HAKI: A RUNTIME VERIFICATION TOOL FOR JAVASCRIPT MVC WEB APPLICATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY
BY

IBRAHIM BILGE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

DECEMBER 2016

HAKI: A RUNTIME VERIFICATION TOOL FOR JAVASCRIPT MVC WEB APPLICATIONS

Submitted by İbrahim Bilge in partial fulfillment of the requirements for the degree of **Master of Science in The Department of Information Systems Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin
Director, **Informatics Institute** _____

Prof. Dr. Yasemin Yardımcı Çetin
Head of Department, **Information Systems** _____

Assoc. Prof. Dr. Aysu Betin Can
Supervisor, **Information Systems** _____

Examining Committee Members:

Assoc. Prof. Dr. Altan Koçyiğit
Information Systems, Middle East Technical University _____

Assoc. Prof. Dr. Aysu Betin Can
Information Systems, Middle East Technical University _____

Assist. Prof. Dr. Erhan Eren
Information Systems, Middle East Technical University _____

Assist. Prof. Dr. Çağdaş Evren Gerede
Computer Engineering, TOBB ETU _____

Assoc. Prof. Dr. Banu Günel Kılıç
Information Systems, Middle East Technical University _____

Date: 22.12.2016

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this wok.

Name, Last name : İbrahim BİLGE

Signature : _____

ABSTRACT

HAKI: RUNTIME VERIFICATION TOOL
FOR JAVASCRIPT MVC WEB APPLICATIONS

Bilge, İbrahim
M.S., Department of Information Systems
Supervisor: Assoc. Prof. Dr. Aysu Betin Can

December 2016, 56 pages

In this thesis, we propose an efficient approach for locating inconsistencies in view-model bindings of JavaScript AngularJS web applications. JavaScript is one of the most common scripting languages used for developing web applications. It can be used to create flexible, efficient code thanks to its highly dynamic nature. In addition, many structural frameworks have been developed for building even more extensible and more dynamic web applications. One of the most popular of these frameworks is AngularJS which uses the MVC (Model-View-Controller) pattern. The dynamism of JavaScript including abstraction and layering of AngularJS can make coding very challenging by making it open for mistakes and vulnerable to inconsistencies that create unreadable, not maintainable, and particularly, unreliable code. In addition, custom web components remain a challenge for verification and consistency of the applications whereas these components are considered the biggest feature of JavaScript frameworks nowadays. Differing from the existing studies in literature, our aim in this study is to propose an effective and compact approach to locate inconsistencies in view-model bindings including type related errors and errors about custom web components. We introduce a tool called HAKI that executes runtime verification process on JavaScript - AngularJS applications and locates errors and warnings by using dynamic analysis. We evaluated our tool using two sets of experiments, one large scale real application and one smaller application with injected faults. Our tool located 55 errors in real application with 8 of them are evaluated as major errors; it also reported 35 warnings that can possibly cause errors. In addition, the runtime verification overhead is minimal.

Keywords: JavaScript; runtime verification; MVC pattern; data binding.

ÖZ

HAKI: JAVASCRIPT MVC WEB UYGULAMALARI İÇİN ÇALIŞMA ZAMANI DOĞRULAMA ARACI

Bilge, İbrahim
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Doç. Dr. Aysu Betin Can

Aralık 2016, 56 sayfa

Biz bu tezde, JavaScript – AngularJS uygulamalarının view ve model bağlantılarında oluşan tutarsızlıkları etkin bir şekilde tespit edebilen bir yaklaşım sunmaktayız. JavaScript, günümüzde web uygulaması geliştirme alanında kullanılan en yaygın betimleme dillerinden biridir. Oldukça dinamik olan yapısı sayesinde esnek ve etkin kod yazma imkânı sağlar. Buna ek olarak birçok yapısal çatı geliştirilmiştir. Bu çatıların en popüler olanlarından biri MVC (Model-View-Controller) mimarisini kullanan AngularJS'dir. JavaScript'in dinamik yapısına, AngularJS'in soyut ve katmanlı mimarisi de eklenince kodlama yapmak oldukça zorlayıcı bir hale gelebilir. Öyle ki geliştirilen yazılımı okunamaz, bakım yapılamaz ve özellikle güvenilemez bir hale getirebilir. Ayrıca günümüzde uygulamaların tutarlılığını ve doğrulamasını zorlaştıran kişiselleştirilmiş bileşen geliştirebilme imkanı da bu JavaScript çatılarının en büyük özelliklerinden biri olarak sayılmaktadır. Literatürdeki diğer çalışmalardan farklı olarak bizim bu çalışmadaki amacımız view ve model bağlantılarında oluşan tutarsızlıkları etkin bir şekilde tespit edebilen bir yaklaşım sunmaktır. Bu amaçla HAKI adını verdiğimiz aracı geliştirdik. Aracımızı 2 farklı uygulama üzerinde test ettik; bir büyük çaplı gerçek bir uygulama ve bir de daha küçük kapsamlı, içerisine hatalar yerleştirilmiş bir uygulama. Aracımız ilk uygulama için 8 tanesi önemli olmak üzere 55 hata ve hataya neden olabileceğini düşündüğümüz 35 tane de uyarı tespit etmiş ve raporlamıştır. Ayrıca bu deneyler sırasında HAKI sebebiyle oluşan performans kaybının göz ardı edilebilir derecede minimum olduğu görülmüştür.

