A Doctor Thesis
博士論文

Interaction-Based Preventive Maintenance of Ajax Web Applications

（相互作用に着目した Ajax Web アプリケーションの予防保守）

by

Yuta Maezawa
前澤 悠太

Submitted to
the Graduate School of the University of Tokyo
on December 12, 2014
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Shinichi Honiden　本位田 真一
Professor of Computer Science

ABSTRACT

The challenge of maintaining Web applications is preventing actual errors in a user environment. Currently, users demand rich user experience of applications; hence, client-side asynchronous JavaScript and XML (Ajax) technologies are increasingly important. Developers build modern Web applications using Ajax technologies (Ajax Web applications), which can handle user events and asynchronously retrieve update data from servers. Thus, Ajax technologies make Web applications responsive, enhancing user experience. However, Ajax event-driven and asynchronous features make the contexts of running applications unpredictable. Despite concerted efforts by developers, these unpredictable contexts might conceal faults in development and testing environments but they will be exposed in a user environment.

Much research has been conducted on state-based analysis and testing for finding faults in Ajax Web applications. The states of these applications correspond to the document object model (DOM) representing Web pages. Since these applications dynamically manipulate DOMs by handling user events or server responses, some have succeeded in leveraging dynamic analysis techniques for extracting a finite state machine (FSM) based on DOM instances captured at runtime. Although current DOM-based testing techniques effectively and efficiently detect executable faults, they do not help finding "potential faults" that are not easily produced in a testing environment but will be exposed in a user environment.

First, we propose a static method for extracting an FSM from Ajax Web applications. Since DOMs cannot be determined in a static manner because they are dynamically manipulated, we focus on interactions with Ajax Web applications (e.g., mouse clicks and server responses), which act as triggers to change the application states. These interactions can be statically distinguished at event handlers in the source code. Hence, our proposed method extracts an interaction-based FSM representing all possible application behaviors. Developers can use the extracted FSM to find the potential faults; however, the cost may not be negligible to manually and carefully determine the correctness of the extracted FSM, which does not enable developers to exhaustively find faults in the applications.

Second, we propose a method for automatically verifying the correctness of the extracted FSM. Model checking techniques are useful for automated verification against given invariants representing correct behaviors; however, there are no generic behavior invariants relevant to the interactions. Therefore, we define interaction invariants from correct and incorrect interaction-based behaviors described in Ajax design patterns, which is a catalog of Ajax Web application development know-how. The model checker leverages the interaction invariants to identify faulty interaction sequences in the extracted FSM. Although the identified faulty interaction sequences should contain suspected faults, they might be spurious counterexamples in the extracted FSM. Therefore, executable evidence that the suspected faults cause errors in Ajax Web applications is required.

Finally, we propose a method for validating applications by revealing actual errors due to potential faults. Although testing all possible scenarios in every environment should reveal these errors, such a method would be unrealistic. Under an assumption that unexpected network latency may make potential faults executable, we implement mutation operators to manipulate the timing of the applications handling server interactions. Our mutation operators are able to produce subtle network delays, which can reveal errors due to delay-dependent potential faults.

From the results of our case studies using real-world Ajax Web applications, our proposed methods revealed errors relevant to vulnerabilities in Web applications, which current analysis and testing techniques cannot detect. Therefore, we conclude that our proposed methods can help developers find and debug potential faults, i.e., help in preventive maintenance of Ajax Web applications.

論文要旨

Web アプリケーションを保守する際の課題は，ユーザ環境でのエラーの顕在化を防ぐことである．近年の Web アプリケーションでは豊かなユーザ体験への需要が大きいため，クライアント側における Asynchronous JavaScript and XML （ Ajax ） 技術の重要性が高まっている．Ajax 技術を実装した Web アプリケーション（ Ajax Web アプリ）は，ユーザイベントに応じて更新データを非同期通信により取得することで，応答性を高めユーザ体験の向上が見込める．しかし， Ajax 技術のイベント駆動性・非同期性により，開発者が Ajax Web アプリの全ての実行状態を把握することは難しく，欠陥のある実行状態を見逃す恐れがあり，ユーザ環境でのエラーの顕在化につながる．

Ajax Web アプリの欠陥発見のために，状態遷移モデルを元に解析・テストする手法が研究されている．Ajax Web アプリでは Document Object Model （ DOM ）構造をその状態と見なせる．DOM は実行時のユーザイベントや非同期通信の結果に応じて動的に操作されるため，Ajax Web アプリの実行結果から状態遷移モデルを抽出しテストする動的解析手法が取り組まれてきた．既存手法は Ajax Web アプリの実行可能な欠陥を効果的・効率的に検出できる．しかし，開発者のテスト環境で再現しないが，ユーザ環境で顕在化する “潜在的な欠陥” の発見は難しい．

本研究ではまず，Ajax Web アプリから静的に状態遷移モデルを抽出する手法を提案する．実行時に決まる DOM は静的に解析できないため，提案手法では Ajax Web アプリの状態を変化させる相互作用（例えば，マウスクリックやサーバレスポンス）に着目する．ソースコード上のイベントハンドラから静的に抽出できる相互作用を元にすると，Ajax Web アプリの実行可能性に依存しない状態遷移モデルを抽出でき，開発者が振舞いを理解し潜在的な欠陥の発見に役に立つと期待できる．しかし，複雑な状態遷移モデルが抽出される場合，開発者が人手で精査することは難しくなる．

次に本研究では，抽出された状態遷移モデルの正しさを自動的に検証する手法を提案する．状態遷移モデルの正しさの検証にモデル検査技術を利用するには，正しい振舞いを表す不変条件が必要だが，Ajax Web アプリの相互作用に関わる正しい振舞いの一般的な定義はない．そこで提案手法では，Ajax Web アプリ開発のノウハウをまとめた Ajax デザインパターンから相互作用に関わる振舞いを整理する．するとモデル検査器は，状態遷移モデルから Ajax デザインパターンに反する状態遷移列を識別できる．しかし，得られた状態遷移列は抽象的なモデルにおける偽反例の可能性があるため，実際に Ajax Web アプリのエラーを引き起こすか確かめる必要がある．

本研究ではさらに，潜在的な欠陥が原因となるエラーを顕在化することで Ajax Web アプリを検査する手法を提案する．Ajax Web アプリの全実行シナリオを全実行環境でテストすれば，そのエラーを顕在化できるが現実的でない．予期せぬ通信遅延がエラーの原因となりやすいため，提案手法では Ajax Web アプリが同期的・非同期的な通信結果を処理するタイミングを調整する変異操作を定義する．変異操作により通信遅延を繊細に調整でき，通信遅延に依存する潜在的な欠陥が原因となるエラーを顕在化できる．

現実の Ajax Web アプリを用いた適用事例の結果，提案手法により脆弱性に関わるエラーを顕在化でき，既存の解析・テスト手法ではそれらを検出できないことを確認した．したがって提案手法は，開発者が Ajax Web アプリの潜在的な欠陥を発見しデバッグする予防保守に役立つと考えられる．

Acknowledgements

My work in doctoral course has been backed up by the unstinting support received from many people, laboratories, universities, and institutions. First and foremost, I would like to sincerely appreciate my adviser Professor Shinichi Honiden, who has educated me on the matter of foundations for research activities. Additionally, he gave me many opportunities to advance my career. With the aid of his continual guidance, I could have tackled challenging and interesting issues and completed this work.

I would like to express my gratitude to Professor Masami Hagiya as a chair and Professor Reiji Suda, Professor Naoki Kobayashi, Lecturer Ichiro Hasuo, and Professor Shigeru Chiba as members of my doctoral committee, whose insightful and constructive feedback allowed me to improve this thesis.

I would also like to extend my sincere appreciation to Professor Bashar Nuseibeh, Dr. Yijun Yu, and Dr. Thein Than Tun at the Open University, Milton Keynes, UK and to Associate Professor Hironori Washizaki at Waseda University, who have provided me lots of valuable feedback on my work. There is no way I would have been able to achieve my tough goal without their mentoring supports.

Moreover, my sincere thanks are due to all members of the Honiden laboratory, who have instructed and inspired me. Especially, I am thankful to members of the software and Web engineering groups at the laboratory, who have discussed our close research topics and practiced our presentations, including Taku Inoue, Johan Nyström-Persson, Tsutomu Kobayashi, Kazuki Nishiura, Keiichiro Hoshi, Hiroki Sawano, Kiichi Ueta, and Tomoya Katagi, and also to elderly and the same year colleagues, who have encouraged me and cared both my research and private activities, including Yukino Baba, Daisuke Fukuchi, Ryuichi Takahashi, Susumu Toriumi, Hirotaka Moriguchi, Soramichi Akiyama, Ryo Shimizu, Atsushi Watanabe, Katsunori Ishino, Masahiro Ito, Yuji Kaneko, Taiken Zen, Chavilai Sombat, Hisayuki Horikoshi, Makoto Fujiwara, Hiroaki Yoshiike, and Shizu Miyamae. In addition, through the participations of many activities and projects of the laboratory, I would like to render my thanks to both past and present members of the laboratory and others, including Shigetoshi Yokoyama, Yoshinori Tanabe, Zu Zhenjiang, Nobukazu Yoshioka, Fuyuki Ishikawa, Kenji Tei, Kazunori Sakamoto, Takuo Doi, Fumihiro Kumeno, Masaru Nagaku, Prabin Karanjit, Kyoko Oda, Ai Tobimatsu, Saki Narimatsu, Kazue Kusama, Rey Abe, Katsushige Hino, Yusaku Kimura, Atsushi Suyama, Nobuaki Hiratsuka, Satoshi Katafuchi, Fan Jiang, Valentina Balijak, Adrian Klein, Florian Wagner, Yoshiyuki Nakamura, Koichi Fujikawa, Souichi Kamiya, Shingo Horiuchi, Naoki Tsurumi, Yuki Inoue, Shengbo Xu, Fernando Tarin, Susumu Tokumoto, Takayuki Suzuki, Kohsuke Yatoh, Shun Lee, Junto Nakaoka, Natsumi Asahara, Yuta Tokitake, Kazuya Aizawa, Miki Yagita, Yasuhiro Sezaki, Katsuhiko Ikeshita, Masaki Katae, Moeka Tanabe, and Yasuo Tsurugai. I do not doubt at all that I owe what I am to them.

Furthermore, I express my thanks to members in the Open University, who welcomed me and helped my life in the UK, including Pierre Akiki, Azadeh Alebrahim, Yago de Quary, Rosalba Giuffrida, Stefan Kreitmayer, Lionel Montrieux, Tu Anh Nguyen, Aleksandra Pawlik, Sunitha Pangala, Nadia Pantidi, Laura Plonka, Brian Pluss, Saad Saleem, Minh Tran and Ann Xambo. I could spend comfortable, interesting, and exciting time at the strange place with their cooperation.

I also wish to thank all my friends, who have made me refreshed by playing, talking, and drinking. It has been necessary for me to maintain my good physical and mental health.

Finally, I am deeply grateful to my family, who have gave me selfless supports and natural belief. My father Isamu and mother Mayumi have promoted all my interests and encouraged me constantly and warmly.

Contents

List of Figures

List of Tables

Citations to Printed Publications

Parts of this thesis have appeared in the following publications.

Journals

1. Yuta Maezawa, Hironori Washizaki, and Shinichi Honiden, "Supporting to Find Faults in Rich Internet Applications by Extracting Interaction-based State Machines", IPSJ Journal, Vol.54 (No.2), pp.820–834, February 2013.

Proceedings

1. Yuta Maezawa, Hironori Washizaki, and Shinichi Honiden, "Extracting Interaction-Based Stateful Behavior in Rich Internet Applications", In Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12), pp. 423–428, March 2012.

2. Yuta Maezawa, Hironori Washizaki, Yoshinori Tanabe, and Shinichi Honiden, "Automated Verification of Pattern-Based Interaction Invariants in Ajax Applications", In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13), pp. 158–168, November 2013.

3. Yuta Maezawa, Kazuki Nishiura, Hironori Washizaki, and Shinichi Honiden, "Validating Ajax Applications Using a Delay-Based Mutation Technique", In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14), pp. 491–502, September 2014.

Chapter 1

Introduction

1.1 Background

Web applications have become essential platforms used in daily life such as information search, e-commerce, and social networking services. Web applications are built on the client-server architecture. Server-side technologies, e.g., Perl [80] and PHP [92], dynamically generate Web page data according to user requests. Currently, end-users demand rich user experience of Web applications [21]. Accordingly, client-side asynchronous JavaScript and XML (Ajax) technologies [25] are increasingly important. Web applications with Ajax technologies (Ajax Web applications) can handle user actions in an event-driven manner, asynchronously retrieve Web page data from the servers, and dynamically update parts of a Web page, so that the applications continuously process user requests on the client side without page transitions. Thus, Ajax technologies improve the performance, interactivity, and responsiveness of Web applications, providing rich user experience [78]. As a result, they are an integral part of the most visited websites [2, 74], such as Google, Amazon, and Facebook, and can be credited with a 676.3% increase in end-users compared to a decade ago [64].

A key factor in attracting end-users is usability of Web applications [71]; usability criteria can be used to assess aspects of user experience [39]. Fortunately, developers can build Ajax Web applications with Ajax design patterns [49], which contain 70 comprehensive findings for increasing usability of Ajax Web applications. However, event-driven, asynchronous, and dynamic features of Ajax technologies make the contexts of running applications unpredictable. Despite concerted efforts by developers, not all possible behaviors of running applications can be predicted. Although developers intend to correctly implement Ajax design patterns, the unpredictable contexts might conceal faults that will violate the properties of the design patterns, decreasing usability. A problem with faults behind these unpredictable contexts is that developers have trouble in detecting faults during testing because they might cause actual errors only when complicated conditions are met [30]. Since it would be unrealistic to test the applications under all possible conditions, these faults are not easily detectable, and users might eventually encounter erroneous behaviors in the applications when a user environment meets the conditions.

Several studies have been conducted on state-based analysis and testing of Ajax Web applications; a state-based approach is effective in finding faults in Ajax Web applications [52]. Since the applications can interactively manipulate an interface by using the document object model (DOM) [94], some have succeeded in leveraging dynamic analysis techniques that can capture DOM instances at runtime and can regard them as states of the applications [53, 22, 4, 7, 61]. With

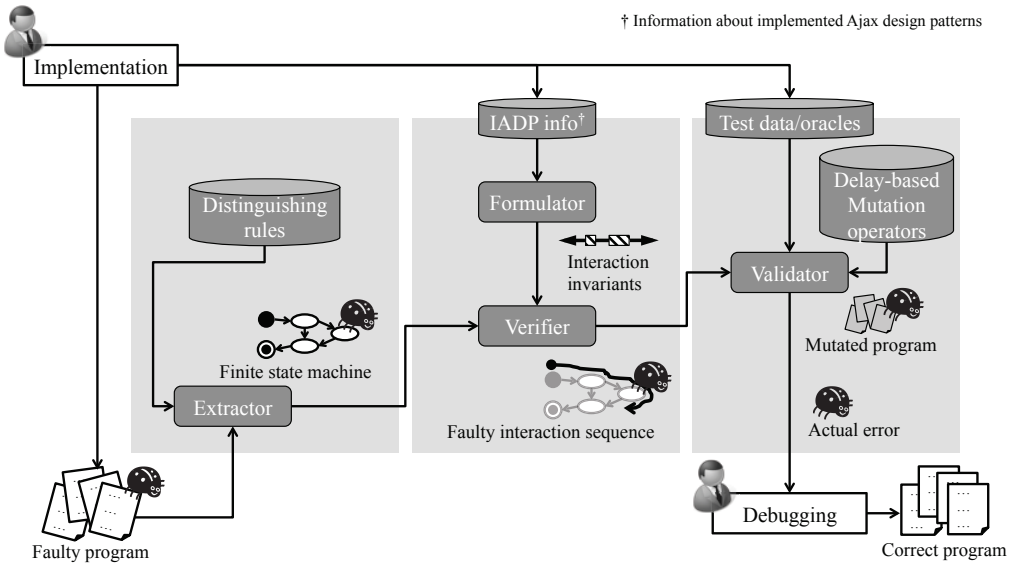


Figure 1.1: Overview of our proposed methods

the aid of the state space based on the execution results, this DOM-based testing can be used for effectively and efficiently identifying *executable faults* in a testing environment of developers.

Our motivation is that Ajax Web applications might have *potential faults*. Potential faults are those that seem to cause actual errors if executed; however, if these faults are not executed in a testing environment, developers have difficulty in detecting them, possibly resulting in users encountering actual errors. However, the DOM-based testing, which relies on the execution results of the applications, does not help verify the correctness in execution paths that are not part of the scenarios and environments given by developers. In contrast, static approaches might have limitations in analyzing all possible DOM instances because interactive DOM manipulation inevitably lead to a state space explosion; for example, an input form value, which is one of the DOM instance constituents, can take all possible string values. Therefore, the challenge from the viewpoint of software engineering is that we need to find a proper aspect of Ajax Web applications, instead of the DOM, that allows to statically analyze stateful behavior of the applications and identify these potential faults.

1.2 Approach Overview

Figure 1.1 shows an overview of our approach in this study. We focus on *interactions* with Ajax Web applications; these interactions correspond to how the applications handle user events, asynchronous server responses, and timeouts, as shown in Figure 2.2. In addition to the fact that interactions implemented in the applications can be obtained from event handlers in the source code, we assume that these interactions act as triggers that can change the application states; hence, the interaction may be the proper aspect on state-based analysis to look for potential faults in Ajax Web applications. Herein, we propose the following methods to support developers in finding and debugging potential faults before actual errors will be exposed in a user environment, i.e., to support preventive maintenance of Ajax Web applications. These methods are implemented in a tool called JSPreventer.

1.2.1 Extraction Method

Developers might incorrectly implement Ajax design patterns in Ajax Web applications due to event-driven, asynchronous, and dynamic features of applications (`Implementation`). For modeling these features, JSPreventer statically extracts a finite state machine from HTML, JavaScript, and CSS codes of Ajax Web applications (`Extractor`). JSPreventer parses the client-side HTML, CSS, and JavaScript codes to find the interactions implemented as transitions of a finite state machine. However, since pure HTML, CSS, and JavaScript parsers cannot distinguish code fragments of event handlers corresponding to the interactions, we define rules for distinguishing interactions implemented in the applications (`Distinguishing rules`). Finally, JSPreventer constructs an interaction-based finite state machine. Although developers can use the finite state machine to manually determine whether the applications run as expected, it does not allow them to exhaustively find faulty behaviors in the finite state machine.

1.2.2 Verification Method

Towards automatic detection of faulty behaviors in the finite state machine, JSPreventer uses the NuSMV model checker [13], which verifies the correctness of nondeterministic automata. Since the NuSMV model checker cannot determine correct and incorrect behaviors of Ajax Web applications, we assume that Ajax design patterns provide invariants relevant to the interactions (`interaction invariants`). Developers can store information about implemented Ajax design patterns (`IADP info`) into a repository when building Ajax Web applications. JSPreventer instantiates interaction invariants with the guided IADP info (`Formulator`). It then runs the NuSMV model checker to verify the correctness of the extracted finite state machine with the invariants (`Verifier`). If the finite state machine does not satisfy the invariants, JSPreventer obtains faulty interaction sequences from counterexamples of the verification results, and reports the presence of "potential faults" that seem to cause actual errors if executed.

1.2.3 Validation Method

Since Ajax design patterns are aimed to improve the usability of Ajax Web applications, code violations against the design patterns do not always lead to actual errors being debugged. Additionally, the identified faulty interaction sequences are counterexamples in the extracted finite state machine, not in the actual code. Counterexamples in an abstract model can be spurious; hence, it is important to reanalyze them in an actual system [98]. Herein, JSPreventer attempts to find executable evidence of potential faults for validating Ajax Web applications (`Validator`). However, it may not easily execute Ajax Web applications on the faulty interaction sequences because a specified environment does not meet specific conditions to reveal actual errors due to these potential faults. Therefore, we assume that an unexpected network latency, which may cause severe problems in Ajax Web applications [105], may make potential faults executable. To emulate an unexpected network latency, we define synchronous and asynchronous delay-based mutation operators (`Delay-based mutation operators`). Although a program mutation technique is commonly used for injecting artificial faults [43], we leverage the technique to allow JSPreventer to make potential faults executable in the specified environment.

Finally, JSPreventer outputs the extracted finite state machine, identified faulty interaction sequences, and revealed actual errors. We expect that devel-

opers can debug the applications by using these outputs (`Debugging`). Consequently, we argue that our proposed methods can help developers conduct preventive maintenance on Ajax Web applications.

1.3 Contributions

We now summarize our contributions in this study. First, we propose the three methods below. Although state-of-the-art studies have proposed effective and efficient testing methods for identifying executable faults in Ajax Web applications, our proposed methods work for identifying potential faults that are not easily executable in a given environment.

- A static extraction method of stateful behavior.
- A verification method of interaction invariants.
- A validation method using a delay-based mutation technique.

Additionally, we implement the following assets in JSPreventer for our proposed methods.

- Distinguishing rules based on HTML, CSS, and JavaScript language and library specifications.
- A fundamental set of interaction invariants based on Ajax design patterns.
- Synchronous and asynchronous delay-based mutation operators.

Finally, we discuss case studies we conducted and evaluate the usefulness of our proposed methods.

- A preliminary case study whose results show that seven participants had difficulty in addressing potential faults in Ajax Web applications.
- A case study on three real-world applications demonstrating that JSPreventer can reveal actual errors due to potential faults in Ajax Web applications.
- Some of the revealed actual errors that are difficult to expose using testing techniques and might cause severe vulnerabilities in the applications.

1.4 Organization

The rest of this thesis is organized as follows.

Chapter 2 We provide background on development of Ajax Web applications, including a motivating example to explain our methods below. In Section 2.7, we define research questions addressed in this study.

Chapter 3 We first propose a static method for extracting a finite state machine from Ajax Web applications. For our extraction method, we define rules for distinguishing the interactions implemented in the applications in Section 3.2. We then explain a workflow of our extraction method in Section 3.3.

Chapter 4 Secondly, we propose a method for verifying the correctness of the extracted finite state machine. Towards automated verification, we leverage a model checking technique in Section 4.2. As behavior oracles of the interaction-based stateful behavior in Ajax Web applications, we define interaction invariants based on Ajax design patterns in Section 4.3.

Chapter 5 Thirdly, we propose a method for validating the applications using a program mutation technique. To make potential faults in the applications executable on the given environment, we develop synchronous and asynchronous mutation operators in Section 5.2.2, resulting that our validation method reveals actual errors due to the potential faults.

Chapter 6 We conduct two case studies to evaluate the usefulness of our proposed methods and discuss the experimental results in Section 6.1 and Section 6.2.

Chapter 7 We present related work to our proposed methods in domains of state-based analysis and testing of Web applications, JavaScript control flow analysis, design pattern verification, client-server codes traceability, mutation analysis and testing, debugging concurrent programs, and automated program repair.

Chapter 8 Finally, we conclude this study and indicate directions for future work.

Chapter 2

Background on Development of Ajax Web Applications

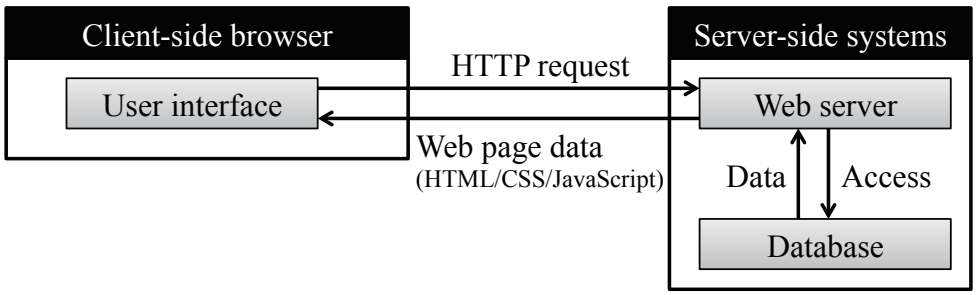
In this chapter, we first describe Asynchronous JavaScript and XML (Ajax) technologies that can be used to build responsive Web applications. We henceforth use the term *Ajax Web applications* instead of Web applications with Ajax technologies. We then explain interactions with the applications as an important development concern and Ajax design patterns that contain comprehensive findings observed in many real-world applications. Next, we give a description of software maintenance, especially, preventive maintenance, as an integral part of the software development process. After that, we present state-of-the-art studies on state-based analysis and testing of Ajax Web applications and give a motivating example that is used to explain our proposed methods in Chapters 3, 4, and 5. Finally, we determine research questions addressed in this study.

2.1 Asynchronous JavaScript and XML (Ajax)

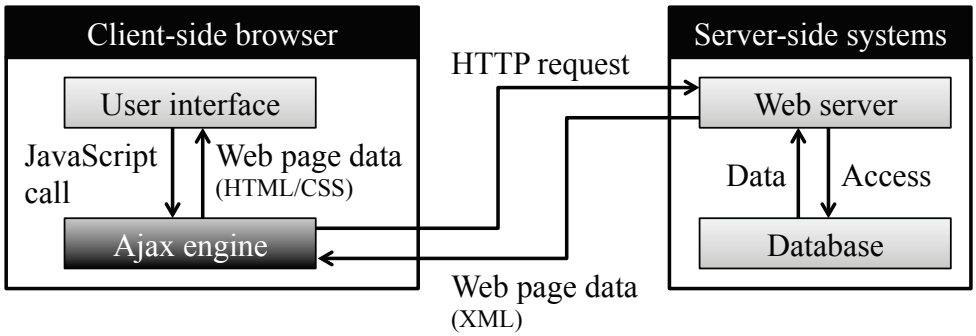
Jesse James Garrett introduced Asynchronous JavaScript and XML (Ajax) in 2005 as the best-of-breed approach to build responsive Web applications [25]. Ajax consist of the following existing technologies.

- JavaScript [66] is an object-oriented language for implementing business logic on the client-side of Web applications, and JavaScript programs bind up all the technologies below.
- HyperText Markup Language (HTML) [96] is used for defining the structure and content of Web pages. The presentational aspects of Web pages are supported by properties of Cascading Style Sheets (CSS) [93].
- Document Object Model (DOM) [94] provides APIs that allow JavaScript programs to dynamically manipulate the page structure, content, and presentation.
- Extensible Markup Language (XML) [95] and Extensible Stylesheet Language Transformations (XSLT) [97] are used for data interchange and manipulation.
- XMLHttpRequest (XHR) [68] is a JavaScript object that provides a method for asynchronous data retrieval.

Figure 2.1 shows a comparison between traditional and Ajax Web application models, as described in [25, Figure 1]. In both of these models, Web applications are built on the client-server architecture. Server-side technologies, e.g., Perl



(a) Traditional Web application model



(b) Ajax Web application model

Figure 2.1: Comparison between traditional and Ajax Web application models

[80] and PHP [92], dynamically generate Web page data[1] according to HTTP requests. Today, end-users demand rich user experience of Web applications [21]; hence, a traditional Web application exhibits problems due to the "page-based client model" [23], in which it refreshes whole of a Web page with round-trip server access in response to each user action, resulting in problems; for example, users cannot interact with the application during page transitions.

Here, client-side Ajax technologies can be used to solve the problems and are increasingly important in Web applications. By leveraging an Ajax engine on the client-side, an Ajax Web application can handle user actions in an event-driven manner, asynchronously retrieve Web page data from the servers, and dynamically update parts of a Web page, so that it can continuously process user requests on the client side without page transitions. Thus, Ajax technologies can improve performance, interactivity, and responsiveness of Web applications, providing rich user experience [78, 91].

As a result, they are an integral part of the most visited websites; Ocariza et al. examined the Alexa top 100 most visited websites [2, 74], such as Google, Amazon, and Facebook, and 97 of them used the client-side JavaScript, i.e., Ajax technologies. From the statistical point of view, Miniwatts Marketing Group publishes World Internet Users and Population Stats that indicates a 676.3% increase in end-users compared to a decade ago and the number of them exceeded 2.8 billions as of December 2013 [64]. Since Ajax technologies had not yet been introduced in this decade ago, they can be credited with this explosive growth of Web applications.

[1]Other resources, e.g., images, audios, and videos, can be interchanged via the HTTP(S) communications.

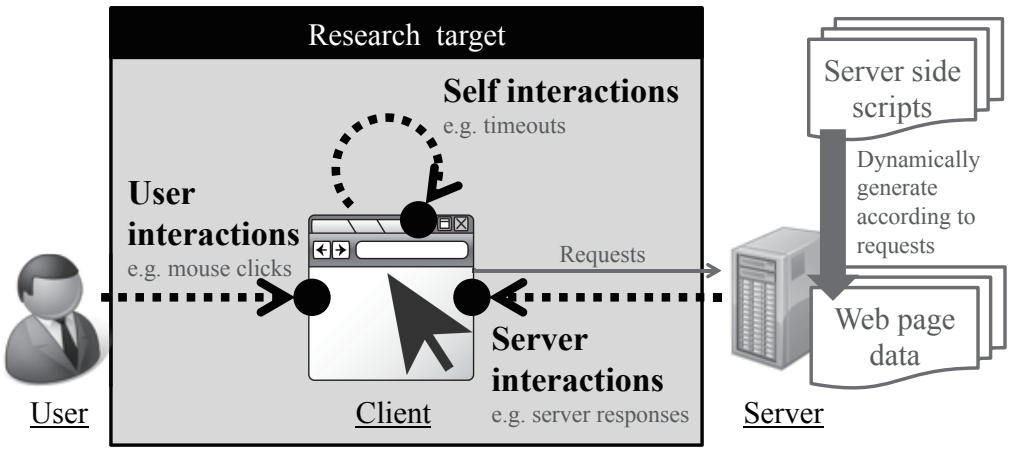


Figure 2.2: Interactions with Ajax Web applications

2.2 Interactions with Ajax Web Applications

Our research target is interactions with Ajax Web applications, as shown in Figure 2.2. When developing and maintaining Ajax Web applications, interactions with the applications need to be considered to improve user experience [23]. Since the applications are primarily aimed at providing rich user experience [21], developers are concerned with the following.

- **Event type:** Interactions the application can handle. We argue that interactions can be classified into user, server, and self interactions corresponding to nondeterministic elements such as user events, asynchronous server responses, and timeouts, as shown in Figure 2.2.
- **Callback function:** Application behavior when handling interactions.

Developers can also control whether the application will handle certain interactions. For determining application behavior, developers need to recognize the effects of enabling and disabling these interactions [33]. Hence, we can argue that the following is also a concern for developers.

- **Enable/Disable statement:** Application behavior when enabling and disabling interactions.

