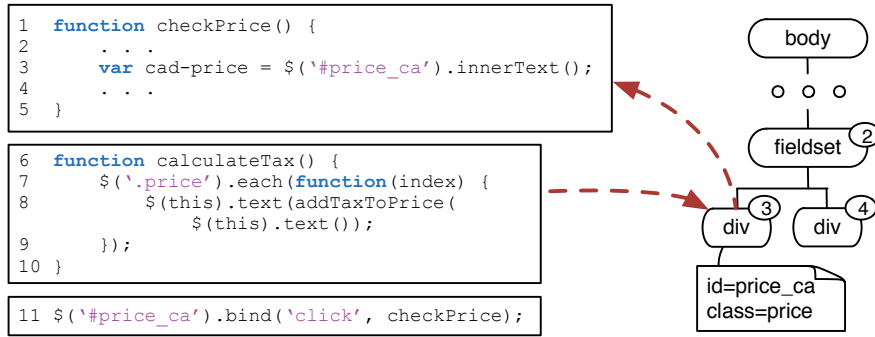
The impact can also be indirectly transferred from entity  $\varepsilon_1$  to  $\varepsilon_2$ , if  $\varepsilon_1$  writes to  $\varepsilon_3$ , which is later read by  $\varepsilon_2$ . We call such this relation a  $WR$  pair.

We use a simple motivating example, presented in Figures 3.1–3.2, to illustrate the challenges and explain each definition. We use different portions of this example in the following subsections. Note that this is a simple example and these challenges are much more potent in large and complex web applications.



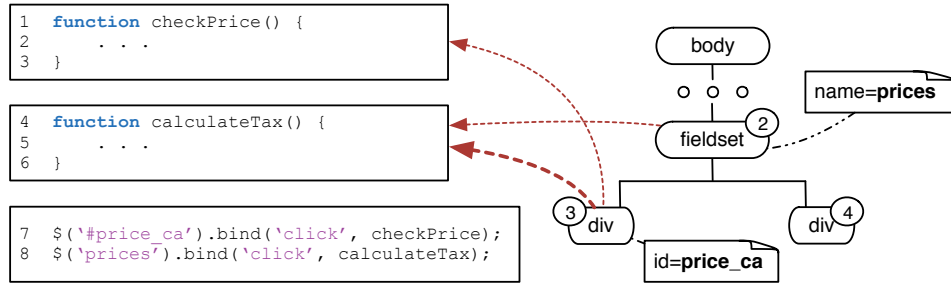
### 3.1.1 Impact through the DOM

In modern web applications, the DOM structure [42] evolves dynamically through the execution of JavaScript code, to update the content, structure, and style of the application in a responsive manner. A JavaScript function can write to a DOM element, which in turn can be read by another function and thus impact its behaviour. Such DOM elements can transfer the impact of a change indirectly. Impact transferred through the DOM introduces an important challenge in identifying change impact in web applications. Hence, in this work, we propose DOM-related dependencies as additional means of impact transfer.



**Figure 3.3: Impact transfer through DOM elements.**

. Figure 3.3 displays a portion of the motivating example that contains a hidden DOM-based dependency. Function `calculateTax()` retrieves all DOM elements having the class attribute `price` (line 7). The function then recalculates the price of each element to include the tax and rewrites the value of the element with the new price (line 8). Later, when the function `checkPrice()` (line 1) is invoked through a user event (registered in line 11), it retrieves the value of a DOM element with id `"price-ca"` (line 3) and uses this value to perform other operations. So far, there is no direct relation between functions `calculateTax()` and `checkPrice()` that shows any dependency between the two code segments. However, looking at the DOM structure shown on the right side of Figure 3.3, we can see that the element with ID `"price-ca"` is also an instance of the `price` class (element ③ on the DOM tree). This means that the value used by `checkPrice()` may be affected by `calculateTax()`. This is a simple example of dynamic



**Figure 3.4: Impact transfer through event propagation.**

DOM dependency, which needs to be taken into account in impact analysis of JavaScript applications.

### 3.1.2 Impact through Event Propagation

In web applications, a single event can propagate on the DOM tree and invoke multiple handlers of the same event-type attached to any of the ancestors of the target element [133]. The direction of the event propagation depends on whether the capturing or bubbling mode is enabled. 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. With *bubbling* enabled, an event first triggers the handler of the target element on which the event was fired, and then it bubbles up and triggers the parents' handlers. The second type of impact dependencies we introduce pertain to the hidden relations between the handlers invoked via propagation of the original event on the target DOM tree. Such invoked handlers can be involved in change impact propagation, i.e., they can affect the control flow of the application and thus need to be considered in impact analysis for JavaScript.

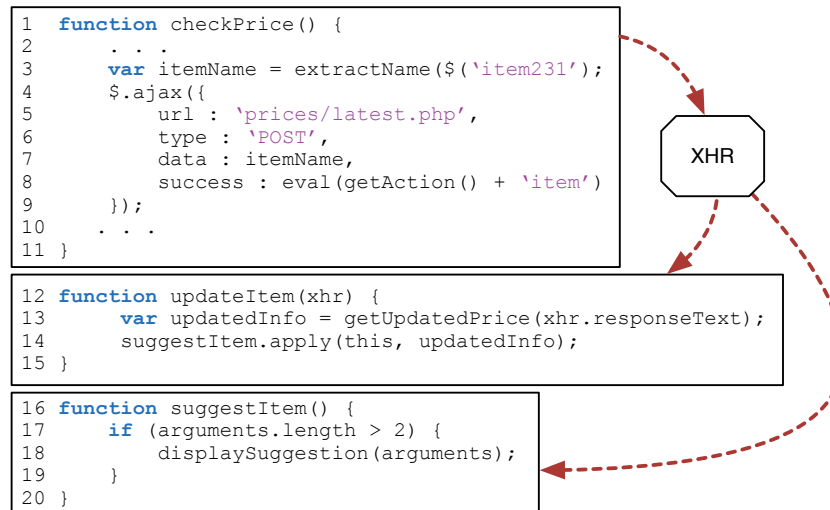
. In a segment of the motivating example shown in Figure 3.4, `checkPrice()` is attached to the element with `id price-ca` as an event handler (line 7). Therefore, if a user clicks on that element (element ③ on the right side DOM tree), function `checkPrice()` gets invoked. However, `price-ca` is contained within a `fieldset` element with the name `prices` (element ② on the DOM tree), which is similarly bound to an event handler for the `click` event (Figure 3.4, line 8).

Due to event *bubbling*, any click on `price-ca` will bubble up to `prices` and trigger its event handler as well. Hence, `calculateTax()` is invoked through propagation of an event originally targeting `price-ca`. As a result, the execution of `calculateTax()` also depends on the `price-ca` element, in addition to the `prices` element.

### 3.1.3 Impact through Asynchronous Callbacks

XMLHttpRequest (XHR) objects help developers enrich user experiences with web applications through asynchronous communication with the server. While increasing the interactivity and responsiveness of applications, XHR usage adds additional complexity to impact analysis. Each XHR consists of three main phases: *open*, *send*, and *response*. A callback function is invoked when the XHR response is received from the server, without a user involvement. A change in opening the request, sending it, or the sent message can impact the response of the server, as well as the behaviour of the application after receiving the response through a callback function. Different components of XHR objects can make the detection of control and data flow relations more troublesome, particularly when these components are not necessarily collocated in the same function or module. This motivates the third type of impact dependencies we introduce in this work.

. `checkPrice()` sends an asynchronous request to the server (lines 4–9 in Figure 3.5). However, the assigned callback function cannot be recognized statically, as the code uses the action chosen by the user dynamically to invoke the appropriate function (line 8, `eval`). Let's assume that in this example the selected action is to “update” the price of an item, and hence the `updateItem()` function is assigned as the callback function of the XHR object. As it can be seen, there are no direct function calls or shared variables between `checkPrice()` and `updateItem()` to enable traditional change impact analysis techniques to derive a dependency relation between the two functions. However, the XHR message along with the data that was sent with it can affect the response that comes back from the server and thus can impact the behaviour of `updateItem()`.



**Figure 3.5: Impact transfer through asynchronous callbacks.**

### 3.1.4 JavaScript Dynamism

Many traditional impact analysis techniques use static aspects of the code to determine the impact set of a change. Dynamic features of the JavaScript language pose a challenge to static analysis techniques. For instance, almost everything in JavaScript, from fields and methods of objects to their parents, can be created or modified at runtime. Also, JavaScript's dynamic policies for invoking functions can add more complexities. One such policy is function variadicity, which is common in web applications [116]; i.e., in JavaScript, functions can be invoked with more or less arguments compared to the parameters specified in a function's static declaration. In addition to the DOM, event, and XHR challenges, the dynamic features of the JavaScript language need to be addressed in an effective change impact analysis technique.

. In line 14 of Figure 3.5, `updateItem()` invokes `suggestItem()` through the `apply()` function, which makes it impossible to infer the number of passed arguments statically. Function `suggestItem()`, the callee, takes no arguments according to its declaration (line 16). Yet, the function is invoked with an arbitrary non-zero number of arguments, which can change the execution of the application (Figure 3.5, line 17). Knowledge of the passed arguments at runtime is crucial for

performing precise data-flow analysis, required for impact analysis.

### 3.1.5 Impact Paths

The concept of WR pairs can be generalized to any mechanism that can transfer impact in JavaScript, such as XHR objects, function arguments, and function return values. Consecutive WR pairs involving all JavaScript entities can form general impact paths, as described in Definition 3.1.5.

**Impact Path.** An impact path of an entity ( $P(\varepsilon)$ ) is a directed acyclic path starting from entity  $\varepsilon$ . The nodes on the path are entities in the system, and the edges are the directed impact relations that connect those entities.

For instance, `updateItem()`  $\rightarrow$  `suggestItem()`, `checkPrice()`  $\rightarrow$  `#price-ca` (DOM element with `id=price-ca`), `checkPrice()`  $\rightarrow$  `#price-ca`  $\rightarrow$  `calculateTax()`  $\rightarrow$  `addTaxToPrice()` are examples of impact paths that exist in the running example (Figure 3.1).

The length of an impact path is defined as the number of entities in the path. The minimum length for propagation of the impact of a change through DOM elements is 3 ( $f W_{\tau_1} d R_{\tau_2} g \mid \tau_1 < \tau_2, d \in D, f, g \in F$ ).

## 3.2 Exploratory Study: DOM-related and Event-based Impacts

We conducted an exploratory study to investigate the role of JavaScript’s DOM-related, event-based and dynamic features in code change propagation. Our goal was to understand whether DOM elements, event handlers, and propagated events contribute to forming new impact paths in JavaScript code.

**Subject Applications.** We selected ten web applications that make extensive use of JavaScript on the client-side for this study. We selected these applications from (1) participants of recent JavaScript programming contests, and (2) trending and popular JavaScript projects on GitHub<sup>1</sup>. They are listed in column 1 of Table 3.1.

---

<sup>1</sup><https://github.com/trending/>

**Design.** We capture JavaScript-DOM interactions as well as any occurrences of event propagation on the DOM tree. For each DOM access that occurs during the execution, we collect the accessed entity, the JavaScript function that accesses the DOM, and the access type. Access types are directed operations performed on DOM elements by JavaScript functions, where the direction of the access is determined by the direction of the flow of data. For instance, assume function *foo* creates DOM element *e* at time  $\tau$ , which means  $foo W_{\tau} e$ . Then the type of access is `element-creation` and the direction of access is from *foo* to *e*. The collected data is analyzed to extract the impact set for each of the JavaScript functions involved in the execution. The DOM elements that are located on at least one impact path of a function are the ones that can contribute to the impact propagation process and are called `WR` elements for simplicity. The considered impact paths are required to have a length of at least three (e.g.,  $foo W_{\tau_1} e R_{\tau_2} bar$ , for functions *foo* and *bar* and DOM element *e*). Moreover, redundant impact pairs (reads and writes between same entities) do not contribute to the impact paths and are therefore eliminated from the analysis.

**Results.** Each application is manually exercised in different scenarios multiple times and the results are integrated. The results are shown in section (A) of Table 3.1. The first column of section (A) displays the total number of DOM elements accessed by JavaScript code during execution. The second column of the section shows the ratio of `WR` DOM elements to the total number of involved DOM elements from the first column. The number of DOM event handlers that were triggered during the execution is shown in column three. Column four represents the percentage of handlers that were invoked through event propagation (capturing or bubbling) to the total number of triggered handlers. The average length of impact paths in each application is depicted in column one of section (B).

The results show that on average, 42% of the DOM elements that were accessed during the execution of these applications, were part of an impact path between two functions. Moreover, about 14% of the executed event handlers were invoked through event propagation mechanisms. The results thus reveal the importance of DOM elements in transferring the impact. Also, the role of propagated event handlers is significant in determining the dynamic behaviour of a JavaScript application.

**Table 3.1: (A) Results of analyzing JavaScript’s DOM-related and dynamic features. (B) Factors in determining impact metrics.**

JavaScript		(A) DOM and dynamic features				(B) Factors of impact metrics				
Application	LOC	# DOM elem.	% WR DOM elem.	# of handlers	% prop. handlers	Avg. Path Length	fan-in elem.	fan-out elem.	fan-in func.	fan-out func.
same-game	229	62	98	20	45	6.3	1.9	2.9	23.1	15.2
ghostBusters	343	44	61	39	0	4.3	3.3	0.4	2.6	20.0
simple-cart	9238	41	51	14	0	3.9	2.1	1.7	2.7	3.3
mojule	522	47	17	18	33	7.0	1.5	2.1	4.6	3.3
jq-notebook	839	1	100	21	38	4.0	16.0	11.0	0.9	1.3
doctored.js	3534	2	50	47	15	5.3	4.0	7.0	1.0	0.8
jointlondon	2498	34	9	16	0	3.7	0.8	2.2	1.6	0.6
space-mahjon	983	61	10	53	4	4.0	1.8	3.0	1.5	0.9
listo	354	5	20	10	0	4.0	0.1	1.5	21.4	1.9
peggame	1274	17	6	23	0	3.0	0.8	1.3	4.3	2.1
Average	1981	31	42	26	14	4.6	3.2	3.3	6.3	4.9

Hence, a CIA technique for JavaScript application should consider the DOM-related and event-based features as media for propagating the impact.

We further analyze the structure of the created dependency graphs to gain more insight into the nature of DOM- and event-based relations within JavaScript applications. Among all structural and semantic aspects of the graphs, the average fan-in and fan-out scores of the functions and DOM elements in the graphs are reported in section (B) of Table 3.1. These factors are selected due to their correlations with the ratio of WR elements in subject applications. We use this information later in the paper, when we propose a set of metrics for ranking the impact set (Section 3.4).

### 3.3 Hybrid Analysis

We propose a hybrid technique, called TOCHAL, which augments static analysis with dynamic analysis to enable a DOM-sensitive and event-aware change impact analysis method for JavaScript applications.

#### 3.3.1 Static Control-Flow and Partial Data-Flow Analysis

Our approach first identifies JavaScript entities that *can* be analyzed statically. Among entities described in Definition 3.1, JavaScript functions ( $F$ ) are the only entities that fit this criterion. The DOM is created and mutated during execution. This limits the static reasoning about its structure and possible event propagations that would affect change impact. Regarding XHR objects, it is not easy to infer

statically what messages are received from the server. Moreover, there is no static information available regarding the order and timing of asynchronous callbacks.

Our static analysis module incorporates *direct* relations between functions into a static call graph (SCG) by analyzing the JavaScript code. In JavaScript, functions are first-class citizens and receive the same treatment as objects; we augment the same static call graph with global variables, which we treat similarly as the functions.

To increase the precision of the static analysis, which in turn improves the quality of the impact set, we perform a pruning algorithm on the extracted dependencies. The pruning is conducted based on a partial data-flow analysis of the call graph. Function invocations are not considered as impact relations unless the two functions have a data dependency through passed arguments or return values, as described in Section 3.3.1. This does not concern data dependencies through global variables shared between two functions, where separate dependency relations are formed.

**Function Dependencies** Let  $\rho, \delta \in P$ . Then impact relations between  $f$  and  $g$  are defined as:

- $f \rightarrow g$  if  $f$  invokes  $g$  and the signature of  $g$  indicates that it takes parameters.
- $g \rightarrow f$  if  $f$  invokes  $g$  and the definition of  $g$  includes a return value.

### 3.3.2 Analyzing the Dynamic Features

To include the dynamic features of the JavaScript language in our impact analysis, our dynamic analysis module intercepts, transforms, and instruments the JavaScript code on-the-fly to collect execution traces. To collect a trace of function executions, the beginning and the end of each JavaScript function are instrumented. Function declarations are modified to collect traces of function invocation and passed arguments. We also trace function terminations and return statements (if they exist in the function). These traces are then used to create a dynamic call graph (DCG) that captures dependencies between function executions at runtime.

The DCG is an under-approximation of the call graph, while the SCG is an over-approximation of the call graph. The DCG contains fewer false positives compared to the SCG, and we augment it to capture DOM-related, event-driven, and asynchronous features of JavaScript, as explained below.



*DOM-Sensitive Impact Analysis* A JavaScript function can impact a DOM element and vice versa, which is defined as:

**Direct Impact between JavaScript and DOM** Consider a DOM element  $d \in D$ , and a function  $f \in F$ . Then  $f$  can directly impact  $d$  and vice versa:

$$\begin{cases} f \rightarrow_{\tau} d & \text{if } fW_{\tau}d \\ d \rightarrow_{\tau} f & \text{if } fR_{\tau}d \end{cases}$$

Furthermore, the impact can travel from function  $f$  to function  $g$ , through a DOM element  $d$ , under certain conditions as defined in the next definition.

**Indirect JavaScript Impact through DOM** Consider two functions  $f, g \in F$ , and a DOM element  $d \in D$ .  $f$  can indirectly impact  $g$  through  $d$ , if and only if the following conditions hold:

$$f \rightarrow_d g \text{ if } \begin{cases} fW_{\tau_1}d & \& \\ gR_{\tau_2}d & \& \\ \tau_2 > \tau_1 \end{cases}$$

In other words, function  $f$  can potentially impact function  $g$  through DOM element  $d$ , if  $f$  writes to  $d$  and  $g$  reads from the same element. Such a write-read ( $WR$ ) pair indicates the existence of a potential impact between the two functions, if the read instruction happens after the write. Such  $WR$  pairs can occur subsequently, involving more elements and functions in the application. The reading function can itself write to a DOM element and augment the propagation path. The same change can then potentially impact all elements and functions that are on such a path.

To analyze how the DOM transfers the change impact (Section 3.3.2), all *read* and *write* accesses to the DOM need to be monitored dynamically. Each access is made from a JavaScript function to a DOM node, element, or an attribute, and through standard DOM API calls (e.g., `getElementById`, `querySelector`). We modify the prototype of the `Node`, `Document` and `Element` classes to be able to dynamically intercept DOM accesses, while preserving the original behaviour of these classes. This allows us to monitor changes to the structure of the DOM tree, as well as the existence, content, and attributes of DOM elements.

It is worth mentioning that the caller functions are extracted from the dynamic context of the intercepted DOM API calls. As a result, if a function does not exist or is not detected statically (e.g., it is created through an `eval` statement), it will still be captured in the dynamic phase if it interacts with other entities and is part of the execution.

For each function and DOM element involved in an access, we augment the dynamic call graph by adding two nodes (if they do not already exist), and connecting them through an edge representing the type of access that was made from the function to the element.

### Event-Based Impact Propagation

TOCHAL also captures all event handlers called directly and indirectly through event propagation on the DOM tree.