Anahtar Sözcükler: JavaScript; çalışma zamanı doğrulama; MVC örüntüsü; veri bağlantısı.

Never laugh at live dragons…

ACKNOWLEDGMENTS

I would like to thank my supervisor Assoc. Prof. Dr. Aysu Betin Can for his encouraging, advice and guidance in this study.

I would like to thank my family, collogue and friends for their support, motivation and encouragement during the study.

I would like to thank my loving wife particularly for her presence and infinite support anytime I need.

I would also like to express my gratitude to the examining committee members for their valuable feedback.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

MVC	Model-View-Controller Design Pattern
DOM	Document Object Model
HTML	Hypertext Markup Language
DSL	Domain Specific Language
IT	Information Technologies
JS	JavaScript
SPA	Single Page Application
XML	Extensive Markup Language
AJAX	Asynchronous JavaScript and XML
XSS	Cross Site Scripting
DTD	Document Type Definition
JSF	Java Server Faces
PHP	PHP Hypertext Preprocessor
E2E	End to End

CHAPTER 1

INTRODUCTION

1.1 Motivation

Since its inception in the mid-90s, JavaScript has become one of the most popular Web development languages. In September 2014, an industry analyst firm, called RedMonk, showed JavaScript as the top language among web development language [1]. Much of this popularity comes from JavaScript's ability to deliver rich, dynamic web content. Also it is relatively lightweight and is easy to use.

Developers prefer JavaScript applications to traditional server-side web applications. The reason is that JavaScript offers a more responsive user experience because of its dynamic nature. Although most languages have some aspect of dynamic behavior, JavaScript has pretty much everything about dynamism like dynamic variables, dynamic types, dynamic functions and objects. JavaScript applications can be developed without using a framework. However, it is so easy to run into trouble because code management and refactoring quickly becomes a challenge with native JavaScript and often leading to a bad structured code. Modern JavaScript frameworks offer a way around the problem of code management by providing well-defined application architectures using the MVC design pattern that can greatly ease development. So choosing one of these frameworks should help to have highly responsive user interfaces along with well-structured and maintainable code, which have considerable benefits in the long run.

Although the MVC framework for JavaScript has certain benefits, there are also potential problems regarding the variable binding between the View and the Model components. For example, AngularJS has automated data binding which is a favorable feature. This feature describes the condition where data is bound to an HTML element in the View and the Model can update the data. That HTML element in the View has the ability to display those data. Although the implementation of this type of data binding reduces the amount of effort to create dynamic views, it complicates debugging and has potential dangers in larger, more complex applications that are

developed by multiple teams from multiple locations, since the errors in bindings do not fire any exception for most of the cases. Consider the example in Figure 1 that creates a loop which renders the inner html content of the div element for each phone of the user defined in the model.

```
1. <div ng-repeat="phone in user.phones">
2.     <input ng-model="phone.number"/>
3. </div>
```

Figure 1. Example for basic data binding

There is the "two-way data binding" for the `ng-model` attribute of the input element. If the View also can update the data, then it is called "two-way". It will update the view automatically when the phone number of the user is changed in the model and vice versa.

The requirement for this script to work properly is the definition of the user object in the related model of this view. Also the user object should have an array field for the phone data. If the `user.phones` or `phone.number` is undefined, this line of code will not work as expected and no error will be generated. Moreover, it is not possible to put a breakpoint at this line of code for debugging purposes. There is no effective way for debugging these types of errors, since it is not a part of the JavaScript code. The code is in a view file which is just an HTML file. The variables like "`user.phones`" and "`phone.number`" are non-executable texts for an HTML file.