Unfortunately, developers have difficulties in correctly implementing interactions so that Ajax Web applications run as expected. This is because the interactions correspond to the nondeterministic elements and they make contexts of running applications unpredictable. Despite concerted efforts by developers, not all possible behaviors of running applications can be predicted, and these difficult-to-predict behaviors may be error-prone. Therefore, we address to extract a behavioral model, i.e., a finite state machine, representing interaction-based behavior from Ajax Web applications, as described in Chapter 3. We expect that developers can use the extracted model for understanding the interaction-based behavior and hopefully finding faults behind the unpredictable contexts of the applications.

2.3 Ajax Design Patterns

The success of modern Web applications lies in asynchronous technologies such as Ajax [24], because the event-driven, asynchronous, and dynamic features can

make the applications interactive and responsive, providing rich user experience with end-users [78]. Web usability is a key factor in attracting an increasing number of end-users [71]; usability criteria can be used to assess aspects of user experience [39]. Fortunately, developers can build Ajax Web applications with Ajax design patterns [49], which contain 70 comprehensive findings in terms of usability observed in many real-world Ajax Web applications. Thus, developers can leverage the Ajax design patterns for increasing usability of Ajax Web applications. Ajax design patterns are divided into the four categories below. For this study, we leverage the findings relevant to the interactions as interaction invariants, i.e., behavior oracles, in Ajax Web applications.

> **Foundamental technology patterns (11 patterns)** are the building blocks that differentiate Ajax from conventional approaches, [...] explain typical usage.
>
> **Programming patterns (23 patterns)** are the features of architecture and code that serve the software design principles [...]. These include, among other things, design of web services; managing information flow between browser and server; populating the DOM when a response arrives; and optimizing performance.
>
> **Functional and usability patterns (28 patterns)** are the things that matter to users, including widgets and interaction techniques; structuring and maintaining what's on the page; visual effects; and functionality that Ajax makes possible.
>
> **Development patterns (8 patterns)** are process patterns advising on best practices for development, as opposite to all the previous patterns, which are "things" that live inside as Ajax applications. The practices are about diagnosing problems and running tests. [sic]

However, even if developers intend to correctly implement Ajax design patterns, unpredictable contexts of running applications might conceal faults that will violate the properties of the design patterns. Although developers test whether the application runs according to the design patterns, testing techniques do not help verify the correctness of all execution paths. For example, the `User Action` design pattern suggests that applications should register user events at page load; we call such a property "`user event handler registration` (`UEHRegist`)" in Section 4.3. This is because executing user event callback functions before displaying Web page elements might result in erroneous behavior. In development and testing environments of developers, the application immediately completes the loading of all page elements; however, running the applications in a user environment might lengthen the loading time, resulting in an erroneous behavior. Such applications evolving over time might have an increasing risk of violating properties of design patterns [9]. Therefore, in Chapter 4, we try to verify whether an interaction-based behavior extracted from Ajax Web applications violates properties of Ajax design patterns implemented in the applications and suggest the presence of *potential faults* that seem to cause actual errors if executed. We expect that developers can debug the detected potential faults before users will encounter actual errors due to them.

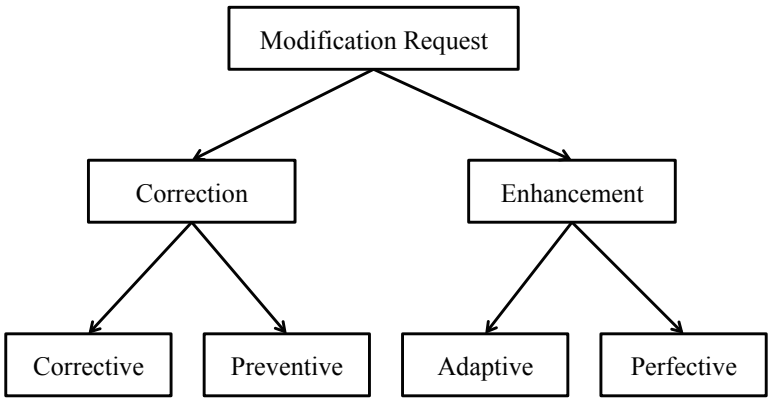


Figure 2.3: Types of software maintenance activities

2.4 Preventive Maintenance

Software maintenance is defined in ISO/IEC 14764:2006 [40] as "the totality of activities to provide cost-effective support to a software system", and it has been crucial in the software development process because its rising cost has not been negligible; 50-90% of software life cycle costs come from software maintenance [6]. According to the standard, developers propose modifications called *modification requests* for a software product being maintained, and these requests are identified as the following types. Figure 2.3 shows these identifications of software maintenance activities, as depicted in [40, Figure 1].

Adaptive maintenance the modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment.

Corrective maintenance the reactive modification of a software product performed after delivery to correct discovered problems.

Perfective maintenance the modification of a software product after delivery to detect and correct latent faults in the software product before they are manifested as failures.

Preventive maintenance the modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults.

Adaptive and perfective types are classified as *maintenance enhancement*, which is "a modification to an existing software product to satisfy a 'new' requirement"; Salehie et al. claimed 'dynamic or runtime' changes in a software product or its contexts as the basis for adaptation [84]. For an 'existing' requirement, another class is *maintenance correction* consisting of corrective and preventive types. Our aim is to prevent users from encountering actual errors due to potential faults detected, i.e., 'existing', in Ajax Web applications; thus, this study is directed to preventive maintenance of the applications.

To accomplish preventive maintenance of Ajax Web applications, developers need to debug detected potential faults. From the perspective of preventive maintenance, in 1985, Jim Gray devised the Bohrbug-Heisenbug hypothesis; software faults that developers have difficulty addressing can be classified as a Bohrbug or Heisenbug [28]. More recently, Michael Grottke et al. claimed that a Heisenbug can be a subtype of a Mandelbug and precisely defined these types as follows [29]:

Bohrbug An easily isolated fault that manifests consistently under a well-defined set of conditions, because its activation and error propagation lack "complexity" as defined below.

Mandelbug A fault whose activation and/or error propagation are complex. "Complexity" can be caused by

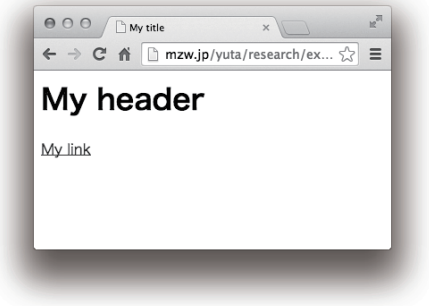
1. a time lag between the fault activation and the occurrence of a failure; or
2. the influence of indirect factors, i.e.,
   a) interactions of the software application with its system-internal environment (hardware, operating system, other applications); or
   b) influence of the timing of inputs and operations (relative to each other, or in terms of the system runtime or calendar time); or
   c) influence of the sequencing of operations; sequencing is considered influential, if the inputs could have been run in a different order and if at least one of the other orders would not have led to a failure.

A Bohrbug is an easy-to-detect fault because it is repeatedly exposed under a specific set of conditions; however, if the set is unknown, the fault is extremely difficult for developers to detect. A problem with a Mandelbug is that developers have trouble detecting it during testing because it causes an actual error only when complicated conditions are met [30]. If developers can specify exact conditions to reveal actual errors due to a Mandelbug, it becomes a Bohrbug, i.e., they may easily debug it with the specified conditions. Eventually, the challenge of debugging these faults is to specify the exact conditions.

Herein, our motivation of this study is that potential faults to be detected in Ajax Web applications are not easily executable during testing. Since the aforementioned complicated conditions may arise from unexpected user operations, Web browser behaviors, and network delays, these potential faults may have little chance of being executed during testing. Additionally, it should be noted that these potential faults do not always lead to actual errors being debugged because they correspond to code violations against Ajax design patterns that are in terms of the usability of Ajax Web applications. Therefore, we specify conditions to make potential faults executable and provide developers with executable evidence of not-easily-executable faults for validating the applications in Chapter 5. We expect that the executable evidence, if there is any, enables developers to debug potential faults, finally enabling preventive maintenance of Ajax Web applications.

2.5 State-Based Analysis and Testing

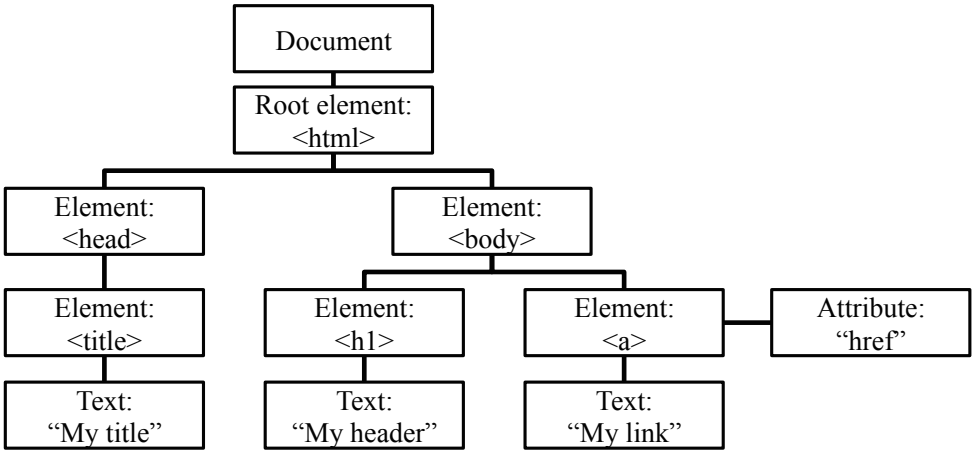
Many studies have been conducted on state-based analysis and testing to find faults in Ajax Web applications; a state-based approach may be more effective to find faults in the applications than navigation-model-based, code-coverage, or black-box ones [52]. The challenge of a state-based approach is to determine variables representing states of subject applications. Although Web pages can represent states of traditional Web applications [82], Ajax technologies enable the applications to interactively manipulate the content, structure, and presentation of a Web page; hence, Web page snapshots at runtime can represent states of Ajax



(a) Example of Web page

```
1 <html>
2   <head>
3     <title>My title</title>
4   </head>
5   <body>
6     <h1>My header</h1>
7     <a href="">My link</a>
8   </body>
9 </html>
```

(b) Example of HTML code



(c) Example of DOM tree

Figure 2.4: Example of DOM instance

Web applications. Figure 2.4 shows examples of Web page, HTML code, and DOM tree, as depicted in [52, Figure 1 and 2]. Herein, Marchetto et al. defined DOM instances and effects of callback executions as states and transitions of Ajax Web applications, respectively [52, 54]. DOM instances correspond to Web page snapshots and can be obtained in a form of tree structure through DOM APIs as states of the applications, whereas transitions between these states lie in DOM manipulations that are executed in callback function associated with user events or server responses.

The most successful work in this domain was conducted by Mesbah et al. whose tool is called Crawljax [60, 14]. Figure 2.5 shows results of this tool[2] from crawling on their portal website[3]. It simulates user events by finding fireable DOM elements, automatically executes Ajax Web applications, and extracts a DOM-based behavioral model, as shown in Figure 2.5. Such crawling of applications can be powerful for finding DOM-related faults such as dead clickable elements [59], detecting security vulnerabilities [8], cross-browser compatibility testing [58, 12], determining unnecessary CSS codes [57], or automated test generations [63]. Additionally, Artzi et al. argued that their technique of prioritizing event sequences might be helpful for Crawljax to effectively explore the state space [7]. Consequently, these state-of-the-art studies work for effectively and efficiently identifying 'executable' faults in development and testing environments.

[2] `http://crawls.crawljax.com/salt.ece.ubc.ca/#graph`
[3] `http://salt.ece.ubc.ca/`

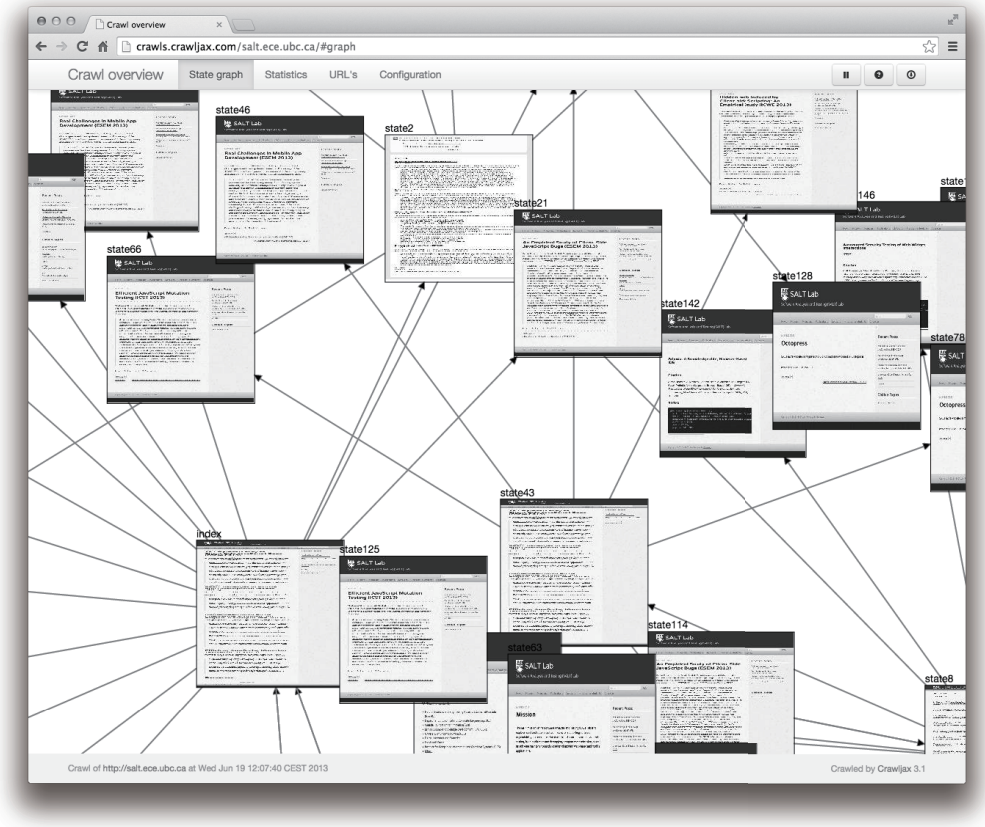


Figure 2.5: Screenshot of crawling results of Crawljax

However, all possible DOM instances cannot be extracted from Ajax Web applications because interactive DOM manipulations inevitably lead to a state space explosion. Although a dynamic analysis can be applicable for extracting a behavioral model based on actual DOM instances from execution results of the applications, such a DOM-based behavioral model might not contain the application behaviors that are not easily or cannot be executed in development and testing environments of developers. Therefore, we argue that existing DOM-based analysis and testing does not help developers address 'potential' faults in Ajax Web applications.

2.6 Motivating Example

We give the source code and screenshots of an Ajax Web application as a motivating example[4] in Figures 2.6 and 2.7. This is a typical Ajax Web application for shopping on a website where (a, b) users select options for an item and (c, d) add the item to their cart. This application has a fault that may cause duplicate orders on e-commerce websites. Such duplicate order problems have been reported in the troubleshooting of e-commerce services such as `Amazon`[5] and `eBay`[6]. We illustrate how developers implement and test this application using Ajax design patterns.

First, developers implement the option selection functionality based on the user event registration property. **(a) Page load:** An `onload` event is first eval-

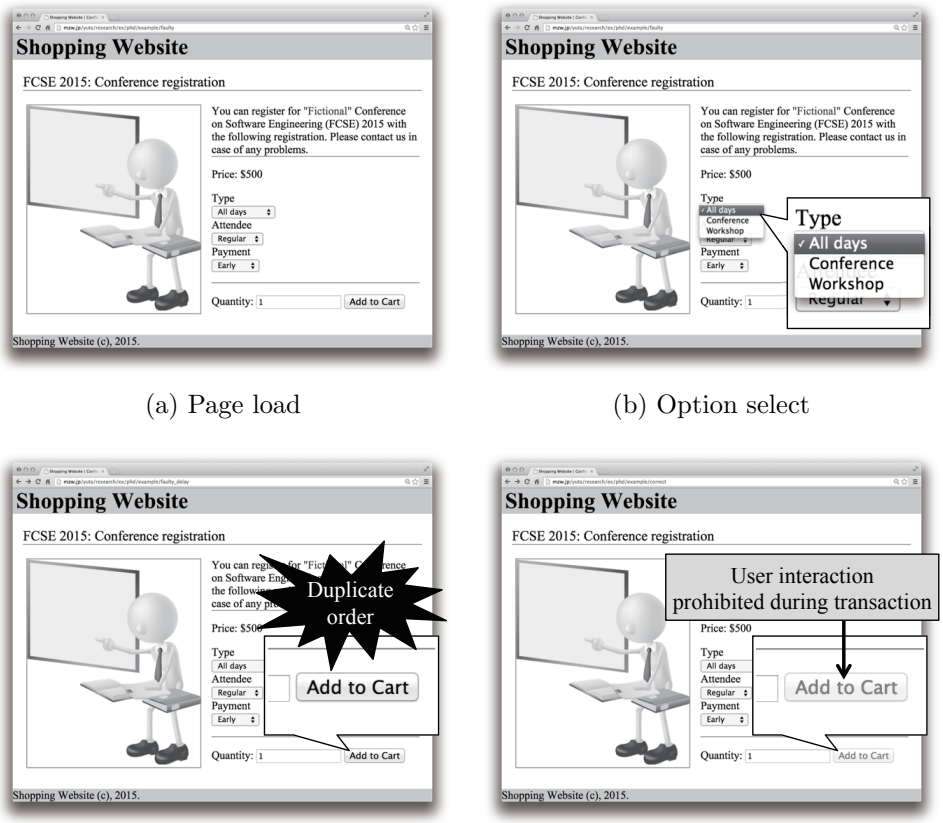
[4]Running examples are available from `http://mzw.jp/yuta/research/ex/phd/example/`
[5]`https://sellercentral.amazon.co.uk/forums/search.jspa?q=duplicate+order`
[6]`http://community.ebay.com/t5/forums/searchpage/tab/message?q=duplicate+order`

```
<html><head>...
  <!-- CSS and JavaScript codes loaded from external files -->
  <link rel="stylesheet" type="text/css" href="css/base.css" />
  <script type="text/javascript" src="js/jquery.js"></script>
  <!-- Embedded JavaScript code -->
  <script type="text/javascript"><!--//
/* User event handler registration */
window.onload = setUserEventHandlers;
function setUserEventHandlers() {
  document.getElementById("reg_type").onchange = calcPrice;
  document.getElementById("reg_attendee").onchange = calcPrice;
  document.getElementById("reg_payment")
    .addEventListener("change", calcPrice);
  document.getElementById("addcart").onclick = addCart;
};
function calcPrice() {
  var reg_type_value = document.getElementById("reg_type").value;
  /* Calculate and display total price */
};

/* User event handler singleton */
function addCart() {
/*  disableAddCard(); // proper disabling */
  if(isValidInput()) {
    reqRunTrans();
  } else {
    alert("Invalid user inputs");
/*  enableAddCart(); // proper enabling */
  }
};
function enableAddCart()  {
  document.getElementById("addcart").disabled = false;
};
function disableAddCart() {
  document.getElementById("addcart").attr(disabled, disabled);
};

function reqRunTrans() {
/* Asynchronous communication */
  jQuery.ajax({
    url: "runTrans.php",
    data: getParams(),
    success: succeeded
  });
};
function succeeded() {
  disableAll();
  jumpToConfirm();
};
...
//--></script>
</head><body>...
Price: $<span id="price">500</span>
<!-- Option select -->
<div>Type</div>
<select id="reg_type">
  <option id="all" value="350">All days</option>
  <option id="cnf" value="250">Conference</option>
  <option id="wsp" value="100">Workshop</option>
</select>
<div>Attendee...</select>
<div>Payment...</select>
<span>Quantity:</span>
  <!-- Inline CSS code -->
  <input id="quantity" style="width:25px;"...
<!-- Add to cart submit button -->
<input id="addcart" type="submit"  value="Add to Cart" />
...</body></html>
```

Figure 2.6: Source code of our motivating example: shopping website



(a) Page load

(b) Option select

(c) Add to cart

(d) Disabled add to cart

Figure 2.7: Screenshots of our motivating example given in Figure 2.6

uated when users visit the website (line 8). Then, the application calls back a function `setEventHandlers` (lines 9-15). **(b) Option select:** When users select options of an item, the browser evaluates an `onchange` event corresponding to the option widget (lines 10-14). In the interface, users can see the total price according to their selections, which is calculated at a callback function `calcPrice` of the events (lines 16-19).

Developers then visit the website and select the options for testing whether this functionality satisfies the user event handler registration property. Since the application displays the correct price, this test is successful.

Next, developers iteratively implement item addition functionality. To prevent the duplicate order problem, developers require the application of handling the add-to-cart click only once. The User Action design pattern also suggests that Ajax Web applications can prevent multiple calls of specific user event handlers; we call such a property "`user event handler singleton` (`UEHSingle`)" in Section 4.3. **(c) Add to cart:** Users can also add an item to their cart by clicking a submit button labeled `Add to Cart`. When the button is clicked, a `click` event occurs (line 14) and the application processes an `addCart` function (lines 22-30). If the selections are valid (line 24), the application sends an asynchronous request to run a transaction for taking inventories on the server side (lines 25 and 38-49). Otherwise, an alert box appears for users to enter valid inputs (lines 26-29). Finally, the application asynchronously receives a server response (lines 43 and 46-49) and jumps to a confirmation page (line 48).

To test the user event handler singleton property of the additional implementation, developers click the button with valid inputs and see that the application

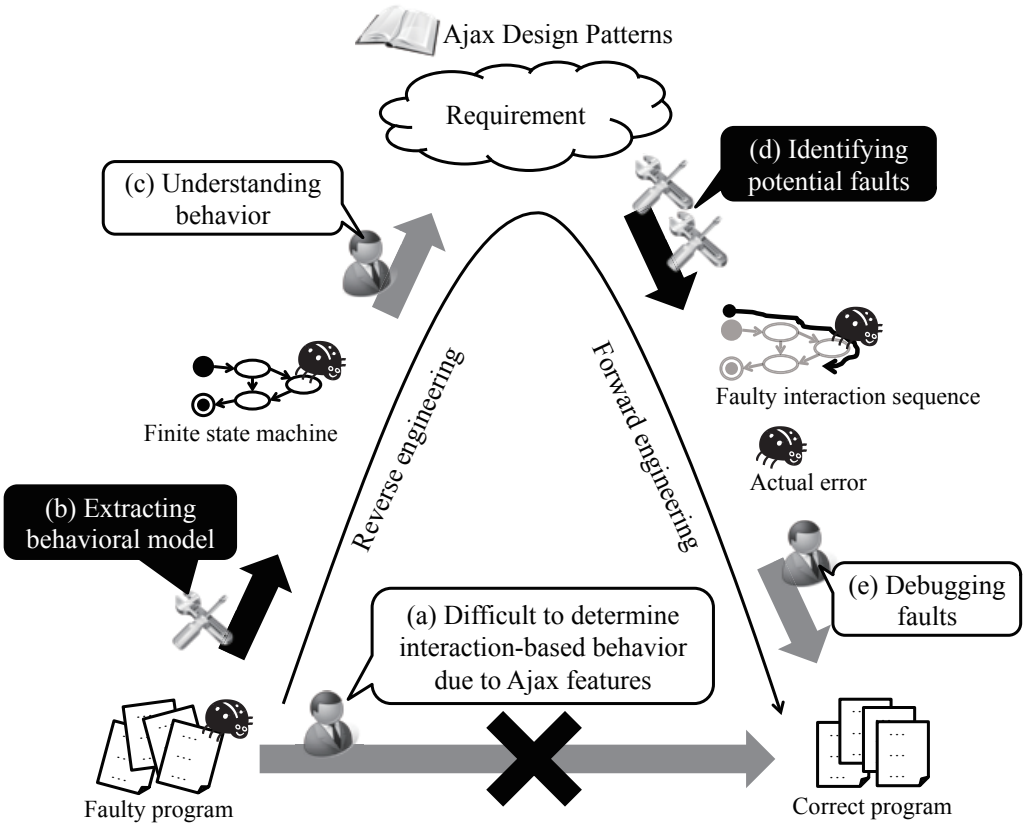


Figure 2.8: Overview of our approach

cannot handle the click due to the immediate jump. Since a previous test case also passed, developers finally confirm that the application expectedly runs according to user event handler registration and singleton properties derived from the User Action design pattern.

However, the duplicate order problem arises when users unexpectedly double-click the add-to-cart button. It is difficult to expose this duplicate order problem using a testing technique that leverages execution results. This is because the application does not execute such faulty paths in a reliable network and quickly processes the lightweight transaction. Otherwise, the duplicate order problem will be revealed in an actual user environment. **(d) Disabled add to cart:** To prevent the duplicate order problem, developers need to implement the appropriate enabling and disabling of the click so that users cannot interact with the button while the transaction is running (lines 23 and 28).

2.7 Challenges and Research Questions

Figure 2.8 shows an overview of our approach in this study. We use the reengineering lifecycle, as described in [16, Figure 1.1], because Demeyer et al. claimed that there may be little difference between software reengineering and maintenance. Here, reengineering consists of reverse and forward engineering, which they defined as follows:

> **Reverse engineering** is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.

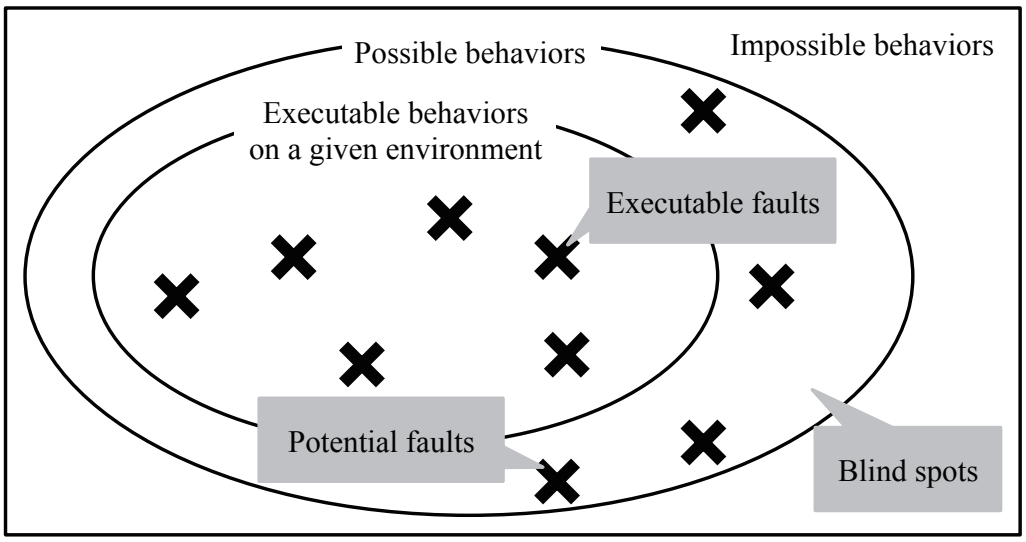


Figure 2.9: Venn diagram illustrating blind spots and potential faults

Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

If developers do not determine how a subject application runs, they will have difficulties in maintaining the application (a). For this situation, reverse engineering can be used to extract a behavioral model focusing on an aspect developers are concerned with (b). The extracted model is expected to help developers understand the application behavior (c). For forward engineering, developers have requirements for the application at the high-level abstractions. With the aid of the extracted model, they may identify potential faults against their requirements (d). Then, developers may debug the identified faults in the application (e).

For Ajax Web application development and maintenance, developers are concerned with interactions with the applications. However, Ajax event-driven, asynchronous, and dynamic features make contexts of running applications unpredictable so that developers have difficulty in determining interaction-based behavior in the applications (a). Therefore, we propose a method for extracting a finite state machine that represents interaction-based stateful behavior in Ajax Web applications (b). The challenge of our extraction method lies in *blind spots* of the application behaviors that may not be executable on given execution scenarios and environments. Figure 2.9 shows a Venn diagram illustrating the blind spots. As we explained in Section 2.5, state-of-the-art studies rely on DOM-based dynamic approaches and they cannot extract such blind spots from the execution results of the applications.

We now focus on the state transitions of Ajax Web applications. Interactive DOM manipulations are triggered when the applications handle user events or server responses [53]. This means that the interactions, as shown in Figure 2.2, can correspond to the state transitions of Ajax Web applications. Since the interactions consist of event types, callback functions, and enable/disable statements implemented in the source code, these constituents can be statically obtained from the source code. Therefore, we discuss our interaction-based extraction method relying on a static approach in Chapter 3.

In addition to reading the source code and reviewing runtime behavior of Ajax Web applications, we expect that developers can use the extracted finite state

machine to understand interaction-based stateful behavior in the applications (c). Hopefully, they can find faults at the blind spots, i.e., potential faults in the applications. Thus, we set the following research question for our extraction method.

RQ1 *Can our extraction method support developers in understanding an interaction-based stateful behavior containing blind spots in Ajax Web applications?*

Additionally, we assume that developers implement Ajax design patterns in Ajax Web applications for increasing usability of the applications. Even if developers intend to correctly implement the design patterns, unpredictable contexts might conceal faults, resulting in unexpected errors and decreasing usability. Although we claim that the extracted finite state machine can be useful for understanding application behavior, developers need to manually determine its correctness against properties of the design patterns, which does not enable them to exhaustively find potential faults. Although the NuSMV model checker [13] is useful for verifying the correctness of nondeterministic automata, the challenge of an automated verification is that there is no generic behavior oracles relevant to interactions implemented in Ajax Web applications. Therefore, we try to define correct and incorrect interaction-based behaviors described in Ajax design patterns as interaction invariants. In Chapter 4, we explain a method towards automated verification of pattern-based interaction invariants in Ajax Web applications.

Developers can use counterexamples from verification results, if there are any, for identifying faulty behaviors associated with potential faults (d). Since we expect that the identified faulty behaviors work as debugging clues (e), the research question below is added for our verification method.

RQ2 *Can our verification method report the presence of potential faults in Ajax Web applications?*

However, the identified faulty behaviors, i.e., code violations against Ajax design patterns, do not always lead to actual errors being debugged because the design patterns are aimed to improve the usability of Ajax Web applications. Additionally, the counterexamples in the extracted finite state machine can be spurious. Thus, it is important to validate whether actual errors are due to potential faults. The challenge of validating the application is to specify complicated conditions to reveal actual errors, such as unexpected user operations, Web browser behaviors, and network delays. Although testing all possible scenarios in every environment should reveal all errors, such a method would be unrealistic. It has been pointed out that unexpected network latency may cause severe problems in Ajax Web applications [105]; therefore, in Chapter 5, we propose a validation method for revealing actual errors that are caused when specific network delays are present (d).