Definition 3.3.2 summarizes the potential impact transmission between a DOM element and all event handlers that are called through event propagation.

**Indirect Impact via Propagation** Let  $d$  be a DOM element that has an event handler for event  $e$ . Consider  $prop_e[d]$  to be the set of all JavaScript functions that are submitted as handlers for event  $e$  to  $d$  or any of its ancestors in the DOM tree, and thus can be triggered by event propagation. Then  $d$  can impact all these handlers indirectly:  $d \rightarrow prop_e[d]$ .

Moreover, the dynamic analysis module yields information on function arguments and return values for all directly and indirectly invoked functions. These variables may differ from what has been declared in static function signatures, due to function variadicity in JavaScript.

*XHR Relations* There are three main phases in the lifecycle of each XHR object: open, send, and response. These three phases can be scattered throughout the code. In addition, callbacks from the server-side could invoke other functions on the client-side, and hence it is not trivial to find the XHR components statically. Our technique instruments and intercepts each component of an XHR object by wrapping around the XHR object of the browser. The gathered information is then

**Table 3.2: Impact transfer through different entities.**

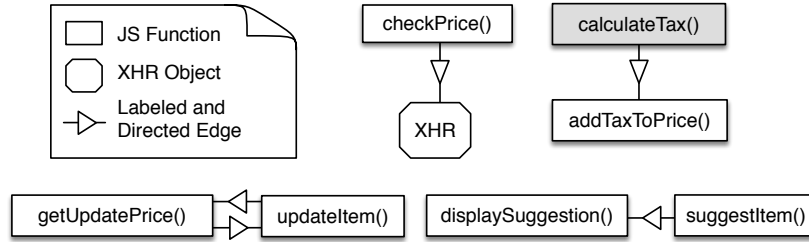
Assume $f, g \in F$ (functions), $d \in D$ (DOM) and $x \in X$ (XHR)	
Relation	Description
$fW_{\tau}g$	<ol style="list-style-type: none"> <li>1. <math>f</math> calls <math>g</math> and passes arguments.</li> <li>2. <math>g</math> calls <math>f</math> and <math>f</math> returns a value.</li> </ol>
$fW_{\tau}d$	<ol style="list-style-type: none"> <li>1. <math>f</math> creates element <math>d</math>, adds it to the DOM tree, deletes it, or detaches or relocates it from the DOM.</li> <li>2. <math>f</math> modifies the content or the attributes of <math>d</math>.</li> </ol>
$dR_{\tau}f$	<ol style="list-style-type: none"> <li>1. <math>f</math> uses information regarding the content, attributes, or location of <math>d</math> in the DOM.</li> <li>2. <math>f</math> is bound to <math>d</math> through an event handling mechanism.</li> <li>3. <math>f</math> is set as an event handler of one of the ancestors of <math>d</math>, that can be triggered via event propagation.</li> </ol>
$fW_{\tau}x$	<ol style="list-style-type: none"> <li>1. <math>f</math> opens <math>x</math> as a new XHR object.</li> <li>2. <math>f</math> sends a previously-created XHR object <math>x</math>.</li> </ol>
$xR_{\tau}f$	<ol style="list-style-type: none"> <li>1. <math>f</math> is set as the callback function of <math>x</math></li> <li>2. <math>f</math> sends a previously-created XHR object <math>x</math>.</li> </ol>

used to augment the dynamic call graph. Similar to the previous features, new nodes representing XHR objects are added to the graph, the involved functions nodes are only added if they were not previously included, and the function nodes are linked to the XHR node based on the type of access they make to the XHR object. The access types are defined by the type of the interaction between a function  $f$  and an XHR object  $x$ , and determine the direction of the impact relation. For instance, if  $f$  creates and opens  $x$ , then  $f \rightarrow x$ , while if  $f$  is registered as the callback function of  $x$ , then  $x \rightarrow f$ .

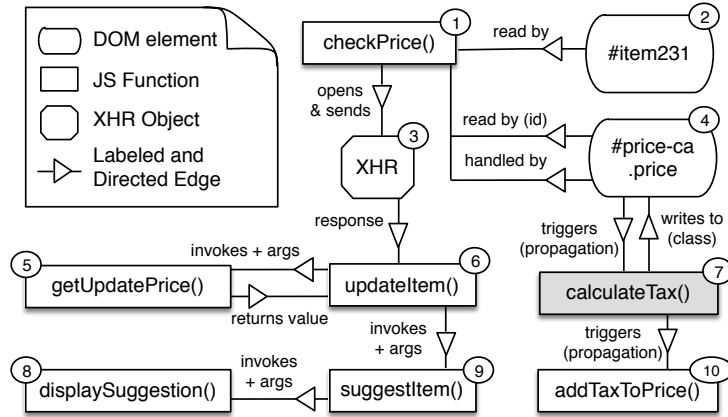
Similar to the static call graph, we enhance the dynamic call graph using interprocedural data-flow analysis. Arguments and return values of functions are used to trim the call graph where there is no data flow between two functions (Definition 3.3.1). However, instead of using the static function code, the dynamic arguments and return values are used to support function variadicity at run time.

### 3.3.3 Hybrid Model for Impact Analysis

At this stage, TOCHAL creates a system dependency graph by integrating the obtained static and dynamic call graphs. This graph is used for performing impact analysis on JavaScript applications. The dynamic part of the model contributes to the precision of the analysis, while its static features make it more complete. We take a best-effort approach for fulfilling soundness, following the soundness



**Figure 3.6: A static call graph, displaying the dependencies extracted from the running example (Figures 3.1 and 3.2).**



**Figure 3.7: A sample hybrid graph, including the dynamic and DOM-sensitive dependencies extracted from the running example (Figures 3.1 and 3.2).**

manifesto [80]. In order to satisfy the practicality of our approach in terms of precision and scalability, complete soundness is not a concern of our approach. The hybrid model is represented as a directed graph. The vertices are system entities and the edges are the potential impact relations as summarized in Table 3.2.

**Vertices.** The vertices in the graph are all entities that are present statically or are created during the execution of an application. The vertices can take one of the following types:

- *JavaScript functions.* JavaScript functions extracted by our static analyzer (Section 3.3.1) are added as vertices. Functions found dynamically are added as well due to their involvement in the impact propagation, even if they are not connected directly (Section 3.3.2).
- *DOM elements.* The importance of DOM elements in transferring the impact dynamically was discussed in Section 3.3.2. Accessed DOM elements (and their contextual information) are captured as vertices in the model.
- *XHR objects.* XHR vertices incorporate information regarding the creation and sending of messages, as well as callback functions and data transmitted dynamically between the server and the browser.

**Edges.** The edges in the graph are labeled and directed.

- *Direction.* The direction of each edge depicts the flow of the data between two vertices. The edges are categorized into *read* and *write* accesses. An edge is directed from the vertex that writes (offers) the data to the vertex that reads it.
- *Labels.* The edge labels indicate the type of dependency relations that connect the vertices. Different labels are used to connect different vertices, since the valid operations vary for each category of vertices.

*Example.* Figure 3.6 shows a dependency graph for the running example (Figures 3.1 and 3.2) obtained through the static analysis module alone. On the other hand, Figure 3.7 depicts a simplified hybrid graph utilizing both the static and dynamic analysis modules of TOCHAL. This is hard to do with the static graph. However, using the hybrid graph, one can find the potential impact set of a change in entity  $\epsilon$  by tracing the graph forward, starting from  $\epsilon$ .

Consider a case where a developer plans to make a change to the `calculateTax()` function (line 21 of Figure 3.1), and would like to find the potential impact before making the actual change. A DOM-agnostic static change impact analysis method (see Figure 3.6) would report that only `addTaxToPrice()` would be affected. The source code shows that next to the `addTaxToPrice()` function, DOM elements with `class=price` (lines 22–23 of Figure 3.1) can also be affected.

However, our hybrid DOM-sensitive analysis reveals that there exist more impact paths. The DOM element with `id=price-ca` is also a member of a

**Table 3.3: Impact Metrics**

Entity	Metric	Description	CC with % WR DOM elem.
$d \in D$	$f_{in}(d)$	Number of functions $f$ such that $fW_{\tau}d$	0.66
$f \in F$	$f_{in}(f)$	Number of elements $d$ such that $fR_{\tau}d$	0.74
	$f_{out}(f)$	Number of elements $d$ such that $fW_{\tau}d$	0.62
$\varepsilon \in DorF$	$\hat{L}[P]$	Average length of impact paths in the application	0.59
	$D_m(\varepsilon)$	Minimum distance of $\varepsilon$ from the change set	-

`class=price` (box 4 of Figure 3.7). This element can thus be impacted by `calculateTax()`, and in turn can propagate the impact to `checkPrice()` indirectly (lines 3 & 26 of Figure 3.1, box 1 of Figure 3.7). Furthermore, evaluating the response of an XHR object, `checkPrice()` can then transfer the impact to `updateItem()` (box 6), which can propagate the impact to more functions (boxes 5, 8 & 9). To summarize, as our hybrid model shows, changing the `calculateTax()` function can affect six more elements in addition to the two elements that can be detected by statically analyzing the code. Thus, the proper impact set consists of functions `addTaxToPrice()`, `checkPrice()`, `updateItem()`, `getUpdatedPrice()`, `suggestItem()`, `displaySuggestion()`, the DOM element with `id=price-ca`, and the anonymous XHR object.

### 3.4 Impact Metrics and Impact Set Ranking

An impact set inclusive of the contributions of both static and dynamic analyses can become large and overwhelm the user. Considering that not all entities in the impact set are equally important, providing a ranking mechanism is essential for helping developers identify relevant impacted entities more efficiently.

We propose a set of *impact metrics* to estimate the importance of each entity in the produced impact set. The impact metrics, outlined in Table 3.3, are variables derived from the semantic and structural characteristics of the hybrid graph. These metrics can affect the probability of impact propagation, through DOM-related and dynamic mechanisms of JavaScript. Based on the impact metrics, we propose an impact ranking mechanism as outlined in Definition 3.4. The *impact rank* score of each entity  $\varepsilon$ , referred to as  $IR(\varepsilon)$ , is an estimation of the importance of  $\varepsilon$  in

propagating the change, relative to other elements in the impact set.

**Impact rank** Let  $\varepsilon$  be an entity in the impact set. Then the impact rank of  $\varepsilon$  is defined as:

$$IR(\varepsilon) = \frac{IR_{pr}(\varepsilon) * \hat{L}[P(\varepsilon)] * Fan_w(\varepsilon)}{D_m(\varepsilon)}$$

where, the value of the impact rank of an element in the impact set depends on four variables. (1)  $IR_{pr}(\varepsilon)$ : the accumulation of impact ranks of immediate precedents of entity  $\varepsilon$  in the hybrid graph that are on an impact path from the change set to  $\varepsilon$ . If the impact is transferred to  $\varepsilon$  from entities with higher ranks, then the importance of  $\varepsilon$  in transferring the impact potentially increases. (2)  $D_m(\varepsilon)$ : the shortest distance of  $\varepsilon$  from the change set in the hybrid graph. The closer  $\varepsilon$  is to the change set, the higher is the probability of being impacted by the change in practice. Hence, a longer distance of  $\varepsilon$  from the change set has a lesser effect on its impact rank. (3)  $\hat{L}[P(\varepsilon)]$ : the average length of an impact path starting from entity  $\varepsilon$ . If  $\varepsilon$  can cascade the impact to more elements and deeper levels in the propagation graph, then  $\varepsilon$  is potentially an important entity in the impact set. (4)  $Fan_w(\varepsilon)$ : the weighted fan-in / fan-out score of functions and DOM elements in the hybrid graph is indicative of the number of entities that can impact and be impacted by  $\varepsilon$  directly; hence, we consider it as a determining factor in impact rank determination.  $Fan_w(\varepsilon)$  is calculated as follows:

$$Fan_w(\varepsilon) = \begin{cases} w_1 * f_{in}(\varepsilon) & \text{if } \varepsilon \in D \\ w_1 * f_{in}(\varepsilon) + w_2 * f_{out}(\varepsilon) & \text{if } \varepsilon \in F \end{cases}$$

$\hat{L}[P(\varepsilon)]$  and  $Fan_w(\varepsilon)$  are extracted from the findings of our exploratory study (Section 3.2). During the course of the study, we measured the semantic and topological properties of hybrid graphs of ten web applications. Among the analyzed variables, the length of impact paths, fan-in of DOM element node, and fan-in and fan-out of function nodes (section (B) of Table 3.1) have a correlation with the number or ratio of DOM elements that can transfer the impact. Hence, these variables can affect the probability of impact propagation through an entity. They thus play a role in these two determining factors influencing the overall impact rank.

### 3.5 Tool Implementation: TOCHAL

We implemented TOCHAL using JavaScript libraries such as Esprima,<sup>2</sup> Estraverse<sup>3</sup> and Escodegen<sup>4</sup> for parsing and transforming the JavaScript code. We use these libraries to instrument functions in a manner that permits us to collect data about function invocations, function entries and function exits. The collected data also include dynamic values of functions' arguments and return values. External files are attached to the beginning of each document that allow instrumentation and interception of DOM events, XHR objects and timeouts. We use the Mutation Summary library [93] to detect JavaScript code appended to the DOM on the fly. Therefore, we can extend function instrumentation to the JavaScript code that gets created dynamically during application execution. We use WALA [128] to extract the static call graphs of applications.

TOCHAL provides an interface for developers to utilize our hybrid change impact analysis. The main goal is to facilitate the comprehension of change impact “during” development and debugging activities. Hence, the analysis needs to be available to developers while they make changes to their application. We have integrated our impact analysis method within Google Chrome’s DevTools [27], a popular web development environment. This decision entails a number of benefits, namely (1) the approach is complementary to existing web development platforms and environments; it does not change the functionality they provide, but augments their capabilities. (2) The developer can perform the impact analysis in the same context as the code, and can preserve her mental model of the code. (3) The developer is not required to learn a new tool, or divide her attention between two different tools.

The interface allows the user to select JavaScript functions (including XHR callbacks) or DOM elements as the change set, and then perform the impact analysis. Chrome’s DevTools includes a set of panels, each providing a window to a subset of functionalities that the platform provides. Two panels are of more interest for us: “elements” and “sources”. The elements panel visualizes and provides inspection mechanisms on the DOM. The sources panel displays all of the JavaScript code that contributes to the application. We add a sidebar to each of these panels, allowing

---

<sup>2</sup><http://esprima.org>

<sup>3</sup><https://github.com/Constellation/estrapverse/>

<sup>4</sup><https://github.com/Constellation/escodegen/>



the user to invoke the CIA unit, on a selected entity, at any stage of the development. TOCHAL is publicly available for download [130].

## 3.6 Evaluation

We empirically evaluate the effectiveness and usefulness of TOCHAL through the following research questions:

**RQ4.1** How does our hybrid method compare to static/dynamic analysis methods?

**RQ4.2** Does TOCHAL help developers in performing change impact analysis in practice?

We address these questions through two studies, each described in the following two subsections, respectively.

### 3.6.1 Study 1: Comparing Static, Dynamic, and Hybrid Analyses

To address RQ4.1, we conduct a study to evaluate the impact sets extracted using TOCHAL in comparison with those detected by static and dynamic analysis techniques separately. We compare TOCHAL with a state-of-the-art static analysis technique. We also examine the differences in the outcomes of TOCHAL with those of the dynamic analysis unit of TOCHAL. The term *dynamic* analysis encompasses the DOM-sensitive, dynamic, event-driven, and asynchronous analysis performed by TOCHAL. Our first hypothesis is that TOCHAL outperforms static impact analysis due to its support for *dynamic* analysis. We also hypothesize that while dynamic analysis is a significant part of Tochal, it is outperformed by tochal’s hybrid analysis.

We decided to compare TOCHAL with static and dynamic approaches, since to the best of our knowledge, TOCHAL is the first change impact analysis technique for JavaScript and there is no similar tool available for JavaScript.

*Design and Procedure.* The only entities that can be analyzed by both static and *dynamic* analysis methods are JavaScript functions. Hence, to be fair to both static and dynamic analyses, we configure TOCHAL to only deliver functions in the impact sets. TOCHAL’s hybrid model and analysis, however, do not differ from what is

described in Section 5.3 and use DOM-based, dynamic and asynchronous entities and relations to extract the functions in the impact sets.

We use the same set of subject applications from our exploratory study of Section 3.2 (Table 3.4, column 1). For each application, we randomly sample three functions and extract their impact sets using each of the methods. We compare the impact sets to assess the completeness of the outcomes of analysis with each approach. The sample functions are selected from a pool of functions that are recognized by all three analysis methods. In other words, static and *dynamic* analysis alone are not able to detect some functions that are detected by TOCHAL and are involved in its hybrid analysis. If static/*dynamic* analysis is performed on any of the functions it does not recognize, the impact set will be empty. We aim at comparing impact sets at this stage and these functions are unable to provide useful information regarding the analysis. Thus, we do not include such functions in the comparison. However, this indicates the need for an investigation of the functions involved in each type of analysis (in the dependency graphs), as described in the next paragraph.

We measure the number of functions that are included in the dependency graphs of each analysis, contributing to the detection of the impact sets. The average number of functions in each type of analysis denotes the extent of the analysis performed for extracting the impact sets. Moreover, the lower number of recognized functions by an analysis method means that there are more functions for which the method is unable to perform CIA.

**Pure Static Analysis.** We use the static analysis part of our approach, which is built using WALA [128]. WALA is a leading static analysis tool for JavaScript, used by many other JavaScript analysis techniques [125, 136, 137]. It should be noted that WALA by itself does not support change impact analysis. For the purpose of this evaluation, we extended and directed it towards performing static impact analysis to conduct the comparisons.

**Pure Dynamic Analysis.** We disable the static module of TOCHAL and only utilize the *dynamic* analysis module. The applications are exercised through their test suites when available and manually within multiple sessions and the results are integrated. Each manual session follows a set of pre-defined scenarios that covers

**Table 3.4: Results of comparison between static, dynamic and TOCHAL (RQ1) (A) Comparison of impact sets (B) Comparison of functions in system dependency graphs**

Application	(A) Impact sets											(B) Functions						
	TOCHAL			Static			Static TOCHAL	Dynamic			Dynamic TOCHAL	TOCHAL	Static TOCHAL	Dynamic TOCHAL	Pure Static	Pure Dynamic		
	avg	min	max	avg	min	max	%	avg	min	max	%	avg	%	%	%	%		
same-game	4.33	2	7	0.67	0	1	15	3.67	2	6	85	16	56	93	6	44		
ghostBusters	1.33	0	2	0.33	0	1	25	1.33	0	2	75	10	80	100	0	20		
simple-cart	1.67	0	3	0.67	0	1	40	1.67	0	3	100	44	74	91	9	26		
mojule	1.00	0	2	0.33	0	1	33	0.67	0	2	67	21	24	90	10	71		
jq-notebook	2.67	0	7	0.67	0	2	25	2.33	0	6	87	19	47	100	0	53		
doctored.js	1.67	0	4	0.33	0	1	20	1.33	0	3	80	38	67	50	56	33		
jointlondon	2.33	0	5	0.67	0	1	29	1.67	0	4	72	36	31	85	15	69		
space-mahjong	2.67	1	5	1.00	0	2	37	2.00	0	5	63	27	56	93	7	44		
listo	1.33	1	2	0.00	0	0	0	1.33	1	2	100	12	75	58	25	42		
peggame	2.67	1	6	1.00	1	1	37	2.00	0	5	75	24	83	75	25	17		
Average	2.07	0.5	4.3	0.57	0.1	1.1	26	1.8	0.3	3.9	80	25	59	84	15	42		

all main use-cases of the application that are accessible to an end-user.