1.2 Problem Definition

Our study focuses on locating errors and warnings of inconsistencies about these data bindings automatically in the development phase. There are a few studies ([35], [36], [37], [38]) about this problem but there is no effective solution for type mismatches and custom web components. Custom web components are critical in web development. Almost all of the JavaScript frameworks support custom components which enable the extension of HTML by developing custom reusable components and create a Domain Specific Language (DSL). This feature became so popular that it was announced as a web standard in 2011 [11] and all major browsers have started implementing the technologies needed to run web components natively. While browser vendors are still working on native implementations, most of the JavaScript frameworks have already made web components available to developers.

We propose an approach to help developers locate errors about bindings in both built-in attributes and attributes of custom components developed within the JavaScript MVC framework. Our approach consists of four main steps. We register the state transitions of the page, then, we search for attribute changes in the "Document Object Model" (DOM). When we find

an attribute with a value in the view, we evaluate that value within the context of the associated model. After getting the result of that evaluation, we check for inconsistencies and specifically for errors about variable definitions and type mismatches.

We implemented our approach as a tool called *HAKI*, which is designed to work for AngularJS applications since it is the most popular JavaScript MVC framework. Our tool is used in a big scale JavaScript MVC web application to measure its ability to locate real errors in a real-world application. In addition, for accuracy assessment, we used HAKI in an application where we applied a systematic fault injection process. HAKI located 55 real errors 8 of which were flagged as blocking defects in the real application and was able to locate all 18 of the injected errors in our application.

1.3 Running Example

The running example is a part of an application used by an imaginary airline company. We refer to this motivating example as *FlightApp* throughout the study.

FlightApp is an AngularJS application with two routes. Customers use the first route, namely `SearchFlights`, to select the travel cities and dates. When they click the "*List Available Flights*" button, they are redirected to the second route called `ListFlights`. Here, the information about the available flights is listed. If there is no flight available, a message is shown to inform the customer. Also there are links for every flight for selection.

```
1. <!-- The View of the route -->
2. <label for="flyingFrom">Flying From?</label>
3. <input id="flyingFrom" type="text"
4.        ng-model="selectedCity.from"/>
5.
6. <label for="flyingTo">Flying To?</label>
7. <input id="flyingTo" type="text"
8.        ng-model="selectedCity.to"/>
9.
10. <label for="departDate">Departure Date?</label>
11. <input id="departDate" type="date"
12.        ng-model="selectedDate.depart"/>
13.
14. <label for="returnDate">Returning Date?</label>
15. <input id="returnDate" type="date"
16.        ng-model="selectedDate.return"/>
17.
18. <button type="submit"
19.         ng-click="searchFlights()">
20.     List Available Flights
21. </button>
```

```
22.
23. <script type="text/JavaScript">
24. //* The Controller of the route -->
25. flightApp.controller("SearchFlightCtrl",
26.     function ($scope, RoutingService) {
27.
28.         //* The Model of the route -->
29.         $scope.selectedCity = {from: "", to: ""};
30.         $scope.selectedDate = {
31.             depart: new Date(),
32.             return: new Date()
33.         };
34.
35.         $scope.searchFlights = function () {
36.             RoutingService.redirect("/flight-list",
37.                 $scope.selectedCity,
38.                 $scope.selectedDate,
39.             );
40.         };
41.     }
42. );
43.
44. </script>
```

Figure 2. The View, Controller and Model of SeachFlights

The View of the `SearchFlights` route includes two inputs for the customer to enter the cities that she wants to travel from and travel to respectively. There are also two date input elements to select the travel dates and a button to submit these values. The Controller populates the model with default values and defines a function to handle the click event of the button (Figure 2). This function redirects application to the `ListFlights` state with entered parameters.

```
1.  <!-- The View of the route -->
2.  <div class="msg" ng-hide="availableFlights.length">
3.    There is no flight available.
4.  </div>
5.  <flight-table
6.      header="Available Flight List"
7.      flight-list="availableFlights"
8.      flight-selected="chooseFlight">
9.  </flight-table>
10.
11. <script type="text/JavaScript">
12. //* The Controller of the route -->
13. flightApp.controller("ListFlightCtrl",
14.     function ($scope, RoutingParams, RemoteService) {
15.
16.         //* The Model of the route -->
17.         $scope.availableFlights = RemoteService.query({
18.             RoutingParams.selectedCity
19.             RoutingParams.selectedDate
```

```
20.         });
21.
22.         $scope.chooseFlight = function (id) {
23.             alert("The flight "+ id +" selected!");
24.         };
25.     }
26. );
27.
28. </script>
```

Figure 3. The View, Controller and Model of ListFlights

The view of the `ListFlights` route includes a message to inform the user when there are no flights available to list. The table `flight-table` is used to list the available flights. This element is a custom web component and it has three attributes: `header`, `flight-list`, `flight-selected` respectively. The Controller invokes a *Service* to query the flights and populates the Model with the result. Also there is a function named `chooseFlight` to set the `flight-selected` attribute of the custom element (Figure 3).