Once our validation method succeeds in revealing the actual errors, it provides developers with executable evidence indicating that they are not executed without subtle network delays. With the aid of the executable evidence, developers can address them in a similar way of executable faults that many state-of-the-art studies have addressed (e). Therefore, we also provide the following research question for our validation method.

RQ3 *Can our validation method find executable evidence of potential faults in Ajax Web applications?*

To answer these research questions, we carried out two case studies and discuss their results in Chapter 6.

Chapter 3

Extracting Interaction-Based Stateful Behavior in Ajax Web Applications

In this chapter, we propose a method for statically extracting a finite state machine that represents an interaction-based stateful behavior in Ajax Web applications. Figure 3.1 depicts the workflow of our extraction method. It requires the following two inputs from developers:

- A URL where developers deploy an Ajax Web application to be analyzed.
- Rules for distinguishing interaction-related HTML, CSS, and JavaScript code fragments (distinguishing rules), as defined in Section 3.2.

Our extraction method involves a rule-based static analysis, as explained in Section 3.3. The extracted finite state machine is an output of our extraction method.

3.1 Overview

When developing and maintaining Ajax Web applications, developers are concerned with interactions implemented in the applications, as shown in Figure 2.2. However, developers have difficulty in determining interaction-based behavior in the applications. This is because the interactions consist of nondeterministic elements in the applications, such as user events, asynchronous server responses, and timeouts. Since these nondeterministic elements make contexts of running applications unpredictable, developers cannot predict all possible behaviors of

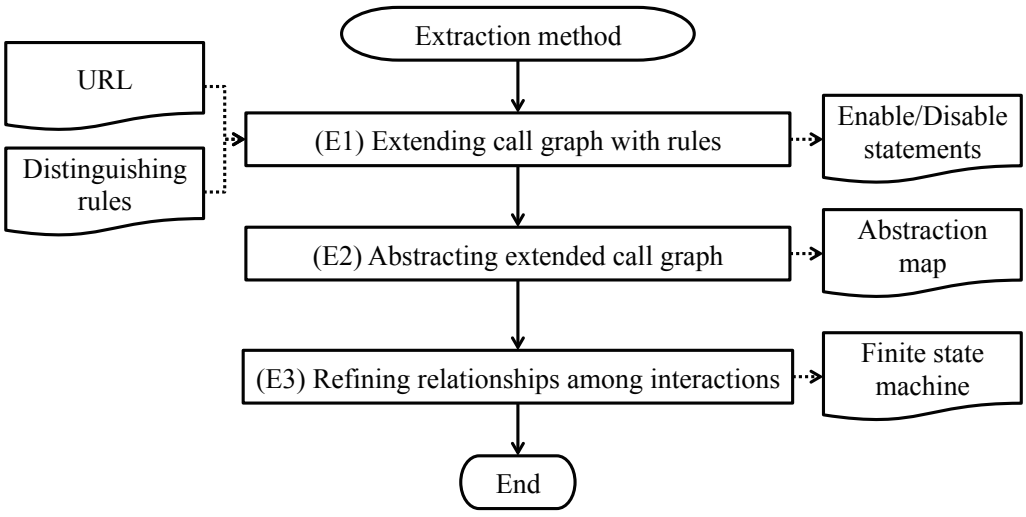


Figure 3.1: Extraction method workflow

running applications in spite of their greatest efforts. These difficult-to-predict behaviors may be error-prone; therefore, it is important to support developers in understanding interaction-based behavior in Ajax Web applications.

To find faults in Ajax Web applications, a state-based approach may be more effective than navigation-model-based, code-coverage, or black-box ones [53, 52]. Successful extraction of stateful behavior in Ajax Web applications is possible by regarding the document object model (DOM) [94] as *states of Ajax Web applications*. Considering that the aspect of interactive DOM manipulation in the applications, many researchers have relied on dynamic approaches to construct a finite state machine representing stateful behavior of the applications [53, 22, 4, 7, 61]. Although a dynamic analysis can be applicable for extracting a finite state machine based on actual DOM instances from the execution results of the applications, such a DOM-based finite state machine might not contain "blind spots" of application behaviors that may not be executable on given execution scenarios and environments. For example, it might not contain erroneous behavior due to communication failures if the analysis was done in a reliable network. In fact, state-of-the-art studies can be used for effectively and efficiently identifying 'executable' faults using the state space of the applications; however, the challenge of identifying 'potential' faults at the blind spots still remains.

Although a static analysis can be used for extracting stateful behavior containing such blind spots from Ajax Web applications, a DOM-based static approach might have limitations in analyzing all possible DOM instances in the applications. This is because the interactive DOM manipulations inevitably lead to a state space explosion. We now focus on *state transitions of Ajax Web applications*. Interactive DOM manipulations are triggered when the applications handle user events and asynchronous server responses [53]; i.e., the interactions can correspond to the state transitions of the applications. Therefore, we assume that

Assumption 1. *interactions implemented in Ajax Web applications act as triggers that can change the application states.*

As described in Section 2.2, the interactions consist of event types, callback functions, and enable/disable statements implemented in the source code; hence, all these constituents can be obtained in a static manner. However, program parsers cannot distinguish these constituents from other code fragments. Therefore, we first define rules for distinguishing these constituents in the source code of Ajax Web applications in Section 3.2. In Section 3.3, we then discuss a static method for extracting stateful behavior in Ajax Web applications under our Assumption 1. Our extraction method extracts a finite state machine based on all possible interactions implemented in the source code. Such an interaction-based finite state machine might contain erroneous behaviors due to potential faults in Ajax Web applications.

3.2 Distinguishing Rules

Developers implement interactions in Ajax Web applications as callback functions according to event firing. As we mentioned in Section 2.2, the interactions consist of event types, callback functions, and enable/disable statements. However, pure HTML, JavaScript, and CSS parsers[1] cannot distinguish them from

[1]We use jsoup [44], Rhino [67], and CSS Parser [15] to parse HTML, JavaScript, and CSS codes, respectively.

Table 3.1: Distinguishing rules

Rule class	HTML attribute	JavaScript element	CSS property
Trigger	Event handler	Callback object	-
Function	-	Event handling function	-
Control	Control and hidden	DOM manipulation function	Display and visibility

Table 3.2: Distinguishing rule examples in XML format

Rule class	Example	XML description
Trigger	`onchange`	`<Trigger interact="User" event="onchange" />`
	`success`	`<Trigger interact="Server" event="success" repeatable="false" />`
Function	`addEvent Listener`	`<Function func="addEventListener" event="arg_0" callback="arg_1" target="PropTarget" />`
Control	`disabled`	`<Control attr="disabled" disabled="true" />`
	`attr`	`<Control func="attr" prop="arg_0" value="arg_1" cond="$prop=='disabled'" disabled="$value=='disabled'" />`
	`display`	`<Control prop="display" disabled="none" />`

other ignorable code fragments in the source code. Therefore, we define rules for distinguishing such interaction-related code fragments as distinguishing rules. Tables 3.1 and 3.2 list three classes of distinguishing rules and their examples in the XML format, respectively. The XML notations used in the distinguishing rules are listed in Table A.1. Here, we explain these rules using code snippets, as shown in Figure 3.2, from the source code of our motivating example described in Figure 2.6.

3.2.1 Trigger Rule

In Ajax Web applications, the interactions are implemented at statements that assign callback functions to event types. In Figure 3.2a, for example, developers implement the user interaction of the option selection at the `reg_type` element by using the `onchange` event type and `calcPrice` callback function (line 10). Our motivating example handles the `success` event type and calls back the `succeeded` function (line 43) as the server interaction. Since the JavaScript parser regards these event types as just property syntax elements, it cannot distinguish them from the `value` property (line 17). Thus, we define trigger rules by using `<Trigger>` whose `event` attribute has HTML event handler attributes[2] or JavaScript callback objects[3]. As for its `repeatable` attribute, this represents whether the event is enabled or disabled after the applications handle it. For example, once the application handle the `success` event, this event is disabled until the `jQuery.ajax` function is invoked (line 40); hence, the `repeatable` attribute of the `success` event is set to "false". With the aid of these trigger rules, our ex-

[2] `www.w3.org/TR/html5/webappapis.html#event-handler-attributes`
[3] `api.jquery.com/?s=callback+object`

```
10| document.getElementById("reg_type").onchange = calcPrice;
```

```
17| var reg_type_value = document.getElementById("reg_type").value;
```

```
40|   jQuery.ajax({
41|     url: "runTrans.php",
42|     data: getParams(),
43|     success: succeeded
44|   });
```

(a) Event handler attribute and callback object being defined in trigger rules

```
12| document.getElementById("reg_paymeny")
13|   .addEventListener("change", calcPrice);
```

(b) Event handling function being defined in function rule

```
32| document.getElementById("addcart").disabled = false;
```

```
35| document.getElementById("addcart").attr(disabled, disabled);
```

(c) Control attribute and DOM manipulation function being defined in control rules

Figure 3.2: Code snippets from Figure 2.6 for distinguishing rules

traction method finds them as event types and analyzes their callback functions in assignment statements.

3.2.2 Function Rule

Developers also leverage JavaScript built-in or library functions for interaction implementations. Figure 3.2b shows a code snippet corresponding to such the built-in function implemented in our motivating example. This `addEventListener` function attaches the `calcPrice` callback function at its first argument to the `change` event type at its zeroth argument (lines 12-13). Such implementations also cannot be distinguished from other function calls such as `reqRunTrans` (line 25). Therefore, we call functions whose arguments are event types or callback functions as `event handling functions` and input API information of these functions as function rules to our extraction method. The function rule is described using `<Function>` whose `func`, `event`, `callback`, `target` attributes have the function name, event type source, callback function source, and target element source, respectively. Our extraction method then leverages the function rules to distinguish the interaction implementations at function calls.

3.2.3 Control Rule

Additionally, developers implement the enable/disable statements by using assignment or function call statements as well as the interaction implementations. As shown in Figure 3.2c, our motivating example enables and disables the add-to-cart button (lines 32 and 35). To distinguish these enable/disable statements, we describe control rules by using `<Control>` whose `attr` attribute has HTML control[4] or hidden[5] attribute and `disabled` attribute has a value for disabling HTML elements. Note that the hidden effect is typically implemented using CSS, so we

[4] `http://www.w3.org/TR/html401/interact/forms.html#h-17.12`
[5] `http://www.w3.org/TR/html5/editing.html#the-hidden-attribute`

also describe <Control> whose prop attribute is CSS display[6] or visibility[7]. JavaScript functions can manipulate these HTML attributes and CSS properties. Thus, we describe cond in <Control>, which indicates whether the function identified by func manipulates these HTML attributes and CSS properties, so that our extraction method can distinguish enable/disable statements by using function calls.

We define these distinguishing rules based on HTML, JavaScript, and CSS language specifications provided by The World Wide Web Consortium (W3C); therefore, these distinguishing rules are application independent. We also refer to JavaScript library specifications, such as jQuery[8] and Prototype[9], and define library-dependent distinguishing rules. This follows the same strategy as the Externs[10] in Google Closure Compiler. This is because these libraries are powerful for implementing rich user experience in Ajax Web applications, but the code is too complex to statically analyze their behavior.

Our extraction method leverages the distinguishing rules to extract all interactions with Ajax Web applications in HTML, JavaScript, and CSS codes. However, an finite state machine by combining the interactions might contain many impossible behaviors, resulting in a state space explosion. Therefore, our extraction method analyzes the relationships among the interactions (interaction relationships) to more precisely extract a finite state machine according to actual Ajax Web application behavior.

3.3 Rule-Based Static Analysis

To obtain possible interaction relationships, our extraction method statically analyzes HTML, JavaScript, and CSS codes with distinguishing rules. As described in Assumption 1, We assume that interactions with Ajax Web applications, as shown in Figure 2.2, act as triggers that can change the states of applications; the interaction that we focus on in this study corresponds to a function call in response to an event firing. Hence, states and transitions in the extracted state machines represent function calls and the relationships between them. Our rule-based static analysis consists of three steps, as shown in Figure 3.1 (E1, E2, and E3).

3.3.1 Source Code Retrieval

As a preliminary to the rule-based static analysis, our extraction method retrieves HTML, JavaScript, and CSS codes. Figure 3.3 shows code snippets illustrating code locations of these codes. It first retrieves HTML code from the Web server by using a given URL and parses the HTML code with the jsoup [44] parser to access all HTML elements through a DOM tree. Our extraction method simultaneously reads XML files containing the distinguishing rules. While parsing the HTML code, our extraction method retrieves inline, embedded, and external JavaScript and CSS codes by analyzing their locations, as shown in Table 3.3. For the event handler attributes, our extraction method leverages the trigger rules. For example, in Figure 3.3a, two <script> elements correspond to external and embedded JavaScript codes, respectively (lines 4 and 6-51). Our motivating

[6] http://www.w3.org/TR/css-display-3/
[7] http://www.w3.org/wiki/CSS/Properties/visibility
[8] http://jquery.com
[9] http://prototypejs.org/
[10] https://code.google.com/p/closure-compiler/wiki/ExternsForCommonLibraries

Table 3.3: JavaScript and CSS code locations in HTML code

Language	Type	Code location
JavaScript	Inline	Event handler attribute of HTML element
	Embedded	Inner HTML element of `<script>`
	External	`src` attribute of `<script>`
CSS	Inline	`style` attribute of HTML element
	Embedded	Inner HTML element of `<link>` element
	External	`href` attribute of `<link>` element

```
4| <script type="text/javascript" src="js/jquery.js"></script>
6| <script type="text/javascript"><!--//
50| ...
51| //--></script>
```

(a) External and embedded JavaScript code locations

```
65| <input id="quantity" style="width:25px;" ...
```

(b) Inline CSS code location

Figure 3.3: Code snippets from Figure 2.6 for HTML and CSS code locations

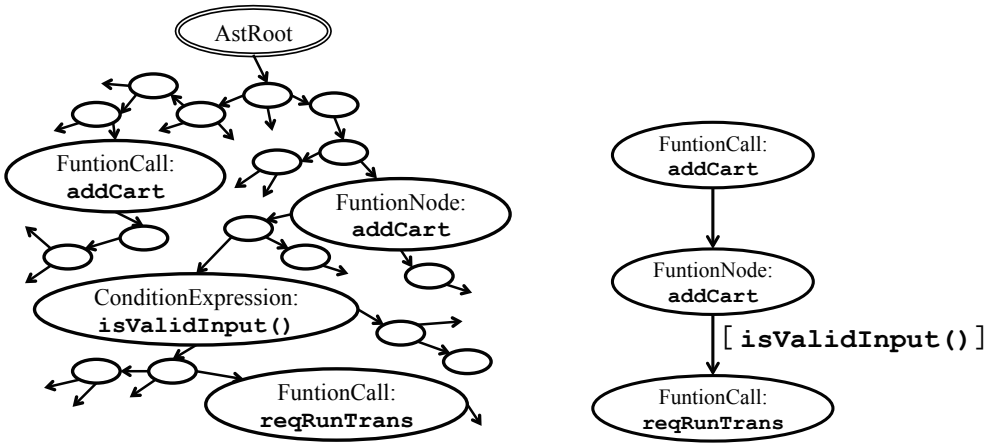
example sets a stylesheet parameter with an inline CSS code (line 65), as shown in Figure 3.3b. When detecting these JavaScript and CSS codes, our extraction method parses them with the `Rhino` [67] and `CSS Parser` [15], and instantiates abstract syntax trees (ASTs) and CSS rules, respectively. Thus, our extraction method can analyze the application behavior and presentation.

3.3.2 Extending Call Graph with Rules

Since the interactions are function callbacks in response to event firing, our extraction method leverages a call graph that represents function caller-callee relationships to obtain possible interaction relationships. Figure 3.4 shows an example of function caller-callee relationships in a call graph. The function caller and callee are represented using `FunctionNode` and `FunctionCall` elements in the ASTs. By traversing the ASTs, our extraction method detects these elements and makes edges between them. It also obtains conditions for calling these functions from the `ConditionExpression` elements and sets them on corresponding edges. Thus, our extraction method represents the function caller-callee relationships as a call graph.

However, the call graph in its original form does not contain the interaction relationships (e.g., event firing and function callbacks). Therefore, our extraction method extends the call graph in terms of the interactions, as shown in Figure 3.5.

Binding JavaScript codes Note that the call graph has its root node regardless of the ASTs. For binding inline, embedded, and external JavaScript codes as application behavior, our extraction method adds an edge from the root node of the call graph to the `AstRoot` element in each AST.



(a) Abstract syntax tree (AST)

(b) Example of caller-callee relationships

Figure 3.4: Example of function caller-callee relationships represented in call graph

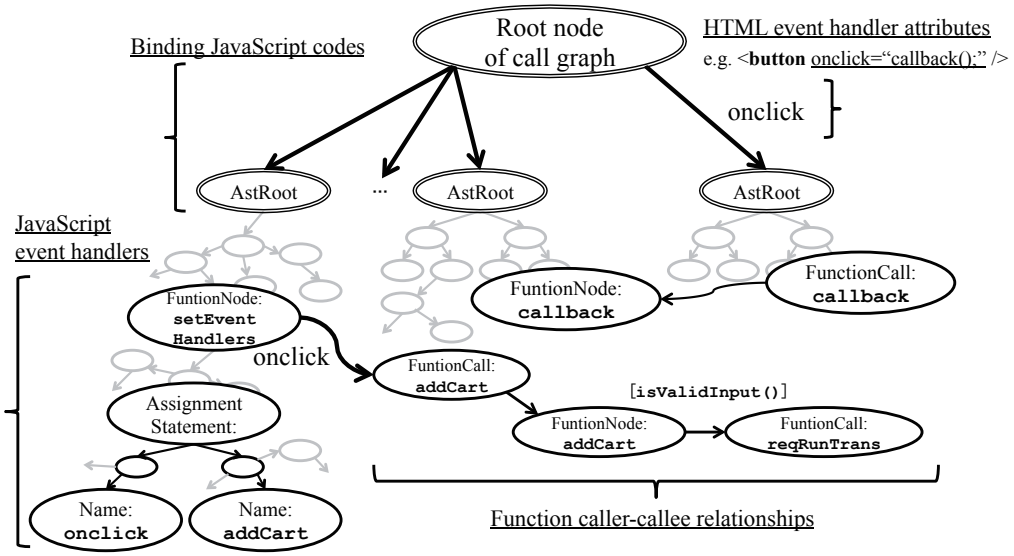
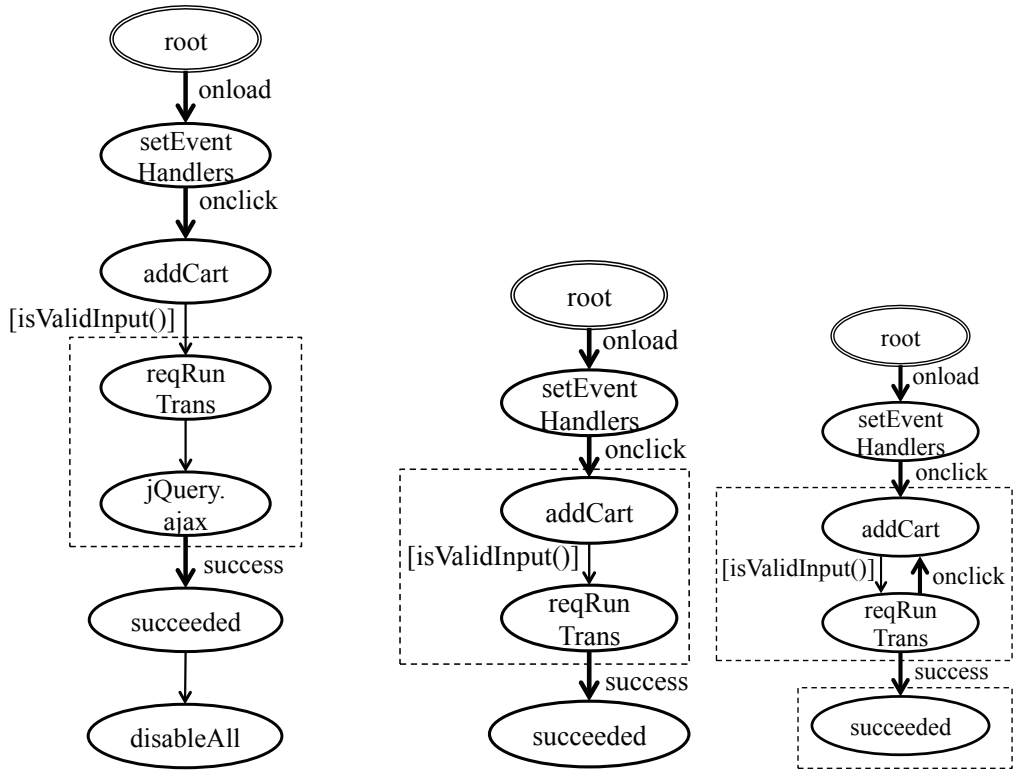


Figure 3.5: Interaction-based extensions to call graph

HTML event handler attributes For the AST from the inline JavaScript code, our extraction method additionally sets its HTML event handler attribute on the edge which is added while the binding. As a temporary example, we give a HTML code snippet at the upper right of Figure 3.5. The `onclick` and `callback` function call are the HTML event handler attribute and inline JavaScript code, respectively. Hence, our extraction method sets the `onclick` event handler attribute on the added edge.

JavaScript event handlers While our extraction method traverses the ASTs, it uses distinguishing rules as a means of detecting event types, callback functions, and enable/disable statements. Note that the enable/disable statements are used in the last step (E3), as described in Section 3.3.4. As interaction relationships, our extraction method adds edges from register functions to callback functions and sets event types on the edges. For our motivating example, `setEventHandlers` and `addCart` functions correspond to the register and callback functions, respectively (lines 9 and 14 in Figure



(a) Extended call graph (b) Abstracted call graph (c) Refined call graph

Figure 3.6: Partial example of finite state machine constructed

2.6). Our extraction method assigns the `onclick` event type on the edge between them, as shown in Figure 3.5.

In this way, our extract method extends a call graph containing interaction relationships. For this purpose, it leverages the distinguishing rules and analyzes HTML and JavaScript codes of an Ajax Web application in a static manner. Figure 3.6a shows a part of an extended call graph of our motivating example. However, the extended call graph might contain many relationships irrelevant to the interactions.

3.3.3 Abstracting Extended Call Graph

Our extraction method, therefore, abstracts elements irrelevant to the interactions in the extended call graph to obtain relationships that focus on the interactions. It first instantiates a map for storing the abstraction information (`abstraction map`), as shown in Figure 3.7. The abstracted nodes and edges are values in the abstraction map, and their key is a corresponding node in the abstracted call graph. When developers want to grasp the correspondence relations between the nodes in the call graph and the elements in the source code, our extraction method can identify the node corresponding to the element with the aid of the abstraction map.

Our extraction method then traverses the extended call graph from its root node. When finding a relationship without an event type and/or conditions, our extraction method removes the relationship and the callee node from the extended call graph. It simultaneously stores them in the abstraction map as values and a key of the values is the caller node. If the extended call graph has other

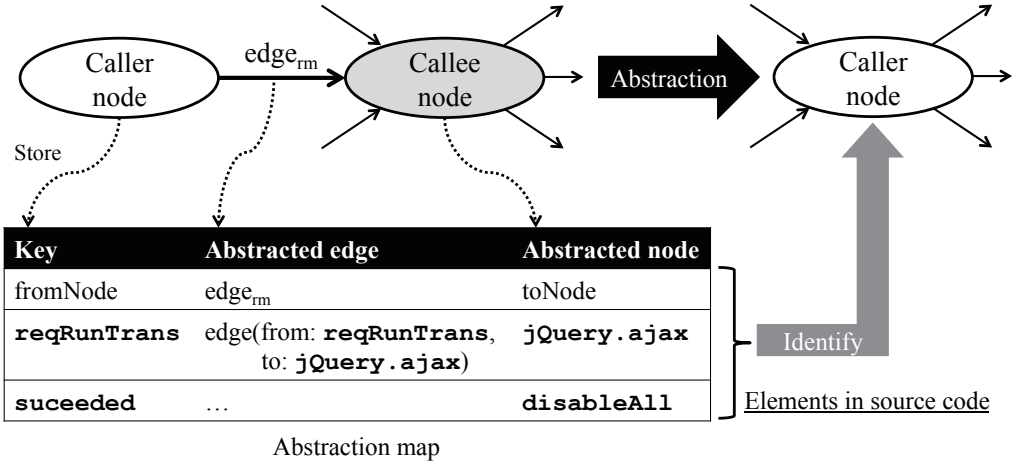


Figure 3.7: Abstraction map

relationships connected with the callee node, our extraction method replaces the connections with the caller node. For example, in Figure 3.6a, our motivating example runs from `reqRunTrans` to `jQuery.ajax` without any interactions (the dashed box area). In this case, our extraction method abstracts this caller-callee relationship into the corresponding invoked function of `reqRunTrans`, as shown in Figure 3.6b. Thus, it can obtain possible interaction relationships from the extended call graph.

3.3.4 Refining Relationships among Interactions

Developers can also implement enable/disable statements in Ajax Web applications, which is important for precisely extracting Ajax Web application behavior [33]. Our motivating example enables and disables the add-to-cart button on lines 32 and 35, respectively, as the enable/disable statements. Our extraction method leverages control rules to find these `disabled` attributes and their values from the ASTs. It also finds enable/disable statements in CSS codes from statements for manipulating the `style` attributes of HTML elements. Our extraction method then associates these enable/disable statements with corresponding nodes in the abstracted call graph by using the abstraction map. Thus, each node in the abstracted call graph possesses information for our extraction method to determine whether the interactions are enabled or disabled,

Our extraction method adds possible interaction relationships and removes impossible ones at each node. It first determines what interactions are registered at each node. If an interaction is repeatable and registered at a parent node, its child nodes also handle the interaction. For example, in Figure 3.6b, our motivating example registers the repeatable `onclick` at the `setEventHandlers`; therefore, `reqRunTrans` and `succeeded` have it. Note that our extraction method does not add the interaction at the branch node (`addCart`) because it represents only conditions for proceeding to the child node (`reqRunTrans`). Our extraction method then analyzes the enable/disable statements at each node. When an interaction is disabled or hidden, our extraction method removes the interaction. For example, in Figure 3.6c, our extraction method adds an `onclick` edge from `reqRunTrans` to `addCart` (the upper dashed box area), but it removes such an edge from `succeeded` (the lower dashed box area) because all interactions are disabled there (line 47 in Figure 2.6).

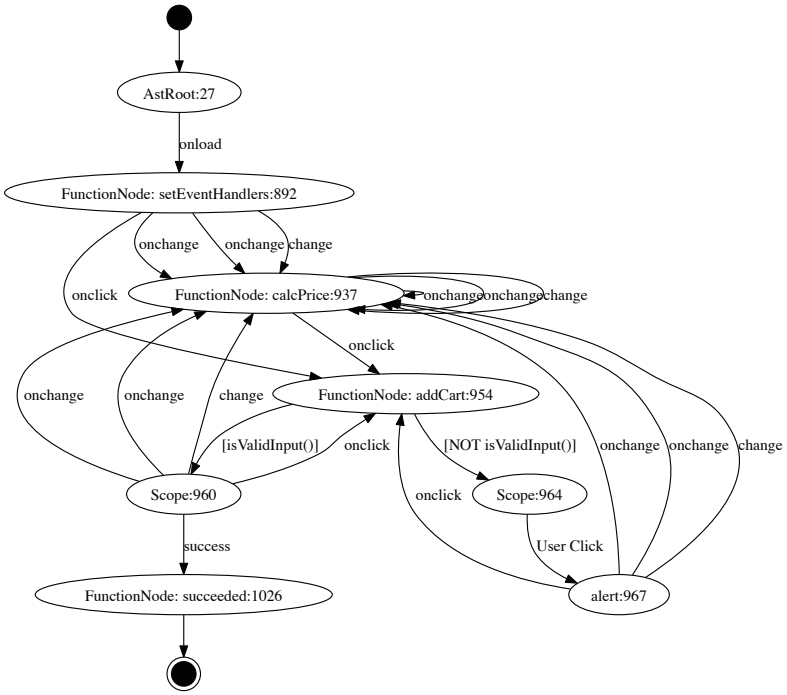
Finally, our extraction method constructs a finite state machine based on the

refined interaction relationships as its output. Since our extraction method relies on a static approach, the extracted finite state machine may contain correct and wrong behaviors of Ajax Web applications that are independent of any execution scenarios and environments. Therefore, the extracted finite state machine can support developers to understand the application behaviors even if they are not easily executed on their environments, i.e., blind spots.

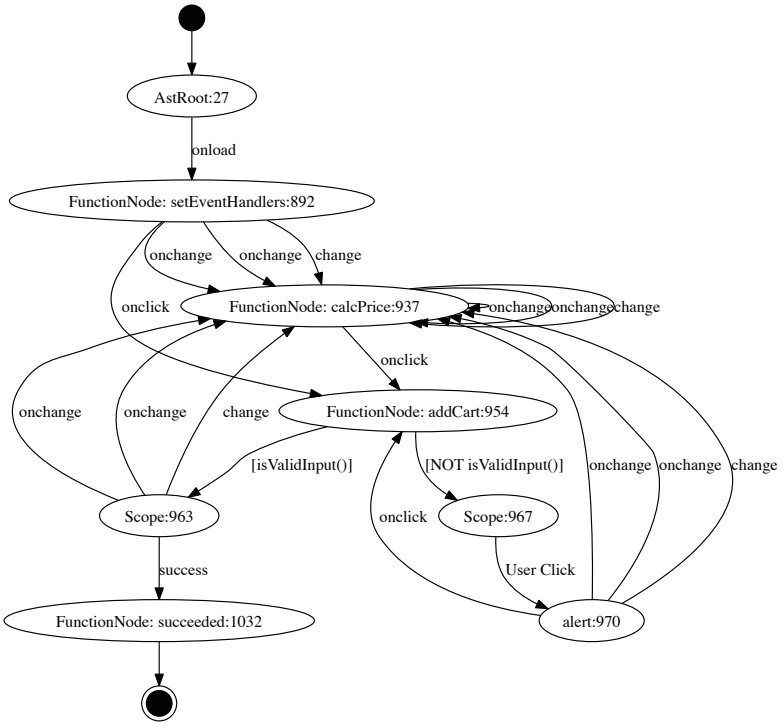
3.4 Use Scenario and Results on Motivating Example

Figure 3.8 shows two results of our extraction method running on our motivating example. In the finite state machine extracted from the faulty version, as shown in Figure 3.8a, developers might find an edge associated with the `onclick` event from the `Scope:960` to the `addCart` nodes, and identify the duplicated order problem in our motivating example; our motivating example might handle the double-click on the add-to-cart button, resulting in the problem. With the aid of their identification, developers then modify our motivating example so as to make the button disabled while asynchronously communicating with the server (lines 23 and 28 in Figure 2.6). Developers rerun our extraction method on the modified version and obtain another finite state machine, as shown in Figure 3.8b. In this finite state machine, they determine the removal of the edge causing the duplicatie order problem, and finally, they can confirm that our motivating example run as expected.