**Hybrid Analysis.** We use the hybrid model of TOCHAL for performing impact analysis and obtaining the set of functions that are involved in the hybrid analysis.

*Results and Discussion* To answer RQ4.1, we discuss the outcomes of the study, summarized in table 3.4.

**Completeness of Impact Sets.** Section (A) of Table 3.4 depicts the results of the impact set detection using static, *dynamic* and hybrid analysis methods. The first column of this section displays the average, minimum and maximum sizes of the impact sets of the selected JavaScript functions, detected by TOCHAL. The second column displays the average, minimum and maximum impact set size by static analysis. The third column represents the percentage of the ratio of the impact set size of static analysis to that of TOCHAL. Similarly, the fourth and fifth columns, respectively, show the impact set size for *dynamic* analysis, and the ratio of the size of impact sets using *dynamic* analysis compared to TOCHAL. We observe an increase in completeness for all applications in favour of TOCHAL. On average, static and *dynamic* analysis methods detected 0.57 and 1.80 functions in the impact set of each sample function, respectively. The hybrid TOCHAL method, however, extracted an average of 2.07 functions to be potentially impacted by each of the sample functions.

Overall, the impact sets extracted by TOCHAL include 74% more functions

on average compared to those detected by static impact analysis. The outcome conforms the findings of our earlier exploratory study, showing the prevalence and importance of DOM-related and dynamic characteristics of JavaScript in impact analysis. TOCHAL takes into account new types of entities in its dependency graph that are more aligned with the nature of JavaScript (DOM elements, XHR objects). It also recognizes new (and mostly hidden) types of relations between these entities, that lead to more complete and more precise dependency graphs and impact sets at the same time. The static analysis still remains useful in TOCHAL since *dynamic* analysis can only cover 80% of the impact sets detected by hybrid analysis and cannot replace it.

It is worth noting the small sizes of static impact sets. Considering the conservativeness of static methods, the dependency graphs in general can include many relations between functions that are not feasible in practice. Therefore, static impact sets in CIA methods for traditional languages are expected to get large and contain infeasible relations. Our results show an *opposite phenomenon* for JavaScript applications. The small sizes of static dependency graphs and the resulting impact sets attest to the difficulties and limitations of static analysis for JavaScript. The findings further confirm that new forms of definitions and usages of functions, objects, DOM elements and asynchronous objects negatively affect analysis of useful dependency graphs. Static analysis confronts more barriers during the analysis JavaScript applications that should be mitigated using an approach that supports these features.

**Necessity of Hybrid Analysis.** Separate data sets are collected from each type of analysis, to let us distinguish between the statically detected and dynamically invoked functions. Through these datasets, we extract the functions that were invoked dynamically but were not detected statically, and the functions that were extracted before the execution, but did not play a role at runtime. The results are summarized in section (B) of Table 3.4. The first column of this section contains the total number of functions that were included in our hybrid analysis. The second and third columns represent the percentages of these functions that were covered by static and *dynamic* analysis units, respectively.

The static analysis method only covers about 56% of the functions that are

covered during hybrid analysis. As expected, this inadequacy is caused due to the dynamism of JavaScript even in simple function invocations. Moreover, the *dynamic* analyzer includes 86% of the functions detected by the hybrid analysis. This confirms our anticipation of incompleteness of dynamic analysis, due to its reliance on specific executed scenarios of the code. The increase in the covered JavaScript functions by our proposed hybrid analysis leads to a more complete system dependency graph, which is used to perform the impact analysis. Hence, the proposed hybrid approach can improve the accuracy of JavaScript impact analysis.

Column 4 in section (A) of Table 3.4 represents the percentage of JavaScript functions that were detected by the static analyzer, but were *not* invoked during any of the executions of the applications. Note that the existence of such functions, that would be missed from pure *dynamic* analysis, is sufficient evidence for the necessity of a hybrid analysis technique. Column 5 of the same section of the table, on the other hand, displays the percentage of functions that were executed during the execution of each application, but were *not* detected by the static analyzer. The results confirm our previous intuition regarding the shortcomings of static JavaScript analysis, even in a well-established type of analysis based merely on function declarations and invocations.

### 3.6.2 Study 2: Industrial Controlled Experiment

We conducted a controlled experiment [139] in an *industrial* environment<sup>5</sup> to address the following two questions, derived from RQ4.2:

**RQ4.2.1** Does TOCHAL increase the CIA task completion *accuracy*?

**RQ4.2.2** Does TOCHAL decrease the CIA task completion *duration*?

*Experimental Subjects* We recruited 10 participants, 5 female and 5 male, with ages ranging from 20 to 34. At the time of conducting the experiment, all participants were employed in a large software company in Vancouver. Their skills in web development ranged from medium to professional. All participants volunteered for taking part in our experiment and did not receive monetary compensation.

---

<sup>5</sup>The experimental material is available at: <http://ece.ubc.ca/~saba/tochal/study-materials.zip>

*Experimental Design* The experiment had a “between-subject” design to avoid carryover effects. We formed two independent groups of participants. The experimental group used TOCHAL for performing the tasks; none of the participants were familiar with TOCHAL prior to attending the experimental sessions. Since TOCHAL is the first CIA tool for JavaScript applications, the control group performed the tasks using the development tool they used in their day-to-day web development activities (without TOCHAL). To avoid bias, it was important for the members of the two groups to have similar proficiency levels. We manually assigned the participants to the groups to ensure this was the case based on a pre-questionnaire evaluating their experience and expertise.

**Tasks.** We designed a set of four tasks that involved finding change impacts during web application maintenance activities. The tasks, outlined in Table 3.5, require the participants to detect and comprehend the potential impact of a change in a JavaScript function or a DOM element. Moreover, two of the tasks require participants to use their understanding of the change impact to find a bug or an inconsistency that occurs after the change.

All features of TOCHAL were available to the experimental group during the experimental session. We were particularly interested in assessing the usefulness of the ranking mechanism of TOCHAL, which was deployed based on our proposed impact metrics (Section 3.4). Hence, we designed a smaller study that enabled us to compare the effects of using the ranking. However, this comparison was only meaningful for the experimental group, who used TOCHAL and had access to its ranking feature. Having this in mind, the two debugging tasks, tasks T3 and T4 in Table 3.5, were designed to have similar levels of difficulty and to require similar amount of effort and expertise. However, we disabled the ranking feature for T3, while leaving it enabled for T4. These two tasks were counterbalanced to avoid order effects.

**Dependent and Independent Variables.** We measured two continuous variable as our dependent variables. Task completion *duration* was measured in minutes and seconds. Tasks completion *accuracy* was measured in marks based on a fixed and predefined grading rubric, and the marks were converted to percentages for consistency across all tasks.

**Table 3.5: Impact analysis tasks used in the controlled experiment.**

Task	Description
T1	Finding the potential impact of a DOM element (the button changing the size of the displayed slideshow images)
T2	Finding the potential impact of a JavaScript function (the function toggling the play/pause state of the slideshow)
T3	Finding a conflict after making a new change (problem in submitting new comments after changing the table containing all comments of a picture). Ranking is disabled.
T4	Finding a bug in JavaScript code (entered email format is not properly checked)

The independent variable is a nominal variable including two levels. One level represents using TOCHAL, and the other level depicts using a different tool (e.g., Chrome DevTools, Firefox Developer Tools, or Firebug).

*Experimental Object* We used Phormer [106], an online photo gallery in PHP, JavaScript, CSS and XHTML, which consisted of around 6,000 lines of code and over 38,000 downloads at the time of conducting the experiment. Throughout the experiment, the participants had to understand and debug parts of the application related to displaying a slideshow of pictures, viewing the pictures, and authoring comments.

*Experimental Procedure.* The experiment consisted of three main phases.

**Pre-Experiment.** The participants were given a pre-questionnaire form before attending the experimental session. They were required to provide information regarding their proficiency and experience in web development and software engineering. This information was used for assigning them into one of the two groups prior to the session. The pre-questionnaire also inquired about the tools the participants normally used for performing their every-day web development tasks. The answers to this question were used to determine the proficiency levels of the participants for assigning them into the control and experimental groups. The information was also used to indicate the tool the control group would use for the experiment. It is worth mentioning that all participants of the control group selected Google Chrome as their preferred web development tool. In the experimental session, the participants were introduced to TOCHAL for the first time and were trained to use it.

Then they were given a few minutes to familiarize themselves with the tool, and ask us questions if needed.

**Tasks.** During the experimental session, the participants were asked to perform a set of tasks, as indicated in Table 3.5. To avoid experimental bias, each task was handed out on a separate sheet of paper to the participant, when the investigator marked the start time of the task. The investigator terminated the measurement of the task duration when the participant returned the paper to the investigator along with her answer. The task accuracy was marked after the session, using a rubric created prior to conducting the experiment.

**Post-Experiment.** We asked the participants to fill a post-questionnaire form after completing the tasks. The questionnaire contained questions regarding both the helpful features and the shortcomings of the tool they used in the experiment. Moreover, we enquired about the metrics our participants considered to affect the importance of an entity in the impact set.

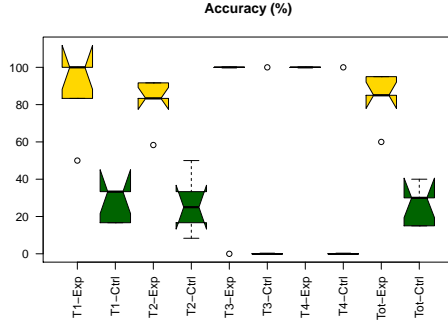
## Results and Discussions

Figures 3.8 and 3.9 depict the results of task completion duration and accuracy for both experimental and control groups.

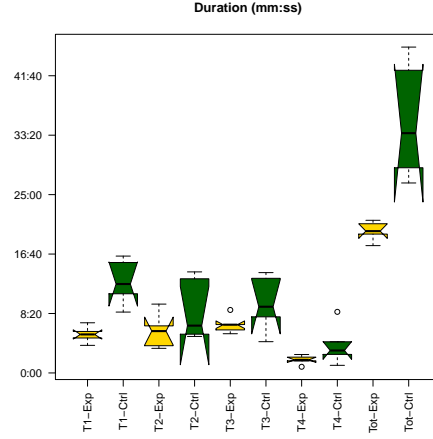
**Accuracy (RQ4.2.1).** We ran the Shapiro-Wilk normality test on the accuracy data. The results showed that the data collected for tasks T1, T3 and T4 were not normally distributed and hence, Mann-Whitney U tests were used for analyzing the results of these tasks. The data gathered for task T2 and the total accuracy of the tasks were normally distributed and were analyzed using t-tests. The results of conducting the tests revealed a statistically significant difference for the experimental group using TOCHAL (Mean=84%, STDDDev=14%) compared to the control group (Mean=26%, STDDDev=11%); (p-value=0.0001). *Overall, participants using TOCHAL perform 223% more accurately compared to the control group, across all tasks in the experiment.*

Further, the accuracy for all tasks was higher when the participants used TOCHAL. The improvement was statistically significant for tasks T1, T2 and T4, but not T3. Recall from the tasks table (Table 4.4), that the ranking mechanism of TOCHAL was disabled for T3. This outcome thus emphasizes the value of ranking





**Figure 3.8: Task completion accuracy per task and in total for the control (ctrl) and experimental (exp) groups (RQ4.2.1).**



**Figure 3.9: Task completion duration data per task and in total for the control (ctrl) and experimental (exp) groups (RQ4.2.2).**

the impact set in helping the user find the important impacts more efficiently. Not having access to the impact ranks, participants had to expend more manual effort. Enforcing more analysis burden on the participant increases the variation in the answers based on the individual differences in abilities of the participants. The high variation prevents the statistical significance of the results of T3, in spite of the 60% higher accuracy average for TOCHAL users.

**Duration (RQ4.2.2).** We first used the Shapiro-Wilk test on the duration data and confirmed that the data was normally distributed. Therefore, we ran a set of t-tests on all individual tasks, as well as on the total time spent on all of the tasks throughout the experiment. The results show a statistically significant difference in the total duration for the experimental group using TOCHAL (Mean=19:54, STDDDev=1:23), compared to the control group using other tools (Mean=35:26, STDDDev=8:21); (p-value=0.01). *Overall, participants using TOCHAL spent an average of 78% less time on the same set of tasks compared to those using other tools.*

Further, the participants using TOCHAL showed improvements for all of the

tasks compared to the control group, with the difference being statistically significant for T1. Considering the higher accuracy scores for other tasks when using TOCHAL, we observe that many participants of the control group terminated the tasks with incomplete and incorrect answers, erroneously assuming that they had completed the task. This caused them to obtain lower accuracy scores, while still spending more time on average.

**Ranking.** The results of the experimental group were analyzed further to investigate the effects of using our proposed ranking system on the duration and accuracy of understanding and debugging an application after a change. T3 and T4 were both debugging tasks requiring similar levels of expertise. As mentioned earlier, the use of TOCHAL's ranking feature was disabled for T3, while it was enabled for T4. Performing a t-test on the results revealed a statistically significant difference in task completion duration between participants who used the ranking mechanism (Mean=1:50, STDDev=39), compared to those who did not (Mean=6:34, STDDev=1:16), with  $p\text{-value} < 0.05$ . We used a Mann-Whitney U test to analyze the accuracy results for the ranking mechanism. Although using the ranks led to an average of 20% higher accuracy of the answers, the results were not statistically significant in this case. However, the participants completed T4 about 3.7 times faster than T3. This significant improvement highlights the importance of ranking the impact set, and is an indication of the usefulness of our impact ranking method (Section 3.4).

*Participants' Feedback.* We gathered qualitative data from both experimental and control groups through a post-questionnaire form. The questionnaire asked participants about the usefulness of the tool used in the study, its strengths, and its shortcomings. Overall, all TOCHAL users mentioned that they found the tool useful. The participants were particularly pleased with the idea of finding the potential impact of JavaScript functions and DOM elements. Understanding the dynamic behaviour and underlying dependencies were mentioned to be most useful. The users found TOCHAL to be helpful in solving the problems faster, especially in the presence of its ranking mechanism. The participants in the experimental group were also interested to see more features in TOCHAL. The feature requests were mostly

attributed to improving the user interface. Some participants were also interested in having direct debugging support in the tool.

Furthermore, we asked our participants about the metrics they thought could determine the importance of an entity in the impact set. We analyzed and categorized the participants' opinions on impact metrics. A few of the final categories were considered in our ranking strategy, as we expected. These metrics included (1) distance of an entity from the change set and (2) number of dependencies of an entity. However, there were many other categories that are not included in our methods, such as: (1) number of invocations of a function, (2) visibility of the impact in the interface, (3) importance of the feature to the customers, (4) "breadth" of usage of the entity: whether it is used by multiple files, or is isolated to one file (5) complexity of the function, and (6) "history" of a function: rate of having faults in the function.

*Threats to Validity.* The first internal threat is the the population selection threat, and specifically the equivalency of the two groups in terms of their expertise. We addressed this threat by first manually dividing the participants into different proficiency levels, which were extracted from the information gathered in the pre-questionnaire forms. We then distributed the members of each level into control and experimental groups by random sampling. The second threat is the investigator's bias while marking the accuracy of the answers. We mitigated this threat by creating a marking rubric while designing the tasks, and using the same rubric for marking the results later. A similar threat can arise from the bias in measuring the duration of the tasks. We resolved this issue by enforcing a mutual supervision on time measurement by both the investigator and the participant. We assigned a separate sheet of paper to each task, which was handed to the participant in the beginning of the task, and was returned to the investigator after task completion. The fourth threat can be introduced by the choice of the tool that the control group used. This threat was mitigated by allowing the control group to choose the tool of their preference. Finally, the compensatory incentives were not a threat to validity as all of our participants volunteered for the experiment, with no monetary compensation.

An external threat is with regard to the representativeness of our sample of population. We mitigated this threat by recruiting professional web developers from

industry as our participants. Another concern is raised regarding the representativeness of tasks used in the experiment. We used general tasks enquiring about understanding the impact of a code change and also detecting potential faults in the code, which are faced by developers in their daily professional activities.

### **3.7 Concluding Remarks**

The dynamic, asynchronous, and event-based nature of JavaScript and its interactions with the DOM make modern web applications highly interactive and responsive to users. These same features also introduce new types of dependencies into the system, making the prediction and detection of change impact challenging for developers. In this paper, we proposed an automated technique, called TOCHAL, for performing a hybrid DOM-sensitive change impact analysis for JavaScript. TOCHAL builds a novel hybrid system dependency graph, by inferring and combining static and dynamic call graphs. Our technique ranks the detected impact set based on the relative importance of the entities in the hybrid graph. Our evaluation shows that the dynamic and DOM-based JavaScript features occur in real applications and can lead to significant means of impact propagation. Furthermore, we find that a hybrid approach leads to a more complete analysis compared with a pure static or dynamic analysis. Finally, our industrial controlled experiment shows that TOCHAL increases developers' performance, by helping them to perform maintenance tasks faster and more accurately.

## **Chapter 4**

# **Understanding Asynchronous Interactions in Full-Stack JavaScript**

JavaScript has become one of the most popular languages in practice. In Chapters 2–3, we discussed understanding the behaviour and impact of change in client-side JavaScript. However, 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.

However, 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 [49] 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 known 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 [10, 61, 84, 101, 143]. In our earlier work, we proposed a technique, called CLEMATIS [4], 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 chapter, 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 [119]. 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.

## 4.1 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 4.1–4.3. These are simple examples and the challenges are more potent in large and complex applications.