1.4 Outline

This thesis organized in seven chapters as follows:

1. **Introduction**. It includes motivation, problem definition and our running example.
2. **AngularJS and MVC**. It focuses on background of the problem and defines MVC Architecture. It also defines properties of the framework.
3. **Literature Review**. It lists related studies.
4. **Problem Statement.** It defines the problem in two sections.
5. **Formal Model and The Tool**. It explains the formal model, approach and implementation.
6. **Experiments and Results**. It contains evaluation of the tool.
7. **Conclusion.** It includes limitations and future work sections.

CHAPTER 2

ANGULARJS AND MVC

JavaScript was initially developed as a browser-agnostic scripting language; however, in recent years, it has evolved beyond the browser to areas such as mobile and server-side web applications. Over the next few years, JavaScript is poised to become the dominant language of the enterprise for IT [12]. This evolution is leading to an increase in the number of the new JavaScript tools and frameworks.

There are back-end frameworks, such as the web application framework NodeJS [3] and real-time application framework EngineIO [4] which are capable of transporting real time information using various methods and are suitable for testing using JS libraries, such as Mocha [5] and Should.js [6]. There are also front-end development frameworks, such as Ember.js [7], Backbone.js [8], Knockout.js [9], and AngularJS [10].

We focus on JavaScript MVC frameworks, particularly on AngularJS. Generally, developers prefer client-side JavaScript to traditional server-side web applications to increase responsiveness of their applications. For example, when a user clicks a button, instead of sending an entire page and waiting for the page to reload as in a traditional server-side web application, JavaScript frameworks only send required data to the server and then load portions of the page as the user interacts with them, thus speeding up the responsiveness of the user interface. The aim is to create a user interface that feels as fast as a native application.

The definition of AngularJS as put by its official documentation is as follows [12];

“AngularJS is a structural framework for dynamic web applications. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application's components clearly and succinctly. AngularJS’s data binding and dependency injection eliminate much of the code you currently have to write. And it all happens within the browser, making it an ideal partner with any server technology.”

AngularJS became the most popular JavaScript framework quickly with its strong features, such as:

- being capable of creating Single Page Applications (SAP) in a very clean and maintainable way,
- providing data binding capability to HTML thus giving user a rich and responsive experience,
- providing unit testable code,
- using dependency injection and making use of separation of concerns,
- providing reusable components,
- letting the developer write less code and get more functionality,
- providing views which are pure html pages, and having controllers written in JavaScript for business processing,
- being able to run on all major browsers and smart phones including Android and iOS based phones/tablets.

Although AngularJS has many advantages, there are also certain concerns:

- Detecting of inconsistencies in data bindings can easily become a problem, since there is not even any error message addressing the problem when an inconsistency occurs for the most of the cases.
- When an inconsistency is detected, debugging the code is problematic as in other MVC JavaScript frameworks due to the fact that exceptions are fired by the mechanism called digest-cycle. Then browser generate a stack trace that shows the mechanism as the source of the error.

As an example of these concerns, consider the code in Figure 1. It is difficult to debug the code when an error occurs. This is due to the mechanism called "*digest cycle*" which is responsible for automated data binding. This script re-renders the view with the values of the variables in the related model every time these variables are updated. The errors that occur in this JavaScript code are caught by the internal interceptors, and interpreted by the browser as a caught error, and then a stack trace is generated that shows the digest cycle as source of the error. A binding error about any of the variables in the whole application will produce the exact same stack trace; therefore, it is hard to locate the actual source of the error using the debugger.

In this study, we focus on locating errors in these bindings using runtime verification methodology. Our approach provides a considerable improvement for the development phase of MVC JavaScript applications by helping to locate the error between the View and the Model as applications scale-up.

2.1 MVC Architecture

Model-View-Controller, or shortly MVC, is a software design pattern which is very popular in web application development. An MVC pattern consists of three parts.