Thus, developers can use the extracted finite state machine to understand stateful behavior in Ajax Web applications in addition to reading the source code and reviewing runtime application behavior. Even if the application have erroneous behaviors that are not executable in the development and testing environments of developers, they may find such erroneous behaviors in the extracted finite state machine and debug the applications so as to run as expected. Therefore, our extraction method can support developers to build reliable Ajax Web applications.



(a) Extracted from faulty version



(b) Extracted from correct version

Figure 3.8: Finite state machines extracted from our motivating example in Section 2.6

Chapter 4

Verifying Pattern-Based Interaction Invariants in Ajax Web Applications

In this chapter, we propose a method for verifying interaction-based stateful behavior in Ajax Web applications. As one input to our verification method, the behavior to be verified is the extracted finite state machine discussed in Chapter 3. Towards automated verification of invariants relevant to the interactions (`interaction invariants`), our verification method follows the workflow shown in Figure 4.1. Our verification requires the following additional developer input.

- Information about implemented Ajax design patterns (`IADP info`) for instantiating interaction invariants, as explained in Section 4.3.

Our verification method consists of three steps, as shown in Figure 4.1 (V1, V2, and V3). As outputs, our verification method either verifies the correctness of application behavior or reports the presence of potential faults in the applications.

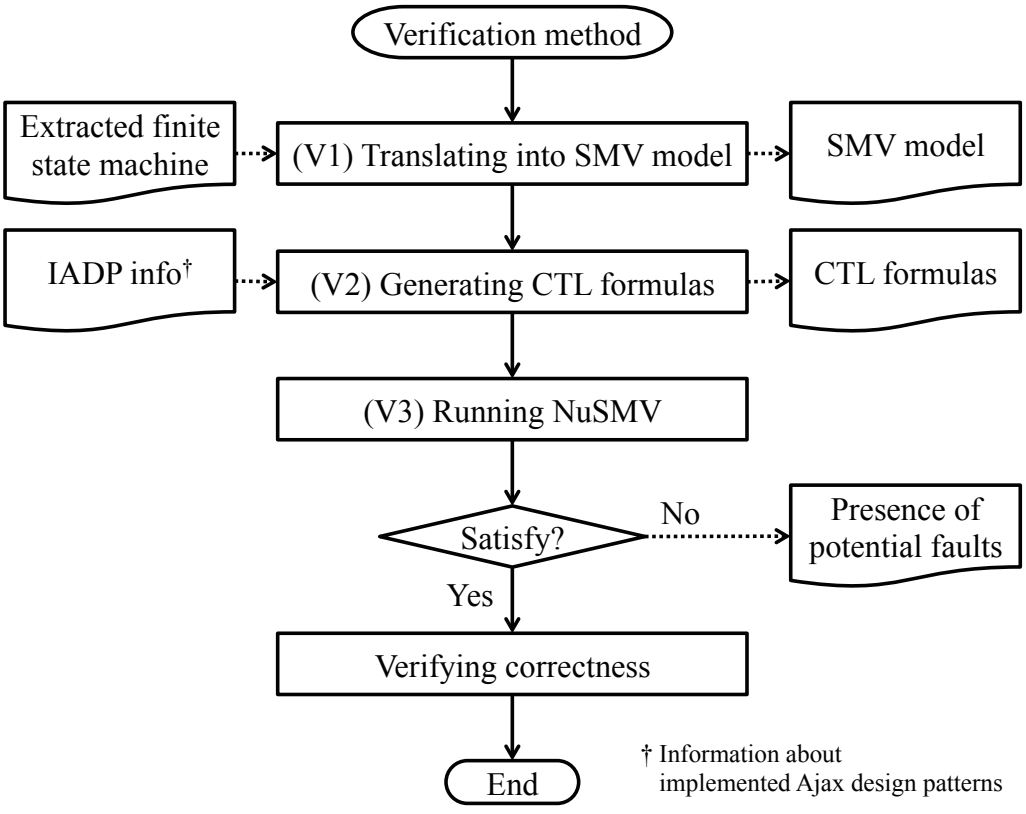


Figure 4.1: Verification method workflow

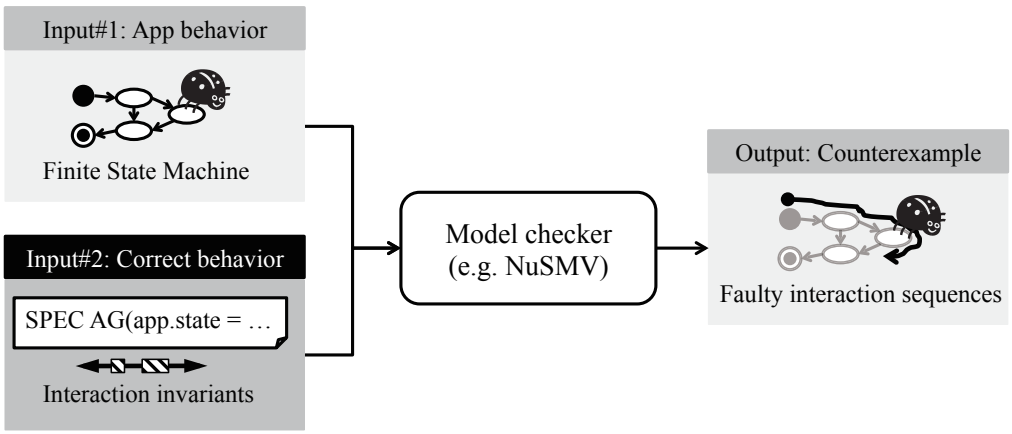


Figure 4.2: Input and output of model checker

4.1 Overview

When developing Ajax Web applications, developers implement Ajax design patterns [49] for increasing the usability of the applications. Although developers intend to correctly implement the design patterns, unpredictable contexts while running applications might conceal faults that will break properties of the design patterns. We claim that such faults decrease usability; therefore, a technique for verifying whether the application correctly runs according to the implemented design patterns is required.

In Chapter 3, we present a static extraction method of stateful behavior in Ajax Web applications to support program understanding. Although developers may be able to simultaneously find faults relevant to the interactions using the extracted finite state machine, the cost may not be negligible for developers to manually and carefully determine the correctness of the behavior. Additionally, the more interactions developers implement in the applications, the larger the finite state machine our extraction method extracts. Therefore, we address the automatic verification of application behavior correctness.

Towards automatic detection of faulty behavior, a model checking technique is useful for verifying the correctness of nondeterministic automata with given invariants. Figure 4.2 shows an overview of the model checking technique. Although the nondeterministic automata representing application behavior correspond to the extracted finite state machine, the challenge is that there are no generic behavior oracles relevant to interactions, i.e., interaction invariants. Developers need to define properties to be verified and correctly express them in verification formulas; however, these tasks are difficult for developers who build Ajax Web applications and are not familiar with a model checking technique.

We assume that developers implement Ajax design patterns in Ajax Web applications. Since the design patterns have been collected from observations in many real-world applications, we assume that Ajax design patterns describe the properties of the application behaviors expected by developers (Assumption 2). Therefore, we define the properties relevant to interactions with Ajax Web applications as the interaction invariants in Section 4.3.

Assumption 2. *Ajax design patterns provide behavior oracles focused on interactions with Ajax Web applications.*

Consequently, our verification method automatically verifies the correctness of the extracted finite state machines. Otherwise, our verification method identifies

faulty interaction sequences from counterexamples in the verification results and reports the presence of potential faults in Ajax Web applications if any.

4.2 Model Checking

Our verification method leverages a widely known model checker, NuSMV [13], for verifying interaction invariants in a finite state machine extracted in Chapter 3. Given flexible invariants expressed as computation tree logic (CTL) formulas, NuSMV verifies the correctness of a finite state machine described in an SMV model. Accordingly, such a model checker is suitable for verifying the extracted finite state machine that models nondeterministic elements of Ajax Web applications.

4.2.1 Translating into SMV Model

Our verification method first translates the extracted finite state machine into an SMV model. Figure 4.3 shows the code of an SMV model translated from part of the extracted finite state machine in Figure 3.6. Table 4.1 lists the definitions of the elements in the SMV model. In this model, our verification method outputs a module called `App` (lines 1-24) for representing application behavior extracted in the finite state machine. A `state` variable is defined for describing invariants in the CTL formulas (lines 2-8). Assignment statements `init` and `next` are used for initializing the state variable to a `root` label and for representing state transitions in the extracted finite state machine, respectively (lines 11 and 12-24). Another module called `main` instantiates the App module to simulate application behavior (lines 26-27).

States in the extracted finite state machine are defined as labels in the `state` variable (lines 2-4). Note that an SMV model unfortunately does not allow describing an event-based state transition; therefore, our verification method also defines labels representing events that the application handles (lines 6-7). Our verification method is designed to deal with the event-based state transition in such a way that the `state` variable is set to the event label then to the next state label. For example, in the `root` state, our motivating example nondeterministically handles an `onload` event (line 13). When handled, our motivating example then transits to a `setEventHandlers` state (line 14).

Thus, our verification method obtains the SMV model that represents the interaction-based stateful behavior in Ajax Web applications. With the aid of this SMV model, the NuSMV model checker can simulate the application behavior. However, any model checker does not know correct and wrong behaviors of the applications so that it cannot verify whether the application behavior is correct or wrong. Additionally, there are no generic behavior oracles relevant to the interactions, i.e., the correct and wrong behaviors. Consequently, it is difficult for developers to define properties to be verified and to correctly express them in verification formulas.

4.3 Pattern-Based Interaction Invariants

Herein, we consider that developers implement Ajax design pattern [49] in the applications. It means that they want to verify whether the applications correctly run according to the implemented Ajax design patterns (IADPs) and have information about the IADPs. Therefore, under Assumption 2, our verification

Table 4.1: Definitions of elements in SMV model we used

SMV model element	Definition
`MODULE` *App*	Application module that expresses behavior of Ajax Web application
`MODULE` *main*	Main module that instantiates application module in `app` variable
`VAR` *state*	State in which application module is
`case ... esac;`	Conditional branch expression of "*state* = **cur_state** : {**state1**, **state2**};" expressing that application module nondeterministically transits to **state1** or **state2** when it is in **cur_state**

```
1  MODULE App
2  VAR state : {
3  -- state labels
4    root, setEventHandlers, addCart,
5      reqRunTrans, succeeded, ...
6  -- event labels
7    onload, click, success, ...
8  };
9
10 ASSIGN
11   init(state) := root;
12   next(state) := case
13     state = root : { root, onload };
14     state = onload : setEventHandlers;
15     state = setEventHandlers :
16       { setEventHandlers, onclick, ... };
17     state = onclick : addCart;
18     state = addCart : reqRunTrans;
19     state = reqRunTrans :
20       { reqRunTrans, success };
21     state = success : succeeded;
22     ...
23     TRUE : state;
24   esac;
25
26 MODULE main
27 VAR app : App;
```

Figure 4.3: Partial example of translated SMV model

method supports the aforementioned difficult tasks by generating correct verification formulas using the information.

4.3.1 Generating CTL Formulas

Ajax design patterns contain comprehensive findings for increasing usability of Ajax Web applications; hence, we first define invariants in terms of the interactions from the findings (interaction invariants). Table 4.2 gives a fundamental set of interaction invariants, which are derived from the category of fundamental technology patterns in Ajax design patterns. In addition, we also define a selec-

Table 4.2: Interaction invariants derived from Ajax design patterns

#	Pattern category	Ajax design pattern	Property name†
1	Fundamental Technology	XMLHttpRequest Call	AyncComm
2			ACFRetry
3		On-Demand JavaScript	SRWait
4		User Action	UEHRegist
5			UEHSingle
6	Programming	Explicit Submission	UESubmit
7	Functionality and Usability	Live Form	FDValid
8		Direct Login	SeedRetrieve
9			LFDisable

† Long forms and explanations of these abbreviations are listed in Table 4.3.

Table 4.3: Explanations of interaction invariants

#	Property name (Abbreviation) Property description of expected application behavior
1	Asynchronous communication (AsyncComm) Handling user events during asynchronous communications
2	Asynchronous communication (ACFRetry) Retrying when asynchronous communication fails
3	Wait for server response (SRWait) Retrieving response before calling response-dependent functions
4	User event handler registration (UEHRegist) Registering user event handlers at page load
5	User event handler singleton (UEHSingle) Preventing multiple calls of specific user event handlers
6	User event and submit (UESubmit) Requiring explicit user operations before form data are submitted
7	Form data validation (FDValid) Validating form data before submission
8	Seed retrieval (SeedRetrieve) Retrieving seed data before login attempt
9	Login form disable (LFDisable) Disabling login form after successful login

tive set of interaction invariants from other Ajax design pattern categories to use them in our case study in Section 6.2. These interaction invariants consist of their property names and descriptions; for example, the user event handler registration (name) property explains that Ajax Web applications should register user event handlers at page load (description).

Table 4.4: CTL template formulas related to interaction invariants

#	Property name	Property pattern	CTL template formula†
1	AsyncComm	Response	**AG**(app.state = **$1CommFunc** -> **AF EX** app.state = **$2UserEv**)
2	ACFRetry	Response	**AG**(app.state = **$1FailEv** -> **AF** app.state = **$2CommFunc**)
3	SRWait	Precedence	**A**[app.state != **$1WaitFunc** **W** app.state = **$2SuccessEv**]
4	UEHRegist	Precedence	**A**[app.state != **$1UserEv** **W** app.state = **$2PageLoadEv**]
5	UEHSingle	Absence	**AG**(!(app.state = **$1PreventFunc** & **EX** app.state = **$2UserEv**))
6	UESubmit	Existence	**A**[app.state != **$1SubmitFunc W** (app.state = **$2UserEv** & app.state != **$1SubmitFunc**)]
7	FDValid	Precedence	**A**[app.state != **$1SubmitFunc W** app.state = **$2ValidateFunc**]
8	SeedRetrieve	Precedence	**A**[app.state != **$1LoginFunc W** app.state = **$2RetrieveFunc**]
9	LFDisable	Absence	**AG**(!(app.state = **$1SuccLoginFunc** & **EX** app.state = **$2LoginEv**))

† **$1*** and **$2*** represents template variables.

To express these interaction invariants in correct verification formulas, we also leverage the property pattern mappings for CTL [1], which classifies raw property specifications of a GUI, concurrency logic, and communication protocol, into occurrence and order patterns. These property patterns contain template verification formulas with given states and events of running applications. By relating these property patterns to the interaction invariants, we can describe CTL templates using the `state` variable in the `App` module in Table 4.1 for our verification methor.

Table 4.4 lists the relationships between interaction invariants and property patterns. For example, the user event handler registration property means that no user event occurrences (`$1UserEv`) precede that of a page load (`$2PageLoadEv`). Therefore, we relate the `Precedence` property pattern in the order pattern to the #4 interaction invariant in Table 4.2. For the user event handler singleton property, the `Absence` property pattern in the occurrence pattern is related because the absence of multiple calls (`$1PreventFunc`) of the user event (`$2UserEv`) expresses the #5 interaction invariant in Table 4.2. Thus, developers i) select interaction invariants and ii) input variables in the CTL template formulas, then our verification method can generate correct CTL formulas using the relationships listed in Table 4.4.

4.3.2 Running NuSMV

When developers implement and test Ajax Web applications based on Ajax design patterns, they can input information about IADPs into a repository of our verification method (`IADP info repository`). Developers can input function and event names in the source code as variables for selected interaction invariants. To determine states corresponding to the given function and event names, our verification method leverages the abstraction map, as explained in Section 3.3. The abstraction map contains to which states the functions are abstracted. Therefore,

developers do not need to deeply understand how our verification method works.

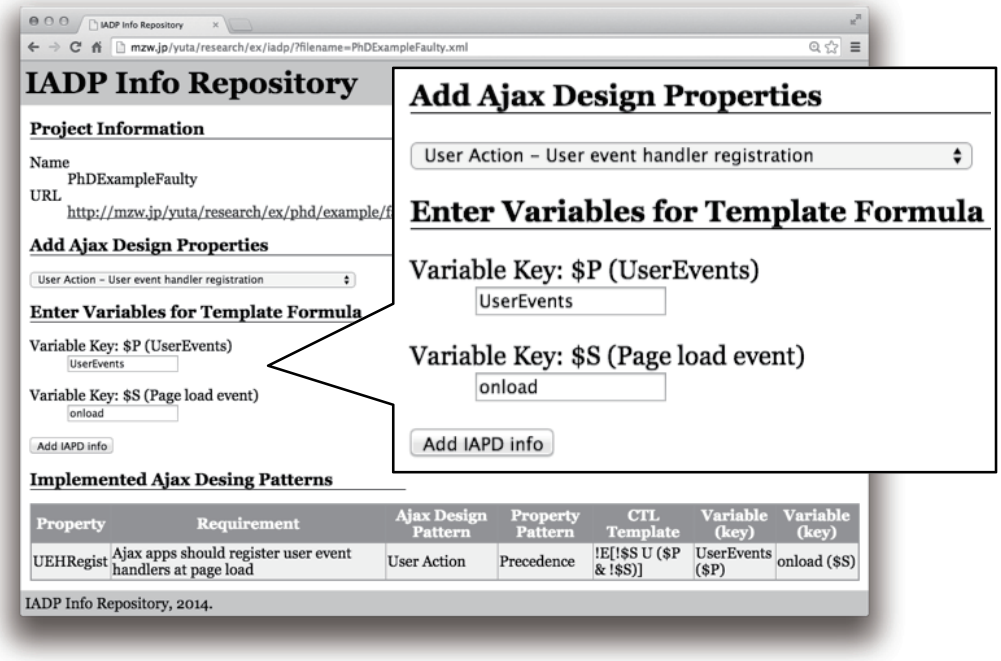
By obtaining the information via the repository as input, our verification method automatically generates correct CTL formulas expressing the interaction invariants. Then, NuSMV traverses in the state space of the translated SMV model and verifies whether the model satisfies the generated formulas. This verification of correctness can assure developers that the application correctly behaves according to their intentions. Otherwise, NuSMV outputs a counterexample of the CTL formula as a fault oracle, then our verification method describes the counterexample on the extracted finite state machine as a faulty interaction sequence. Finally, our verification method automatically reports the presence of potential faults in Ajax Web applications to the developers as output.

4.4 Use Scenario and Results on Motivating Example

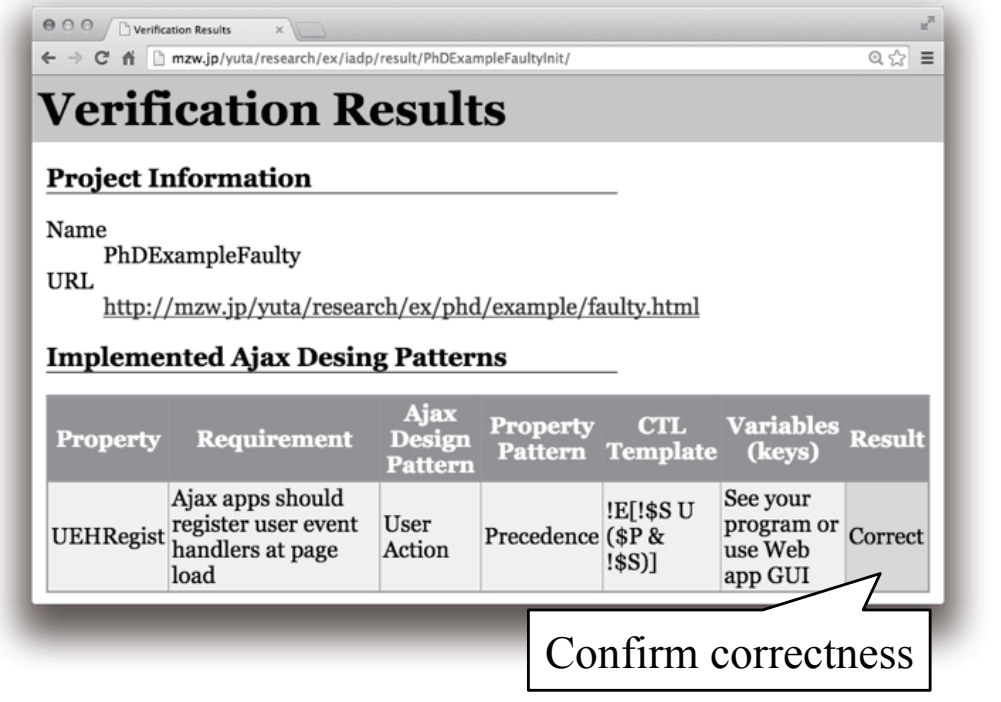
In this section, we explain a use scenario of our verification method and results on our motivating example described in Section 2.6. We assume that our verification method can be used in the context of test-driven development, where developers first give test cases of additional functionalities and then improve the source code to pass the test cases. Considering such the development process, we argue that developers first input interaction invariants of implemented Ajax design patterns into the IADP info repository of our verification method, then, they can debug until the invariants are verified as correct.

We now illustrate a use scenario of our verification with our motivating example. In our motivating example, developers first implement the option select functionality based on the user event handler registration property in Ajax design pattern. Thus, they first select the user event handler registration property and input its variables of `UserEvents` and `onload` when implementing the option selection functionality, as shown in Figure 4.4a. Note that our verification method interpret this `UserEvents` value as all user event types implemented in the source code such as `onclick` and `onchange`. Then, our verification method verifies the correctness of this implementation, as shown in Figure 4.4b. Next, developers implement the item addition functionality based on the user event handler singleton and give the IADP info for this additional functionality, as shown in Figure 4.5a. At this time, our verification method determines that the application behavior does not satisfy the additional invariant and reports the presence of potential faults, as shown in Figure 4.5b. This report contains faulty interaction sequences[1] associated with the potential faults, as shown in Figure 4.6a. With the aid of the faulty interaction sequences as clues to debugging, we expect that developers can debug the faulty version of our motivating example. Finally, developers confirm that the application correctly runs according to the invariants, as shown in Figure 4.6b.

[1]Our verification method actually outputs faulty interaction sequences in the form of slideshow corresponding to the arrowed line drawn in Figure 4.6a.

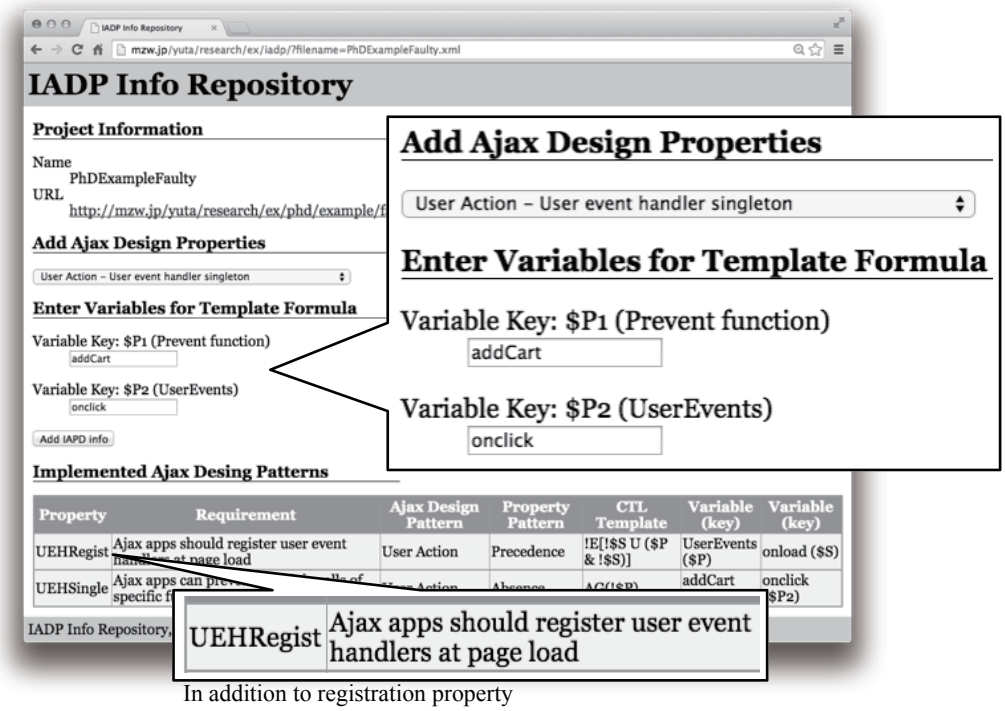


(a) Input of IADP info for user event registration property

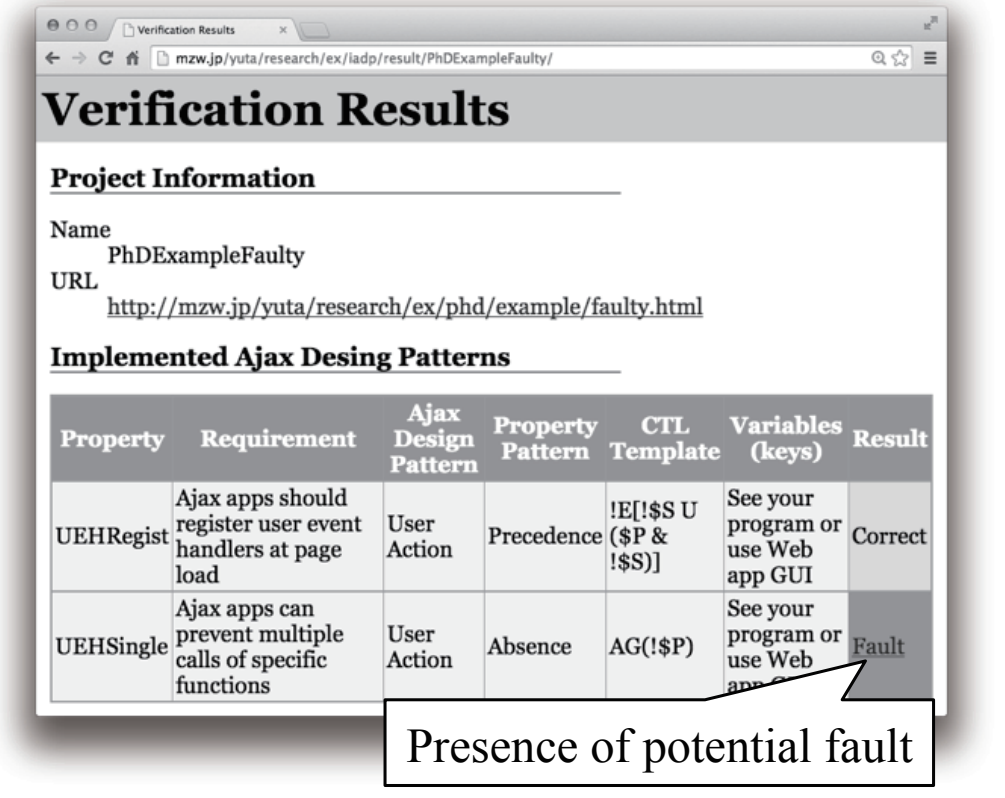


(b) Confirm correctness of user event registration property

Figure 4.4: Verification results of user event registration property in faulty finite state machine extracted in Figure 3.8a

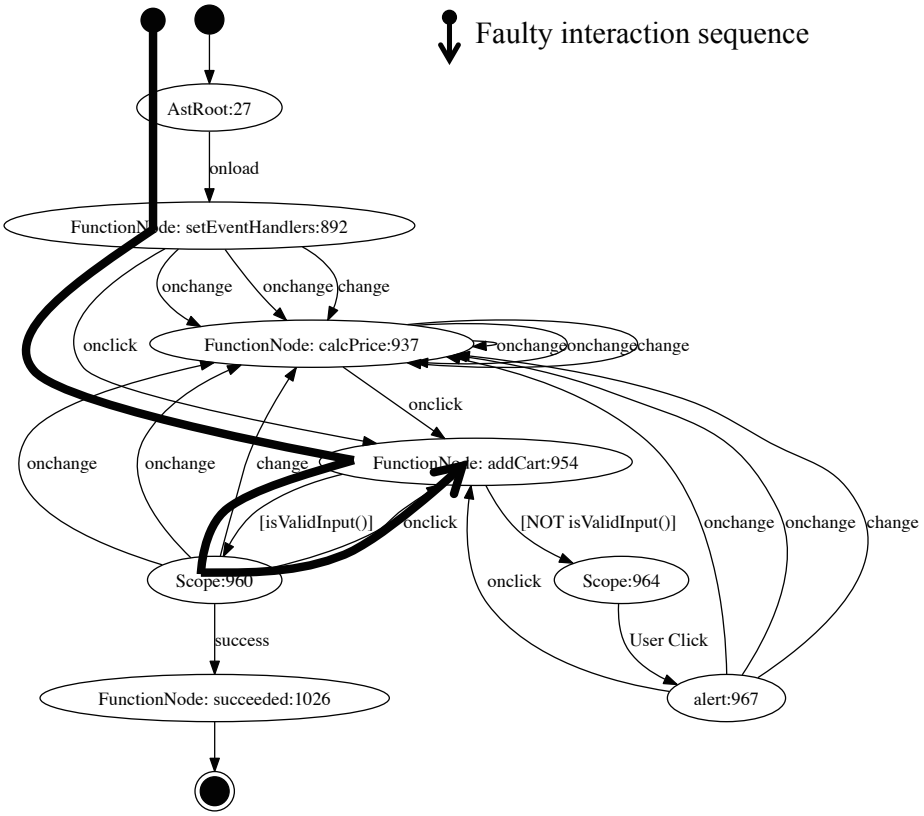


(a) Input of IADP info for user event singleton property

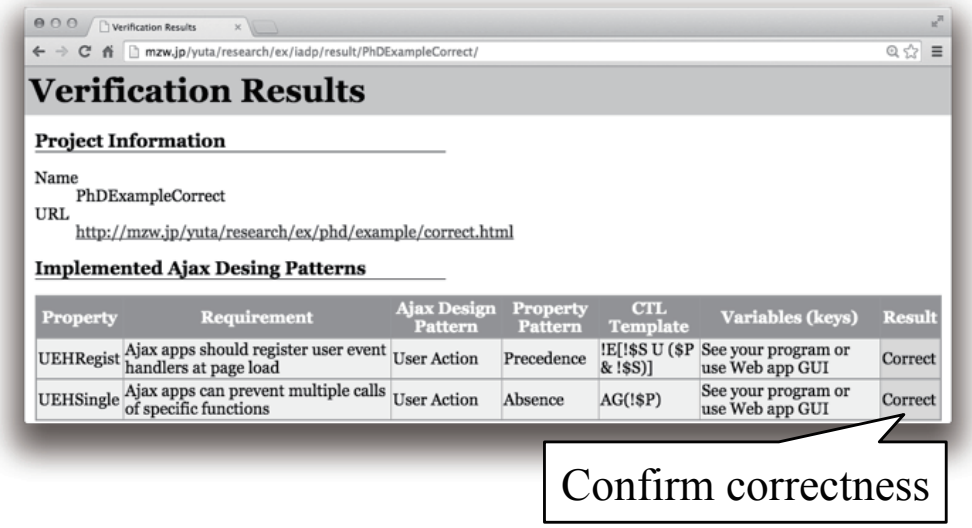


(b) Report presence of potential fault

Figure 4.5: Verification results of user event singleton property in faulty finite state machine extracted in Figure 3.8a



(a) Faulty interaction sequence associated with potential fault



(b) Confirm correctness of both user event registration and singleton properties

Figure 4.6: Confirm correctness of our motivating example in Figure 3.8b

Chapter 5

Validating Ajax Web Applications Using Delay-Based Mutation Technique

In this chapter, we propose a method for validating whether actual errors are due to potential faults in Ajax Web applications. Figure 5.1 depicts a workflow of our validation method. In this workflow, two of three inputs, i.e., the original code and faulty interaction sequences, are given using our extraction and verification methods, as described in Sections 3.3.1 and 4.3.2, respectively. Hence, our validation method requires the following input from developers:

- Test data and oracles to test whether the potential faults cause actual errors (See Section 5.2.1).