### 4.1.1 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 4.1 depicts an end point for receiving a GET request (lines 12–18 use Express.js APIs [46]). 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

```

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 4.1: Receiving HTTP requests at an end point**

used in JavaScript applications [49]. However, It is not trivial to understand the JavaScript code that deploys callbacks. In many cases callbacks are nested (up to eight levels deep [49]) 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 [24]. 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.

*Example.* The code in Figure 4.2 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.

#### 4.1.2 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



```

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 (←
5         instType) {
6           row.forEach(function(arg, i) {
7             resolveAliases(instType, arguments0);
8           }) }) }) })
9   })
10  // send response back
11 })

```

**Figure 4.2: Callback hell**

```

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   }
10  function periodicUpdate() {
11    $.get('/pupdate', function(data) {
12      // do stuff
13    });
14  }
15  setInterval(periodicUpdate, updateCycle);

```

**Figure 4.3: Asynchronous client-side JavaScript**

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 4.3 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 function (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 pe-

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.

### 4.1.3 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.

## 4.2 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.

### 4.2.1 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:

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

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:

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

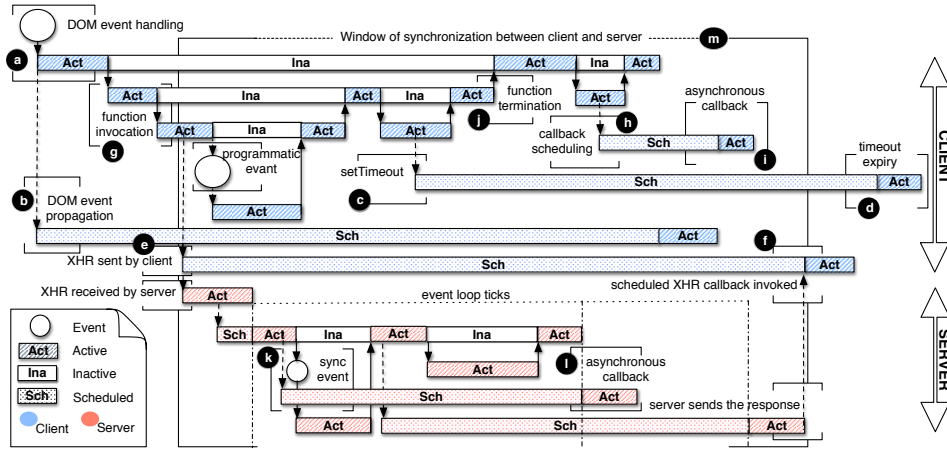
To understand the lifeline of each execution, the model must account for all these phases. There can be a maximum of one scheduling phase per function execution, depending on whether it was triggered asynchronously. This means  $Sch(f)$  can occur 0 or 1 times in the beginning of a lifeline. Each execution 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 4.1 summarizes the information required for analyzing various types of vertices that is captured by our approach, in addition to the time of event occurrence.

**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



**Figure 4.4: A sample temporal, context-sensitive and asynchronous model of events, lifelines, and interactions.**

**Table 4.1: 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

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

**Table 4.2: 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

various types of relations, which are summarized in Table 4.2.

The direction of an edge represents the direction of the control flow between the involved nodes, which depends on the type of the edge.

Table 4.3 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 4.4 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.

#### 4.2.2 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 [6] 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 [133]. Thus, we capture propagated handlers and their relations with the original events.

**Table 4.3: 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 unhandled 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 $\langle ev, el \rangle$	$e := newV_e(ev, el)$ $ll := newV_{ll}(ev \rightarrow \text{handler})$ $\sqcup_{ll} \leftarrow \sqcup_{ll} \cup ll$ $\Pi_{fe} \leftarrow \Pi_{fe} \cup e$	$d = newE_e(\text{src} : e, \text{dst} : ll, \text{action})$	if(JS active) $ll.\text{init}(\text{Phase.Sch})$ $\Pi_{ue} \leftarrow \Pi_{ue} \cup e$ O.W. $ll.\text{init}(\text{Phase.Act})$
2	Propagated DOM events ( $\Sigma pe$ )	$e_p := \Pi_{fe} \rightarrow \text{head}$ $\forall e_i \in \Sigma pe$ $ll_i := newV_{ll}(pe \rightarrow \text{handler})$ $\sqcup_{ll} \leftarrow \sqcup_{ll} \cup ll$	$\forall e_i \in \Sigma pe$ $d := newE_e(\text{src} : e_p, \text{dst} : ll_i, e_p \rightarrow \text{action})$	$\forall e_i \in \Sigma pe$ $ll_i.\text{init}(\text{Phase.Sch})$ $\Pi_{fe} \leftarrow \Pi_{fe} \cup \{e_i\}$
3	Timeout set	$to - id := newuniqueTOID()$ $ll_c := \sqcup_{ll} \rightarrow \text{head}$ $ll := newV_{ll}(to - id, \text{delay})$	$d := newE_{ts}(\text{src} : ll_c, \text{dst} : ll, \text{delay})$	$ll.\text{init}(\text{Phase.Sch})$ if(server side) $\bigcirc \leftarrow \bigcirc \cup \langle \text{TO}, ll \rangle$
4	Timeout callback	$ll := \sqcup_{ll}.\text{get}(\text{TO} \rightarrow to - id)$		$ll.\text{end}(\text{phase.Sch})$ $ll.\text{start}(\text{phase.Act})$ if(server side) $\bigcirc.\text{pop}(ll \rightarrow to - id)$
5	XHR send	$xhr - id := newuniqueXHRID()$ $ll_c := \sqcup_{ll} \rightarrow \text{head}$ $ll := newV_{ll}(xhr - id, \text{url}, \text{method})$	$d := newE_{xs}(\text{src} : ll_c, \text{dst} : ll, \text{data})$	$ll.\text{init}(\text{Phase.Sch})$ if(server side) $\bigcirc \leftarrow \bigcirc \cup \langle \text{XHR}, ll \rangle$
6	XHR callback	$ll := \sqcup_{ll}.\text{get}(\text{XHR} \rightarrow xhr - id)$		$ll.\text{end}(\text{phase.Sch})$ $ll.\text{start}(\text{phase.Act})$ if(server side) $\bigcirc.\text{pop}(ll \rightarrow xhr - id)$
7	Server events	$ll_c := \sqcup_{ll} \rightarrow \text{head}$ $e := newV_e(ev)$ $ll := newV_{ll}(ev \rightarrow \text{handler})$ $\sqcup_{ll} \leftarrow \sqcup_{ll} \cup ll$	$d = newE_e(\text{src} : ll_c, \text{dst} : e, e \rightarrow \text{type})$ $d = newE_e(\text{src} : e, \text{dst} : ll)$	$ll.\text{init}(\text{Phase.Act})$ $ll_c.\text{init}(\text{Phase.Ina})$
8	Callback scheduling	$ll_c := \sqcup_{ll} \rightarrow \text{head}$ $ll := newV_{ll}(\text{callback})$	$d = newE_{cs}(\text{src} : ll_c, \text{dst} : ll)$	$ll.\text{init}(\text{Phase.Sch})$ if(server side) $\bigcirc \leftarrow \bigcirc \cup \langle \text{CB}, ll \rangle$
9	Callback invokation	$ll := \bigcirc \rightarrow \text{head}$		$ll.\text{end}(\text{Phase.Sch})$ $ll.\text{start}(\text{Phase.Act})$ if(server side) $\bigcirc.\text{pop}(\langle \text{CB}, ll \rangle)$
10	Function invokation	$ll_c := \sqcup_{ll} \rightarrow \text{head}$ $ll := newV_{ll}(\text{function})$	$d = newE_c(\text{src} : ll_c, \text{dst} : ll)$	$ll.\text{start}(\text{Phase.Act})$ $ll_c.\text{start}(\text{Phase.Ina})$
11	Function termination	$ll_c := \sqcup_{ll} \rightarrow \text{pop}$ $ll_p := \sqcup_{ll} \rightarrow \text{head}$	$d = newE_t(\text{src} : ll_c, \text{dst} : ll_p)$	$ll_c.\text{start}(\text{Phase.Ter})$ $ll_p.\text{start}(\text{Phase.Act})$

Upon invocation of the original handler, we create a node representing the event and add it to the model (Table 4.3, row 1 & Figure 4.4, 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 4.3, row 2 & Figure 4.4, 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 `setTimeout`'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 [4].

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 initialized with a *scheduled* phase in the beginning. An edge is created from current *active* lifeline to the newly created *scheduled* lifeline (Table 4.3, row 3 & Figure 4.4, 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 4.3, row 4 & Figure 4.4, 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 4.3, row 5 & Figure 4.4, 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 4.3, row 6 & Figure 4.4, 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 4.3, row 10 & Figure 4.4, 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 4.3, row 8 & Figure 4.4, 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 4.3, row 9 & Figure 4.4, 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 *terminated* phase, and is popped (Table 4.3, row 11 & Figure 4.4, 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.

### 4.2.3 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 4.3, rows 8–9 & Figure 4.4, 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 `EventEmitters`. A major difference between `EventEmitters` 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 4.3, row 7 & Figure 4.4, l).

#### 4.2.4 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 4.3, rows 5–6 & Figure 4.4, 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.

#### 4.2.5 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 [2].

**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 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

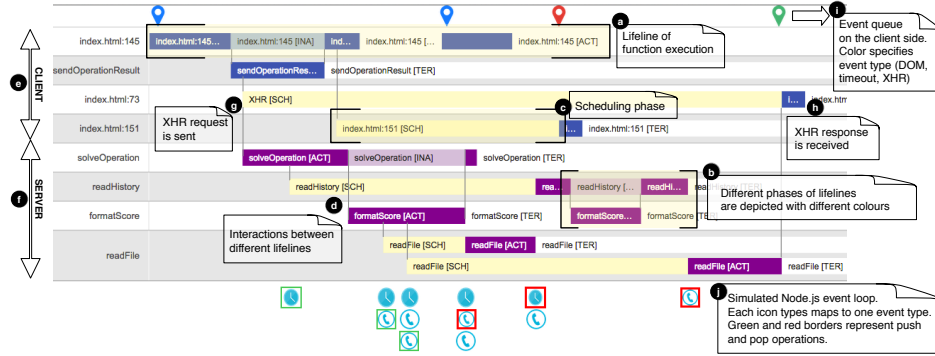


Figure 4.5: A snapshot of the visualization.

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 [108]. We incorporate custom lifeline visualization in the interface of our behavioural model.

**Visualization example.** Figure 5.7 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.

#### 4.2.6 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 [44], instrument the AST using Estraverse [45], and serialize the AST back into JavaScript code with Escodegen [43]. The visualization is built on top of the timeline view of Google chart tool [53]. SAHAND is publicly available [119].

## 4.3 Evaluation

We conducted a comparative controlled experiment [139] to investigate the effects of using SAHAND on the performance of developers when understanding full-stack web applications. Our experimental dataset is available online [119].

### 4.3.1 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 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 [82] 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 [27], 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 JetBrains WebStorm [135], 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

**Table 4.4: 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.

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. [102]. Each of our tasks covers multiple activities, and also involves elements specific to JavaScript comprehension. The tasks are summarized in Table 4.4.

*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 4.4). 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 [119]. 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.

### 4.3.2 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



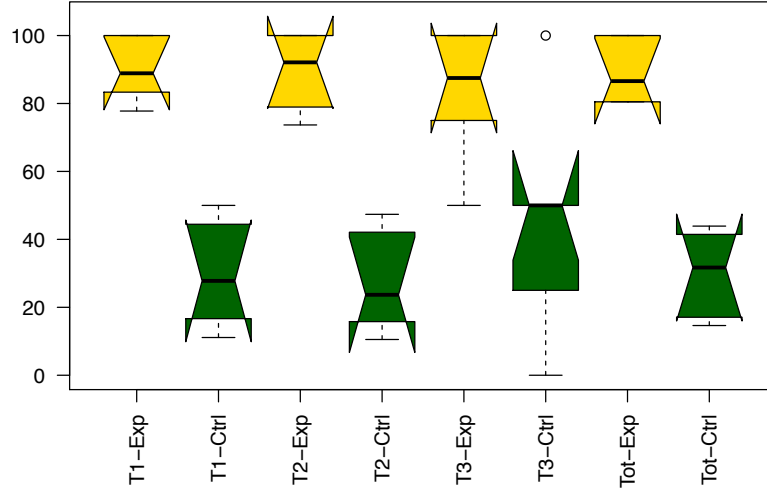
tasks). Next, we ran a MANOVA test on each individual task. If the results of a MANOVA test were significant, we examined the univariate tests (ANOVA) to see if the significance in performance improvement was due to the duration, accuracy, or both.

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 5.8. *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.*

### 4.3.3 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



**Figure 4.6: Accuracy results. Gold plots display experimental (SAHAND) group, and green plots display the control group. Higher values are better.**

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 [102]. 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 [119].

## 4.4 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

[119]. 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.

## Chapter 5

# Inferring Hierarchical Motifs from Execution Traces

### 5.1 Introduction

Program comprehension is an essential first step for many software engineering tasks. Developers spend a considerable amount of time understanding code. About 50% of maintenance effort is spent on comprehension alone [31]. Unfortunately, code understanding is challenging. To understand code, developers typically start by *searching* for clues in the code and the environment. Then they go back and forth on the incoming and outgoing dependencies to *relate* pieces of foraged information. Throughout the process, they *collect* information they find relevant for understanding the code on an “as-needed” basis [71]. However, developers often fail in searching and relating information, and lose track of relevant information when using such ad-hoc strategies [117]. Further, developers form mental models of code, that are often inaccurate [92]. Thus, there is a need for systematic and automated techniques for program comprehension [76].

Dynamic analysis, which collects and utilizes data (traces) from program execution [34], is a popular technique for program comprehension. However, due to the amount of information obtained during the execution, the traces tend to become complex and overwhelming and thus difficult to understand [33, 141, 142]. Existing techniques target this problem, e.g., by summarizing traces [55], structuring and

visualizing collected data [7, 48], or inferring system specifications [100]. However, the first technique loses some of the data that may still be valuable, and the rest become overwhelming for developers and are not flexible to small variations of data. The problem can also be approached by finding patterns in the execution. However, prior work in the area has predominantly focused on generic and predefined design patterns, low-level architectural relations between program artifacts, or visualizations of all details of execution [7, 21, 60, 73]. While useful, these approaches do not capture the behavioural patterns that are not defined or even known prior to analysis, but form and recur (with small variations) throughout the execution of a program. Even in more traditional programming languages, patterns in execution do not repeat exactly in the same manner or same sequence. Further, presence of programming languages features such as dynamism, asynchrony and non-determinism in the execution makes the analysis more problematic and burdensome, and renders conventional techniques ineffective. Hence, program comprehension through dynamic analysis still remains challenging [34].

In this paper, we propose a novel technique for program comprehension by inferring a model of execution *motifs*. Motifs are abstract and flexible recurring patterns in program execution that serve a specific purpose in the functionality of the program. The term is inspired by *sequence motifs*, which are recurring patterns in DNA sequences that have a biological function [40]. Our approach discovers motifs from traces containing function executions and events. Our proposed algorithm compares a trace obtained from an interaction session against a database of previously-collected traces. It iteratively examines segments of traces for detecting sequences of function executions that may recur in execution. It is tolerant of small variations in different manifestations of each motif, allowing abstraction in inferred motifs. The algorithm discovers hierarchies between motifs as they emerge from details of execution. The hierarchical structure of inferred motifs reveals how higher-level key points of execution are formed. It allows users to have an overview of the trace, while still having access to all execution details as well as all intermediate levels.

The main contributions of our work are as follows:

- We propose an automated approach for inferring a model of program be-

haviour, which encompasses hierarchies of abstract recurring motifs extracted from execution traces. Our approach is inspired by techniques from bioinformatics, where similar challenges arise in investigating similarities in large sequences of DNA. The motifs facilitate program comprehension by highlighting the main characteristics of behaviour, and abstracting the details and variations of execution.

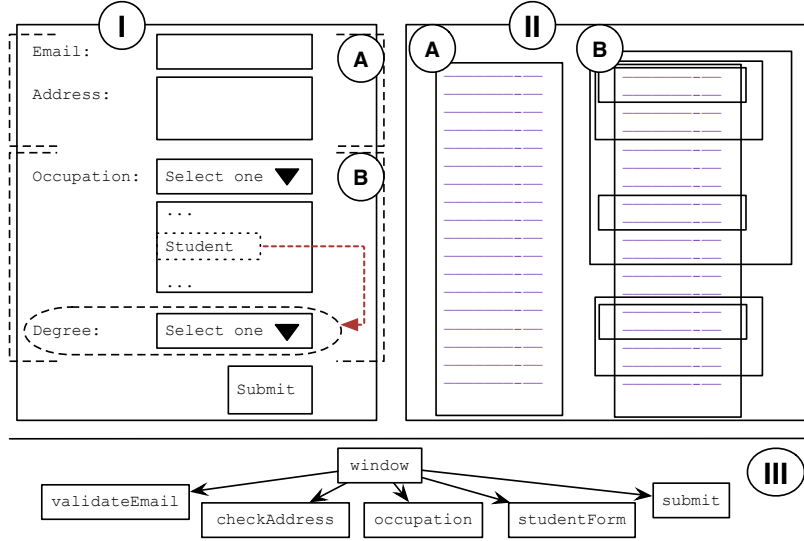
- We design and build a visualization technique for presenting the motifs to developers, to provide assistance with program comprehension. Our method is complementary to existing tools and techniques, and is designed to be utilized alongside existing programming environments.
- We implement our approach in a tool called SABALAN that supports JavaScript-based web applications. Our tool is non intrusive for generating traces, and infers models of recurring motifs from execution trace in an automated manner.
- We evaluate our approach through a controlled experiment conducted with 14 participants, on a set of real-world program comprehension tasks. The results show that using SABALAN helps developers perform program comprehension tasks 54% more accurately than other tools.

## 5.2 Challenges and Motivation

To assist the process of *searching*, *relating* and *collecting* information, many techniques collect execution traces, analyze them, and/or visualize the results for the developers. Despite providing the grounds for precise analyses, dynamic traces become very large and cause information overload. Further, they become very complex due to dynamism, asynchrony and non-determinism in program execution. These challenges render large traces ineffective in assisting program understanding.

In this section, we use a simple example to illustrate these challenges (Figures 5.1–5.3). We selected JavaScript for the examples since it is the lingua franca of web development. It is voted as the most popular language [126] and is the most used language on GitHub [74]. JavaScript applications are highly dynamic, asynchronous and event driven, and heavily interact with the Document Object Model (DOM) and the server code. These features can help demonstrate trace complexity within small





**Figure 5.1: I: A sample registration form. II: A) Sample execution trace, and B) hierarchy of inferred motifs. III: Dynamic call graph of example**

code segments. While our approach is general, we use JavaScript in this paper to demonstrate it.

**Overload by Information in Large Traces.** The amount of information a trace carries matters due to the cognitive overload understanding the trace imposes on developers [35], e.g., a study found that one GB of trace data was generated for every two seconds of executed C/C++ code [113]. For modern applications, which are often distributed among many nodes with many components involved, the traces become incomprehensible for developers very quickly. Some techniques try to address the problem by reducing the trace during/after its collection [19, 55, 113] by focusing on more important entities, or filtering the details of the executions. These techniques have been able to make traces more useful by decreasing the information contained in the traces [61]. However, even with a technique that creates a smaller trace, the trace is still not necessarily understandable for developers, as some of the data might be lost or missed by developers.

**Complex and Hidden Dependencies.** Revealing abstract and higher-level patterns that highlight the key points of a program’s behaviour can facilitate comprehen-

```

1 <form>
2   Email: <input type="email" id="email">
3   Address: <input type="text" class="addr">
4   Occupation: <div class="dropdown" id="occupation">
5     <button class="dropbtn">
6       Choose one</button>
7     <div class="dropdown-content">
8       <a href="#">Academic</a>
9       <a href="#">Industry</a>
10    </div>
11  </div>
12  <input type="submit" value="submit">Submit</input>
13 </form>

```

**Figure 5.2: Initial DOM state of the running example.**