2.1.1 Model

The model is responsible for managing application data. It responds to the requests from the View and to the instructions from the Controller to update itself. It corresponds to JavaScript objects for AngularJS as shown in Figure 4.

2.1.2 View

The view is responsible for displaying all or a portion of the data to the user. View presents the data in a particular format, triggered by the controller's decision. The particular format is HTML for AngularJS and it means values of attributes are just texts with no meaning from the point of View. On the other hand, these values may be defined as variables in the model and the framework handles that binding. The View corresponds to DOM in Figure 4 which is generated by browsers from the HTML code.

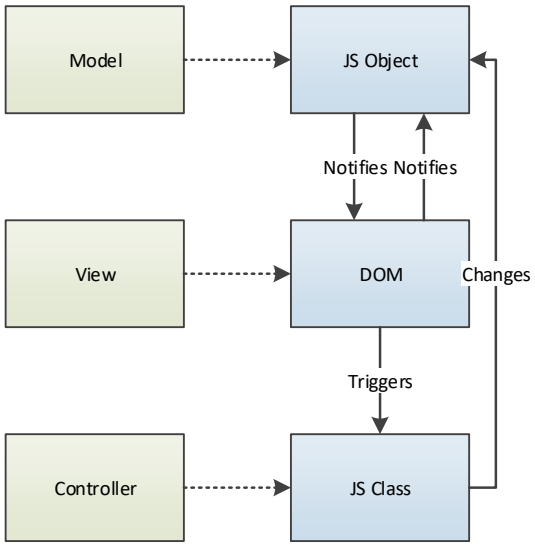

Figure 4. MVC Structure of AngularJS

2.1.3 Controller

Controller is the part of the software that controls the interactions between the Model and the View. The controller responds to user input and performs interactions on the data model objects. The controller receives input, validates it, and then performs business operations that modify the state of

the data model. It is a JavaScript class for AngularJS as shown in Figure 4. Controller is related to a subtree of the DOM. This relation creates inheritance between controllers since they may relate each other's subtrees.

MVC is popular because it isolates the application logic from the user interface layer and supports separation of concerns. The controller receives all requests for the application and then works with the model to prepare any data needed by the view. The view then uses the data prepared by the controller to generate a final presentable response.

2.2 Data Binding

Data binding is the automatic synchronization of data between model and view components. It is the most known feature of AngularJS as many supporters, especially Google, list two-way binding as a main framework benefit [12].

Indeed, automated data binding is a good feature, and its easy implementation is impressive for those with a background in an imperative library like JQuery [13] where such behavior requires several lines of code to implement. All the details are hidden in the black box that drives the logic of data binding in AngularJS.

In traditional MVC frameworks, models are the application's connection to the backend data source. When the application is used, models are accessed (by the controller) and the retrieved data is blended with the view template to form the page which the user ultimately interacts with. As the user clicks around, the controller continues to query the model for data. Once the data is returned, it becomes available to the view at "render time" – usually on page reload. "Asynchronous JavaScript and XML" (AJAX) helps with some parts of this process by eliminating the burden of page reloading just to update potentially small parts of data. Nevertheless, an explicit view change is still required and, more importantly, additional effort is required from the developer to provide this functionality.

AngularJS takes a different approach to the model concept. The view and the model are intertwined in an AngularJS application. Views are considered a projection of the current model state, as data sourcing from the view is handled by the model and then turned around and fed right back into the view immediately. In fact, no developer effort is required to provide this binding effect.

2.3 Scope Object

Scope is a special object in AngularJS. It is injected to the Controller and populated with the Model references to be ready for use in the View. In

other words, scopes are objects that refer to the Model. They act as the glue between the Controller and the View. The Controller should populate the scope object using plain JavaScript objects or services, which represent the real model, and then, the scope is used in the render phase of the view.

We use the term "*model"* in the remaining part of this paper instead of the term "scope" for simplicity, since the scope refers to the model.

Controllers are related to a certain subtree of the DOM as mentioned above. They are not directly related to the View, but use the Model to render the variables in the View. There can be many active models on an HTML page and they can be nested, which means the inner ones inherit from the parents as in inheritance in OOP. This makes locating of inconsistencies (manual or automated) harder.

2.4 Directive (Custom Component)

Directives are markers on DOM elements (such as elements or attributes). These can be used to create custom HTML tags that serve as new, custom widgets to create a Domain Specific Language. AngularJS has some built-in directives to support generic web development requirements. For example, `ng-show` is a built-in attribute that takes a value in the View. This value has to be an expression of Boolean type within the context of the model. If the evaluated result is false, then `ng-show` will hide the HTML element that it is related to. There are many built-in special attributes like `ng-show` and it is possible to implement custom components with custom special attributes.