Our validation method is mainly divided into three steps, as shown in Figure 5.1 (M1a/b, M2, and M3). As outputs, our validation method reveals actual errors due to delay-dependent potential faults if any.

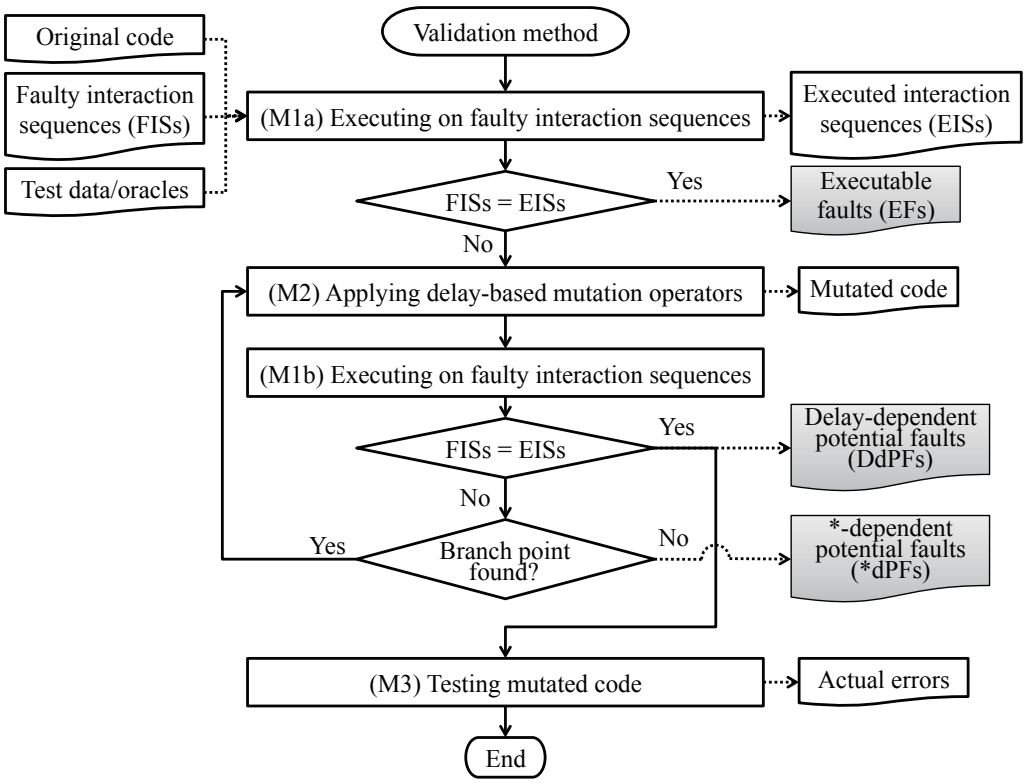


Figure 5.1: Validation method workflow

5.1 Overview

In Chapters 3 and 4, we investigated static methods for identifying Ajax Web application behaviors that seem to cause errors when specific conditions are met; for example, the duplicate order problem in our motivating example because of an unexpected double-click on the add-to-cart button. Although developers can use our extraction and verification methods for identifying the presence of "potential faults" that seem to cause actual errors if executed, they need to confirm whether the potential faults are actually executable. Testing all possible scenarios in every environment should reveal all errors; however, such a method would be unrealistic.

Therefore, we present a mutation-based validation method for specifying conditions to reveal actual errors due to these potential faults. Our validation method obtains the faulty interaction sequences as input from our verification method and tries to find executable evidence of the potential faults. However, our validation method may not easily execute Ajax Web applications on the faulty interaction sequences because a specified environment does not meet specific conditions to reveal actual errors due to the potential faults. Since Zheng et al. pointed out that an expected network latency may cause severe problems in Ajax Web applications [105], we assume that

Assumption 3. *an unexpected network latency may make potential faults in Ajax Web applications executable.*

To emulate an unexpected network latency, we define synchronous and asynchronous delay-based mutation operators, as described in Section 5.2.2. Although a program mutation technique is commonly used for injecting artificial faults [17], we leverage the technique to allow our validation method to make potential faults executable in the specified environment. This is because our mutation operators change the nonfunctional aspects of Ajax Web applications, but the applications can handle specific server responses after a given delay time has elapsed. We expect that testing the mutated source code with the given test data and oracles will reveal actual errors. Thus, our validation method provides developers with executable evidence of not-easily-executable faults during testing, i.e., subtle network delays are required for revealing actual errors. In addition to outputs from our extraction and verification methods (i.e., an extracted finite state machine and identified faulty interaction sequences), developers can use revealed actual errors from our validation method to debug the original source code of Ajax Web applications.

Compared with our extraction and verification methods, the novelty of our validation method lies in revealing actual errors due to potential faults in Ajax Web applications. Since Ajax design patterns are aimed to improve the usability of Ajax Web applications, code violations against the design patterns do not always lead to actual errors being debugged. Additionally, the faulty interaction sequences that are identified are counterexamples in the extracted finite state machine. Counterexamples in an abstract model can be spurious; therefore, it is important to reanalyze them in an actual system [98]. Consequently, we argue that the novelty of our validation method is that it helps developers validate Ajax Web applications.

5.2 Delay-Based Program Mutation

The potential faults reported by our verification method can be exposed under a specific set of conditions but have little chance of being found if the set is unknown. A problem with such faults is that developers have trouble detecting them during testing because they cause errors only when complicated conditions are met [30]. As for Ajax Web applications, despite concerted efforts by developers, they have difficulty in specifying unexpected conditions in unpredictable contexts of a running application, such as subtle network delays. Therefore, the challenge of our validation method is to specify complicated conditions to reveal actual errors due to potential faults in testing environments of developers before users encounter erroneous behaviors in the applications. Consequently, our validation method is designed to validate whether Ajax Web applications run as expected on faulty interaction sequences that might not be easily executable in the given environments.

Figure 5.1 depicts the three-step process that our validation method uses for validating Ajax Web applications. Our validation method initially executes Ajax Web applications on the faulty interaction sequences identified by our verification method (M1a). If the potential faults associated with these sequences are not easily executable, our validation method mutates the source code until the Ajax Web applications do execute the potential faults (M2 and M1b). Then, developers test for the unexpected behavior due to the potential faults by using the mutated applications (M3). Additionally, our validation method classifies potential faults into executable, delay-dependent potential, and *-dependent potential faults, which are defined below.

5.2.1 Executing Faulty Interaction Sequences

Because executing Ajax Web applications results in actual errors if there are any, our validation method first attempts to execute the applications in accordance with the identified faulty interaction sequences (M1a in Figure 5.1). For this execution, our validation method require that developers implement test cases with test data and oracles by using Selenium WebDriver [86] and JUnit [45]. Figure 5.3 shows an example of the test case for the duplicate order problem in our motivating example. This test case first instantiates the Firefox Webbrowser at the `setupBrowser` method annotated with `@Before` (line 5-7). Then, at the `testDuplicateOrder` method annotated with `@Test` (lines 10-20), it open our motivating example using `URL` (line 11), emulates user action of the click on the add-to-cart button (line 13), and check whether or not the button is enabled (line 14). If our motivating example can handle the duplicate order, this test case fails (line 15). Finally, it quit the browser at the `quitBrowser` method annotated with `@After` (line 23-25). Thus, our validation method uses such test cases in order to make its execution as automated as possible.

Our validation method then obtains the executed interaction sequences from the execution results. If the executed interaction sequences and faulty interaction sequences are identical, our validation method determines that potential faults in the applications are `executable faults` against the implemented Ajax design patterns. As described in Section 2.5, executable faults can be revealed through state-of-the-art studies.

If the potential faults are not executed, our validation method focuses on the differences between the faulty interaction sequences and the executed interaction sequences. Differences may occur because Ajax Web applications cannot handle

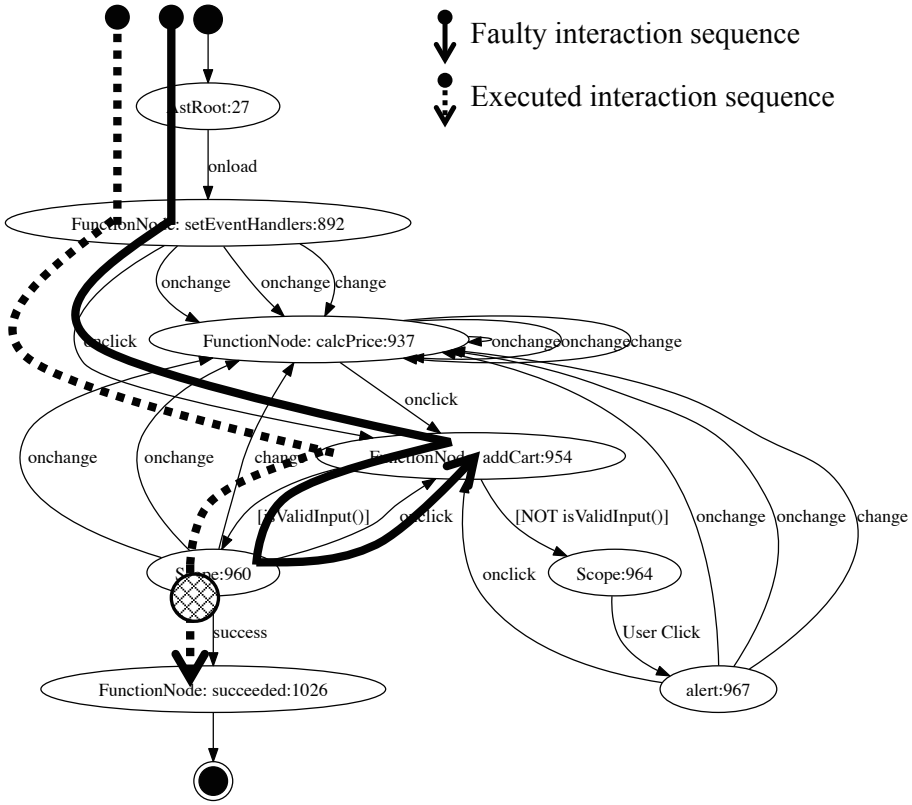
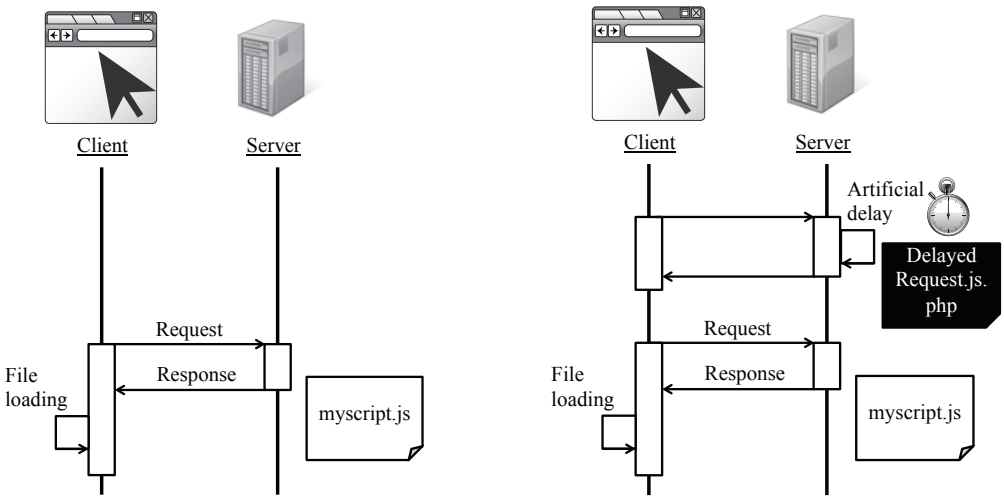


Figure 5.2: Executed interaction sequence and branch point

```
1  WebDriver driver;
2  String URL = "http://...";
3
4  @Before
5  public void setupBrowser() {
6    driver = new FirefoxDriver();
7  }
8
9  @Test
10 public void testDuplicateOrder() {
11   driver.get(URL);
12   try{
13     driver.findElement(By.id("addcart")).click();
14     if(driver.findElement(By.id("addcart")).isEnabled()) {
15       fail("dupliate order");
16     }
17   } catch(Exception e) {
18     e.printStackTrace();
19   }
20 }
21
22 @After
23 public void quiteBrowser() {
24   driver.quit();
25 }
```

Figure 5.3: Example of test cases using Selenium WebDriver and JUnit



(a) Synchronous communication

(b) Delayed synchronous communication

Figure 5.4: Synchronous delay mutation operator

certain interactions in faulty interaction sequences. The solid and dashed arrows in Figure 5.2 respectively represent faulty interaction sequence and its executed interaction sequence. At the meshed circle in the figure, WebDriver emulates the `onclick` user action, but our motivating example actually handles the `success` event because of an immediate server response. However, the ability to manipulate the timing of the applications handling the interactions allows Ajax Web applications to run as faulty interaction sequences. Thus, a mutation technique can be used to manipulate the timing.

5.2.2 Applying Delay-Based Mutation Operators

By mutating the source code of the Ajax Web applications, our validation method enables them to run on faulty interaction sequences in a given environment (M2 in Figure 5.1). We assume that an unexpected network latency might allow potential faults to be executable and define the following two delay-based mutation operators, as shown in Figure 5.4 and Figure 5.7.

Synchronous Delay Mutation Operator

A Web browser parses the HTML code from top to bottom while loading a Web page. When the browser finds a `script` element whose `src` attribute has the location of an external JavaScript file, it begins loading the JavaScript file. For example, in Figure 5.4a, such an external JavaScript file corresponds to `myscript.js`. Because the JavaScript code can dynamically manipulate the Web page content, the browser blocks render the remaining HTML code until the JavaScript file is completely loaded. Therefore, script elements can be implemented at the bottom of the HTML code in order to render all page elements as fast as possible. This is a well-known way to increase the perceived performance [89].

Loading a JavaScript file at the improper time may cause an actual error. Figure 5.5a shows brief code fragments to explain our synchronous delay mutation operator. In this application, developers implement the `handleClick` function in the `handleClick.js` JavaScript file and set the function as the `onclick` event handler of the button element. Following common practice, the script element is

```
40 <button onclick="handleClick()" />
41 ...
42 <script type="text/javascript" src="handleClick.js" />
```

(a) Synchronous communication

```
40 <button onclick="handleClick()" />
41 ...
42 <script src="text/javascript"
43     src="http://.../DelayedRequest.js.php?millisecond=3000" />
44 <script type="text/javascript" src="handleClick.js" />
```

(b) Artificially delayed synchronous communication

Figure 5.5: Loading a JavaScript file with improper timing

```
1 <?php
2   $sleepMillis = intval($_GET['millisecond']);
3   usleep($sleepMillis * 1000);
4   header("content-type: application/javascript");
5   echo '';
6 ?>
```

Figure 5.6: DelayedRequest.js.php: mock server-side script

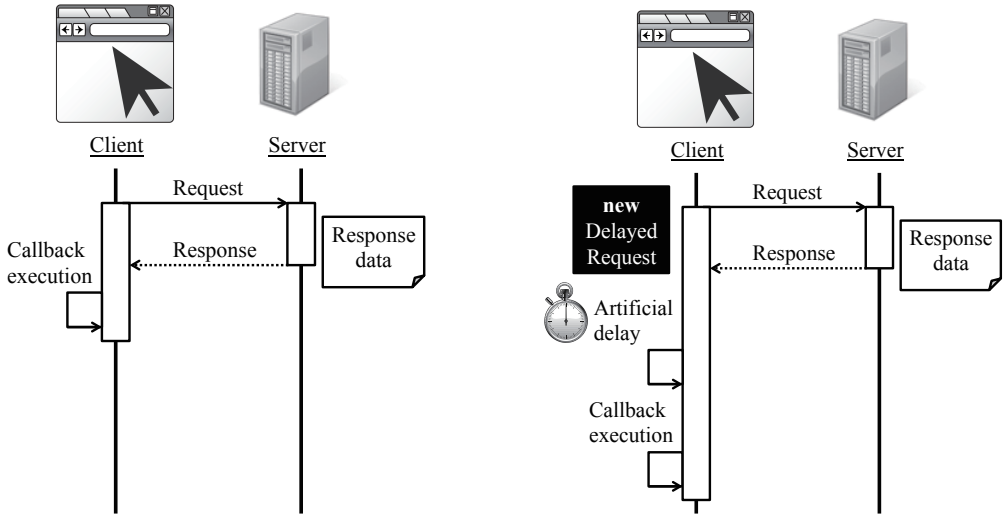
then implemented at the bottom of the HTML code. However, an unexpected network latency may delay loading of the JavaScript file. The application cannot respond to the `onclick` event occurrence because this scenario allows a user to click a button even though the browser has yet to register the event handler.

Figure 5.4b illustrates the aforementioned synchronous communication delayed by our validation method. To emulate a JavaScript file loading delay due to an unexpected network latency, our validation method inserts an artificial script element whose src attribute is located in our server-side program (lines 42-43 in Figure 5.5b). Figure 5.6 depicts our server-side program called `DelayedRequest.js.php`. After the artificial script element sends an HTTP request, this program then sends an HTTP response with a given delay time, causing a synchronous delay. This HTTP response consists of header and body sections in which are set "content-type: application/javascript" and an empty string, respectively. In this manner, our validation method can manipulate the timing of handling interactions relevant to page loading, such as an `onload` event.

Asynchronous Delay Mutation Operator

The most significant feature of Ajax Web applications is the asynchronous communications between the client and server, as shown in Figure 5.7a. However, an unexpected network latency can significantly impact this feature. Unlike a synchronous delay, an unexpected network latency does not inhibit rendering of Web page content or handling of user interactions. Consequently, it is extremely difficult to consider all possible states where Ajax Web applications may have to handle a delayed asynchronous server response.

To emulate a handling delay with an asynchronous server response, our validation method rewrites the corresponding JavaScript code fragment by using our wrap function, `DelayedRequest`, as shown in Figure 5.8. Our validation method finds a target function for an asynchronous server request in the original code and parses its arguments, such as URL, query string, and callback function. Instead of the original code, our validation method inserts an instantiation statement of



(a) Asynchronous communication

(b) Delayed asynchronous communication

Figure 5.7: Asynchronous delay mutation operator

```
40  jQuery.ajax({           /* taget function */
41    url: "runTrans.php", /* URL */
42    data: getParams(),    /* query string */
43    success: succeeded    /* callback */
44  });
```

(a) Asynchronous communication

```
40  new DelayedRequest(3000).applyFunction(
41    jQuery.ajax,      /* target function */
42    "runTrans.php", /* URL */
43    getParams(),      /* query string */
44    succeeded         /* callback */
45  );
```

(b) Artificially delayed asynchronous communication

Figure 5.8: Code snippet from Figure 2.6 for asynchronous delay mutation

the DelayedRequest with a given delay time, e.g., 3000 msec. As illustrated in Figure 5.7b, the DelayedRequest sends the same asynchronous server request as the original one, but does not invoke the callback function even if a response is immediately received. After the given delay time has elapsed, the DelayedRequest invokes the callback function.

Thus, our mutation operators mutate the source code of Ajax Web applications in order to artificially manipulate the timing of handling interactions with the applications. It should be noted that our mutation operators do not change the functionalities of the applications. This is because the applications can still handle a specific interaction after the given delay has elapsed.

To determine where to apply our mutation operators, our validation method searches for a `branch point` between the faulty interaction sequences and executed interaction sequences, such as the meshed circle in Figure 5.2. The branch point may prevent Ajax Web applications from running on the faulty interaction sequences. The mutation operators are then applied to an interaction in the executed interaction sequences at this point, e.g., the success event handler.

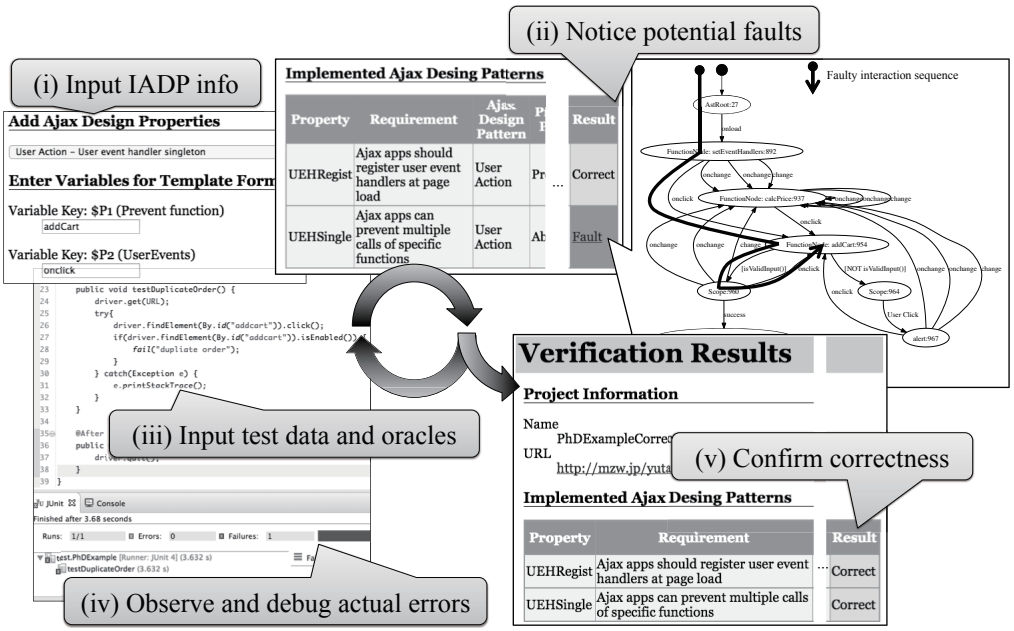


Figure 5.9: JSPreventer use scenario

Even if the server immediately responds, our motivating example does not handle the server response within the delay period. By iterating M1b and M2 in Figure 5.1 until the executed interaction sequences equal the faulty interaction sequences, our validation method artificially sets up conditions in which potential faults can be exposed. Finally, it classifies the potential faults that become executable by applying our mutation operators as `delay-dependent potential faults`. Thus, our validation method provides developers with executable evidences of the delay-dependent potential faults which are not-easily-executable faults because they are not executed without subtle network delays.

However, not all applications can be mutated to produce identical executed interaction sequences and faulty interaction sequences. Our validation method classifies these potential faults as `*-dependent potential faults`, which might cause actual errors for reasons other than delayed injection.

5.2.3 Testing Mutated Code

Because developers implement Ajax design patterns according to their intentions, we assume that they can also implement test oracles to verify whether the patterns are correctly implemented in Ajax Web applications or not. With such test oracles, our validation method can determine whether the mutated code has passed the test (M3 in Figure 5.1). If the code fails a test, developers can determine the actual errors from the test results.

Our validation method outputs actual errors in addition to a finite state machine and faulty interaction sequences from our extraction and verification methods. We assume that these outputs will help developers debug the original source code because they allow the complex behaviors of Ajax Web applications to be understood, the faulty behaviors of a finite state machine to be identified, and unexpected behaviors due to potential faults to be observed. Thus, developers can use our validation method to prevent actual errors due to potential faults in Ajax Web applications.

5.3 Use Scenario and Results on Motivating Example

Figure 5.9 illustrates a use scenario of JSPreventer, which includes our extraction, verification, and validation methods, with the results of our motivating example. JSPreventer is applicable in the context of iterative and incremental development [48]. In fact, developers often select a simplified and rapid iteration development life cycle [36].

The scenario involves five steps: (i) the implemented Ajax design pattern information (IADP info) is initially inputted to JSPreventer; (ii) JSPreventer suggests faulty interaction sequences in the applications against the IADP info and reports the presence of potential faults. (iii) Developers provide test data and oracles to JSPreventer; (iv) to reveal unexpected runs, JSPreventer mutates and tests the applications. In this step, if the potential faults become to be executable with subtle network delays, we assume that the executable evidences and the revealed actual errors can be of help in validating and debugging the applications; (v) through iterative development, highly reliable Ajax Web applications can be released.

Chapter 6

Evaluation

We conducted case studies to evaluate the usefulness of our proposed methods implemented in a tool called JSPreventer. As described in Section 2.7, we answer the following research questions.

RQ1 *Can our extraction method support developers in understanding an interaction-based stateful behavior containing blind spots in Ajax Web applications?*

RQ2 *Can our verification method report the presence of potential faults in Ajax Web applications?*

RQ3 *Can our validation method find executable evidence of potential faults in Ajax Web applications?*

We first discuss the preliminary case study we conducted on sample Ajax Web applications in Section 6.1. This case study involved seven participants and demonstrated that they had difficulties in understanding interaction-based stateful behavior in the sample application. In Section 6.2, we discuss an additional case study on real-world Ajax Web applications to evaluate whether JSPreventer could reveal actual errors due to potential faults in real-wold applications. We then discuss the threats to validity in our case studies in Section 6.3. Finally, we describe the limitations of JSPreventer in Section 6.4.

6.1 Preliminary Case Study

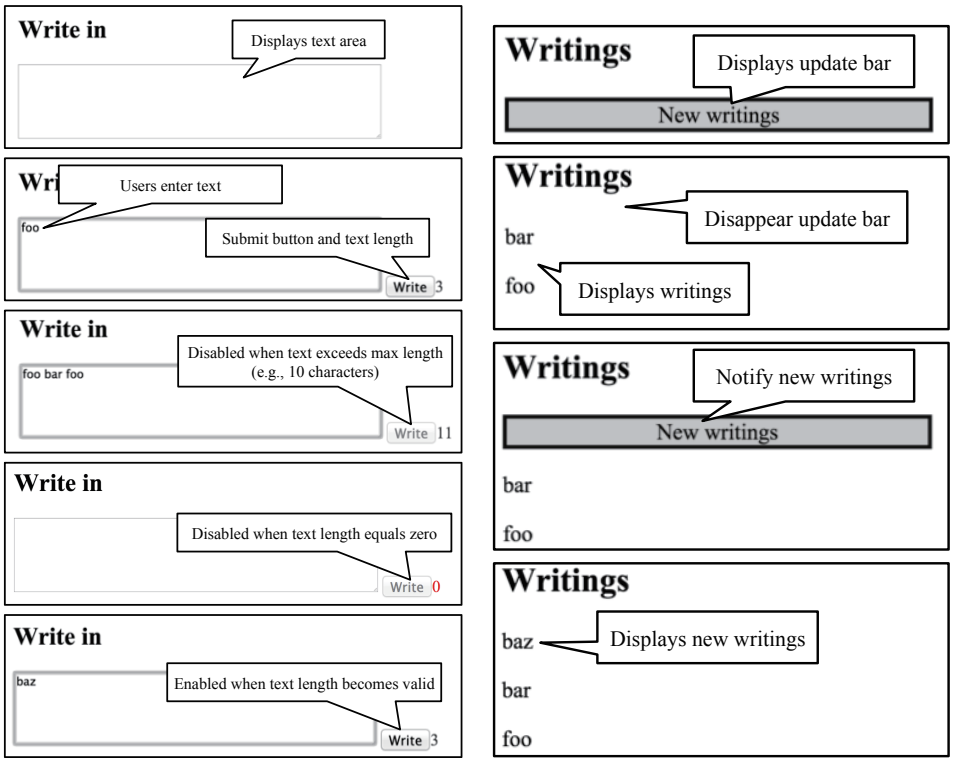
We conducted a preliminary case study to evaluate interaction-based behavior in Ajax Web applications that are difficult for developers to understand and a finite state machine extracted with our extraction method that is helpful for them to find erroneous behaviors in the applications. In this case study, we invited seven computer science students (`P#1`, `P#2`, `P#3`, `P#4`, `P#5`, `P#6`, and `P#7`) to participate.

6.1.1 Subject Applications

For this preliminary case study, we implemented a sample Ajax Web application, as shown in Figure 6.1. This sample application has the following two functionalities and functions as a bulletin board like Twitter[1].

Write functionality This Web page initially displays a text area where users enter their "writings", which correspond to tweets in Twitter. After users

[1] `https://twitter.com/`



(a) Write functionality (b) Watch functionality

Figure 6.1: Screenshots of sample Ajax Web application

first enter in the text area, it then displays a submit button and the length of the string. Next, when users change their writings, it enables and disables the submit button depending on whether the length is less than the given maximum one and is not zero. At the same time, it updates the displayed length. When users submit their writings, this Web page asynchronously sends the string to a server. On the server side, this application receives and stores the string in a file.

Watch functionality Another Web page is for displaying the writings. Once this page is loaded, it creates an update bar, which handles mouse click events. It also asynchronously inquires the server on whether there are new writings. If the server responds that new ones exist, this page displays the update bar. When users click the bar, this page asynchronously retrieves the new writings from the server. Once this page updates the new ones, it hides the bar. Otherwise, it recursively sends asynchronous messages to check the new ones at given intervals.