```

1 $("#email").addEventListener("change", validateEmail, false);
2 $(".addr").click(checkAddress);
3 $(".dropdown-content").addEventListener("change", occupation, false);
4 function validateEmail () {
5   // do stuff
6 }
7 function checkAddress() {
8   // do more stuff
9 }

```

**Figure 5.3: [Partial] JavaScript code of the running example.**

sion. The focus of the developer can be guided through a hierarchy of recurring patterns of execution, while all collected information are still preserved for further inquiry. However, extracting such patterns (motifs) is challenging due to dynamism, asynchrony and non-determinism in program execution.

First, there are many complex and hidden dependencies between entities in the system, that can affect the execution. Understanding the impact of a user action or asynchronous communication with the server are examples of relations that are difficult, if not impossible, to capture from merely analyzing the code or the call graph. They act as media for connecting segments of execution together, that otherwise would not be related in the code itself. Further, for a part of behaviour to be distinguished as a motif, it should recur during the execution. Different executions of what is conceptually the same motif, may vary in details and thus may not converge to reveal the same motif. The alterations are intensified when programs have user interfaces, are distributed, or involve general dynamism and asynchrony.

However, the caused variations should not prevent the analysis to recognize the high-level blueprint of the behavioural motif they all manifest. An analysis that is overly dependent on all execution details cannot permit higher-level motifs to reveal themselves.

*Example.* Consider the example shown in Figure 5.1, showing a part of a form required for registering a user. Specific events on the input fields of the form have handlers that validate the input before the form can be submitted. `verifyEmail()` and `checkAddress()` (lines 1–2 of Figure 5.3) are handlers for `email` and `address` fields of part (A) of the form (lines 2–3 of Figure 5.2). The two functions are always executed together in a successful registration scenario, and are a consistent part of the motif representing that scenario, due to their placement in the DOM. However, this relation cannot be inferred from the code (Figure 5.3) or the call graph (Figure 5.1.III).

Moreover, a successful submission requires filling all the fields properly and submitting. However, the form change in section (B) of Figure 5.1 is based on the input of `occupation` (lines 4–11 of Figure 5.2). If the user chooses `Student`, a drop-down menu appears and the appearance, content, and functionality of the form changes a bit based on user’s input. However, the conceptual purpose of submitting the form, and hence the motif, remain the same. Should an analysis be too dependent on exact execution details, these two executions will be considered different. However, a more representative analysis should recognize that regardless of occupation of the user, the essence of the motif is the same and it should support both options. There are no prior knowledge or templates of the application-specific motifs. Hence, a useful comprehension method should allow a degree of flexibility in inferring motifs, to allow abstract motifs to form independently from unimportant contextual details.

### 5.3 Overview of the Methodology

For execution traces, such as the one depicted in Figure 5.1.II.A, our goal is to infer a hierarchy of its recurring motifs, by utilizing the knowledge of previous executions of the application. The model of extracted motifs assists comprehension of the program behaviour by facilitating the cycle of *searching*, *relating*, and

*collecting* information. Having our proposed approach, developers are able to gain an overall understanding of the highlights of execution, manifested as motifs, at a glance. Further, they would have the means to understand the details of such motifs, their hierarchies, and relations upon inquiry (Figure 5.1.II.B). Our approach takes advantage of precision of dynamic analysis, but prevents developers from being trapped and overwhelmed by the execution details.

Our proposed approach first instruments and intercepts the application on the fly, to obtain traces. Having a knowledge-base of previously collected traces and a query trace, our algorithm then extracts motifs of different lengths within traces, and infers hierarchies and relations of motifs. Our algorithm is inspired by bioinformatics algorithms for aligning biological sequences. Our approach creates a behavioural model from the motifs and their relations, which we visualize for developers in the final step.

**Execution Traces.** To obtain the traces required for the algorithm, we *instrument* the applications and collect dynamic execution information automatically. Our instrumentation allows our technique to *intercept* all function executions and collect their context-sensitive information. It also intercepts all events that can occur during the execution. Their added knowledge can assist the algorithm to infer conclusions about motifs and their causal and temporal relations with (a)synchronous events. Next, our approach eliminates low-level details included in the raw trace, such as auxiliary events, low-level and library method calls, and framework-specific details. The pruned trace is then used as the input to the algorithm. Note that our method is non intrusive, and preserves the original behaviour of the application under investigation.

## 5.4 Algorithm for Inferring Motifs

In this section, we propose our algorithm for detecting motifs in order to create a higher-level model of behaviour. We define motifs as abstract and hierarchical sequences of function executions that recur throughout the lifetime of an application. While each motif eventually supports concrete sequences of function executions, it is by nature a composite element, and can represent more complicated structures. We define a motif as an ordered set of two or more members ( $m_0$  to  $m_i$ ), which

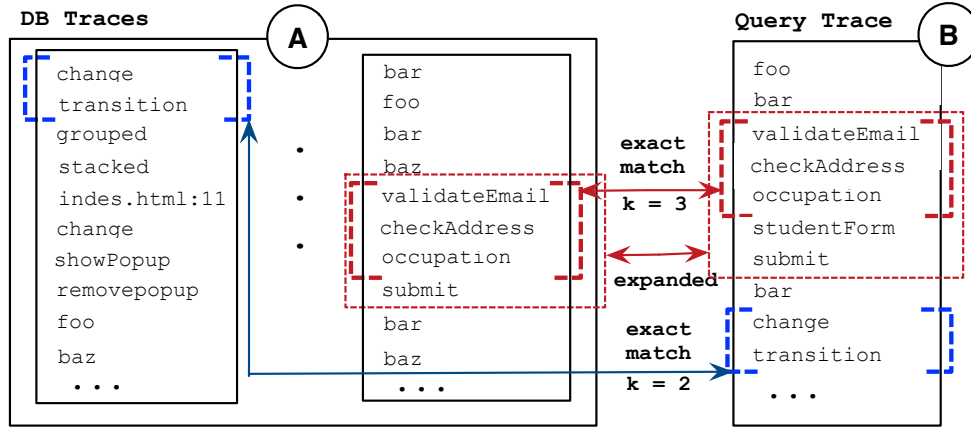
include [sub-]motifs, abstract entities, and context-sensitive function executions. The confidence of a motif in each of its members is a representative of the manner that member is observed within different executions of the motif, and is shown as  $c_0$  to  $c_i$  for all motif's members, respectively.

$$M = \{\langle m_0, c_0 \rangle \dots \langle m_i, c_i \rangle\}_{i=1}^{\infty} \quad , \quad \begin{array}{l} m_i \quad ::= \quad \text{function execution} \\ | \quad \text{sub-motif} \\ | \quad \text{abstract entity} \end{array}$$

Our approach draws the attention of developers to the main observed motifs, presumed to represent highlights of behaviour, preventing their view to be obstructed by low-level details. The underlying model still preserves details that can be demanded by users as necessary.

#### 5.4.1 Inspiration from Analyzing Biological Sequences

In designing the algorithm, we were inspired by bioinformatics, where there is a constant need to explore, compare, and analyze large data sequences. Most relevant to our approach are *sequence alignment* algorithms, which find similarities in sequences of DNA, RNA, and protein by arranging and comparing them [52]. We begin our core algorithm by using a heuristic for finding exact matches between trace sequences. For comparison of the trace sequences, we adapt the idea of BLAST (Basic Local Alignment Search Tool) [9], a *local sequence alignment* algorithm which we modify to fit the domain of execution traces. We then expand the matches by maximizing similarities in their neighbouring entities in the traces. This phase is accomplished using a dynamic programming algorithm in order to allow more flexible and more abstract motifs. Throughout this process, our method infers existing motifs and reveals their relations and hierarchies. In this section, we use Algorithms algorithm 2 – algorithm 3 and Figures 5.4 – 5.5 to explain our algorithm. Our algorithm takes the application (`app`) and its knowledge-base ( $\Sigma\delta b$ ) as input and returns the extracted motifs as output. Please note that we have eliminated and/or merged many details for the sake of brevity. The details can be found in the repository of our open-source tool [118].



**Figure 5.4:** This figure depicts a DB of traces (A) and a sample query trace (B) of an application, on the left and right side, respectively. Exact matches of length 2 and 3 between the query trace and different DB traces are marked.

### 5.4.2 Forming a Knowledge Base

Our algorithm requires a knowledge base of multiple previous executions, i.e., a set of traces, named *database (DB) traces* ( $\Sigma\text{Edb}$ ), which can initially be collected by executing the test suite, crawling, or exploratory testing and exercising the application multiple times. During each interaction session, our approach collects a trace, called the *query trace* ( $\Sigma q$ ), which will be analyzed and compared against all DB traces for finding its motifs. Each query trace is itself added to the DB traces after the algorithm is finished. This part is depicted in lines 21–23 of Algorithm algorithm 2. Parts A and B of Figure 5.4 display sample DB traces and query trace of the running example (Figure 5.3).

### 5.4.3 Finding Exact Matches

Next, the algorithm finds all the exact matches of length  $k$  between the query trace and the DB traces. We start by matches of length 2 (function pairs). We then increment the length of exact matches iteratively and repeat the search at each iteration, until we have found all exact matches. Two sets of exact matches of length 2 and 3 are shown in Figure 5.4, between 2 DB traces (part A) and the query trace (part B). Lines 25–26 of algorithm 2 iterate over the query trace for finding matches

---

**Algorithm 2** Finding exact matches and expanding them

---

**input** :  $app, \Sigma db$ **output** : motifs**Procedure** EXTRACTPATTERNS() **begin**

```
21   modifiedApp  $\leftarrow$  INSTRUMENT( $app$ )
22   rawTrace  $\leftarrow$  INTERCEPT(modifiedApp)
23    $\Sigma q \leftarrow$  PRUNE(rawTrace)
24    $k \leftarrow 2$ 
25   for  $i \leftarrow k; i \leq \Sigma q.length; i++$  do
26       for  $j \leftarrow 0; j \leq \Sigma q.length - k; j++$  do
27            $subQ \leftarrow$  EXTRACTSUBTRACE( $k, \Sigma q, i, j$ )
28           matches  $\leftarrow$  EXACTDBMATCHES( $\Sigma \Sigma db, i, subQ$ )
29           for  $m \leftarrow 0; matches.length; m++$  do
30               for  $n \leftarrow 0; n < matches[m].length; n++$  do
31                    $dir \leftarrow$  INITIALEXPANSIONDIRECTION()
32                    $\Sigma I \leftarrow [qI_{start} \quad qI_{end} \quad dbI_{start} \quad dbI_{end}]$ 
33                   if EXPANDABLE( $\Sigma q, \Sigma \Sigma db[m], \Sigma I$ ) then
34                        $subDb \leftarrow matches[m][n]$ 
35                       expandedQ  $\leftarrow \Sigma q.EXPAND(sub_q, \Sigma I_q, dir)$ 
36                       expandedDb  $\leftarrow \Sigma \Sigma db[m].EXPAND(sub_{db}, \Sigma I_{db}, dir)$ 
37                        $dir \leftarrow$  DIREXPANSION(toggle: true,  $\Sigma \Sigma db[m], \Sigma q, \Sigma I$ )
38                        $k.INCREMENT()$ 
39                       MOTIF  $\leftarrow$  COMPARE(expandedQ, expandedDb,  $k$ )
40                       matches.PUSH(MOTIF,  $k$ )
41                       motifs.ADD(MOTIF)
42                        $\Sigma I \leftarrow$  ADJUSTEXPANSIONINDICES( $dir$ )
43
44
45
46
47
```

---

of length  $k$ , and increment  $k$  at each iteration. Lines 27–28 show that subsequences of length  $k$  are extracted from the query trace at each iteration, and are compared against all  $k$ -length subsequences of all DB traces to find matches.

#### 5.4.4 Allowing Abstraction in Motifs

In the next step, we expand each match to progress towards finding flexible and abstract motifs, that are tolerant of small alterations. This technique decreases dependency on specific execution details, provides a higher-level overview of the

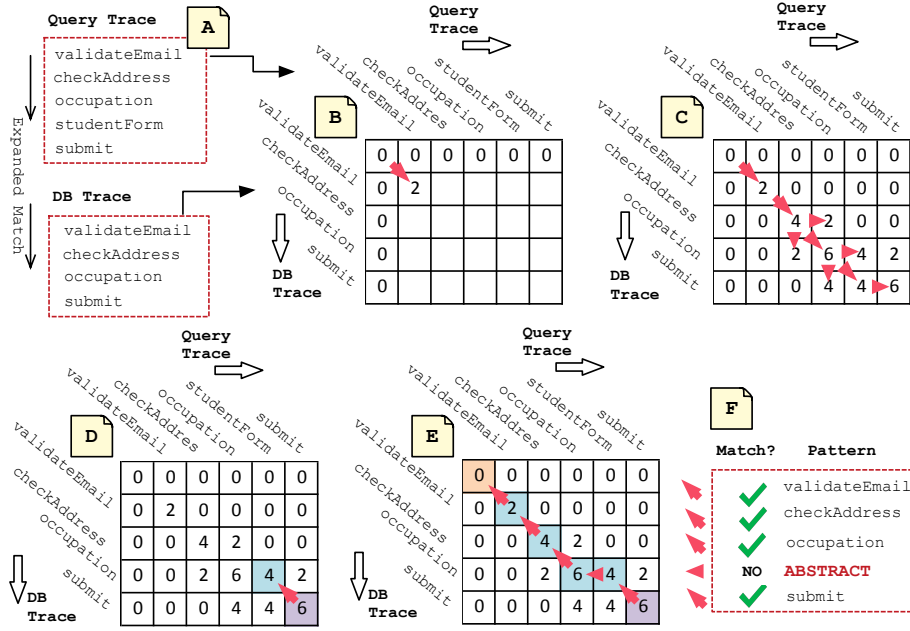
semantics of the application, and permits flexible motifs of variable length that may include gaps.

At this stage, our algorithm iteratively performs the following steps. **First**, it selects two matches from existing matches, which were determined as the result of previous step (Figure 5.4). Then, it iteratively expands the query and DB matches from both directions, while gradually incrementing the length of the motif under investigation (Lines 29–45 of Algorithm algorithm 2). Figure 5.4 also shows an expanded match of two [partially] different sequences, for measuring their similarities (Figure 5.4. A). This phase continues until the accumulated penalty of gaps interrupts expansion of the motif.

*Next*, the algorithm finds a [sub-]sequence of those matches that have the maximum similarity. At this step, we adapt a dynamic programming algorithm called *Smith-Waterman* for finding patterns in two molecular sequences [123]. This algorithm quantifies the sequence alignment process by assigning scores for matches and mismatches, and penalties for gaps. Aligned sequences are then found by searching for the highest scores in scoring matrices. To adapt this algorithm to our domain and compare similarities of two traces, we propose a similarity matrix that determines the similarity of two members in traces. The similarity of the traces are determined by a combination of similarities of all their members, based on a dynamic programming heuristic.

**Similarity Matrix.** We propose a similarity measure for quantifying the similarities between function executions in traces. Our comparison is based on two metrics, function names and parameters. We devise three scores for comparison: strong match, weak match, and mismatch (leading to a carryover penalty). If two functions match in terms of names and parameter numbers they are a strong match. The match is weak if only the names are equal (and not parameter numbers). The reason for considering parameter count as a separate metric is the nature of JavaScript, where a function call does not need to be faithful to the function signature in terms of arguments (known as function variadicity). Two function executions do not match if both metrics are different. The strong and weak matches are assigned positive scores, while the mismatch is assigned a negative score as it can represent gaps in the motif, which can accumulate and disrupt a motif. The base scores for matches and penalties are determined using empirical data.





**Figure 5.5:** This figure briefly depicts (A) the process of taking two expanded trace subsets, (B, C) forming a scoring matrix based on similarities between sub-traces, and (D, E) finding a match in manner that maximizes the similarities between sub-traces. The final motif can be seen in (F).

Moreover, our matrix needs to support comparison of motifs, to accommodate another extension of the original algorithm, which permits hierarchies between motifs. The function execution members of a motif are compared as explained above. Should a motif contain an abstract node, then all valid executions of the abstract node should be compared with the other sequence. Throughout the process of comparing motifs, our algorithm infers hierarchies and abstractions of motifs should they exist, as explained below.

Then, we perform our adaptation of the Smith-Waterman algorithm on the two expanded sequences, as shown in line 48 of Algorithm algorithm 3, which creates a scoring mechanism for comparing the two sequences based on a one-by-one comparison of all their entities, based on our similarity matrix. The result is a scoring matrix of overall scores of comparing two sequences ( $M^{k+1,k+1}$ ). This

process is shown for the two sequences of the running example from the previous step in Figure 5.5.A–C. To find the common motif in these sequences, we find the sub-sequences that hold the highest collective similarity as a group. We start by finding the highest score in the matrix (line 49 of algorithm 3, Figure 5.5.D), and then trace the matrix back, determining the aligned motif at this stage (lines 50–58 & Figure 5.5.E). For navigating the motif back in the matrix, our dynamic programming algorithm chooses the maximum neighbouring score at each step (line 51). Based on the selected neighbour, the algorithm determines whether that motif member comes from one or both of sequences, and whether an abstract entity should be injected to show different alternatives of the motif. The motif’s confidence in that member is then updated based on how it is selected (lines 53–55 of Algorithm 3). The inferred motif of the running example (Figure 5.5.F) has five members, one of which is abstract. The abstract member was advised to enable to motif to support both sequence shown in Figure 5.5.A, which serve the same purpose (registration), but are executed in a slightly different manner. The abstract member demonstrates that observing the `studentForm` function in the motif is arbitrary, and the motif can either be observed with this function and five total members, or with only four members which do not include said function.

### 5.4.5 Inferring Hierarchies of Motifs

In the next step, we devise another extension to the original algorithm, which enables us to infer and reveal hierarchical relations between motifs. By definition, our motifs are composite entities, which can contain other motifs as their members. During the analysis, our algorithm may encounters cases where (1) the match that is being compared is a motif itself, and (2) the expansion leads to discovery of a new motif. In such cases, a hierarchical relation is added between the two motifs. Meaning not only the [sub-]motif was observed independently within the execution, it also contributes to the formation of the new and larger motif. Our analysis follows a bottom-up approach, starting with function executions as building blocks of the trace. It iteratively works the way up to higher-level and more abstract motifs that allow flexibility in execution. At each iteration of the algorithm, new motifs can be revealed which may have hierarchical relations with existing motifs. As such motifs

---

**Algorithm 3** Inferring a motif

---

**input** :  $S_1, S_2, k$   
**output** : motif

**Procedure** EXPANDPATTERNS() **begin**

```

48    $M^{k+1,k+1} \leftarrow \text{SMITHWATERMAN}(S_1, S - 2, k)$ 
49    $\langle i_{max}, j_{max} \rangle \leftarrow \text{MAXSCORELOCATION}(M^{k+1,k+1})$ 

50   while  $i_{max} > 0$  AND  $j_{max} > 0$  do
51      $dir \leftarrow \text{MAXNEIGHBOUR}(i_{max}, j_{max})$ 
52     if  $dir == \text{DIAG}$  then
53        $\text{MOTIF.ININSERT}(\text{ABSTRACT}(S_1[i_{max}], S_2[j_{max}]))$ 
55     else if then
56        $\text{motif.insert}(\text{FUNCTION}(S_1[i_{max}], S_2[j_{max}]))$ 
57      $\langle i_{max}, j_{max} \rangle \leftarrow \text{BACKTRACK}(M^{k+1,k+1}, S_1, S_2, dir)$ 
58   end while

59   if  $S_2.type == \text{MOTIF}$  then
60      $\text{BUILDHIERARCHY}(S_2, \text{MOTIF})$ 
61   end if