2.5 Routing

The Routing is the concept of switching views; it enables creation of Single Page Applications (SPA) easily as it can switch views just in a portion of the page. The state of the application changes when the route transition happens. AngularJS has a built-in router that enables creation of routes with view and controller groupings. These routes can be used for transition when triggered by an event such as the clicking of a link by the user.

CHAPTER 3

LITERATURE REVIEW

JavaScript is becoming the most popular web application development language. The number of studies about JavaScript has started to increase in recent years [1]. Although there are studies about the analysis of JavaScript code, the most of these studies are only inside the JavaScript code itself. There are limited studies about the relation between JavaScript code and HTML code (data binding) to the best of our knowledge. This limitation is reasonable since the application of the MVC framework to web application programming is a fairly recent improvement.

3.1 Code Analysis for JavaScript Applications

Chugh et al. focus on the staged analysis of JavaScript code and finding information flow violations [15]. They study information flow properties by reading document cookies and URL redirects. A valuable feature of that work is its support for dynamically loaded and generated JavaScript in the context of what is generally thought of as whole-program analysis. The technique is similar to our approach, but HTML bindings are not considered.

The Gatekeeper project [16], [17] proposes a points-to analysis together with a range of queries for security and reliability as well as support for incremental code loading. Sridharan et al. [18] analyze the characteristics of common JavaScript frameworks, such as jQuery. Their technique allows them to reason precisely about properties that are copied from one object to another as is often the case in the jQuery library. These studies mostly focus on usability, maintainability and performance issues instead of reliability and inconsistency issues, which are our main focus in this study.

There are studies that have proposed dynamic analysis techniques to detect client-side errors in JavaScript web applications. Li et al. [19] analyzed user behavior to find failures in AJAX web applications. Robustness testing of web applications studied by Pattabiraman et al. [20]. There is a study about invariant-based testing of web application by Mesbah et al. [21]. Differing

from these studies our approach uses dynamic analysis to detect errors about two-way data bindings even if it is not causing a problem in the user interface.

Many researchers focused on reliability of the web applications have also used static analysis techniques. For example, Guha et al. used the static analysis to detect intrusion [22] and Zheng et al. [23] has also used static techniques to locate bugs caused by asynchronous calls in web applications. There are also tools created during researches such as Mugshot [24] to capture and replay for JavaScript web applications and WaRR [25] to reproduce client-side errors. However, these studies used static analysis unlike our work and they have not focused on errors about two-way data binding between controller and view of JavaScript MVC web applications. Static analysis lacks the detection of variable types because of the nature of JavaScript. In other words, type mismatches cannot be located statically in an interpreted programming language like JavaScript on the other hand dynamic techniques are applied after the interpretation process and for that reason our approach uses dynamic analysis.

Some researchers have studied the performance issues of JavaScript web applications in recent years. For instance, Richards et al. [26] conduct an empirical study of dynamic JavaScript behavior based on collected traces. A similar work was done by Ratanaworabhan et al. [27] with their JSMeter tool. Fortuna et al. [28] perform a limit study on the parallelism available in JavaScript code. However, none of these studies focus the reliability of web applications.

There are empirical studies focus on the security and privacy of JavaScript web applications. For instance, Yue et al. [29] characterized insecure JavaScript practices on the web analyzing the Alexa top websites and Jang et al. [30] studies privacy issues violating information flows in JavaScript web applications. These papers also differ from our study in that they do not study web applications' errors, which may or may not lead to security vulnerabilities.

Researchers have noticed that a more useful type system in JavaScript could prevent safety violations and inconsistencies. Since JavaScript does not have a rich type system to begin with, the work here is forming a correct type system for JavaScript and then building on the proposed type system. Soft typing [31] might be one of the first steps toward a type system for JavaScript. Several other projects propose type systems for JavaScript [32], [33], [34]. These projects focus on a subset of JavaScript and provide sound type systems and semantics for their restricted subsets. To the best of our knowledge, none of these approaches have been applied to large scale JavaScript code. Also there is no approach that proposes a solution to type inconsistencies in HTML bindings.