6.1.2 Evaluation Methodology

To evaluate whether the participants understood interaction-based stateful behavior in the sample application, we injected faults relevant to interactions with the sample application, as shown in Table 6.1. These faults caused the following erroneous behaviors in the sample applications (`Error#1`, `Error#2`, and `Error#3`). If the participants could identify these erroneous behaviors, we determined that they could understand the application behavior.

Error#1 The write functionality initially enables the submit button. Since

Table 6.1: Faults deployed in sample Ajax Web applications

Interaction	Faults due to improper implementation	Faults due to no implementation
User	`Fault#3`	`Fault#1`
Server	`Fault#4`	`Fault#6`
Self	`Fault#5`	`Fault#2`

the text area does not have any user inputs at that time, developers should prevent users from clicking the submit button (`Fault#1`) at the loaded time (`Fault#2`). If missing this execution scenario, the participants cannot identify this erroneous behavior; otherwise, they may find this 'executable' erroneous behavior by reviewing runtime application behavior.

Error#2 This functionality also disables the text area and submit button when users send their writings to prevent changing the content and multiplying submissions. If this functionality fails to communicate with the server, this text area is permanently disabled; hence, developers should release the text area (`Fault#3`) regardless of the results of the communication (`Fault#4`). This erroneous behavior cannot be revealed in reliable network conditions; therefore, the participants had trouble identifying this erroneous behavior because it may have been a bind spot in the sample application.

Error#3 The watch functionality uses timeouts to periodically check the occurrence of writings. However, this functionality may fail in the first communication with the server to inquire of the existence of new writings. Users cannot operate the sample application at all. This is because this functionality sets the timeouts for sending the inquiry messages only when it succeeds in the first communication; as a result, nothing happens if the first communication fails. Therefore, developers should implement a behavior of sending the inquiry messages after the given elapsed time (`Fault#5`) in case of communication failure (`Fault#6`), as well as communication success. As with `Error#2`, this erroneous behavior cannot be revealed if in localhost network environments.

We describe the experimental procedure in this case study as follows.

1. First, we gave the source code of the sample application. At this time, we taught behaviors of the server-side programs, which are out of the scope of this study, such as receiving writings from users and storing them in a file on the server-side.

2. Second, the participants tried to find erroneous behaviors relevant to the interactions of the sample application by reading the source code and reviewing the runtime application behavior for 30 minutes.

3. Next, we provided finite state machines extracted from these two functionalities with our extraction method.

4. Then, the participants tried to find erroneous behaviors for an additional 30 minutes.

Table 6.2: Results of faults participants found in sample application

	Fault#1	Fault#2	Fault#3	Fault#4	Fault#5	Fault#6
P#1	–	–	✓	–	✓	–
P#2	✓	✓	–	–	✗	✗
P#3	✓	✓	✗	–	–	✗
P#4	✗	✗	✓	✓	✗	–
P#5	✓	✓	✗	✗	✗	✗
P#6	✓	✓	–	–	–	✓
P#7	✓	✓	–	–	–	–

– indicates that participants found faults only with the source code
✓ indicates that participants found faults with the extracted finite state machines
✗ indicates that participants did not find faults

Table 6.3: Results of errors that participants identified in sample application

Participants identified with	Error#1	Error#2	Error#3
source code and runtime behavior	1	3	1
extracted finite state machine	5	2	2
Participants could not identify	1	2	4

5. Finally, the participants reported erroneous behaviors found with or without the aid of the extracted finite state machines.

In this case study, we asked the participants to provide their heuristic feedback about the extracted finite state machines in free format.

6.1.3 Results

Tables 6.2 and 6.3 list the results of faults and errors the participants found and identified in the sample application, respectively. Additionally, we collected feedback from the participants as follows.

- The extracted finite state machines provided suspicious clues and helped the participants find erroneous behaviors.
- Although code fragments relevant to the interactions were dispersed in the source code (for example, HTML inlines and external JavaScript files), the participants could receive a quick overview of interaction-based stateful behavior in the sample application with the aid of the extracted finite state machines.
- The extracted finite state machines were helpful for the participants to reproduce suspicious behaviors they found from the runtime application behavior.
- The participants pointed out the scalability of our extraction method for manually checking whole of the extracted finite state machines.

- The participants could intuitively determine the lack of necessary state transitions in the sample application. However, it was also pointed out that, if there was a state where many state transitions were concentrated, the participants mistook it as a central part of the sample application behavior. This means that the extracted finite state machines did not always match an aspect the participants wanted to focus on. The participants also claimed that our extraction method might divide more states than what they imagined.
- The participants had difficulties in determining code fragments in the source code corresponding to elements of the extracted finite state machines.
- The participants suggested that our extraction method should separately deal with user, server, and self interactions in the extracted finite state machines.

6.1.4 Discussion

We discuss the difficulties in understanding interaction-based behavior in Ajax Web applications and the usefulness of our extraction method based on the results of our preliminary case study.

Support Understanding Ajax Web Application Behavior (RQ1)

In our preliminary case study, the participants identified errors in the sample applications with the aid of finite state machines extracted with our extraction method. From the results described in Table 6.3, only one of seven participants identified Error#1 by using the source code and runtime behavior. Although this error was executable in the experimental environment, six of them missed the execution scenario causing this error. Five of the participants tried to execute the sample application on execution paths in the extracted finite state machine and identified this error by using the extracted finite state machine. Thus, if developers miss execution scenarios in which Ajax Web applications cause erroneous behaviors, the extracted finite state machine may help them determine how to execute the applications for finding erroneous behaviors.

Error#2 and Error#3 were not executable in the experimental environments; i.e., potential faults in blind spots. Although we argue that such potential faults are extremely difficult for developers to find, more participants could identify these errors only with the source code and runtime behavior than those who identified Error#1. This is because Error#2 and Error#3 were caused by asynchronous communication failures which the participants had the most concerns when they tried to find faults. However, even if the participants carefully reviewed the asynchronous communication failures, most could not identify these errors; hence, we want to make sure of our aforementioned argument. Our extraction method gave the finite state machine to the participants; therefore, in both cases of Error#2 and Error#3, two of them could identify these errors. From these results, we confirm that our extraction method can support developers in understanding an interaction-based behavior containing blind spots in Ajax Web applications.

The participants mentioned the difficulties in determining interaction-based stateful behavior in Ajax Web applications. One of the difficulties lies in that code fragments relevant to the interactions are dispersed in the source code. In spite of the fact that the sample application was small, the participants struggled to

determine the complex behavior that came from these code fragments. Therefore, our extraction method is useful for developers because it analyzes these fragments and models the application behavior in the form of a finite state machine. In fact, the participants said that they could improve their understanding of application behavior.

Since we focused on the interactions as state transitions of Ajax Web applications, our extraction method may not extract a suitable finite state machine for concerns of developers. Additionally, some participants claimed that it was difficult to search corresponding code fragments when they found suspicious clues and erroneous behavior in the applications. While we were observing the activities of the participants, we found that they were confused with the unsuitable structure and layout of the extracted finite state machines. Thus, a remaining issue of our extraction method is that developers need to carefully read the extracted finite state machine. Therefore, our verification and validation methods are useful because they automatically identify potential faults and reveal actual errors so that developers do not need to deeply understand how our extraction method works.

6.2 Case Study on Real-World Applications

To evaluate whether or not JSPreventer could reveal actual errors due to potential faults in Ajax Web applications, we additionally conducted a case study on real-world applications.

6.2.1 Subject Applications

Table 6.4 lists the subject Ajax Web applications used in this case study. To find real-world Ajax Web applications, we leveraged `NerdyData`[2], which provides a search engine for the source code on websites. We needed applications in which Ajax design patterns were implemented to verify the properties of the design patterns; therefore, we searched using keywords relevant to Ajax design patterns.

2020m We searched for "login_controller.js" where James Dam[3] implemented the Direct Login pattern[4]. The optical accessory supplier website was found at the top of the search results.

UCDChina and ESA In addition, we searched for "onblur=checkInput" and "onsubmit=validate", which are representative of the Live Form pattern. The Live Form pattern suggests that Ajax Web applications should check form data before making a submission. From the search results, we found Chinese and British companies' portal websites.

6.2.2 Experimental Setup

Our case study experiments were conducted on a 64-bit Mac OS X 10.9.2 machine with an Intel Core i5 (2.3 GHz) and 16 GB of memory. The experimental procedure is as follows:

1. We first obtained HTML, CSS, and JavaScript source code files by using the completely-save-web-page functionality in `Mozilla Firefox`.

[2] `http://nerdydata.com`
[3] James Dam (Internet Archive): `goo.gl/S47AVV`
[4] Ajax Login System Demo (Internet Archive): `goo.gl/yCcxtN`

Table 6.4: Subject Ajax Web applications

Subject	URL	Keyword (Ajax design pattern)
2020m	`2020m.com`	login_controller.js (Direct Login)
UCDChina	`ucdchina.com`	onblur=checkInput (Live Form)
ESA	`www.easyservicedapartments.com`	onsubmit=validate (Live Form)

2. We then leveraged `Code Beautifier`[5] to count lines of these codes, as listed in Table 6.5 (HTML, CSS, and JavaScript). The 1K-10K lines of code range represents medium-largish Ajax Web applications. Note that only client-side source code could be obtained; therefore, we implemented mock server-side scripts in PHP.

3. Next, we used our extraction method to extract the finite state machine from each subject. Table 6.5 lists the extraction time (T_e) and the numbers of states and transitions of the finite state machine (#states and #trans).

4. To verify the correctness of the extracted finite state machines by using our verification method, we determined interaction invariants to be verified and corresponding variables in the subject applications, as listed in Table 6.6. Since we did not know the intent of the original developers, we determined the invariants based on the search keywords and source code.

   In the source code of all the subjects, we found implementations of asynchronous communications and page load event handlers; `jQuery.get`[6] and `jQuery.post`[7] are representative functions for asynchronous communications and the `jQuery.ready`[8] event is usually implemented in order to attach all other event handlers. Therefore, we determined that Invariants #1 (XMLHttpRequest Call) and #2 (User Action) should be satisfied in all the subjects. Additionally, considering the search keywords, we also determined that Invariants #4 and #5 (Direct Login) in 2020m and Invariant #3 (Live Form) in UCDChina and ESA should be satisfied.

5. Our verification method then verified all the invariants for each subject and reported the verification time (T_v in Table 6.7) and the verification results ($Result_v$ in Table 6.7).

6. If the verification result was incorrect, our validation method executed the classification workflow shown in Figure 5.1. We set the delay time to be 3 seconds. Additionally, we provided all the necessary test data and oracles, such as the correct username and password, for successful logins in 2020m. Finally, our validation method classified the potential faults that caused incorrect verification results into executable, delay-dependent potential, or

[5] `http://ctrlq.org/beautifier/`
[6] `http://api.jquery.com/jQuery.get/`
[7] `http://api.jquery.com/jQuery.post/`
[8] `http://api.jquery.com/ready/`

Table 6.5: Size of subject Ajax Web applications and extracted finite state machines

Subject	HTML	CSS	JavaScript	#state	#trans	T_e (sec)
2020m	188	420	2468	15	57	9.434
UCDChina	1978	898	880	32	508	14.166
ESA[†]	4751	964	3523	34	602	17.705

[†] Easy serviced apartments `ESA` had an implementation for asynchronous communications by directly manipulating the `XMLHttpRequest` object [68]. Since our extraction method was not designed to analyze a dataflow of the object, we replaced this implementation with the equivalent function call by using jQuery which was originally loaded in `ESA`.

Table 6.6: Determined interaction invariants for subject Ajax Web applications

Subject	#	Interaction invariant	**$Var1**[‡]	**$Var2**[‡]
2020m	1	AsyncComm	jQuery.post	UserEvents
	4	UEHRegist	UserEvents	ready
	8	SeedRetrieve	validateLogin	getSeed
	9	LFDisable	window.location.replace	submit
UCD	1	AsyncComm	jQuery.post	UserEvents
China	4	UEHRegist	UserEvents	ready
	7	FDValid	sendMail	checkInput
ESA	1	AsyncComm	jQuery.get[†]	UserEvents
	4	UEHRegist	UserEvents	ready
	7	FDValid	onsubmit	validate

[†] As described in Table 6.5, we replaced an implementation for asynchronous communications using `XMLHttpRequest` object with `jQuery.get` in `ESA`.
[‡] **$Var1** and **$Var2** correspond to **$1*** and **$2*** in Table 4.4.

*-dependent potential faults. The $Result_c$ column in Table 6.8 corresponds to these classification results. JSPreventer also reported the execution, mutation, and testing times (T_x, T_m, and T_t in Table 6.8).

7. In addition, we conducted comparative experiments with `Crawljax`, which is a state-of-the-art tool for finding faults in Ajax Web applications, as described in Section 2.5. The default setting of Crawljax could not find any delay-dependent potential faults in the subjects.

6.2.3 Results and Discussions

Automated Verification (RQ2)

Our verification method could semi-automatically verify correct and incorrect application behavior. As inputs to our verification method, we selected the interaction invariants and entered their template variables listed in Table 6.6 for the subject applications, which might require additional tasks for developers. To

Table 6.7: Verification results of subject Ajax Web applications

Subject	#	Interaction invariant	T_v (msec)	$Result_v$
2020m	1	AsyncComm	65	Correct
	4	UEHRegist	84	Fault
	8	SeedRetrieve	96	Fault
	9	LFDisable	86	Fault
UCDChina	1	AsyncComm	95	Correct
	4	UEHRegist	293	Fault
	7	FDValid	235	Fault
ESA	1	AsyncComm	111	Correct
	4	UEHRegist	254	Fault
	7	FDValid	194	Fault

Table 6.8: Validation results of subject Ajax Web applications

Subject	#	Invariants	T_x (sec)	T_m (sec)	T_t (sec)	$Result_c$[†]
2020m	4	UEHRegist	9.643	0.039	11.868	*dPF
	8	SeedRetrieve	8.147	0.116	8.175	DdPF
	9	LFDisable	8.553	-	-	*dPF
UCDChina	4	UEHRegist	11.677	0.183	11.816	DdPF
	7	FDValid	9.692	-	-	EF
ESA	4	UEHRegist	10.520	0.133	10.335	DdPF
	7	FDValid	15.307	0.208	14.965	*dPF

[†] EF, DdPF, and *dPF in the column of $Result_c$ represent executable, delay-dependent potential, and *-dependent potential faults, respectively.

mitigate the additional tasks, we plan to define language-level semantics of Ajax design patterns towards full-automated verification [9].

Table 6.7 lists results of our verification method. In 2020m, UCDChina, and ESA, our verification method reported the presence of potential faults which indicated that the subject applications violated the #4 UEHRegist property. Although these violations indicated that user events might be handled before page load completions, the immediate page load in our testing environment concealed the faulty behavior. We then searched the user events on faulty interaction sequences suggested by our verification method and found them in the HTML source code of the subject applications. These implementations conformed to undesirable ones described in the User Action Ajax design pattern. We debugged them according to a solution suggested in the design pattern so that our verification method could output the correct results.

Our verification method also reports the presence of potential faults that indicated that the login functionality in 2020m might not work properly; 2020m might send account information without the seed data for password hashing (#8 SeedRetrieve) and might redundantly handle the login attempts (#9 LFDisable). Since our reliable network allowed 2020m to retrieve the seed data and to jump

to the logged-in page immediately, these faulty behaviors were also concealed. We tested UCDChina and ESA to run on the faulty interaction sequence for #7 FDValid. Although we observed that UCDChina actually handled the form submission without any user inputs in the form, ESA did not run on the faulty interaction sequence.

From the results of our case studies, we argue that our verification method could report the presence of potential faults that were concealed in our testing environment but would be executable in an actual user environment. Our remaining issue is to investigate whether potential faults will actually cause errors in Ajax Web applications. Therefore, our validation method allows Ajax Web applications to run on faulty interaction sequences suggested by our verification method.

Revealing Actual Errors due to Delay-Based Potential Faults (RQ3)

Our validation method could classify potential faults reported by our verification method into executable, delay-dependent potential, and *-dependent potential faults. As indicated in Table 6.9, it found actual errors in the subjects. We reviewed the source code of the subjects and confirmed that the actual errors could be exposed.

In 2020m, users could not log in even with their correct username and password because 2020m sent the login request with the initial value at the declaration statement of the `seed` variable. For secure password hashing, 2020m used the MD5 algorithm to a value obtained by adding the `seed` variable to the user password. Although 2020m should use the `seed` variable generated at runtime, it hashed the value with the initial value "0"; hence, 2020m sent the hashed value using the raw user password. The MD5 Reverse Lookup[9] might allow interceptors to reverse-engineer the hashed value, obtain the user password, and log in 2020m illegally. Thus, we could infer a vulnerability to intrusion.

Additionally, our validation method revealed that UCDChina sent an empty text on its search form, despite developers having incorporated a `goSearch` function to prevent it. A cause of this erroneous behavior was that UCDChina could display the search form before loading the code fragment containing the `goSearch` function. Since the search form is the source where users enter the search query, developers in this case had intended to improve the search form. Although they had expected only valid search queries would be sent from the search form, the error indicates that UCDChina did not validate the search query at all, and therefore, it possibly sent every search query. If so, there would be an additional possibility that a server would receive malicious search queries (e.g., an SQL code fragment) that might lead to an SQL injection attack.

Our validation method could find an undefined `addthis_close` function call in ESA by using the synchronous delay operator upon loading an external JavaScript `addthis_widget.js` file. The undefined function calls can be fatal errors, which might cause ESA to crash.

Although Crawljax [14] ran until it completed its exploration of the state space, it did not reveal the actual errors found by our validation method. These results suggest that our validation method reveals actual errors due to delay-dependent potential faults.

[9] `http://search.cpan.org/~blwood/Digest-MD5-Reverse-1.3/`

Table 6.9: Actual errors due to delay-dependent potential faults

#	Subject	Brief explanation	Inferred vulnerability
1	2020m	Login failed with correct username and password	Intrusion
2	UCDChina	Content search with an empty query	SQL injection
3	ESA	Undefined function call	Application crash

Table 6.10: Results of code and runtime behavior reviews in subject Ajax Web applications

Subject	#	Invariants	Reviewed behavior	Confirmed result
2020m	4	UEHRegist	Proper rendering block	False positive
	8	SeedRetrieve	Error#1 in Table 6.9	Potential faults
	9	LFDisable	Proper page transition	False positive
UCDChina	4	UEHRegist	Error#2 in Table 6.9	Potential fault
	7	FDValid	Form data submitted without validated	Executable fault
ESA	4	UEHRegist	Error#3 in Table 6.9	Potential fault
	7	FDValid	Data-intensive impossible behavior	False positive

Reasonable Analysis Time

For each test subject, JSPreventer, consisting of our extraction, verification, and validation methods, revealed the actual errors within one minute. Most of the T_e, T_x, and T_t values were required to initialize parsers and to launch the test browser. Although T_v increased linearly with the size of the extracted finite state machine, the extracted finite state machines were small enough to be verified with the NuSMV model checker. T_m was much shorter than the other amounts. These results indicate that JSPreventer is practical.

False Positives in *-dependent Potential Faults

In our case studies, three potential faults could not be executed using JSPreventer and the delay-based mutation operators. As for the potential faults at Invariants #2 and #5 in 2020m, we observed that Mozilla Firefox prevented 2020m from running on the identified faulty interaction sequences by making a proper rendering block and page transition, respectively. Since the extracted finite state machine did not contain the Web browser behavior, we determined that these potential faults were false positives due to spurious counterexamples. Additionally, JSPreventer could not make the potential fault at Invariant #3 in ESA executable. In the source code of ESA, we found that a conditional branch was implemented to prevent ESA from executing the potential fault. Analyzing data-intensive impossible behavior is currently beyond the scope of our research; hence, this result was a false positive due to our methods. However, this problem

can be mitigated by using execution results to refine the extracted finite state machine, i.e., by conducting further dynamic analyses together with JSPreventer. Finally, we confirmed results of our proposed method in the subject applications, as listed in Table 6.10.

Applying to Generic Ajax Web Applications

We designed JSPreventer for developers who use Ajax design patterns to build applications. In actual Ajax Web application development, developers have their own design patterns. If there are not given IADP info for the own design patterns, JSPreventer cannot determine code locations where our mutation operators should be applied. To test Ajax Web applications made with their own design patterns, developers need to specify the code locations of synchronous or asynchronous communications which they suspect as the cause of errors when specific network delays are present. JSPreventer then mutates the code fragments at the specified code locations and tries to reveal the errors.

Additional Ajax design patterns

We assume that interaction invariants in Ajax Web applications derive from Ajax design patterns. In fact, developers have their original Ajax design pattern and flexible requirements. When adding new design patterns, developers need to define verification properties in the design patterns and relate appropriate property patterns to the properties. Otherwise, developers can use JSPreventer with raw CTL verification formulas.

Debugging Potential Faults

We assume that the actual errors revealed by JSPreventer can help developers debug potential faults. However, this debugging task depends on the skills and experience of the developers. In the future, we plan to establish a method to support debugging. For example, we are interested in combining JSPreventer with automated program repair techniques [104, 27].

6.3 Threats to Validity

6.3.1 Internal validity threats

We considered two external factors that might affect results in our case study experiments. The results from 2020m, UCDChina, and ESA demonstrate the usefulness of JSPreventer because these real-world applications were obtained via a public search engine provided by NerdyData. However, their actual server-side scripts were not available; therefore, we implemented mock server-side scripts in order to run these real-world applications on our machine. Because the mock server-side scripts may be a threat to internal validity, we intend to conduct additional case studies using real-world open-source Ajax Web applications.

Additionally, we provided the necessary information to run the test subjects (e.g., IADP Info, test data, and oracles). Although the ability of developers to input correct information into JSPreventer may affect the internal validity, the results of our case studies show that JSPreventer can reveal actual errors due to potential faults. It should be noted that the test scenarios involved typical tests, such as verifying user login success with the correct username and password. In

```
1  <input type="submit" id="submit" onclick="handleClick();">click</input>
2  <script type="text/javascript"><!--//
3  var mystate = 0;
4  function handleClick() {
5    if(mystate == 0) {
6      init();
7      mystate = 1;
8    } else if(mystate == 1) {
9      proc();
10   }
11 }
12 //--></script>
```

Figure 6.2: Example of changing behavior depending on state variable

the future, we intend to use JSPreventer in actual Ajax Web application development projects to evaluate whether developers can input correct information into JSPreventer. We are also interested in combining a search-based testing technique [3] with JSPreventer in order to generate test data automatically.

6.3.2 External validity threats

Regarding the generality of our approach, JSPreventer can only reveal actual errors due to delay-dependent potential faults. However, potential faults may depend on other reasons, such as unexpected user operations and Web browser behaviors. Therefore, we are going to collect *-dependent potential faults by conducting additional case studies with JSPreventer and define effective mutation operators for them.

Although 2020m, UCDChina, and ESA are practical Ajax Web applications, it would be interesting to determine the scalability of JSPreventer by obtaining experimental results using many real-world Ajax Web applications.

6.4 Limitations

We now describe the limitations of JSPreventer.

6.4.1 State Changes Using Variables

Developers can implement *state variables* that represent states of Ajax Web applications. Figure 6.2 shows an example source code to illustrate the state variables. In this case, the `mystate` variable corresponds to the state variable (line 3). When handling the `onclick` event (lines 1 and 4-11), this applications initially calls back the `init` function (line 6), but it then changes their state (line 7) to call back the `proc` function from the next time (line 9). Thus, developers expect that such an application exhibits stateful behavior according to the values of the state variables.

From Ajax Web applications using the state variables, our extraction method in JSPreventer extracts a finite state machine representing interaction-based stateful behavior, which is a different view from what developers expect. Since our extraction method does not evaluate contexts of variables at conditional statements, it extracts only one state change from the initial state to the state of the `handleClick` function in response to the `onclick` event. Analyzing state changes depends on the state variables, we can consider combining our extraction method with constructing the *state variable definition graph* [88].

```
24|   if(isValidInput()) {
25|     reqRunTrans();
26|   } else {
27|     alert("Invalid user inputs");
28| /* enableAddCart(); // proper enabling */
29|   }
```

Figure 6.3: Example of changing behavior depending on state variable

```
24| var script_from_users = document.getElementById("my_text_area").value;
25| eval(script_from_users);
```

Figure 6.4: Example of changing behavior depending on state variable

6.4.2 Data-Intensive Impossible Behaviors

Our extraction method analyzes only enable/disable statements to determine whether an Ajax Web application can handle interactions. In fact, developers can also implement such interaction controls using data flows. Figure 6.3 gives the code snippet from our motivating example in Figure 2.6 to illustrate the data-intensive impossible behaviors. In this code snippet, user inputs for selecting options can never be invalid (line 24), which means that the application can never proceed to the state corresponding to invalid user inputs (lines 26-29). Such data-intensive impossible behaviors can be addressed using DOM-based dynamic approaches such as Crawljax. Hence, we will extend JSPreventer to leverage the contributions of these state-of-the-art studies to construct a hybrid approach.

However, we want to claim that such application behavior may be executable faults even if developers consider it as impossible at first glance, for example, in the case in which other developers modify the source code of open source Ajax Web applications or in which users install other application plugins. Therefore, we argue that our pessimistic verification method is valuable for verifying application behavior containing potential faults.

6.4.3 Behaviors Added at Runtime

Additionally, Ajax Web applications can add their behaviors at runtime by using `eval`, `innerHTML`, `document.write`, etc [31]. The additional behaviors can be generated on the server side or can be entered by users via input forms, resulting in false negatives with our verification method in JSPreventer. This is because our extraction method analyzes the interactions implemented in the source code and the additional behaviors at runtime are implemented out of the source code. For example, Figure 6.4 demonstrates behaviors added at runtime. In this case, an application obtains a string from the text area identified by `my_text_area` (line 1) and evaluates the string as its additional behavior by using the `eval` function (line 2). However, developers should not implement or take great care in implementing them. This is because they might cause security vulnerabilities such as enabling the stealing of cookies and misusing user authorizations with servers [49]. To address the false negatives, we intend to combine JSPreventer with dynamic analysis techniques that allow it to analyze how Ajax Web applications add their behavior at runtime.

6.4.4 Complicated Conditions for Making Potential Faults Executable

Although our validation method in JSPreventer revealed actual errors in the subject applications, a remaining issue is that it is unknown whether Ajax Web applications actually execute *-dependent potential faults. It should be noted that these *-dependent potential faults might be impossible for the subject applications to actually execute in every environment. Such actually-not-executable faults can be defined as false positives in JSPreventer. However, current testing techniques cannot be used for testing the applications in every environment; hence, JSPreventer has a limitation in precisely determining whether the *-dependent potential faults are potential faults or false positives. To address this issue, we are interested in defining additional effective mutation operators to make *-dependent potential faults executable. With the aid of these mutation operators, we want to specify complicated conditions, in addition to subtle network delays, to reveal actual errors caused by the potential faults.

Chapter 7

Related Work

In this chapter, we survey related work according to the following categories.

7.1 State-based Analysis and Testing of Web Applications

In traditional Web applications, Ricca et al. introduced ReWeb for model-based analysis and testing [82, 83]. They claimed that HTML Web pages are central entities of the applications and extracted page transition models, i.e., navigation models, by analyzing hyperlinks, frames, and forms among the pages. Such extraction of models from applications correspond to reverse engineering [16], which is aimed to provide alternative views from software artifacts for re-documentation, design recovery, etc. [10]. Tramontana claimed that developers rarely describe sufficient documentations including these models to understand complexities in Web applications due to the very short time-to-market situation [99]; Jazayeri also mentioned early releases and frequent specification updates of Web applications [41]. In this situation, reverse engineering of Web applications can be used for supporting in developers understanding the complexities in Web applications.

In terms of supporting program understanding, several researchers have conducted reverse engineering of Web applications. Vanderdonckt et al. developed a tool called VAQUISTA for reverse engineering user interfaces from traditional Web applications [100]. This tool statically analyzes HTML pages and translates them into a *presentation model* representing elements of these pages. Katsimpa et al. also used a static approach for reverse engineering of ASP.NET Web applications [46]. Their proposed tool parses ASPX code, extracts Web page elements, and creates ASP.NET tag trees for content management of the Web page. Regarding dynamic approaches, Antoniol et al. focused on dynamic Web page generation in Web applications and presented their tool called WANDA, which reverse engineers models, including class diagrams, sequence diagrams, etc., from the server-side PHP programs [5]. Licca et al. presented the Web Application Reverse Engineering tool (WARE-tool) [19, 20], where they used both static and dynamic analyses; as the first step, it analyzes static information such as static Web pages, then it retrieves dynamic information such as execution results of PHP `print` statements. In the early 2000's, these researchers argued that the complexities of Web applications lied at the server-side dynamic Web page generation. However, the advent of Ajax technologies in 2005 [25] moved the complexities to the client-side event-driven, asynchronous, and dynamic features of the applications.

To address the complexities derived from Ajax technologies, Marchetto et al. introduced the concept of state-based analysis and testing to Ajax Web ap-

plications [53, 52] in 2008. Their tool called ReAjax [54] extracts finite state machines from the execution results of Ajax Web applications. The finite state machines consist of the document object model (DOM) [94] instances and the effects of callback executions as states and transitions; Duda et al. also proposed the *transition graph* based on a similar concept [22]. However, ReAjax requires developers to manually execute Ajax Web applications to sufficiently trace the execution results.

Towards automated execution, Mesbah et al. implemented Crawljax [56, 60], which crawls Ajax Web applications by automatically emulating user actions. They proposed many tools involving Crawljax; Automatically Testing UI States of Ajax (ATUSA) can be used for detecting Ajax-specific faults, such as malformed HTML codes, 404 not found errors, dead clickable elements [59], and security-related DOM change and HTTP request violations [8]. Additionally, they ran Crawljax on different browser environments and compared these crawling results for testing the cross browser compatibility [58]; Choudhary et al. also used Crawljax to improve the accuracy of identifying cross browser issues [12]. Mesbah et al. also presented a tool called CILLA based on Crawljax, which determines unnecessary CSS rules from the results of sufficiently crawling the applications, resulting in reducing the size of CSS files to be transferred [57]. Since Crawljax is powerful for finding executable faults by automatically crawling Ajax Web applications, this has been the most successful study in the domain of state-based analysis and testing of the applications. In addition, Arzti et al. presented Artemis [7] as a way to improve code coverage by using the feedback-directed technique [77]. Artemis analyzes historical test execution data and generates test cases to explore the state space of Ajax Web applications. The authors noted that Artemis may help Crawljax determine what user actions should be emulated. However, Crawljax and Artemis cannot be combined in a way that would determine potential faults in Ajax Web applications.