62   return MOTIF

```

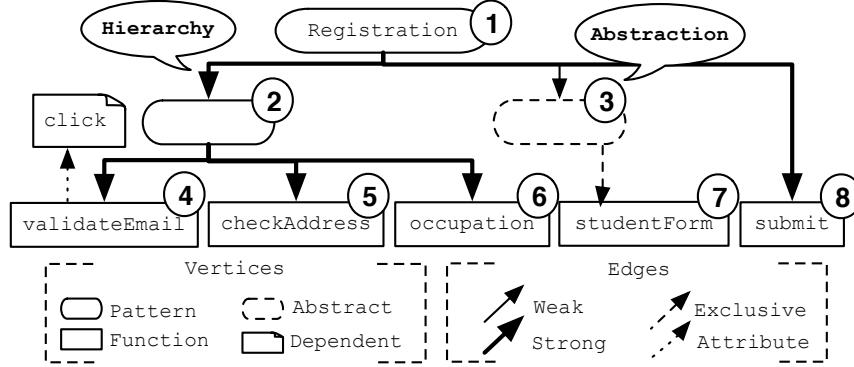
---

emerge, our algorithm captures the process of their formation and hierarchies in a model, as explained in the next section.

In the running example, we first find an exact match with  $k = 3$ , which is a motif itself, but with no abstraction (Figure 5.4). Later, during expansion, we find that this motif is a member of a larger motif (Figure 5.5.F) with two other members (an abstract member and a function execution). The algorithm creates a hierarchy (line 59 of algorithm 3), which then manifests in the model (Figure 5.6), as an edge from the new abstract motif (node 1) to the sub-motif (node 2).

## 5.5 Creating and Visualizing the Model

In this section, we explain our methodology for inferring the hierarchical model of behavioural motifs and visualizing it.



**Figure 5.6: Sample model of the running example.** The root node (1) is the highest-level inferred motif. Node 2 is a sub-motif of node (1), marked by the hierarchical edge between the two. Node 3 is abstract allowing variations of its child node to occur in the motif. The leaves of (nodes 4–8) are concrete function executions in the trace.

### 5.5.1 Creating the Motif Model

As mentioned above, during the process of extracting motifs, our approach infers the hierarchies and other potential relations between them. Such structural relations are preserved in a model, represented as a directed acyclic graph (DAG), which evolves as the algorithm proceeds, as explained below.

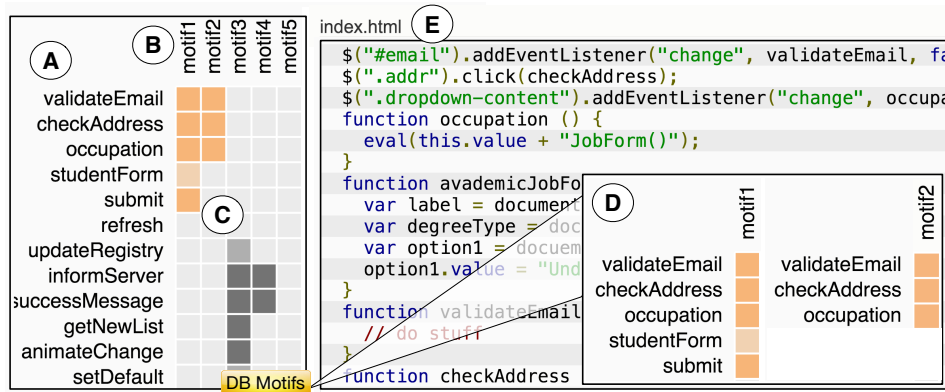
**Vertices.** Vertices of the graph can be functions, motifs, abstract entities, or dependent vertices. *Function* vertices are atomic nodes representing specific and context-sensitive function executions. *Motif* vertices are semantically composite classes. Each motif conceptually contains an ordered set of its members. *Abstract* vertices are the model’s means for supporting flexibility in motifs. Should there be alterations in different observations of a motif, an abstract node is used for accommodating all valid cases. *Dependent* vertices hold additional attributes of other types of vertices. and exist only to provide more information about other vertices. e.g., an event contributing to a function execution is shown as a dependent vertex.

**Edges.** The vertices of the graph are connected through directed and ordered edges. The edges are responsible for connecting motifs to their members. The *direction* of an edge is from a motif node to its members, which are *ordered* based

on the time they were observed in traces. The edges also represent the *confidence* of the algorithm in the respective member (strong or weak), based on the manner of observation of the member. Edges may contain other *special attributes*, depending on its type and the purpose it serves in the model. For instance, the “edge exclusion” property is used to show that only one of the variations of an abstract node is valid at a given time.

Figure 5.6 represents the model of the running example of Figure 5.1. The root of the DAG (node 1) is a motif representing registration. It consists of three members: a sub-motif (2), an abstract node (3), and a function execution (8). The first member is a motif itself, which contains a sequence of three functions from the trace, marked 4–6. The *strong* edge to the sub-motif (and from there to its children) shows the high confidence of the algorithm in the sub-motif. Node 3 is an abstract node, acting as a place holder for valid versions of the node, manifested in its children. This node exists due to the variation in two observations of the motif (Figure 5.5.A). In the case of our example, the *exclusive* type of the child (7) edge demonstrates that occurrence of this node is optional in the motif (`studentForm` is observed or not). Further, the *weak* edge type displayed the algorithm low confidence in this node. Node 8, the final member of 1, is an execution of function `submit`. All leaves of the DAG are concrete function executions from the trace. Nodes at higher levels of a graph involve abstractions and hierarchies, to represent the incremental process of emergence of motifs from details of trace.

**Motif Relations.** The hierarchies that form between motifs are one type of relations that are preserved through the model. The algorithm also discovers other types of relations between motifs, such as *temporal* or *causal* relations. The integration of all these relations depicts how semantics of the program are shaped from bottom (small specific motifs) to the top (larger and more abstract motifs representing key points of behaviour). The motifs may be semantically related in manners that are not quite obvious from the code. For instance, motif `m1` may *cause* motif `m2`, or they may be ordered (but not dependent) due to the design and the architecture of the system. Querying the model allows us to reveal patterns of motifs themselves and even discover patterns that are not known by the developer, are created unintentionally, or are imposed on the system by other factors such as third-party frameworks.



**Figure 5.7: A [modified] screenshot of visualization. (A): Query trace. (B): Inferred motifs depicted on the table. (C): Motif hierarchies. (D): All motifs. (E): Code panel displaying selected function/motif code.**

### 5.5.2 Visualizing the Model

Finally, we visualize the motifs to further assist program comprehension by taking advantage of information visualization techniques. Our web-based visualization provides two main views for displaying (1) the motifs recorded in a specific query trace, and (2) all motifs discovered in the behaviour (DB traces).

**Trace Motifs.** To allow developers to focus only on a part of behaviour that is of interest to them, this view displays motifs that are found within the query trace, freshly recorded from an interaction session (Figure 5.7, A and B). Section (A) of the figure displays the pruned query trace, where time proceeds from top to bottom. Section (B) displays the motifs, distinguished by colour and index. The saturation of each cell of a motif displays the motif's confidence in that member. Each motif may recur multiple times in the same trace, or may contain hierarchies of motifs (Figure 5.7C).

**All Motifs.** The second view is meant to provide a global overview of application behaviour by displaying all its motifs, extracted from all DB traces (Figure 5.7D). These motifs may display system usecases, feature implementations, or other higher-level sequences that somehow describe the functionality of the system. They do not conform to a single trace, and thus each motif has its own separate

trace. In both views, hovering the mouse over each entity displays more information regarding that entity in the tooltip. Clicking on an entity displays its respective code (function or motif) on the code panel (Figure 5.7E). Displaying only the code relevant to the motif, allows developers to only focus on their specific task, without the added burden of understanding the whole application.

## 5.6 Implementation: SABALAN

We implemented our approach in an open-source tool, called SABALAN. The entire tool is implemented in JavaScript. We create our own Express.js server for implementing the algorithm, the instrumentation unit, and the visualization. We develop the bioinformatics-inspired algorithms from scratch for execution traces. We use a proxy to automatically inspect applications [63]. For instrumenting the code, we create an AST of the code, modify it, and serialize it back into JavaScript [43–45]. SABALAN is publicly available [118].

## 5.7 Evaluation

We empirically evaluate our approach by investigating the characteristics of the extracted motifs, as well as the usefulness of our approach for developers and its overhead through the following research questions.

**RQ5.1.** What are the characteristics of typical motifs inferred by SABALAN from execution traces?

**RQ5.2.** Does using SABALAN improve developers’ performance for common comprehension tasks?

### 5.7.1 Motif Characteristics

To address RQ5.1, we performed our analysis on seven JavaScript applications, listed in Table 5.1.

**Design.** We selected seven open-source JavaScript applications from GitHub (Table 5.1). These applications cover various software domains and were selected based on their popularity and usage. Based on each application’s specifications, we selected a method for collecting its traces (running the test suite or designing

scenarios for exploratory testing). We provided the traces (DB and query) as input to our approach, and analyzed their extracted motifs and investigated their main characteristics. We measured the number of unique motifs inferred from all traces for each application, calculated their lengths, and analyzed their hierarchies. We registered the size of DB traces in terms of number of different traces as well as the size of a typical trace.

**Results and Discussion.** The results of the analysis are depicted in Table 5.1. The second column displays the number of lines of code for each application, while the third column contains the number of motifs our approach found in the applications. The next column represents the number of DB traces that were collected for each application. Column five, shows the average size of traces of each application, collected by SABALAN in one-minute interaction sessions. Note that our tool performs a level of filtering while logging execution details. Column six shows the average trace size collected by Google Chrome’s JavaScript profiling and Timeline. It can be seen that the average trace size using SABALAN is 77 KBs, while without SABALAN there is an average of 96 MBs of data for the same interaction session. These values emphasize the extent of the information contain in the raw traces, even for modest-sized applications, which make them challenging for developers to analyze. However, using our approach, developers have an average of 8 recurring high-level motifs for each interaction session, each with an average length of 4 (columns 7–9), to guide them through the understanding the behaviour. The last column displays the number of unique hierarchical relations between unique inferred motifs. The numbers show the existence of hierarchies of motifs. Further assessment of the structures of model graphs depict the bottom-up formation of higher-level key points of behaviour based on smaller motifs through such hierarchies.

There are a few cases in the results where the algorithm was not able to find many (meaningful) motifs, or any hierarchies. Upon further investigation, we found that these applications rely heavily on external and graphic libraries, which were disabled in our analysis. These features can be activated in future if needed.

An important factor that can impact the efficacy of the algorithm is the *requirements of the DB traces*. The number of initial traces in the knowledge base, their coverage of the application’s functionality, and their similarities (or differences), are factors that can impact the quantity and quality of the final motifs. We aimed to



**Table 5.1: Characteristics of traces and inferred motifs**

Application						Motif Length			
	LOC #	# of	# of	Trace	Raw	Avg	Min	Max	#
		of	DB	size	Trace				of
		M.		(KB)	(MB)				unq
									H.
Phormer	6000	13	20	84	86	4	2	11	4
same-game	229	4	7	255	143	3	2	4	0
simple-cart	9238	4	19	45	67	4	2	8	3
browserQuest	36206	17	15	67	125	5	3	9	2
adarkroom	15612	6	15	41	40	4	2	6	2
doctored.js	3534	4	10	16	102	3	2	5	1
hextrix	5154	7	16	30	110	4	2	6	2
Average	10853	8	14.5	77	96	4	2	7	2

maximize the features we covered with the DB traces. We stopped collecting new DB traces when we observed that adding a trace did not affect the inferred motifs (average of 14.5 DB traces per application).

### 5.7.2 Controlled Experiment

Next, we conducted a controlled experiment to assess the effectiveness of our technique for developers in practice and address RQ5.2. We divided the participants into control and experimental groups. The experimental group used our approach, while the control group used the tool of their choice. The participants accomplished a set of comprehension tasks, and their performance was measured. The tasks were designed based on common software comprehension activities [102]. We defined the performance of a developer by the combination of time and accuracy of completing the tasks. Our hypothesis was that using our approach would enhance developers' performance in understanding the overall behaviour, main usecases, and recurring motifs of a web application.

#### Experiment Planning

The goal of our experiment is to investigate the following research questions.

**RQ5.2.1.** Does using SABALAN decrease task completion *duration* for common comprehension tasks?

**RQ5.2.2.** Does using SABALAN increase task completion *accuracy* for common

comprehension tasks?

**RQ5.2.3.** Is SABALAN better suited for certain types of comprehension tasks?

**Variable Selection.** Our design involved one *independent variable (IV)*, the variable we controlled, which was the type of tool used in the experiment, i.e., a nominal variable with two levels. We refer to the first level as SABALAN, since they had to use our tool. The second level represented usage of other tools, which we refer to as OTHER. Our goal was to measure developers’ performance in completing the tasks. Since performance is not measurable, we quantified it using two variables, namely task completion *duration* and *accuracy* (both continuous), which were our *dependent variables (DV)*.

**Selection of Object.** We chose Phormer photo gallery application as our object [106], which has about 6,000 lines of code and over 43,000 downloads. It is an open-source PHP-based application that allows users to store photos, categorize and rate them, view them as a slideshow. Since we had allocated limited time for each session, we had to choose an application that is simple, and yet exhibits realistic motifs in its behaviour - these criteria are met by Phormer.

**Selection of Subjects.** We recruited 14 participants for the experiment. They were all graduate students in computer science and engineering, and many of them had professional software development experience. The participants consisted of 2 female and 12 male participants, aged between 23 and 35, and they volunteered for the experiment. Knowledge of programming and familiarity with web development (and particularly JavaScript) were our only requirements for picking the participants. Overall, our participants had 1–10 years of web development and 1–18 years of software development experience, respectively.

**Experimental Design.** Our experiment had a “between-subject” design. To avoid the carryover effect, we divided our participants into two groups. The *experimental* group were given access to SABALAN for performing the tasks, while the *control* group used Google Chrome’s Developer Tools for completing the session. All our participants were familiar with DevTools according to their answers to the pre-questionnaire form and chose to use it during the experiment. No member of the experimental group were familiar with SABALAN prior to the study session. To avoid bias in favour of one of the groups in terms of their proficiency levels, we

**Table 5.2: Comprehension tasks used in the study.**

Task	Description	Activity
T1.A	Understanding all common usecases	A1, A7, A9
T1.B	Determining the most used scenarios	A6, A7
T2.A	Locating the implementation of a feature for reuse	A1, A3
T2.B	Estimating the quality of said implementation	A4, A5, A8
T3	Understanding the addition of a new feature	A1, A2, A3

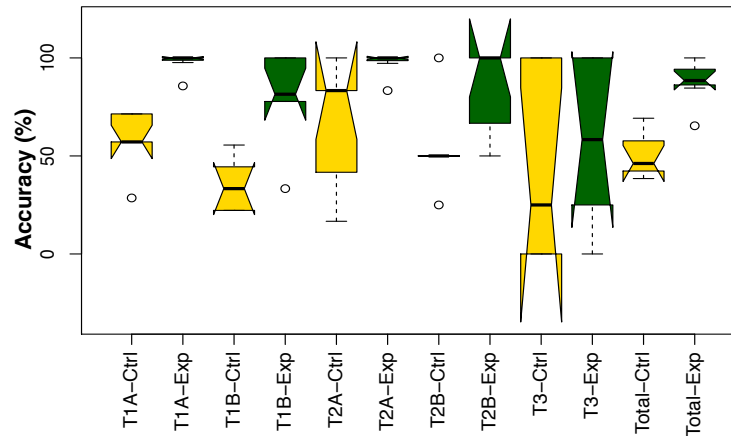
collected historical data about our participants prior to scheduling the sessions. We assigned each participant a proficiency score, based on a combination of metrics, including their experience with software development, knowledge of JavaScript, and how they perceived their own expertise. We balanced the proficiency levels in both experimental and control groups.

**Experiment Tasks.** We designed five comprehension tasks, as outlined in Table 5.2. The design of the tasks was based on common program comprehension activities, proposed by Pacione et al. [102]. As the name suggests, these activities represent fine-grained activities that developers need to perform for understanding software, regardless of the language and the platform used. Table 5.2 shows how each of our tasks covers one or more activities - all activities are covered in our design. Moreover, each task also included a mini questionnaire, which asked about how participants perceived the difficulty of the task, the required time, and the required expertise level for accomplishing the task. We have made all the tasks and datasets publicly available [118].

### Experimental Procedure

The procedure of the experimental sessions consisted of three main phases.

- **Pre-study.** We required all participant to fill a pre-questionnaire form to gather some demographic data about them. Further, we used the data regarding their experience, programming habits, and self-perceived expertise level, to assign participants expertise scores. The score allowed us to fairly balance the expertise levels in both experimental and control groups.
- **Training.** At this step, the experimental group were given a tutorial on SABALAN, which they were encountering for the first time. Then both



**Figure 5.8: Notched box plots of accuracy results. Green plots display experimental (SABALAN) group, and gold plots display the control group. Higher values are better.**

groups were given some time to familiarize themselves with the setting of the experiment. We then started the tasks when the participants were ready.

- **Tasks.** During this phase, the participants completed the five comprehension tasks summarized in Table 5.2. Based on our design, we wanted to measure both *duration* and *accuracy* of completing the tasks. To measure time, we prepared each task on a separate sheet of paper. We started a timer when we handed a task sheet to a participant, and asked her to return it to us (with the answer) as soon as she had completed the task, which is when we stopped the timer. This allowed us to record the time they spent on each task. We evaluated the accuracy of each task later, based on rubrics we had prepared prior to conducting the experiment.

Moreover, we wanted to gather some data regarding how the participants perceived the tasks. Thus, we provided them with a set of meta tasks, that questioned them about the perceived difficulty, time-consumption, and required expertise level for each task. Finally, the participants filled a post-questionnaire form regarding their experience in the study.

## Results

We first ran the Shapiro-Wilk test on all collected data sets, to determine if they were normally distributed. For normally distributed data of accuracy we used *two-sample t-tests*. The duration data did not pass the normality test and thus we used the *Mann-Whitney U test*.

For the accuracy, the results of running the tests showed a significant difference, with a high confidence, for the experimental group using SABALAN (Mean=87.8%, STDDDev=11.6%), compared to the control group using Chrome DevTools (Mean=50.5%, STDDDev=11.6%); ( $p - value = 6.2e - 05$ ). The accuracy results are shown in Figure 5.8. *Overall, using SABALAN increased developers' accuracy in performing comprehension tasks by an average of 54%, over other tools.* (RQ5.2.1). We further analyzed the impact of using SABALAN in accuracy of individual tasks. The results of running the statistical tests showed significant difference in favour of SABALAN for all tasks, except T3. The accuracy of results of T1A through T2B were significantly higher using SABALAN. The results for task T3, although not statistically significant, were on average 23% more accurate when participants used SABALAN.

For the times, the collected task completion duration data were comparable for participants of both experimental and control groups. Running the tests did not reveal any statistically significant difference in task duration between the two groups (RQ5.2.2).

Finally, we analyzed the collected data from the questionnaire form participants filled regarding each task. They perceived the difficulty of tasks from 2.15 to 2.54, based on a 5-point Likert scale, which shows an average level of difficulty for all tasks. We compared the difficulty of each task as perceived by participants with the results of duration and accuracy of the same task. We found no correlation between perceived difficulty of a task by participants and how they actually perform the task based on the Pearson correlation coefficient.

### 5.7.3 Discussion

The results of the experiment revealed that SABALAN improves developers' performance in comprehension by significantly increasing their accuracy by 54%

(RQ5.2.1). The results however did not show a significant difference for duration (RQ5.2.2).

**Domain Knowledge and System Use-cases.** One of the first steps towards program understanding is general understanding of its domain and overall dynamic behaviour by identifying the components that provide a solution to the domain. Our results show that SABALAN significantly increases the accuracy of such tasks (T1). This task consisted of two main parts, understanding the overall behaviour and use-cases of the experimental object (T1.A), and deciding on their importance (T2.B). Using SABALAN significantly improved the accuracy of these two tasks by 49% and 78%, respectively. The results show that using SABALAN not only provides a more accurate overview of an application's behaviour compared to ad-hoc approaches, but also helps developers obtain a better understanding of the importance and usage of main system components and their interactions (RQ5.2.3).

**Feature Location.** Feature location is one of the main tasks performed during program comprehension, and has many applications, such as reuse and testing. Our results show that using SABALAN significantly improved the accuracy of feature location (RQ5.2.1, RQ5.2.3). The experimental group were able to find components involved in the implementation of a feature and infer their relations 42% more accurately than the control group (T2.A). They were also 42% more accurate in estimating the quality of the implementation of the said feature (T2.B). Investigating the answers revealed that the control group missed many connections in the code that lead to discovery of different parts of the implementation and thus failed to create a complete and accurate model of the involved code. Due to their incomplete understanding of the feature, the control group was not able to estimate and measure the *quality* of the respective part of application as well. The experimental group, however, could assess the quality based on the more accurate model of the behaviour that extracted the feature as a behavioural motifs (RQ5.2.3).

**Software Change and Root-Cause Detection.** The last task (T3) involved understanding the system in order to make a change, by finding the root cause of a particular observed behaviour. The experimental group were able to perform the task 23% more accurately with SABALAN, although the results were not statistically significant (RQ5.2.1). Using SABALAN, they were able to focus on a much smaller part of the code that was relevant to the feature that needed change. However, be-

cause we do not have debugger support within SABALAN, using common debugging techniques such as setting breakpoints and watching variables in such tasks required the participants to frequently switch between the visualization and the application. We believe that we could achieved statistical significance for task if we extend our research prototype or integrate it with a debugger such as Google Chrome's DevTools.