The most similar work to our study is by Mesbah et al. [35] for detecting undefined values in HTML bindings. They use static analysis differing from our study. Therefore, the approach they use lacks analysis of custom web components and type mismatches about these components. Custom web components are used in almost every JavaScript MVC application and continuing to gain more and more popularity. In addition, the type checking feature of their tool cannot handle many cases since most of the variable types are determined at runtime in JavaScript. Our focus is to offer a more compact approach covering most of the real world cases.

3.2 Code Analysis for Server Side MVC Applications

MVC based web applications has been the target of various studies, since it is comparable in popularity to JavaScript. The pattern has been applied to the server side of web applications [39], where the Model and the Controller are implemented on the server and the View is represented by the HTML output on the client. Considerable work has been done on the application of MVC on the server-side [40], [41], [42], [43], where frameworks such as Spring MVC and JSF are used. Artzi et al. [44] implemented a tool that validates the output of the PHP web applications. The validation in this approach is the confirmation to the HTML specification and it does not consider inconsistencies about data bindings.

Braband et al. [45] studied HTML documents to conform validation of the code according to the official DTD using statically analysis techniques. Another study by Jovanovic et al. [46] focused server side PHP scripts, and used static analysis to detect cross-site scripting vulnerabilities. In addition to previous work Wassermann et al. [47] used the same analysis over server-side generated HTML and JavaScript codes to detect particularly XSS attacks. Wojciechowski et al. [48] compared different MVC-based design patterns on the server-side, and analyzed the frameworks' characteristics, such as their susceptibility to file upload issues. In contrast, our work is concerned with the client-side of web applications.

CHAPTER 4

PROBLEM STATEMENT

MVC frameworks for JavaScript have certain benefits, there are also potential problems regarding the variable binding between the View and the Model components. For example, AngularJS has two-way data binding which is a favorable feature. This feature describes the condition where data is bound to an HTML element in the View and that element has the ability to both update and display those data. Model can also update the data, thus the "two-way" descriptor is used. Although the implementation of this type of data binding reduces the amount of effort to create dynamic views, it complicates debugging and has potential dangers in larger, more complex applications that are developed by multiple teams from multiple locations, since the errors in bindings do not fire any exception or are not caught by the browser.

4.1 Binding Errors

In this study, we refer the errors in the View with respect to the Model as *binding inconsistencies*. There are two types of inconsistencies: undefined variables and type mismatches.

4.1.1. Variable Definition

Variables of the model should exist in the Controller where they are defined or used in JavaScript code. These same variables are also used in the View inside the HTML code. It is straightforward to define a JavaScript variable in the Controller and use the same identifier of variable in the HTML code. However, it gets harder to ensure existence of a variable for both sides in MVC applications because of the following reasons:

- The View is written in HTML and the Controller is a JavaScript file. The same variable used in different languages makes the application susceptible to identifier inconsistencies.

- As the application grows, views and controllers are separated in separate files and usually maintained by separate programmers.

When they are merged in the render time of the DOM, inconsistencies are highly likely to occur.

- A view can be associated with multiple controllers. While this feature enables flexibility and reusability, it becomes harder to refactor a piece of code without affecting other parts of the code.

- Sometimes an undefined variable in model hides a piece of the View (`ng-show`, `ng-hide`…) unintentially. For example if a developer refactors the FlightApp in our motivating example and changes the name of the variable `availableFlights` in Figure 3 line 17 while forgetting to reflect this change for the `ng-hide` attribute in Figure 3 line 2. The View will be totally empty if there is no flight available for the selected dates instead of showing a message to the user. This is the hardest type of inconsistency to notice with manual testing. The tester should know and remember all of the fields that the View should have. The Automated End-to-End (E2E) tests usually catch these inconsistencies, however, the maintenance is difficult as it depends on the values in HTML and HTML is edited often during the application development.

4.1.2. Type Mismatch

In addition to ensuring variable definition, the developer of the MVC JavaScript application should also need to ensure that the type of the variable matches the expected type in the View. For example, in AngularJS, the `ng-include` attribute in the View must be assigned a String value, which is the path of another HTML fragment to fetch and include in the View. It should cause type mismatch error if a variable that contains a non-string value is attached to the `ng-include` attribute in the View. Locating these types of errors becomes complicated since JavaScript has a loosely typed structure.

Our approach handles this problem by simplifying expected types as;

- String
- Number
- Object
- Function
- Array
- Boolean
- Date
- File
- Undefined

This set of variable types is sufficient for almost all of the cases. There are methods to query the type of a variable, such as the "`typeof`" operation in AngularJS. We discuss such methods in Approach section.