In contrast, Amalfitano et al. proposed several state change criteria in Ajax Web applications and an interactive process for extracting finite state machines [4]. They constructed a tool called CreRIA that suggests state changes based on the criteria and developers can accept or reject the suggestions while executing Ajax Web applications. Fard et al. also leveraged human efforts towards automated test generation for Ajax Web applications [63]. Their tool called Testilizer first reads test cases given by developers and a state-flow graph based on the test cases. It then refines the state-flow graph by using Crawljax and finally generates test cases based on the refined state-flow graph. Such strategies involving developers in automated processes by tools are feasible for effectively and efficiently analyzing and testing Ajax Web applications. In this study, we also considered manual efforts of developers to input the IADP info, test data, and oracles.

However, the above dynamic approaches cannot verify the correctness of application behaviors because they leverage execution results. Our motivation for constructing JSPreventer is that Ajax Web applications may have not-easily-executable faults to be exposed.

7.2 JavaScript Control Flow Analysis

Regarding static approaches, Guha et al. proposed a static method to prevent Ajax Web applications from handling invalid server requests [33]. Their framework analyzes control flows in the JavaScript code of applications and constructs a *request graph* through control flow analysis, which represents how Ajax Web ap-

plications handle asynchronous server requests with an invariant order. Although their analysis results can be used for detecting invalid server requests at runtime, it was presumed with their approach that developers can correctly understand and implement application behavior. However, if developers cannot correctly implement Ajax Web applications, the request graph cannot reject invalid server requests relevant to potential faults. Thus, developers debugging the potential faults using JSPreventer can construct a more proper request graph from the debugged applications. Additionally, they pointed out that analyzing disabling event handlers is necessary to precisely monitor Ajax Web application behavior, which was their limitation. Our analysis scope covers application behavior containing such enabling and disabling interactions.

Zheng et al. also conducted a rules-based static analysis to detect data races due to asynchronous calls in Ajax Web applications [105]. Their proposed system first extracts JavaScript codes from Web pages that are dynamically generated by server-side scripts. It then parses the retrieved codes and identifies asynchronous calls, which are event handlers for user actions and asynchronous server responses; we deal with these asynchronous calls as the sort of user and server interactions in this study. To detect the *data inconsistency* and *atomicity violations* in global variables shared by these asynchronous calls, this system uses T.J. Watson Libraries for Analysis (WALA)[1], which performs pointer analysis for JavaScript programs. Here, JavaScript codes are implemented to dynamically manipulate Web page elements, which are also shared by these asynchronous calls; however, these elements are not always set to global variables of JavaScript codes. Since JSPreventer leverages Ajax design patterns as behavior oracles, it can identify faulty asynchronous calls without global variable-based data races. Additionally, Zheng et al. were not interested in testing unexpected behaviors in the applications if a detected data race occurred. Although they suggested two methods to fix data races, we argue that the actual errors found by JSPreventer can help developers debug their applications and confirm the correctness of the codes.

Jensen et al. modeled the HTML DOM and browser APIs as a set of abstract JavaScript objects [42]. This model can be used for precisely analyzing the control flow and dataflow of 'JavaScript' Web applications. Their objective was to detect or show absence of potential programming errors, such as unreachable code, in JavaScript programs used in the applications. However, since the basic building blocks of the client-side of Web applications are HTML, CSS, and JavaScript [101, 51], we argue that their abstraction of HTML and CSS might be too approximation for analyzing 'Ajax' Web application behavior. Therefore, we designed JSPreventer to analyze all the client-side codes. Additionally, Jensen et al. discussed the challenge of modeling a browser environment. For this challenge, we fortunately found ENVJS[2], which allows the JavaScript parser [67] used in JSPreventer to simulate a browser environment.

Guarnieri et al. introduced a pure static taint analysis for JavaScript code in order to identify security vulnerabilities such as cross-site scripting and SQL injection [32]. Wei et al. pointed out the dynamic feature of JavaScript; JavaScript code can dynamically get additional code from the server at runtime. They proposed a blended taint analysis of the JavaScript code that can be collected by executing test cases [102]. Although these analyses can output precise control flows of only the JavaScript code, JSPreventer analyzes stateful behaviors in the HTML, CSS, and JavaScript code of Ajax Web applications.

[1] `http://wala.sourceforge.net/`
[2] `http://www.envjs.com/`

7.3 Design Pattern Verification

Blewitt et al. conducted detection of the Gang of Four (GoF) design patterns [35] in Java using semantic constraints [9]. Their concern was that software evolution over time would violate properties of the design patterns in their original forms on the implementations. Additionally, Ghabi et al. addressed an issue of maintaining requirements-to-code traces [26] because software evolution also invalidates a requirements traceability matrix. In this study, we assume that information about IADPs is correct, and it would be interesting to determine how JSPreventer works with the incorrect information.

7.4 Client-Server Codes Traceability

Although we focused on HTML, JavaScript, and CSS codes of Ajax Web applications, these client-side codes can be generated using the server-side program codes such as PHP. Therefore, Nguyen et al. proposed mapping algorithms for tracing from the client-side to the server-side codes [70]. They proposed a tool called PhpSync, which finds the PHP `print` and `echo` statements from the server-side programs as the source of dynamically generated client-side Web pages and outputs approximate client-side HTML Web pages in the *D-model*. The PhpSync tool then uses the Tidy validator [90] to fix validation errors in the approximate HTML Web pages. This tool keeps relationships among the PHP code fragments and elements in the D-model so that PhpSync can propagate the fixes to corresponding parts of server-side PHP programs; Samimi et al. also presented an approach to automatically repair malformed HTML Web pages generated from PHP programs by using string constraint solving [85]. Additionally, Nguyen et al. extended PhpSync to a tool called Dangling Reference Checker (DRC), which detects dangling references across HTML, JavaScript, PHP, and SQL statements by using the D-model [69]. We plan to combine their traceability analysis technique with JSPreventer so that it will be able to locate actual faults on the server-side codes.

7.5 Mutation Analysis and Testing

In the late 70s, mutation testing was first proposed by DeMillo et al. [18] and Hamulet [34]. It seeds artificial faults into the program under test and then measures whether the seeded artificial faults can be detected in the test cases. Afterwards, many researchers developed effective mutation testing techniques specific to programming languages and paradigms, program specifications, and testing activities [75, 43]. Accordingly, Web-application-specific mutation testing techniques have been proposed. To address distinctive features of Web applications, Mansour et al. developed mutation operators based on a fault model of .NET Web applications [50]. Praphamontripong et al. also presented mutation operators for JavaServer Pages (JSP) Web applications by categorizing JSP-related faults [81]. Additionally, Shahriar et al. designed PHP- and JavaScript-specific mutation operators to inject cross site scripting (XSS) vulnerabilities to Web applications [87]. These studies were focused on the server-side complexities, i.e., Web applications that dynamically generate Web pages according to user requests at runtime. However, Ajax event-driven and asynchronous features have raised diverse complexities to be addressed in mutation testing.

Mutation testing for Ajax Web applications has been recently proposed as follows. **Bottom-up approach:** Since injected faults should be all-too-common,

mutation operators can be defined based on real faults previously made by developers. Indeed Ocariza et al. pointed out that public bug repositories for Web applications had few bug reports relevant to the client-side Ajax technologies [73], but Mirshokraie et al. collected real faults from the best practices by experienced programmers, such as JavaScript design patterns [76] and ant-patterns [38], for formalizing mutation operators [65]. **Top-down approach:** In contrast, Nishiura et al. proposed mutation operators specific to Ajax Web applications by conducting feature analysis of the applications [72]. Their proposed mutation operators inject comprehensive types of faults in Ajax Web applications; hence, developers may be able to improve test sets for detecting faults they did not make but will make. To the best of our knowledge, these approaches are state-of-the-art studies on mutation testing for Ajax Web applications.

In contrast, JSPreventer leverages a deformed mutation analysis technique, wherein seeded artificial delays do not act as faults to be detected but expose existing potential faults in Ajax Web applications. Although JSPreventer cannot be used to assess the adequacy of test cases, it can reveal actual errors due to potential faults. In addition, we assume an unexpected network latency may make potential faults executable, but JSPreventer could not make all the identified potential faults executable. Since the mutation operators proposed in the state-of-the-art studies were designed for injecting actual errors in JavaScript programs of Ajax Web applications, they may be used for reveal actual errors doe to the potential faults. Therefore, we are interested in using these mutation operators in addition to our delay-based mutation operators.

7.6 Debugging Concurrent Programs

As a basic approach to debugging a program under test, developers cyclically stop such a program during execution and check whether the program runs as expected. However, such a *cyclical debugging* approach may not be applicable for debugging concurrent programs, such as Ajax Web applications, because of not-easily-reproducible faults due to nondeterministic behavior of the programs [55]. Therefore, Carver et al. proposed a *deterministic execution debugging* technique, which uses semaphores and monitors to make a concurrent program under test sequentially runnable [11], enabling developers to debug concurrent programs in the same way as for sequential programs.

Hong et al. have a concern similar to our validation method, and addressed to find concurrency errors due to the improper timing of Ajax Web applications handling the interactions [37]. Their tool called WAVE is an extended implementation of the WebKit[3] browser framework for managing event invocations on the JavaScript engine; the WAVE preliminarily records an order of event invocations and handles the events in that order. However, exposed errors on a customized Web browser might not be equal to actual errors in a user environment. This is because the HTML rendering engine can also affect event invocations, such as the rendering block, as we mentioned in Section 5.2.2. Since our validation method modifies the source programs and enables the applications to sequentially run, resulting in revealing errors due to potential faults of Ajax Web applications on standard Web browsers that users actually use. Thus, our validation method can more precisely suggest erroneous behavior of the applications to developers.

[3] `https://www.webkit.org/`

7.7 Automated Program Repair

A remaining issue in this study is to support developers to debug Ajax Web applications. For addressing this issue in future, we consider to leverage automated program repair techniques.

Weimer et al. presented the *generate-and-validate* program repair technique, which automatically generates fault-fixed candidate programs and validates them through testing [103]. They mentioned that, since 2009, many researchers have been concerned with this technique to reduce software maintenance costs. To generate the candidates, their tool called GenProg uses mutation operators that add, remove, or replace statements in a program under test [104, 27]. They used mutation operators for repairing the program instead of injecting artificial faults; hence, their direction is similar to that in our validation method. Additionally, Kim et al. found fix-patterns from human-written patches in Java programs for automatic patch generation [47]. It would be interacting how the fix-patterns work for repairing Ajax Web applications.

Chapter 8

Conclusion

8.1 Summary

We addressed the challenges in preventive maintenance of Ajax Web applications. The first challenge was the blind spots of application behaviors that might not be executable on given execution scenarios and environments. Developers have troubles identifying potential faults from the blind spots; as a result, the potential faults will cause actual errors in a user environment. Therefore, our aim was to reveal these errors in a testing environment of developers to prevent users from encountering these errors.

Although the state-based approach may be effective in detecting executable faults in Ajax Web applications, we argued that the state-of-the-art studies relied on DOM-based dynamic approaches, so they could not detect potential faults from the execution results of the applications. Therefore, we investigated a static extraction method of a finite state machine representing interaction-based stateful behavior in Ajax Web applications (Chapter 3). From the results of our preliminary case study, we confirmed that the extracted finite state machines helped the participants find the errors even if some of them were not executable on the experimental environments. However, the participants claimed the non-negligible cost of manually determining the correctness of the extracted finite state machines.

Towards automated verification, the second challenge was that there were not generic behavior oracles relevant to interactions with Ajax Web applications. For this challenge, we leveraged Ajax design patterns to define the interaction invariants, which enabled our verification method to semi-automatically verify the properties described in the design patterns (Chapter 4).

Although our verification method reported the presence of potential faults in the applications, the third challenge was unknown conditions to reveal actual errors due to the potential faults; reported potential faults might not always lead to actual errors. To specify these conditions, we developed delay-based mutation operators, which can be used to find executable evidence of potential faults (Chapter 5). From the results of our case study on real-world applications, our proposed methods revealed actual errors and some of them indicated severe vulnerabilities in the applications. We expect that developers can use the revealed errors to debug potential faults in a similar way to executable faults. Therefore, we conclude that our proposed methods can support developers in conducting preventive maintenance for building highly reliable Ajax Web applications.

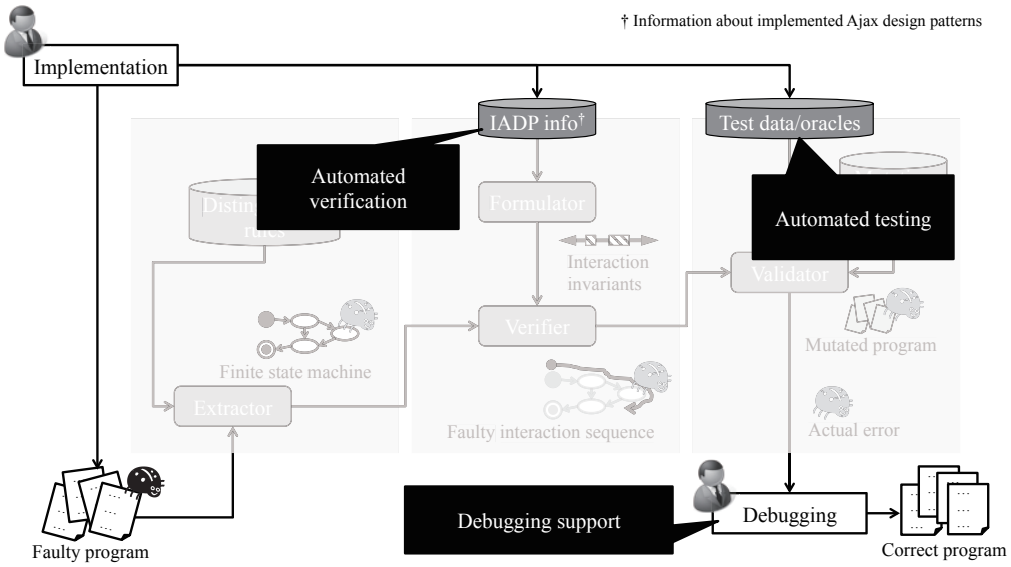


Figure 8.1: Future research directions

8.2 Future Work

Our future research will proceed in three separate directions described in Sections 8.2.1, 8.2.2, and 8.2.3. Figure 8.1 illustrates these directions based on the overview of our proposed methods in Figure 1.1. We also plan to follow up this thesis with additional studies described in Sections 8.2.4, 8.2.5, 8.2.6, and 8.2.7.

8.2.1 Debugging Support

Although we argued that developers can debug the potential faults in Ajax Web applications by using the revealed errors, debugging support is a remaining issue of this study. For addressing this issue, we are interested in leveraging automated program repair techniques [104, 27, 103]. These state-of-the-art studies on automated program repair relies on mutation operators that are originally designed to inject artificial faults. Such mutation operators should be defined specific to programming languages and paradigms, program specifications, and testing activities [75, 43]. Fortunately, several researches have been recently conducted to define effective mutation operators specific to Ajax Web applications [65, 72]. Hence, we can consider to leverage such Ajax-specific mutation operators to automatically repair Ajax Web applications so as to remove the potential faults.

8.2.2 Automated Verification

The IADP info should be given by developers to our verification method for instantiating the interaction invariants. We need to mitigate this additional task for developers. Blewitt el al. mentioned that *Design by Contract* (DbC) [62] can be used for automated verification purposes [9]. In DbC, developers implement program specifications, such as pre/postconditions and invariants, into the source code. Furthermore, Pei et al. have recently presented that these contracts can be used for automated program repair [79]. Therefore, we want to establish how developers can effectively implement interaction-related specifications into the source code of Ajax Web applications.

8.2.3 Automated Testing

To reveal actual errors, our validation method additionally requires test data and oracles from developers. Towards full-automation, we plan to combine our proposed method with the state-of-the-art studies on the state-based approaches; especially, the Crawljax tool is quite powerful to detect many types of faults in Ajax Web applications [56, 59, 8, 58, 7, 57, 60, 12]. The Crawljax team has recently presented the test data generation technique [63]. It would be interesting that this technique may work instead of the manual inputs. Actually, we are also interested in other techniques such as the search-based testing [3].

8.2.4 Extracted Finite State Machine

We designed our extraction method to extract a finite state machine that was sufficiently small for developers to manually review, whereas our verification method might mitigate the cost of this manual review with a model checking technique. Since this model checking technique can be applied to a large finite state machine, we plan to redesign our extraction method to extract a larger finite state machine that represents more precise stateful behavior in Ajax Web applications, might resulting in less false positives and negatives in JSPreventer, as described in Sections 6.4.2 and 6.4.3. To more precisely analyze the application behavior, we intend to extend our extraction method to analyze the execution context of events and functions implemented in Ajax Web applications. For this extension, we are interested in combining our extraction method with dynamic analysis techniques.

8.2.5 Expansion and Extension of Interaction Invariants

We heuristically defined interaction invariants from the literature of Ajax design patterns [49]; however, developers might have difficulty in defining additional interaction invariants from their own requirements. Therefore, we plan to build a domain-specific language (DSL) for this definition task in our verification method. We expect that this DSL supports developers in determining verification properties and relating appropriate property patterns to the properties.

Additionally, our validation method was developed to provide evidence that potential faults cause actual errors in Ajax Web applications if there are subtle network delays; however, it does not help for verifying that any network delays do not make potential faults executable. For this verification, we will extend our verification method to verify the correctness of timed automata for UPPAAL[1].

8.2.6 Program Mutation for Diverse Potential Faults

In our validation method, we focused on unexpected network delays that might cause severe problems in Web applications. However, these severe problems may be caused by a variety of factors. For example, in our case study described in Section 6.2, we observed that the Web browser prevented the subject applications from running on the faulty interaction sequences identified by our verification method, but anomalous Web browser behaviors may cause the severe problems such as cross browser compatibility. Thus, we need to establish a way of designing mutation operators for making diverse potential faults executable.

[1] `http://www.uppaal.org/`

8.2.7 Additional Case Studies

We will expand the range of case studies on real-world, large-scale, and practical Ajax Web applications. In addition, we also intend to use JSPreventer in actual Ajax Web application development projects.

References

[1] Hamid Alavi, George Avrunin, James Corbett, Laura Dillon, Matt Dwyer, and Corina Pasareanu. Property Pattern Mappings for CTL. http://patterns.projects.cis.ksu.edu/documentation/patterns/ctl.shtml.

[2] Alexa Internet, Inc. Alexa Top 500 Global Sites. `http://www.alexa.com/topsites`.

[3] Nadia Alshahwan and Mark Harman. Automated Web Application Testing Using Search Based Software Engineering. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 3–12, November 2011.

[4] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. An Iterative Approach for the Reverse Engineering of Rich Internet Application User Interfaces. In *Proceedings of the Fifth International Conference on Internet and Web Applications and Services*, ICIW '10, pages 401–410, May 2010.

[5] Giuliano Antoniol, Massimiliano Di Penta, and Michele Zazzara. Understanding Web Applications through Dynamic Analysis. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, WCPC '04, pages 120–129, June 2004.

[6] Alain April and Alain Abran. *Software Maintenance Management: Evaluation and Continuous Improvement*. Wiley-IEEE Computer Society Press, 2008.

[7] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Moller, and Frank Tip. A Framework for Automated Testing of JavaScript Web Applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, May 2011.

[8] Cor-Paul Bezemer, Ali Mesbah, and Arie van Deursen. Automated Security Testing of Web Widget Interactions. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 81–90, August 2009.

[9] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic Verification of Design Pattern in Java. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 224–232, November 2005.

[10] Gerardo Canfora and Massimiliano Di Penta. New Frontiers of Reverse Engineering. In *Proceedings of the Future of Software Engineering*, FOSE '07, pages 326–341, May 2007.

[11] Richard H. Carver and Kuo-Chung Tai. Replay and Testing for Concurrent Programs. *IEEE Software*, 8(2):66–74, 1991.

[12] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 702–711, May 2013.

[13] Alessandro Cimatti, Marco Roveri, Roverto Cavada, Roverto Sebastiani, Stefano Tonetta, Alessandro Mariotti, Andrea Micheli, Sergio Mover, and Michele Dorigatti. NuSMV: a new symbolic model checker. `http://nusmv.fbk.eu`.

[14] Crawljax. Crawljax. `http://crawljax.com/`.

[15] CSS Parser. CSS Parser - Welcome to CSS Parser. `http://cssparser.sourceforge.net/`.

[16] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2008.

[17] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34 –41, April 1978.

[18] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34 –41, April 1978.

[19] G.A. Di Lucca, A.R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini. WARE: a tool for the Reverse Engineering of Web Applications. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, CSMR '02, pages 241–250, March 2002.

[20] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, and Porfirio Tramontana. Reverse Engineering Web Applications: The WARE Approach. *Journal of Software Maintenance and Evolution: Research and Process*, 16(1-2):71–101, January 2004.

[21] Mark Driver, Ray Valdes, and Gene Phifer. Rich Internet Application Are the Next Evolution of the Web. Technical report, Gartner, Inc., May 2005.

[22] Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong Zhou. AJAX Crawl: Making AJAX Applications Searchable. In *Proceedings of the 25th IEEE International Conference on Data Engineering*, ICDE '09, pages 78–89, 2009.

[23] Joshua Duhl. White paper: Rich Internet Applications. Technical report, IDC, November 2003.

[24] Jason Farrell and George S Nezlek. Rich Internet Applications The Next Stage of Application Development. In *Proceedings of the 29th International Conference on Information Technology Interfaces*, ITI '07, pages 413–418, June 2007.

[25] Jesse James Garrett. Ajax: A New Approach to Web Applications. `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`.

[26] Achraf Ghabi and Alexander Egyed. Code Patterns for Automatically Validating Requirements-to-Code Traces. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, pages 200–209, September 2012.

[27] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '12, pages 3–13, May 2012.

[28] Jim Gray. Why Do Computers Stop and What Can Be Done About It? Technical report, Tandem Computers, June 1985.

[29] Michael Grottke, Allen P. Nikora, and Kishor S. Trivedi. An Empirical Investigation of Fault Types in Space Mission System Software. In *Proceedings of 2010 IEEE/IFIP International Conference on Dependable Systems and Network*, DSN '10, pages 447–456, June 2010.

[30] Michael Grottke and Kishor S. Trivedi. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. *IEEE Computer*, 40(2):107–109, 2007.

[31] Salvatore Guarnieri and Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM '09, pages 151–168, August 2013.

[32] Salvatore Guarnieri, Macro Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the World Wide Web from Vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 177–187, July 2011.

[33] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using Static Analysis for Ajax Intrusion Detection. In *Proceedings of the 18th International World Wide Web Conference*, WWW '09, pages 561–570, April 2009.

[34] Richard G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering (TSE)*, 3(4):279–290, 1977.

[35] Erich Helm, Richard Johnson, Ralph Vlissides, and John Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, October 1994.

[36] Anthony T. Holdener, III. *Ajax: The Definitive Guide*. O'Reilly Media, Inc., January 2008.

[37] Shin Hong, Yongbae Park, and Moonzoo Kim. Detecting Concurrency Errors in Client-side JavaScript Web Applications. In *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation*, ICST '14, pages 61–70, March 2014.

[38] Jack Hsu. JavaScript Anti-Patterns. `http://jaysoo.ca/2010/05/06/javascript-anti-patterns`.

[39] ISO 9241–210. Ergonomics of human-computer interaction – Part 210: Human centered design process for interactive systems. *International Standardiziation Organization (ISO)*, 2010.

[40] ISO/IEC 14764. Software Engineering — Software Life Cycle Processes — Maintenance. *International Standardiziation Organization (ISO)*, 2006.

[41] Mehdi Jazayeri. Some Trends in Web Application Development. In *Proceedings of the Future of Software Engineering*, FOSE '07, pages 199–213, May 2007.

[42] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 59–69, 2011.

[43] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering (TSE)*, 37(5):649–678, September 2011.

[44] jsoup HTML parser. jsoup Java HTML Parser, with best of DOM, CSS, and jquery. `http://jsoup.org/`.

[45] JUnit. JUnit. `http://junit.org/`.

[46] T. Katsimpa, Y. Panagis, E. Sakkopoulos, G. Tzimas, and A. Tsakalidis. Application modeling using reverse engineering techniques. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1250–1255, 2006.

[47] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '13, pages 802–811, May 2013.

[48] C. Larman and V.R. Basili. Iterative and Incremental Developments: A Brief History. *IEEE Computer*, 36(6):47–56, 2003.

[49] Michael Mahemoff. *Ajax Design Patterns*. O'Reilly Media, Inc., 2006.

[50] Nashat Mansour and Manal Houri. Testing Web Applications. *Information and Software Technology*, 48(1):31–42, 2006.

[51] Josip Maras, Maja Štula, Jan Carlson, and Ivica Crnković. Identifying Code of Individual Features in Client-Side Web Applications. *IEEE Transactions on Software Engineering (TSE)*, 39(12):1680–1697, 2013.

[52] Alessandro Marchetto, Filippo Ricca, and Paolo Tonella. A Case Study-based Comparison of Web Testing Techniques Applied to AJAX Web Applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(6):477–492, oct 2008.

[53] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based Testing of Ajax Web Applications. In *Proceedings of the First International Conference on Software Testing, Verification and Validation*, ICST '08, pages 121–130, April 2008.

[54] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. ReAjax: a reverse engineering tool for Ajax Web applications. *IET Software*, 6(1):33–49, 2012.

[55] Charles E. McDowell and David P. Helmbold. Debugging Concurrent Programs. *ACM Computer Survey*, 21(4):593–622, 1989.

[56] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling Ajax by Inferring User Interface State Changes. In *Proceedings of the 8th International Conference on Web Engineering*, ICWE '08, pages 122–134. IEEE Computer Society, July 2008.

[57] Ali Mesbah and Shabnam Mirshokraie. Automated Analysis of CSS Rules to Support Style Maintenance. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '12, pages 408–418, May 2012.

[58] Ali Mesbah and Mukul R. Prasad. Automated Cross-Browser Compatibility Testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 561–570, May 2011.

[59] Ali Mesbah and Arie van Deursen. Invariant-based Automatic Testing of AJAX User Interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, May 2009.

[60] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[61] Ali Mesbah, Arie van Deursen, and Danny Roest. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Transactions on Software Engineering (TSE)*, 38(1):35–53, 2012.

[62] Bertrand Meyer. Applying 'design by contract'. *IEEE Computer*, 25(10):40–51, October 1992.

[63] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. Leveraging Existing Tests in Automated Test Generation for Web Applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 67–78, September 2014.

[64] Miniwatts Marketing Group. World Internet Users Statistics and 2014 World Population Stats. `http://www.internetworldstats.com/stats.htm`.

[65] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript Mutation Testing. In *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*, ICST '13, pages 74–83, March 2013.

[66] Mozilla Developer Network and individual contributors. JavaScript. `https://developer.mozilla.org/docs/Web/JavaScript`.

[67] Mozilla Developer Network and individual contributors. Rhino. `https://developer.mozilla.org/docs/Mozilla/Projects/Rhino/`.

[68] Mozilla Developer Network and individual contributors. XMLHttpRequest. `https://developer.mozilla.org/docs/Web/API/XMLHttpRequest`.

[69] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Dangling References in Multi-configuration and Dynamic PHP-Based Web Applications. In *Proceedings of the 28th*

IEEE/ACM International Conference on Automated Software Engineering, ASE '13, pages 399–409, November 2013.

[70] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Auto-Locating and Fix-Propagating for HTML Validation Errors to PHP Server-side Code. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 13–22, November 2011.

[71] Jakob Nielsen and Hoa Loranger. *Prioritizing Web Usability*. New Riders Press, Berkeley, CA, 2006.

[72] Kazuki Nishiura, Yuta Maezawa, Hironori Washizaki, and Shinichi Honiden. Mutation Analysis for JavaScript Web Applications Testing. In *Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering*, SEKE '13, pages 159–165, June 2013.

[73] Frolin S. Ocariza, Karthik Pattabiraman, and Ali Mesbah. AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript. In *Proceeding of the 5th IEEE International Conference on Software Testing, Verification and Validation*, ICST '12, pages 31–40, April 2012.

[74] Frolin S. Ocariza Jr., Karthik Pattabiraman, and Benjamin Zorn. JavaScript Errors in the Wild: An Empirical Study. In *Proceedings of 2011 IEEE 22nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 100–109, November 2011.

[75] A. Jefferson Offutt and Ronald H. Untch. Mutation 2000: Uniting the Orthogonal. In *Mutation Testing for the New Century*, pages 34–44. Kluwer Academic Publishers, 2001.

[76] Addy Osmani. *Learning JavaScript Design Patterns*. O'Reilly Media, Inc., 2012.

[77] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, May 2007.

[78] Linda Dailey Paulson. Building Rich Web Applications with Ajax. *Computer*, 38(10):14–17, 2005.

[79] Yu Pei, C.A. Furia, M. Nordio, Yi Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *Software Engineering, IEEE Transactions on*, 40(5):427–449, May 2014.

[80] Perl.org. The Perl Programming Language. `https://www.perl.org/`.

[81] Upsorn Praphamontripong and Jeff Offutt. Applying Mutation Testing to Web Applications. In *Proceeding of the 3rd International Conference on Software Testing, Verification and Validation*, ICST '10, pages 132–141, April 2010.

[82] Filippo Ricca and Paolo Tonella. Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 25–34, May 2001.

[83] Filippo Ricca and Paolo Tonella. Understanding and Restructuring Web Sites with ReWeb. *IEEE MultiMedia*, 8(2):40–51, April 2001.

[84] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14:1–14:42, May 2009.

[85] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '12, pages 277–287, May 2012.

[86] SeleniumHQ. Selenium WebDriver. `http://docs.seleniumhq.org/projects/webdriver/`.

[87] Hossain Shahriar and Mohammad Zulkernine. MUTEC: Mutation-based testing of Cross Site Scripting. In *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems*, SESS '09, pages 47–53, May 2009.

[88] Stéphane S. Somé and Timothy C. Lethbridge. Enhancing Program Comprehension with recovered State Models. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 85–93, June 2002.

[89] Steve Souders. *High Performance Web Sites*. O'Reilly Media, Inc., 2007.

[90] SourceForge. HTML Tidy Library Project. `http://tidy.sourceforge.net/`.

[91] Brent Stearn. XULRunner: A New Approach for Developing Rich Internet Applications. *IEEE Internet Computing*, 11(3):67–73, 2007.