**Accuracy over Speed.** The results did not show any significant difference for task completion duration in favour of SABALAN. We believe this is not a significant issue due to three reasons. *First*, accuracy of performing a task is more important than its speed [7]. The significant improvement of task completion accuracy with SABALAN (54%), and the test's high confidence in the result, emphasize the challenges of comprehending traces, as well as the usefulness of SABALAN in improving developers' performance for completing said tasks. Investigating the answers further, we found that many participants in the control group had finished the tasks early, assuming they had the right answers. While in fact, they were not even aware that they are missing crucial parts of the answer, which resulted in them having lower accuracy than SABALAN users. *Next*, we believe that the unfamiliarity of our participants with SABALAN might have caused them to spend more time trying to use it. This theory is strengthened when we analyze individual tasks results. We observed that the experimental group had the worst speed ratio compared to the control group for the first task (T1.A), after which they quickly improve and surpass the control group in later tasks. *Finally*, dividing the locus of attention may have also played a role in the results. While the control group were only focusing on the browser, the SABALAN group had to switch back and forth between the application (browser) and the tool. We believe this problem can be solved by either extending the tool or integrating it into an existing programming environment.

**Participants' Perception of Tasks vs. Performance Reality.** There were no correlations between the difficulty of a task as perceived by participants, and their measured performance scores. All participants deemed all tasks to be of moderate difficulty. However, the control group scored significantly lower accuracy marks for all tasks. Considering the participants rated tasks after their completion, shows their interpretation of the task requirements did not match the reality of the task they had just performed. The result confirm the challenging nature of trace comprehension.

Due to difficulty of finding, relating, and collecting elements of execution with an ad-hoc approach, developers miss crucial elements of analysis, without even knowing that there is more to the task.

**Performance Overhead.** We used the experimental object of our user study, Phormer, to obtain data regarding the additional overhead of our approach, in 10 one-minute interaction sessions. We measured three sources of potential performance overhead into account. The overhead caused by *instrumentation* phase, the imposed overhead on *execution* of instrumented code and data collection, and the overhead of *analysis* of traces and motif extraction, which were respectively measured as 1.2, 0.1, and 2.1 seconds on average. This is negligible for all practical purposes, and is barely noticeable during the interaction with the application. Thus, the performance overhead of SABALAN was entirely acceptable for this application.

#### 5.7.4 Threats to Validity

The **external threats** of conducting an experiment such as ours, typically arise from representativeness of tasks, participants, and object selected for the experiment. We mitigated the threat of task selection by designing our tasks in a manner that covered all Pacione’s common comprehension activities [102], to show that the design of our tasks is not biased, and that they are representative of routine comprehension tasks. A valid concern is regarding the representativeness of the participants of the developer population, since we recruited students. We tried to address this concern by recruiting only graduate students who had prior experience with JavaScript, many of whom had experience working in industry. To address the threat of representativeness of the experimental object, we chose an open source JavaScript application Phormer, [106] with about 6,000 lines of code and over 43,000 downloads at the time of conducting the study. An **internal threat** that concerns our method is the bias towards assigning participants into control and experimental groups, or the population-selection problem, which we addressed by balancing the expertise levels between the two groups. Other threats can arise due to the possible bias of the examiner (us) regarding the measurement of both duration and accuracy of task completion. We mitigated the time measurement threat by designing a method for time measurement that both the participant and the examiner agreed



upon, namely physically exchanging the task sheet between the participant and the examiner. We mitigated the bias towards measuring accuracy by creating a rubric prior to conducting the experiments, and abiding by the rubric for marking the tasks in order to address this threat. The final threat we address is the tool used in the experiment. We chose Google Chrome’s Developer Tools, which is very popular for client-side web development, and all our participants were previously familiar with it (based on the pre-questionnaire they filled out).

## **5.8 Concluding Remarks**

In this paper, we proposed a generic technique for inferring a hierarchical model of application-specific motifs from execution traces. Our motifs, inspired by bioinformatics algorithms, are recurring abstract patterns of execution that abstract out alterations and are closer to the higher-level features of a system. We designed a visualization for our technique that allows users to observe and query the motifs for program understanding. Our technique is implemented in a tool called SABALAN, which is publicly available. The results of our user experiment showed that using the systematic analysis of SABALAN enabled participants to perform comprehension tasks 54% more accurately than other state of art tools.

## Chapter 6

# Related Work

Previous research has approached understanding program behaviour and change through different perspectives.

**Program Analysis.** EventRacer is a tool for facilitating dynamic race detection for event-driven applications [112]. Compliant with its goal, EventRacer traces only the events and not other dynamic and asynchronous feature of JavaScript. Moreover, unlike all our methods, theirs requires using an instrumented browser. Wei and Ryder [136] use both static and dynamic analysis to perform a points-to analysis of JavaScript. However, they do not take into account the DOM-based and asynchronous interactions of JavaScript. Ghezzi et al. [51] extract behavioural models from a different perspective. They focus on users' navigation preferences in user-intensive software. Their approach, called BEAR, depends on server logs to capture user interactions. Unlike CLEMATIS and SAHAND, BEAR only focuses on direct user interactions in order to fulfill its purpose, which is classifying the behaviour of users.

Originally proposed by Weiser [138], *program slicing* techniques can be classified in two categories, namely static and dynamic slicing [72]. WALA [124] performs JavaScript slicing by inferring a call graph through static analysis. Since JavaScript is such a dynamic language, WALA yields conservative results that may not be reflective of an application's actual execution. It also ignores the JavaScript-DOM interactions completely. Although not used for slicing purposes, others [94, 140] have utilized static analysis to reduce the execution overhead incurred

from code instrumentation. CLEMATIS determines JavaScript slices through a selective code instrumentation algorithm.

There are numerous static analysis techniques proposed for JavaScript analysis in different domains [47, 66, 69, 83, 95, 125]. We did not choose a pure 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 [4, 6, 96, 137]. 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 [84]. There are two differences between their work and ours. First, our method 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, our technique 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.

*Analysis of Asynchrony.* Different approaches target asynchrony in different domains, such as comprehension, debugging and testing. Frameworks such as Arrows [70] have been proposed to help developers understand and avoid asynchronous errors. Zheng et al. [145] used static analysis to find asynchronous bugs in web applications. WAVE [62] is a testing platform for finding concurrency errors on the client side. Libraries and features such as Async.js [25] and Promises [109] 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.

*Fault Localization and Debugging.* Delta debugging [144] is a technique whereby the code change responsible for a failure is systematically deduced by narrowing the state differences between a passing and a failing run. Other fault localization techniques have been proposed that compare different characteristics of passing and failing runs for a program [1, 28, 54, 110]. CLEMATIS is different in that it focuses on a single web application test assertion at a time and does not

require a passing test per se to operate.

There is limited research geared towards web application fault localization in the literature [14, 99]. Google has recently provided some support for debugging asynchronous JavaScript in Chrome DevTools [26]. Our work is different from previous techniques since it aims at making the implicit links between test failures and faulty JavaScript code more explicit to enhance debugging. In addition, calculating and displaying the JavaScript code slice for a test assertion poses new challenges not faced by previous techniques. This is stemmed from the disconnect between a test assertion failure, the DOM, and the JavaScript code interacting with the DOM.

**Analysis of Change.** *Static Analysis.* Static CIA is performed by analyzing the source code without executing it. A common pattern in many traditional CIA techniques is the usage of dependency-based impact analysis methods [18]. Static impact analysis techniques typically find syntactic dependencies by performing forward slicing on the graph. This type of analysis, however, is based on assumptions made for all possible executions of the software, and hence incurs false positives, which hinders its adoption [58]. The dependency graph can become large and may contain invalid paths. Hence, the resulting impact set can be large and difficult to comprehend. More recently, static analysis has been applied to analyze JavaScript applications. Sridharan et al. [125] adapt traditional points-to analysis for JavaScript through correlation tracking of dynamic properties in the code. Jensen et al. [66] statically model the role of the DOM and browser in their analysis. However, they acknowledge gaps and shortcomings in their analysis, which can result in many false-positives. Feldthaus et al. [47] present an approach for constructing approximate JavaScript call graphs. However, their analysis completely ignores dynamic property accesses and interactions with the DOM. Madsen et al. [83] combine pointer analysis with use analysis to investigate the effects of JavaScript libraries and frameworks on the applications' data flow. These techniques neglect the dynamic DOM interactions as well as event-driven, and asynchronous features of the JavaScript language. Therefore, their analysis can be incomplete for performing change impact analysis for JavaScript applications.

*Dynamic Analysis.* Existing dynamic methods produce a precise but incomplete analysis. Apiwattanapong et al. [12] propose a dynamic technique in which execute-after relations are used to reduce the overhead caused by the amount of collected

dynamic information. Dynamic CIA tools have been applied to various fields of software engineering. For instance, Ramanathan et al. [111] avoid testing unchanged test cases by comparing strings of different traces in their tool. Chianti [114] is a CIA tool for Java that reports the change impact in terms of the subset of the test suite affected by the change. These techniques provide more precision compared to static analysis, especially when integrated with other techniques such as information retrieval [41][50]. However, they do not target JavaScript code and its unique analysis challenges, such as DOM interactions, dynamic function calls involving event propagations, and asynchronous callbacks.

Wei and Ryder’s recent JavaScript blended analysis approach [136] and state-sensitive points-to analysis [137] are perhaps the closest to our work. Their work integrates the information gathered during both static and dynamic analyses to perform a points-to analysis of JavaScript applications. However, their methods do not focus on analyzing the change impact, and hence do not incorporate the dependencies that are formed through DOM interactions and asynchronous JavaScript mechanisms. Moreover, they do not take into account the important role of events and event propagation [133] on the DOM tree, which connects JavaScript functions, unlike our analysis which does.

**Feature Location, Capture and Replay, and Tracing.** Many papers have focused on *locating the implementation* of UI- and interaction-based features [23, 81, 85, 86] 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 [143] 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. Li and Wohlstadter [78] present a tool called Script Insight to locate the implementation of a DOM element in JavaScript code. Similarly, Maras et al. [85, 86] 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.

Extensive reliance on user interactions is an important characteristic of modern web applications. *Capture and replay* tools are used in the literature to address this issue [34]. Record and replay techniques aid the understanding and debugging tasks of web applications [11, 22, 89, 91]. The goal of these techniques, however, is to provide a deterministic replay of UI events without capturing their consequences. Montoto et al. [91] propose a set of techniques for generating a navigation sequence for Ajax-based websites and executing the recorded trace. Mugshot [89] is a system which employs a server-side web proxy to capture events in interactive web applications. It injects code into a target web application in order to record sources of nondeterminism such as DOM events and interrupts. The recorded information is used by Mugshot to dispatch synthetic events to a web browser in order to replay the execution trace. WaRR [11] is another system for capturing and replaying events. Capturing is accomplished by altering a user's web browser in order to record keystrokes and mouse clicks. In the event of a failure, end users of a web application may send a record of their keystrokes to the developer for debugging purposes. Jalangi [122] is another record-replay tool that supports dynamic analysis by shadow execution on shadow values. Burg et al. [22] integrate their capture/replay tool with debugging tools.

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

*Tracing* techniques such as FireCrystal [101] and DynaRIA [10] collect traces of JavaScript execution selectively. Unravel [61] 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.

**Visualization.** There are many tools that use visualization to improve the process

of understanding the behaviour of software applications [10, 101, 143]. However, these methods are not concerned with creating a behavioural model and inferring high-level motifs of execution, while ours is.

Matthijssen et al. [87] conduct a user study for investigating the strategies that web developers use for code comprehension. Extraviz [35] is a visualization tool that represents the dynamic traces of Java applications to assist with program comprehension tasks. However, their approach does not concern itself with building a model of the web application, while ours does.

Zaidman et al. [143] 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 [101] 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 [10] 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.

*Trace Visualization.* Several papers assist program comprehension through dynamic analysis and visualization of traces. Their proposed techniques allow users to explore large traces [115], or perform reduction, compaction and pruning techniques on traces [19, 55–57, 113]. A popular trend is using standard visual protocols, such as UML diagrams [21, 38, 127]. Other papers propose more customized visualization techniques through synchronized views [33], provide program’s landscape focusing on communications [48], allow user interactions with the visualization [104], visualize similarities in traces [32], or present many other techniques for representing the traces [17, 37, 47, 64, 65, 68, 79, 115, 120, 131]. Extravis [33] is the first such technique that was quantitatively measured through a controlled exper-

iment [35]. Another group of methods capture and analyze low-level information in execution traces using techniques such as extracting behavioural units described in usecase scenarios [134], profiling [73], dividing the trace into segments [107], identifying feature-level phases by defining an optimization problem [16], or similar methods [30, 39, 103]. Heuzeroth et al. [59, 60] propose to find patterns in execution. Other papers aim at providing higher-level representations of trace [7, 142]. However, unlike our approach, these approaches do not address the problem of large traces, do not infer a higher-level model of program behaviour, and do not application-specific and hierarchical abstract motifs for facilitating comprehension.



## Chapter 7

# Concluding Remarks

Program comprehension is vital for performing many software engineering tasks, consuming much of the effort in software engineering. 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, the DOM, and the server. However, web developers face numerous challenges when trying to understand these interactions.

### 7.1 Contributions

In this thesis, we introduced our novel techniques for understanding the behaviour, root causes of failures, and impact of change for JavaScript, as well as inferring higher-level motifs of behaviour from execution traces. The main contributions of the thesis are as follows.

- A portable and fully-automated technique, called CLEMATIS, for extracting episodes of interaction in JavaScript-based web applications in Chapter 2. We also proposed a strategy for helping developers understand the root causes of failing test cases. We presented a novel interactive visualization mechanism based on focus+context techniques, for facilitating the understanding of these complex event interactions. 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.

- An automated technique, called TOCHAL, for performing a hybrid DOM-sensitive change impact analysis for JavaScript (Chapter 3). TOCHAL builds a novel hybrid system dependency graph, by inferring and combining static and dynamic call graphs. Our technique ranks the detected impact set based on the relative importance of the entities in the hybrid graph. Our evaluation shows that the dynamic and DOM-based JavaScript features occur in real applications and can lead to significant means of impact propagation. Furthermore, we find that a hybrid approach leads to a more complete analysis compared with a pure static or dynamic analysis. Finally, our industrial controlled experiment shows that TOCHAL increases developers' performance, by helping them to perform maintenance tasks faster and more accurately.
- A novel technique, called SAHAND, for aiding developers' comprehension of full-stack JavaScript applications by creating a behavioural model of the application. The model, described in Chapter 4, 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. 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.
- A generic technique for inferring a hierarchical model of application-specific motifs from execution traces (Chapter 5). Our motifs, inspired by bioinformatics algorithms, are recurring abstract patterns of execution that are accommodating to alterations and are closer to the higher-level features of a system. We designed a visualization for our technique that allows users to observe and query the motifs. Our technique is implemented in a tool called SABALAN, which is publicly available. The results of our user experiment showed that using the systematic analysis of SABALAN enabled participants to perform 54% more accurately.

## 7.2 Research Questions Revisited

We introduced a set of five research questions in Chapter 1, which were addressed by the work presented in the following chapters (chapters 2–5).

### Research Question 1

*How can we enhance developers' performance in understanding the event-based interactions in client-side JavaScript?*

This research question focuses on understanding the behaviour of modern web applications, which is 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. To address RQ1, in Chapter 2, we proposed 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 showed that CLEMATIS is capable of improving the comprehension task accuracy by 157%, while reducing the task completion time by 47%. Further, the results showed that CLEMATIS is most helpful for complex and implicit relations in program execution that are troublesome for developers to understand, if not impossible. For instance, in the experimental object, the effect of using CLEMATIS was most visible when the tasks involved timing events, event propagations, or server communications. It can also 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. Unlike the control group, 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.

## **Research Question 2**

*How can we enhance developers' performance in understanding the root-causes of test assertion failures in client-side JavaScript?*

While RQ1 addresses understanding the behaviour of client-side JavaScript applications, RQ2 focuses on understanding the behaviour specifically when a failure occurs. The goal of this research question is to investigate the means by which we can help developers bridge the gap between test cases and program code by localizing the fault related to a test assertion. Our approach for addressing RQ2 is built on top of CLEMATIS. We proposed an automated technique to help developers localize the fault related to a test failure. Through a combination of selective code instrumentation and dynamic backward slicing, our technique bridges the gap between test cases and program code. We extended the visualization of CLEMATIS to help developers understand the relation of observed program behaviour to the test cases. A follow up experiment reveals that extended CLEMATIS improves the fault localization accuracy of developers by a factor of two. This approach is also discussed in Chapter 2.

## **Research Question 3**

*How can we improve developers' understanding of the temporal and asynchronous behaviour of full-stack JavaScript?*

RQ1 and RQ2 both target understanding different aspect of JavaScript applications, but only on the client side. While JavaScript is the lingua franca of client-side web development, it is also used for server-side programming, leading to “full-stack” applications written entirely in JavaScript. RQ3 targets the challenges that rise when understanding distributed components of a full-stack application on the client- and server-side, as well as their interactions. The temporal nature of program execution particularly the asynchronous and implicit relations of JavaScript entities spread over the client and the server make comprehension a rigorous task.

In Chapter 4, we proposed 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.

#### **Research Question 4**

*How does providing a model of the dependencies in the application improve developers' understanding of the change impact in JavaScript applications?*

While all the previous research questions focus on understanding the behaviour of a JavaScript application exactly as captured from an execution session, this research questions target understanding the behaviour in the presence of a change in the code. It aims at helping developers understand and analyze the impact of a change in the system, and predict the potential impact even before the actual change happens.

In Chapter 3, we propose a change impact analysis for JavaScript which addresses challenges such as the seamless interplay with the DOM, event-driven and dynamic function calls, and asynchronous client/server communication. We first perform an empirical study of change propagation, the results of which show that the DOM-related and dynamic features of JavaScript need to be taken into consideration in the analysis since they affect change impact propagation. We propose a DOM-sensitive hybrid change impact analysis technique for JavaScript through a combination of static and dynamic analysis. The proposed approach incorporates a novel ranking algorithm for indicating the importance of each entity in the impact set. Our approach is implemented in a tool called TOCHAL. The results of our evaluation reveal that TOCHAL provides a more complete analysis compared to static or dynamic methods. Moreover, through an industrial controlled experiment, we find that TOCHAL helps developers by improving their task completion duration by 78% and accuracy by 223%.

## Research Question 5

*How does providing high-level and semantic motifs of a application's behaviour improve comprehensibility of the application?*

All previous research questions rely on all details of program execution, captured through traces. However, due to the amount of information that can be gathered through the execution, the captured traces tend to become very large, complex, and difficult to understand. None of these approaches provide a higher-level abstraction of the program behaviour or infer recurring and behavioural patterns representing the semantics of a particular application, which is the focus of this research question. Further, the scope of this research question is not limited to JavaScript and includes all program traces. In Chapter 5, we propose a generic technique for facilitating comprehension by creating an abstract model of software behaviour. The model encompasses hierarchies of recurring and application-specific motifs. The motifs are abstract patterns extracted from traces through our novel technique, inspired by bioinformatics algorithms. The motifs provide an overview of the behaviour at a high-level, while encapsulating semantically related sequences in execution. We design a visualization that allows developers to observe and interact with inferred motifs. We implement our approach in an open-source tool, called SABALAN, and evaluate it through a user experiment. The results show that using SABALAN improves developers' accuracy in performing comprehension tasks by 54%.

## 7.3 Reflections and Future Directions

In this thesis, we took the first steps towards using static and dynamic analysis for assisting program comprehension, particularly for JavaScript. Our approaches aimed at algorithmically structuring program entities and their execution traces into behavioural models and making them more comprehensible. We also used information visualization techniques to facilitate developers' understanding of the application. However, there still remains much left to be addressed.

As a next future step to this thesis, researchers can further investigate the means of helping comprehension by providing semantic models and patterns of execution of software systems. There is great need for techniques that bridge the gap

between low-level details of code and execution and the mental model of developers. Our preliminary studies showed that such techniques are effective in facilitating comprehension by significantly improving developers' performance. However, there is not much research conducted for providing higher-level and semantic overviews into the behaviour of software that better match the mental model of developers.

Conducting long-term experiments with professional developers can greatly guide the design of such techniques and tools. Investigating how developers understand program behaviour can provide valuable insight for the current state of research in program comprehension. Studying developers' expectations of comprehension tools can drive designing, building, and deploying such tools. An iterative process of studying, designing, evaluating, and improving the design based on feedback can lead to more useful comprehension techniques that are more likely to be adopted in industry as well.

Further, designing and building more sophisticated visualizations using information visualization techniques can allow programmers to gain a better insight into program behaviour. Enabling visual pattern recognition and clustering methods in an interactive visual interface can greatly benefit the viewers. The interaction mechanisms such as querying, filtering, semantic zooming, bookmarking, and adding notes can be utilized to help developers locate and understand their required information easier, faster, and more accurately.

Moreover, more tool support for such techniques can greatly increase their impact in real-world software engineering. Integration with programming IDE's, debugger, and other developments environments can have a significant effect on adaptation of these techniques, and improve developers' performance in their everyday tasks. Another direction these approaches can pursue is in debugging. Specially improving CLEMATIS's fault localization unit to further help developers detect and localize faulty behaviour of JavaScript applications.

Another possible future direction in program understanding is understanding other program artifacts, such as test cases. The test suite is a major part of an application that needs to be understood and maintained by the developers. Despite this necessity, there has been far less research on aiding developers' understanding of the test suites. Researchers can extend program comprehension to understanding the test suites of applications and to address the challenges specific to test compre-

hension and maintenance. Understanding test code is difficult, firstly due to the same challenges that make production-code comprehension rigorous. Moreover, compared to testing traditional languages, JavaScript-specific features such as DOM interactions and server communications further complicate comprehension of JavaScript test suites. Developers and testers use assertions in test cases to investigate the correctness of the code under test. Assertions are vital to functionality and quality of test suites. They can examine the validity of a condition and the conformance of the target code to a requirement. However, they do not contain information regarding the procedure of a test case, its semantics, and its purpose. In order to understand a test case, developers need to understand the process that led to the state of the application that is asserted in the test case. Furthermore, coverage metrics are used for distinguishing parts of the code that are covered by a test suite. Code coverage can reveal parts of the application that are not tested. Although useful, coverage tools do not provide any information regarding how test cases interact with the code. Moreover, client-side JavaScript applications consists of more than just the code. The challenges of understanding JavaScript application propagate to understanding test cases as well. With existing techniques, it is difficult to understand how the test cases interact with the dynamic and event-driven JavaScript code, the DOM, and the server. We hypothesize that having an always-on visualization in a development environment is beneficial for developers to improve their performance in understanding the testing process of a given application. Deploying such approaches by embedding them into development environments will provide realtime visual cues for representing the test cases, their dynamic procedure, and their interactions with the code, DOM elements, and the server.

It is worth mentioning that this thesis is only a step towards facilitating comprehension of program behaviour and motifs. Our findings show the usefulness of our proposed techniques, as well as great promise for further research in the area. We have made all our tools and supporting documents open source and available to be used in future research.



# Bibliography

- [1] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE)*, pages 143–151, Toulouse, France, 1995. IEEE. → pages 3, 10, 146
- [2] 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. → pages 106
- [3] S. Alimadadi. Understanding behavioural patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, pages 1076–1078. ACM, 2016. → pages 8
- [4] 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. → pages v, 4, 5, 7, 93, 102, 146
- [5] 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>. → pages 39
- [6] 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. → pages v, 5, 7, 100, 146
- [7] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *Proceedings of the*

- ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1169–1180. ACM, 2016. URL <http://salt.ece.ubc.ca/publications/docs/icse16.pdf>. → pages vi, 6, 8, 118, 142, 151
- [8] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions with Clematis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, page 17 pages, 2016. URL <http://salt.ece.ubc.ca/publications/docs/tosem16.pdf>. → pages v, 7
  - [9] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990. ISSN 0022-2836. → pages 124
  - [10] 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. → pages 4, 93, 149, 150
  - [11] 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. → pages 149
  - [12] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 432–441. ACM, 2005. → pages 5, 147
  - [13] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society, 1996. ISBN 0818673842. → pages 5
  - [14] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 265–274. ACM, 2010. → pages 147
  - [15] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 571–580. ACM, 2011. → pages 3, 10
  - [16] O. Benomar, H. Sahraoui, and P. Poulin. Detecting program execution phases using heuristic search. In *International Symposium on Search Based Software Engineering*, pages 16–30. Springer, 2014. → pages 151

- [17] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 468–479. ACM, 2014. → pages 150
- [18] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. → pages 147
- [19] J. Bohnet, M. Koeleman, and J. Döllner. Visualizing massively pruned execution traces to facilitate trace exploration. In *International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 57–64. IEEE, 2009. → pages 120, 150
- [20] B. Breech, M. Tegtmeier, and L. Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 55–65. IEEE, 2006. → pages 5
- [21] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006. → pages 6, 118, 150
- [22] 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. → pages 149
- [23] 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. → pages 148
- [24] callbackhell. Callback hell, a guide to writing asynchronous JavaScript programs. <http://callbackhell.com>, July 27, 2017. → pages 95
- [25] Caolan McMahon. Async.js. <https://github.com/caolan/async>, July 27, 2017. → pages 146
- [26] P. Chen. Debugging asynchronous JavaScript with chrome DevTools, July 27, 2017. URL <http://www.html5rocks.com/en/tutorials/developertools/async-call-stack/>. → pages 147

- [27] chromedevtools. Chrome devtools.  
<https://developers.google.com/web/tools/chrome-devtools/>, July 27, 2017.  
 → pages 79, 109
- [28] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 342–351, New York, NY, USA, 2005. ACM. → pages 3, 10, 146
- [29] 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. → pages 31
- [30] J. E. Cook and Z. Du. Discovering thread interactions in a concurrent system. *Journal of Systems and Software*, 77(3):285–297, 2005. → pages 151
- [31] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989. → pages 1, 117
- [32] B. Cornelissen and L. Moonen. Visualizing similarities in execution traces. In *Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 6–10, 2007. → pages 150
- [33] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. Van Wijk, and A. Van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *International Conference on Program Comprehension (ICPC)*, pages 49–58. IEEE, 2007. → pages 6, 117, 150
- [34] 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. → pages 2, 117, 118, 149
- [35] 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. → pages 120, 150, 151
- [36] J. W. Creswell. *Qualitative inquiry and research design: Choosing among five approaches*. Sage, Thousand Oaks, California, 2nd edition, 2012. → pages 45
- [37] W. De Pauw and S. Heisig. Zinsight: A visual and analytic environment for exploring large event traces. In *Proceedings of the 5th international symposium on Software visualization*, pages 143–152. ACM, 2010. → pages 150

- [38] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 151–162. 2002. → pages 150
- [39] M. Denker, J. Ressa, O. Greevy, and O. Nierstrasz. Modeling features at runtime. In *International Conference on Model Driven Engineering Languages and Systems*, pages 138–152. Springer, 2010. → pages 151
- [40] P. D’haeseleer. What are DNA sequence motifs? *Nature biotechnology*, 24(4):423–425, 2006. → pages 118
- [41] B. Dit, M. Wagner, S. Wen, W. Wang, M. Linares-Vásquez, D. Poshyvanyk, and H. Kagdi. Impactminer: A tool for change impact analysis. In *Companion Proceedings of the International Conference on Software Engineering (ICSE)*, pages 540–543. ACM, 2014. → pages 148
- [42] Document Object Model (DOM). Document Object Model (DOM). <http://www.w3.org/DOM/>, July 27, 2017. → pages 64
- [43] escodegen. Escodegen. <https://github.com/esutils/escodegen>, July 27, 2017. → pages 108, 134
- [44] esprima. Esprima. <http://esprima.org/>, July 27, 2017. → pages 108
- [45] estraverse. Estraverse. <https://github.com/esutils/estraverse>, July 27, 2017. → pages 108, 134
- [46] express. Express. <http://expressjs.com/>, July 27, 2017. → pages 94
- [47] 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. → pages 5, 29, 146, 147, 150
- [48] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: The explorviz approach. In *IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4. IEEE, 2013. → pages 118, 150
- [49] 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. → pages 4, 92, 95

- [50] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 430–440. ACM, 2012. → pages 148
- [51] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 277–287. ACM, 2014. → pages 145
- [52] M. Gollery. Bioinformatics: Sequence and genome analysis, 2nd ed. *Clinical Chemistry*, 51(11):2219–2219, 2005. → pages 124
- [53] googlecharts. Google chart tools. <https://developers.google.com/chart/>, July 27, 2017. → pages 108
- [54] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Workshop on Model Checking of Software*, pages 121–135. Springer Berlin Heidelberg, 2003. → pages 146
- [55] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *International Conference on Program Comprehension (ICPC)*, pages 181–190. IEEE, 2006. → pages 117, 120, 150
- [56] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004. → pages
- [57] A. Hamou-Lhadj, T. C. Lethbridge, and L. Fu. Challenges and requirements for an effective trace exploration tool. In *International Workshop on Program Comprehension (ICPC)*, pages 70–78. IEEE, 2004. → pages 150
- [58] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damásio. On the precision and accuracy of impact analysis techniques. In *Proceedings of the International Conference on Computer and Information Science*, pages 513–518, 2008. → pages 147
- [59] D. Heuzeroth, T. Holl, and W. Löwe. *Combining static and dynamic analyses to detect interaction patterns*. Univ., Fak. für Informatik, Bibliothek, 2001. → pages 151

- [60] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Löwe. Automatic design pattern detection. In *International Workshop on Program Comprehension (ICPC)*, pages 94–103. IEEE, 2003. → pages 6, 118, 151
- [61] J. Hibschan 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. → pages 4, 93, 120, 149
- [62] 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. → pages 146
- [63] hoxy. Hoxy proxy. [greim.github.io/hoxy/](https://greim.github.io/hoxy/), July 27, 2017. → pages 134
- [64] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the art of performance visualization. *EuroVis 2014*, 2014. → pages 150
- [65] S. Jayaraman, B. Jayaraman, and D. Lessa. Compact visualization of Java program execution. *Software: Practice and Experience*, 2016. → pages 150
- [66] 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. → pages 5, 146, 147
- [67] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05*, pages 273–282. ACM, 2005. → pages 3, 10
- [68] B. Karran, J. Trumper, and J. Dollner. Synctrace: Visual thread-interplay analysis. In *Software Visualization (VISsOFT), 2013 First IEEE Working Conference on*, pages 1–10. IEEE, 2013. → pages 150
- [69] 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. → pages 146

- [70] Y. P. Khoo, M. Hicks, J. S. Foster, and V. Sazawal. Directing JavaScript with arrows. *ACM SIGPLAN Notices*, 44(12):49–58, 2009. → pages 146
- [71] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006. → pages 1, 117
- [72] B. Korel and J. W. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988. → pages 145
- [73] J. Koskinen, M. Kettunen, and T. Systa. Profile-based approach to support comprehension of software behavior. In *International Conference on Program Comprehension (ICPC)*, pages 212–224. IEEE, 2006. → pages 6, 118, 151
- [74] A. La. Language trends on GitHub.  
<https://github.com/blog/2047-language-trends-on-github>, July 27, 2017. → pages 2, 119
- [75] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, pages 492–501. ACM, 2006. → pages 1, 6
- [76] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 492–501. ACM, 2006. → pages 117
- [77] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. → pages 5
- [78] 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. → pages 148
- [79] S. Lin, F. Taïani, T. C. Ormerod, and L. J. Ball. Towards anomaly comprehension: using structural compression to navigate profiling call-trees. In *Proceedings of the 5th international symposium on Software visualization*, pages 103–112. ACM, 2010. → pages 150



- [80] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundness: A manifesto. *Communications of the ACM*, 58(2): 44–46, 2015. → pages 75
- [81] 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. → pages 148
- [82] I. Loire. Math-race. <https://github.com/iloire/math-race>, July 27, 2017. → pages 109
- [83] 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. → pages 146, 147
- [84] 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. → pages 4, 93, 146
- [85] 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. → pages 148
- [86] 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. → pages 148
- [87] N. Matthijssen, A. Zaidman, M.-A. Storey, I. Bull, and A. Van Deursen. Connecting traces: Understanding client-server interactions in Ajax applications. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 216–225. IEEE Computer Society, IEEE, 2010. → pages 150
- [88] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)*, 38(1):35–53, 2012. → pages 3, 10
- [89] 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. → pages 149

- [90] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Guided mutation testing for JavaScript web applications. *IEEE Transactions on Software Engineering (TSE)*, page 19 pages, 2015. → pages 50
- [91] 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. → pages 149
- [92] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 18–28. ACM, 1995. → pages 1, 6, 117
- [93] Mutation Summary. Mutation summary. <https://code.google.com/p/mutation-summary/>, July 27, 2017. → pages 79
- [94] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, May 2005. → pages 145
- [95] 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. → pages 146
- [96] 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. → pages 146
- [97] nodejs. Node.js. <https://nodejs.org/>, July 27, 2017. → pages 4
- [98] 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. → pages 10

- [99] F. J. Ocariza, K. Pattabiraman, and A. Mesbah. AutofloX: An automatic fault localizer for client-side JavaScript. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 31–40. IEEE Computer Society, 2012. → pages 147
- [100] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun. Behavioral resource-aware model inference. In *Proceedings of the ACM/IEEE international conference on Automated software engineering*, pages 19–30. ACM, 2014. → pages 118
- [101] 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. → pages 2, 4, 93, 149, 150
- [102] 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. → pages 38, 57, 110, 115, 136, 138, 143
- [103] V. K. Palepu and J. A. Jones. Visualizing constituent behaviors within executions. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–4. IEEE, 2013. → pages 151
- [104] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, and M. Haupt. Immediacy through interactivity: online analysis of run-time behavior. In *Working Conference on Reverse Engineering (WCRE)*, pages 77–86. IEEE, 2010. → pages 150
- [105] M. Petrenko and V. Rajlich. Variable granularity for improving precision of impact analysis. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 10–19. IEEE, 2009. → pages 5
- [106] Phormer. Phormer PHP photo gallery. <http://p.horm.org/er/>, July 27, 2017. → pages 86, 137, 143
- [107] H. Pirzadeh and A. Hamou-Lhadj. A novel approach based on gestalt psychology for abstracting the content of large execution traces for program comprehension. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 221–230. IEEE, 2011. → pages 151
- [108] 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.  
→ pages 107

- [109] Promises. Promises/A+. <https://promisesaplus.com>, July 27, 2017. → pages 146
- [110] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *International Workshop on Automated and Algorithmic Debugging*, pages 273–276, Ghent, Belgium, 2003. → pages 146
- [111] M. K. Ramanathan, A. Grama, and S. Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 241–252. IEEE, 2006. → pages 5, 148
- [112] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 151–166. ACM, 2013. → pages 145
- [113] S. P. Reiss and M. Renieris. Encoding program executions. In *proceedings of the 23rd International Conference on Software Engineering*, pages 221–230. IEEE Computer Society, 2001. → pages 120, 150
- [114] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 432–448. ACM, 2004. → pages 5, 148
- [115] M. Renieris and S. P. Reiss. Almost: exploring program traces. In *Proceedings of the 1999 workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM international conference on Information and knowledge management*, pages 70–77. ACM, 1999. → pages 150
- [116] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2010. → pages 60, 67

- [117] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004. → pages 1, 117
- [118] Sabalan. Sabalan. Anonymized for double-blind review., July 27, 2017. → pages 124, 134, 138
- [119] sahand. Sahand: tool implementation and dataset. <http://salt.ece.ubc.ca/software/sahand/>, July 27, 2017. → pages 93, 108, 109, 111, 115, 116
- [120] R. R. Sambasivan, I. Shafer, M. L. Mazurek, and G. R. Ganger. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE transactions on visualization and computer graphics*, 19(12):2466–2475, 2013. → pages 150
- [121] selenium. Selenium. <http://seleniumhq.org>, July 27, 2017. → pages 3
- [122] 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. → pages 149
- [123] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981. → pages 127
- [124] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. *SIGPLAN Notices*, 42(6):112–122, June 2007. → pages 145
- [125] 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. → pages 5, 81, 146, 147
- [126] Stack Overflow. Developer survey. <http://stackoverflow.com/research/developer-survey-2017>, July 27, 2017. → pages 1, 119
- [127] T. Systä, K. Koskimies, and H. Müller. Shimba—an environment for reverse engineering java software systems. *Software: Practice and Experience*, 31(4):371–394, 2001. → pages 150

- [128] T. J. Watson Libraries for Analysis. WALA. <http://wala.sourceforge.net/>, July 27, 2017. → pages 79, 81
- [129] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 162–171. IEEE, 2013. → pages 3, 10
- [130] Tochal. Tochal. <https://github.com/saltlab/tochal>, July 27, 2017. → pages 61, 80
- [131] J. Trümper, J. Bohnet, and J. Döllner. Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the 5th international symposium on Software visualization*, pages 133–142. ACM, 2010. → pages 150
- [132] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459 – 494, 1985. → pages 3, 10
- [133] W3C. Document Object Model (DOM) level 2 events specification. <http://www.w3.org/TR/DOM-Level-2-Events/>, July 27, 2017. → pages 10, 12, 19, 60, 65, 100, 148
- [134] Y. Watanabe, T. Ishio, and K. Inoue. Feature-level phase detection for execution trace using object cache. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 8–14. ACM, 2008. → pages 151
- [135] webstorm. WebStorm. <https://www.jetbrains.com/webstorm/>, July 27, 2017. → pages 109
- [136] S. Wei and B. G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 336–346. ACM, 2013. → pages 5, 81, 145, 148
- [137] 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. → pages 5, 81, 146, 148
- [138] M. Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 439–449. IEEE, 1981. → pages 145

- [139] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012. → pages 37, 84, 109
- [140] S. Yong and S. Horwitz. Using static analysis to reduce dynamic analysis overhead. *Formal Methods in System Design*, 27(3):313–334, 2005. → pages 145
- [141] A. Zaidman. Scalability solutions for program comprehension through dynamic analysis. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 4–pp. IEEE, 2006. → pages 6, 117
- [142] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329–338. IEEE, 2004. → pages 6, 117, 151
- [143] 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. → pages 2, 4, 40, 93, 148, 150
- [144] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10. ACM, 2002. → pages 3, 10, 146
- [145] 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. → pages 146