[92] The PHP Group. PHP: Hypertext Preprocessor. `http://php.net/`.

[93] The World Wide Web Consortium (W3C). Casscading Style Sheets. `http://www.w3.org/Style/CSS/`.

[94] The World Wide Web Consortium (W3C). Document Object Model (DOM). `http://www.w3.org/DOM/`.

[95] The World Wide Web Consortium (W3C). Extensible Markup Language (XML). `http://www.w3.org/XML/`.

[96] The World Wide Web Consortium (W3C). HTML. `http://www.w3.org/html/`.

[97] The World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 2.0. `http://www.w3.org/TR/xslt20/`.

[98] Cong Tian and Zhenhua Duan. Detecting Spurious Counterexamples Efficiently in Abstract Model Checking. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 202–211, May 2013.

[99] Porfirio Tramontana. Reverse Engineering Web Applications. In *Proceedings of the 21st International Conference on Software Maintenance*, ICSM '05, pages 705–708, September 2005.

[100] Jean Vanderdonckt, Laurent Bouillon, and Nathalie Souchon. Flexible Reverse Engineering of Web Pages with VAQUISTA. In *Proceedings of the 8th Working Conference on Reverse Engineering*, WCRE '01, pages 241–248, October 2001.

[101] Web Education Community Group. The web standards model - HTML CSS and JavaScript. `http://www.w3.org/community/webed/wiki/The_web_standards_model_-_HTML_CSS_and_JavaScript`.

[102] Shiyi Wei and Barbara Ryder. Practical Blended Taint Analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 336–346, July 2013.

[103] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE '13, pages 356–366, November 2013.

[104] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, May 2009.

[105] Yunhui Zheng, Tao Bao, and Xiangyu Zheng. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *Proceedings of the 20th International World Wide Web Conference*, WWW '11, pages 805–814, apr 2011.

Appendix A

Notations in Distinguishing Rules

Table A.1 lists notations used in the distinguishing rules.

Table A.1: Lists of notations used in distinguishing rules

Notation	Brief explanation
`Trigger`	A tag of the trigger rule.
`interact`	An attribute of `<Trigger>` representing the interaction type. (value: `User`, `Server`, or `Self`)
`event`	An attribute of `<Trigger>` representing the event type.
`repeatable`	An attribute of `<Trigger>` representing that an application repeatably handle the event or not. (value: `true` or `false`)
`Function`	A tag of the function rule.
`func`	An attribute of `<Function>` representing the function name.
`event`	An attribute of `<Function>` representing the source of event type that the function handles.
`callback`	An attribute of `<Function>` representing the source of callback function of the event handler.
`target`	An attribute of `<Function>` representing the source of target element where the function attaches the event handler.
`event_modifier`	An attribute of `<Function>` to modify the displayed text of the event type (value: `user_click` or `after_msec`)
`masking`	An attribute of `<Function>` representing that the target element is the popup box (value: `true` or `false`)
`Control`	A tag of the control rule.
`attr`	An attribute of `<Control>` representing the attribute name.
`disabled`	An attribute of `<Control>` representing the disable value.
`prop`	An attribute of `<Control>` representing the property source.
`value`	An attribute of `<Control>` representing the value source.
`cond`	An attribute of `<Control>` representing the condition to make the target element disabled.
`arg_`N	A value representing the Nth argument of the function.
`PropTarget`	A value representing the object property.
`ret`	A value representing the return value of the function.
`user_click`	A value representing the implicit user click event type.
`after_msec`	A value representing the event handled after the msec elapsed.

Appendix B

Benchmark Applications

To determine that our proposed methods properly work, we implement three benchmark Ajax Web applications. Since one of them is our motivating example described in Section 2.6, we describe other two applications called `FileDLer` and `QAsite`. Additionally, we show results of our proposed methods on these benchmark applications.

B.1 FileDLer

B.1.1 Implementation

We give the source code and screenshots of an Ajax Web application called `FileDLer`[1] in Figure B.1 and Figure B.2. FileDLer is a typical application of a file downloader Web service. This application (i) initially starts a countdown; (ii) displays a random password string and an input form when the count reaches zero; (iii) allows users to enter and submit a string; and then (iv) enables users to download the file if they give the correct password. In this application, we make two erroneous behaviors to be exposed, as listed in Table B.1.

(i) Countdown When users access this application, it first evaluates the page load event of `onload` (line 5), and then invokes the `countDown` callback function (lines 6-13). In this function, if the `count` is greater than zero, this application updates the count progress and calls back the `countDown` function after `1000` msec elapsed (lines 7-9), as shown in Figure B.2a. Otherwise, it proceeds in `getPwd` (lines 11 and 14-27) and then sends an asynchronous message (lines 15-25) by using the included `prototype.js` library (line 2). The `createPwd.php` returns a randomly-generated string as a correct password (line 16). After evaluating the `onSuccess` event, this application sets the response data to the `pwd` variable and displays it (lines 17-21).

(ii) Set input form This application sets an input-form and a submit-button. As the erroneous behavior #1, although developers expect that this application receives user inputs after it displays the correct password, it improperly

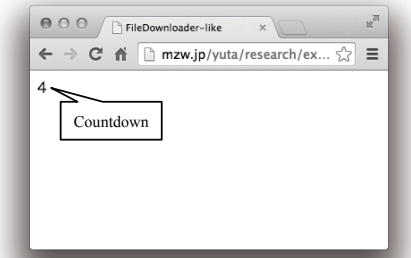
Table B.1: Erroneous behaviors in FileDLer

#	Brief explanation of erroneous behavior	Line(s)
1	Receiving user inputs before displaying correct password	20 and 26
2	Receiving user inputs after given password is validated	49

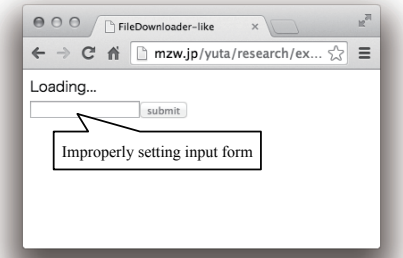
[1]Running examples are available from: `http://mzw.jp/yuta/research/ex/fd/`

```
<html><head>
  <script type="text/javascript" src="js/prototype.js"></script>
  <script type="text/javascript"><!--//
var count = 5, pwd;
window.onload = countDown;
function countDown() {
  if(0 < count) {
    updateProgress(count--);
    setTimeout(countDown, 1000);
  } else {
    getPwd();
  };
};
function getPwd() {
  new Ajax.Request(
    "createPwd.php", {
    onSuccess: function(request) {
      pwd = request.responseText;
      updateProgress(pwd);
      /* setForm(); // Proper */
    },
    onFailure: function(Request) {
      alert("Fail to get password");
    }
  });
  setForm(); /* Improper */
};
function setForm() {
  var ftext = document.createElement("input");
  ftext.onkeyup = inputFormText;
  var fsubmit = document.createElement("input");
  fsubmit.disabled = true;
  fsubmit.onclick = doSubmit;
  /* append ftext and fsubmit */
};
function inputFormText() {
  var len = $("ftext").value.length;
  if(0 < len) $("fsubmit").disabled = false;
  else $("fsubmit").disabled = true;
};
function doSubmit() {
  var val = $("ftext").value;
  if(val == pwd) {
    disableForm();
    enableDownload();
  } else alert("Input password  is invalid");
};
function disableForm() {
  /* $("ftext").disabled = true; // Improper */
  $("fsubmit").disabled = true;
};
function enableDownload() {
  appendTextContent("Click the following button to download");
  var dl_btn = document.createElement("input");
  dl_btn.onclick = doDownload;
  /* append dl_btn   */
};
function doDownload() {
  window.location.href = "path/to/file.ext";
  $("dl_btn").disabled = true;
  appendTextContent("Thank you for using our service");
};
function updateProgress(str) { /* set string in a progress field  */ };
function appendTextContent(str) { /* set string in a download field */ };
//--></script></head><body>
  <div id="progress"></div><div id="form"></div>
  <div id="download"></div>
</body></html>
```

Figure B.1: Source code of our benchmark application: FileDLer



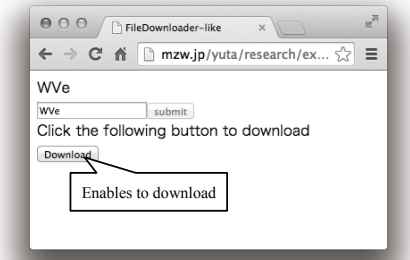
(a) Countdown



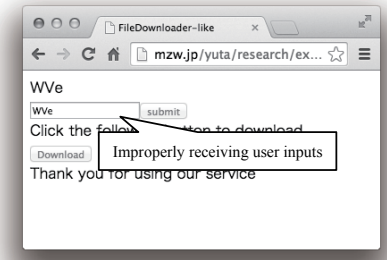
(b) Improperly set input form without displaying correct password

(c) Initially disabled submit button

(d) Enable to submit



(e) Enable to download



(f) Improperly available input form

Figure B.2: Screenshots of FileDLer

deploys the input-form regardless of the asynchronous communication result (line 26), as shown in Figure B.2b. If the communication fails, users can enter a string in the input-form without the correct password after clicking an alert box (lines 22-24) (Error#1 in Table B.1). Additionally, even if this application succeeds the communication, users might be confused to action in the input-form during this application is waiting for the response. Thus, this application might not run as expected by developers.

(iii) Input password and submit In the `setForm` function (lines 28-35), this application creates and appends two `input` elements corresponding the input-form and submit-button. To handle user actions on these elements, it then sets the `inputFormText` and `doSubmit` callback functions to the `onkeyup` and `onclick` event handler attributes of these elements, respec-

Table B.2: Given IADP info for FileDLer and verification results

#	Interaction invariant	**$Var1**	**$Var2**	$Result_v$
1	AsyncComm	Ajax.Request	UserEvents	Correct
2	ACRetry	onFailure	Ajax.Request	Fault
3	SRWait	inputFormText	onSuccess	Fault
4	UEHRegist	UserEvents	onload	Correct
5	UEHSingle	doDownload	UserEvents	Fault

tively (lines 30 and 33). Since the input-form does not initially have any user inputs; hence, the submit-button should be disabled (line 32), as shown in Figure B.2c. Users then cannot submit until they input in the input-form. Next, when users enter in the input-form, this application enables or disables the submit-button according to given string in the input-form (lines 37-39), as shown in Figure B.2d. If the given string equals to the correct password, this application enable users to download the file (lines 43-45). Otherwise, this application displays an alert box (line 46).

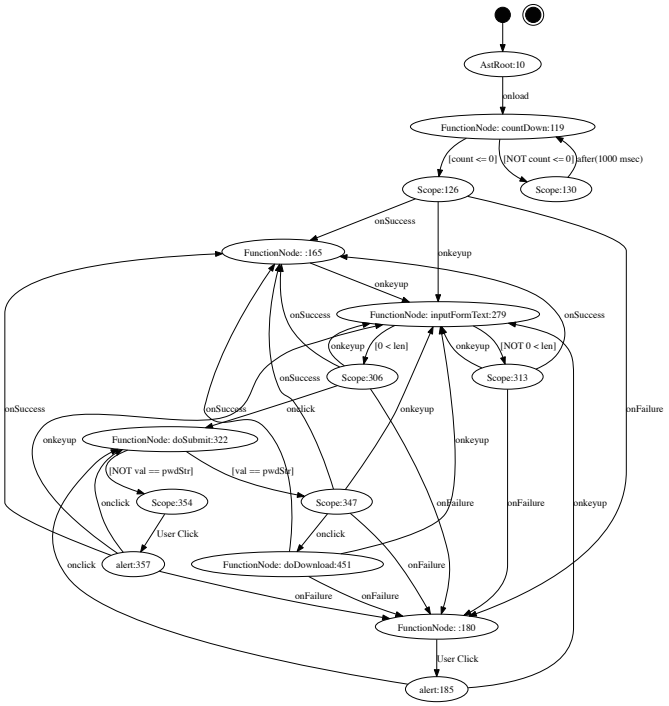
(iv) Download When users enter a valid password and submit it, this application displays the download-button in the `enableDownload` function (lines 52-57), as shown in Figure B.2f. It then disables the download-button after users start to download the file (line 60). Users no longer need to action on the input-form; however, developers might miss to disable the input form (line 49), as shown in Figure B.2f. In that case, this application might be confusing to users (Error#2 in Table B.1).

B.1.2 Results of Proposed Methods

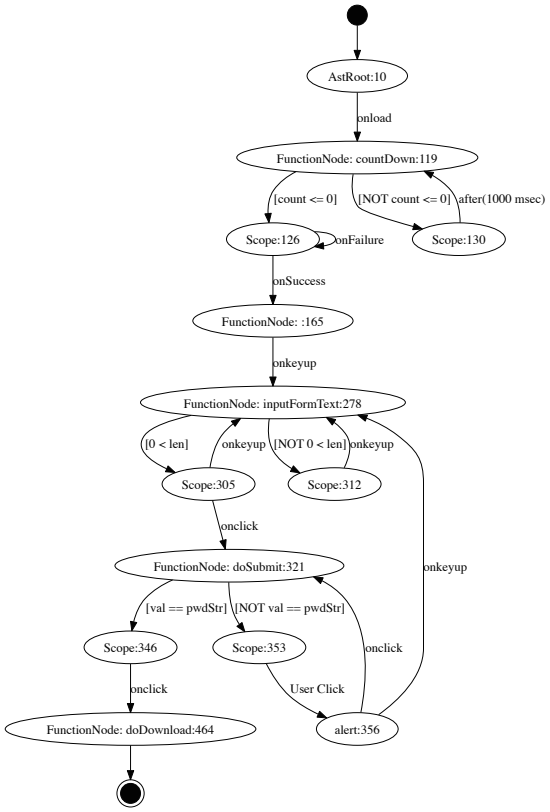
Figure B.3 shows finite state machines extracted from the faulty and correct versions of FileDLer by using our extraction method. We expect that these extracted finite state machine help developers to find errors in Ajax Web applications. For example, developers may find Error#1 in Table B.1 at the edge between `Scope:126` and `inputFormText` nodes. Additionally, developers may use the finite state machine extracted from the correct version, as shown in Figure B.3b, to determine the correctness of the application behavior. These extracted finite state machines are conformed to those we expect; therefore, our extraction method properly works in FileDLer.

We then execute our proposed verification method on the faulty version of FileDLer. When implementing this application, we did not consider any Ajax design patterns [49]; hence, we determine that all fundamental interaction invariants listed in Table 4.3 should be satisfied in FileDLer. As listed in Table B.2 **$Var1** and **$Var2**, we give IADP info to our verification method based on our heuristics. From the verification results are listed in Table B.2 $Result_v$, we find that FileDLer improperly runs when the asynchronous communication fails (invariant#2 in Table B.2). The fault verification results at invariant#3 and invariant#4 in Table B.2 correspond to Error#1 and Error#2 in Table B.1, respectively. Additionally, all the verification results on the correct version are correct. Thus, our verification method also properly works in FileDLer.

Next, we apply our validation method to the fault verification results. Table B.3 lists results of our validation results. As for invariant#5, this fault is



(a) From faulty version



(b) From correct version

Figure B.3: Finite state machines extracted from FileDLer

Table B.3: Validation results in FileDLer

#	Interaction invariant	Fault class
2	ACRetry	*-dependent potential fault
3	SRWait	Delay-dependent potential fault
5	UEHSingle	Executable fault

executable in our testing environments. If developers know the presence of this fault, they can easily find it. Our validation method works well for potential faults against invariant#3. This is because this fault is not executable in our localhost environment but our asynchronous delay mutation operator make this fault executable, resulting in revealing Error#1 in Table B.1. However, our mutation operators are not designed for making asynchronous communications failed; consequently, potential fault at invariant#2 is not exposed by our validation method. Note that we manually cause the communication failure and make sure of an occurrence of Error#2. Since all these validation results are those we expect, we can say that our validation method works properly in this application as well as our extraction and verification methods.

B.2 QAsite

B.2.1 Implementation

We implemented a simple Ajax Web application called *QAsite*[2]. QAsite is a typical Q&A website where users can ask and answer each other's questions, such as in `Experts Exchange`[3] and `Quora`[4]. In implementing QAsite, we referred to the `User Action`, `On-Demand JavaScript`, and `Direct Login` patterns described in the Ajax design patterns [49].

Figure B.4 briefly depicts the source code, while Figure B.5 shows screenshots of QAsite, which runs as follows.

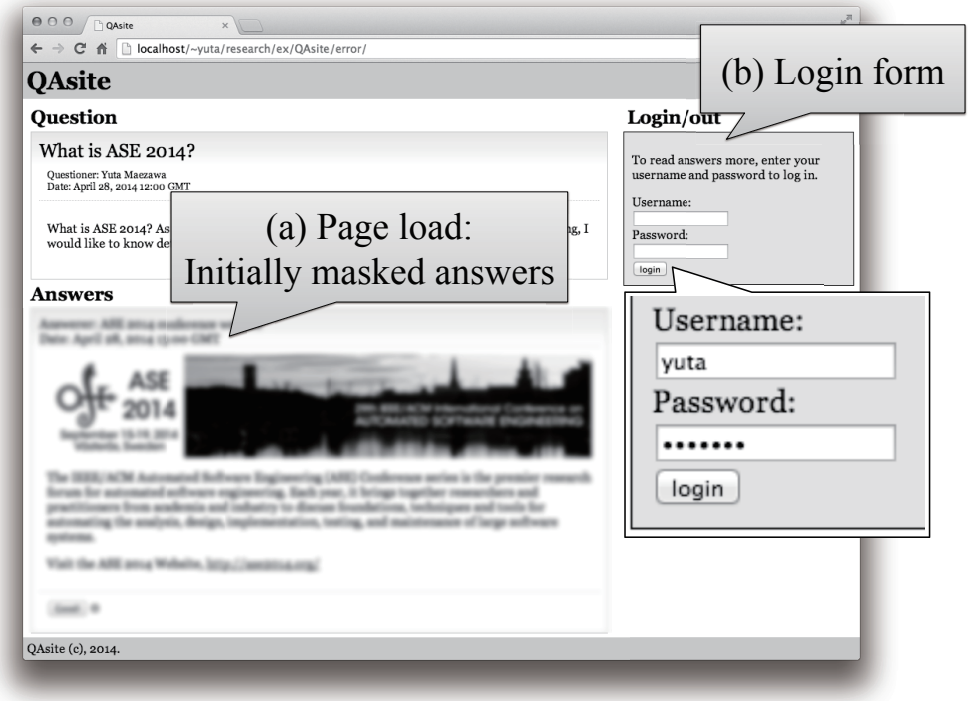
(i) Page load: Upon a page load, QAsite registers user event handlers for the login form and the "Good!" button (lines 6-11). The User Action pattern notes such implementations so that Ajax Web applications can handle user events anytime. Additionally, the QAsite masks the answers (line 60) to prohibit guest users from viewing them. This implementation is similar to the On-Demand JavaScript pattern, in which QAsite should handle the click on the "Good!" button after users login.

(ii) Login: We implemented the login form according to the Direct Login pattern. When a user sets the cursor on the input forms (lines 7-8), QAsite determines that the user intends to log in, and QAsite asynchronously retrieves `seed` data from the server (lines 13-21). When a user clicks on the login button (line 9), QAsite sends the username and a hash value with the password and seed data for secure communications to validate the account on the server side (lines 26-32). After receiving the validation result (line 30), QAsite creates a logout widget and disables the login form if the information is valid (lines 35-37).

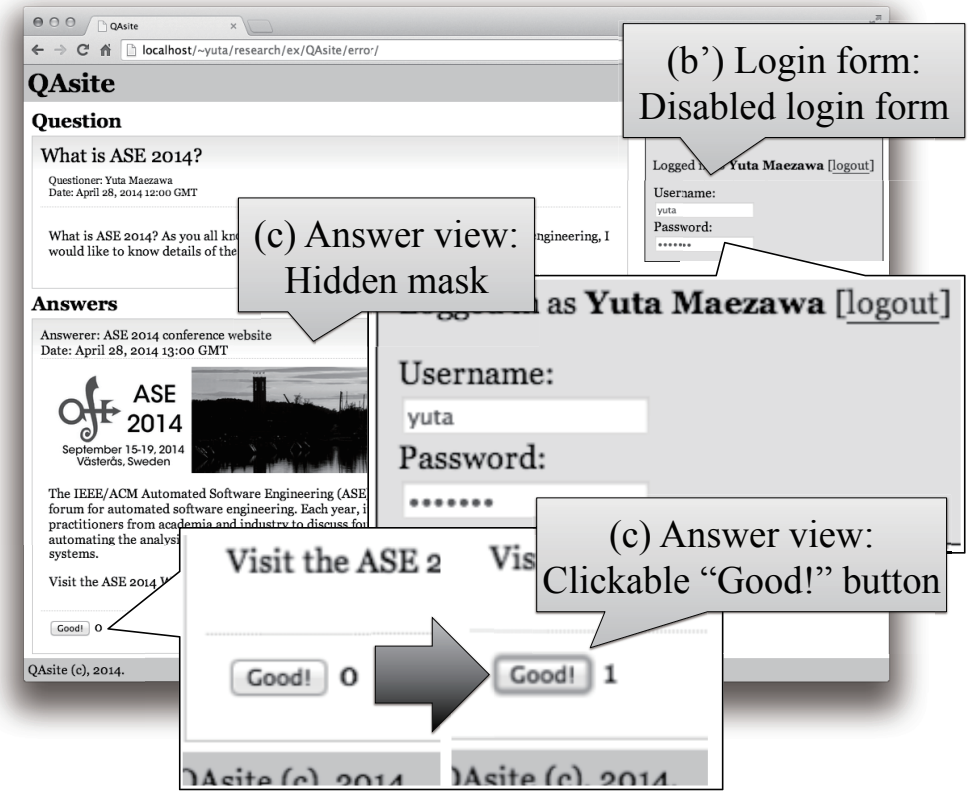
[2]Running examples are available from: `http://mzw.jp/yuta/research/ex/QAsite/`
[3]`http://www.experts-exchange.com`
[4]`http://www.quora.com`

```
<html><head>
  <!-- Load external JavaScript file  -->
  <script type="text/javascript" src="js/prototype.js"></script>
  <script type="text/javascript"><!--//
... /** User Action **/
window.onload = function() {
  $("username").onfocus = getSeed;
  $("password").onfocus = getSeed;
  $("login").onclick = validateLogin;
  $("good").onmousedown = onGood;
}
... /** Direct Login **/
function getSeed() {
  if(!loggedIn && !hasSeed) {
    new Ajax.Request(LOGIN_PREFIX, {
      method: "GET",
      parameters: "task=getseed",
      onSuccess: handleHttpGetSeed
    });
  }
}
function handleHttpGetSeed(res) {
  /** Set response to "seed" variable **/
}
...
function validateLogin() {
  new Ajax.Request(LOGIN_PREFIX,
    method: "GET",
    parameters: "task=checklogin&...",
    onComplete: tryLogin
  });
}
function tryLogin(res) {
  if(isSuccess(res)) {
    /* Create logout anchor
       Disable login form
       Make answer readable */
  } else {
    alert("Invalid account");
  }
}
... /** On-Demand JavaScript **/
function onGood() {
  /** Handle "Good!" button clicks **/
}
...
function tryLogout() {
  /* Disable logout anchor
     Enable login form
     Make answer masked */
}
//--></script>
</head>
<body>...
<h1>QAsite</h1>...
<h2>Question</h2>...
<h2>Answers</h2>...
  <button id="good">Good!</button>
...
<div id="mask"></div><!--for masking answer-->
...
<h2>Login/out</h2>...
  <input id="username" type="text" />
  <input id="password" type="password" />
  <input id="login" type="submit" />
...
</body>
</html>
```

Figure B.4: Source code of QAsite



(a) Before login



(b) After login

Figure B.5: Screenshots of QAsite

Table B.4: Given IADP info for QAsite and verification results

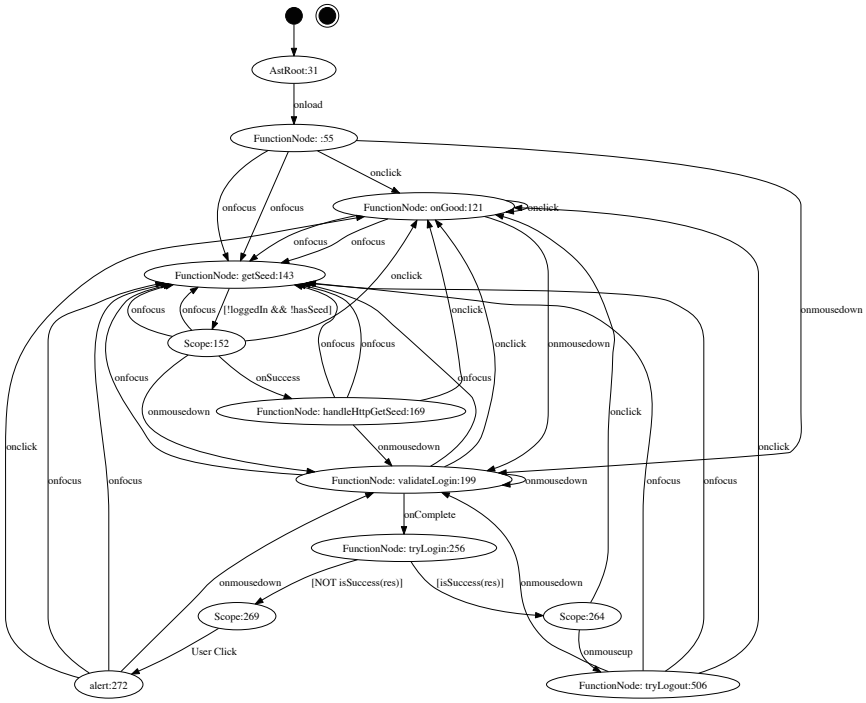
#	Interaction invariant	**$Var1**	**$Var2**	$Result_v$	Error# in Table B.5
3	SRWait	onGood	successLogin	Fault	2
4	UEHRegist	UserEvents	onload	Correct	–
4	UEHSingle	validateLogin	onmousedown	Fault	1
8	SeedRetrieve	validateLogin	handleHttpGetSeed	Fault	3
9	LFDisable	successLogin	onmousedown	Correct	–

(iii) Answer view: While it is creating the logout widget, QAsite removes the mask (lines 35-37). The user can then view the answers and click the "Good!" button (lines 10 and 43-45). When the user logs out, QAsite disables the logout widget, enables the login form, and masks the answers again (lines 47-51).

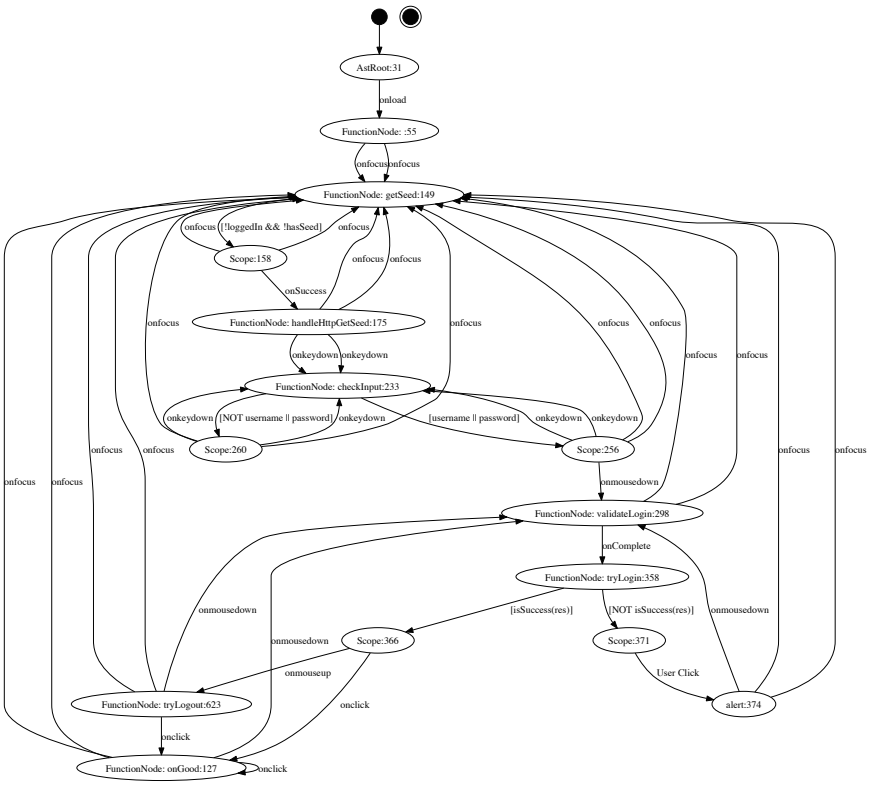
B.2.2 Results of Proposed Methods

Our extraction method extracts finite state machines, as shown in Figure B.6. Although we cannot find errors at a glance by using the extracted finite state machine, our verification method reports the presence of three potential faults, which are listed in Table B.5. Since we implement QAsite while considering Ajax design patterns, we can determine IADP info for this application, as listed in Table B.4, from the implemented Ajax design pattern while implementing.

Table B.3 lists results of our validation method. Since Error#1 is executable in our testing environment, we are easily find it. As for the erroneous behavior for the invariant#8, if QAsite runs on the representative Web browsers in a reliable network (e.g., with Firefox and Chrome browsers in a local host environment), Error#3 is not easily exposed; however, our validation method provides an executable evidence that subtle network delays makes it executable. In addition, we determine the code violation against the invariant#3 as a false positive in our verification method. This is because the "Good!" button is disabled due to the overlap with the mask element. Our proposed methods are limited to analyze such data-intensive impossible behaviors, as described in Section 6.4. Note that the element position corresponds to the data. Since we implement this application for illustrating limitations of our proposed methods, these results are conformed to our expectations i.e., our proposed methods properly works in QAsite.



(a) From faulty version



(b) From correct version

Figure B.6: Finite state machines extracted from QAsite

Table B.5: Erroneous behaviors in QAsite

#	Brief explanation of erroneous behavior	Actual error
1	QAsite might not prevent multiple calls to `validateLogin` when a user unexpectedly double-clicks on the login button.	Unnecessary server sessions
2	QAsite can handle `onGood` before successful login due to a dead link to the mask image file.	Invalid user operations
3	QAsite can send requests for login attempts without `seed` data. Users then cannot log in with their username and password.	Invalid login attempts

Table B.6: Validation results in QAsite

#	Interaction invariant	Fault class
3	SRWait	False positive
4	UEHSingle	Delay-dependent potential fault
8	SeedRetrieve	Delay-dependent potential fault