It is even harder to track the types of variables if they are used for bindings in the View as custom HTML elements and attributes. These custom components are created using the directives for AngularJS. There are usually different developers for developing custom components and creating views in large-scale projects. It may not be possible to enforce the expected type requirement for a custom attribute by the developers who use this component in their view.

For instance, there is a custom component definition named `flight-table` in Figure 5.

```
1. myModule.directive("flightTable", function() {
2.
3.     var component = {};
4.
5.     component.restrict = "E";
6.
7.     component.scope = {
8.         flightList: "=",
9.         header: "@",
10.        flightSelected: "&"
11.    };
12.
13.    component.template =
14."<table>" +
15.  "<th><td>" +
16.    "<label ng-bind='header'></label>" +
17.  "</td></th>" +
18.  "<tr ng-repeat='flight in flightList'>" +
19.    "<td ng-click='flightSelected()'>" +
20.     "<label ng-bind='flight.desc'></label>" +
21.    "</td>" +
22.  "</tr>" +
23."</table>";
24.
25.     return component;
26.});
```

Figure 5. A custom web component definition

This definition creates the ability to use HTML element `<flight-table>` and when this element is used in the View, the browser will replace the element with the HTML content defined in its template property (Line 13 to 23 in the Figure 5). This is a very useful feature for refactoring a repeated block on HTML files. We are more interested in the scope property (Lines 7 to 11 in the Figure 5) of the custom component; since this property is the

map of the special attributes and their types that the component expects from users of the component.

There is an example usage of this custom web component in the motivating example, Figure 3, lines 5 to 9. This usage is inconsistent if there is no variable called `availableFlights` defined in the related model (Figure 3 line 17). It is also inconsistent if the type of the variable `availableFlights` is not an array. There has to be a function defined and named `chooseFlight` in the model as well (Figure 3 line 22). If these requirements are not met, the application will still be loaded but an inconsistent HTML will be rendered from the template of the custom component since non-existent variables are used in the bindings of the template.

The users of the component need to know these custom attributes of the component and their expected types exactly but that is not always an easy task. For example, it may not be possible to know the details of the custom web components in our motivating example FlightApp before looking at its actual implementation in Figure 5. Most of the web applications use many third party component libraries. They usually do not have well maintained documentation and the documentation is not used properly by developers. It is useful to automate the process of finding inconsistencies between the definition of custom components and the usage of them.

CHAPTER 5

FORMAL MODEL AND THE TOOL

We propose an approach to help developers locate errors about bindings in both built-in attributes and attributes of custom components developed within the JavaScript MVC framework. Our approach consists of four main steps. We register the state transitions of the page, then, we search for attribute changes in the DOM. When we find an attribute with a value in the view, we evaluate that value within the context of the associated model. After getting the result of that evaluation, we check for inconsistencies and specifically for errors about variable definitions and type mismatches.

5.1 Formal Model

Here we first give a formal model for JavaScript MVC applications. Then we define the binding inconsistencies on this model. Finally, we describe our approach to detect these inconsistencies based on the model.

Let M be the set of models, C be the set of controllers, V be the set of views. Let Type = {object, string, number, function, array, boolean, date, file, undefined} and AttrName be the set of special attribute names, which is explained later in this section. Also, let Value be the set of all values of each variable defined in the models in M. We define;

Val = Value $\cup$ {undefined}.

A JavaScript MVC application is a tuple <S, A> where S={<m,v,c> | m $\in$ M, v $\in$ V, c $\in$ C} is the set of states and A={<name, type>| name $\in$ AttrName, type $\in$ Type} is the set of attributes. We need to explain states and attributes briefly since our approach is based on these terms.

First of all, we need to define the "state" of an application. Almost every JavaScript MVC framework offers a grouping of Controller and View for page rendering. We use the routes in AngularJS as the states of the application (see Section 2.5). We collect state information of the application while the application loading (Figure 6 Step 1). We use this information in our analysis process when the state transition occurs.

States can be nested in a page. In this study, we call the states with no substates as non-nested states. A substate has its own view. A substate may have its own controller. If there is a controller, the substate uses the variable of the model defined in its controller in addition to the controller of its superstate. In case of name conflict, the model definition in the inner controller is used. This relationship between controllers is called prototypical inheritance in AngularJS.

In this case, it is almost impossible to find an inconsistency with a static analysis as variables in the view do not have values in the controller because controller gets these values from model which can be a remote service. Controller also can get these values from a local model which can be created in a previous state dynamically from the user inputs. So it is not possible to find out the model values which are not exist in the compilation time.

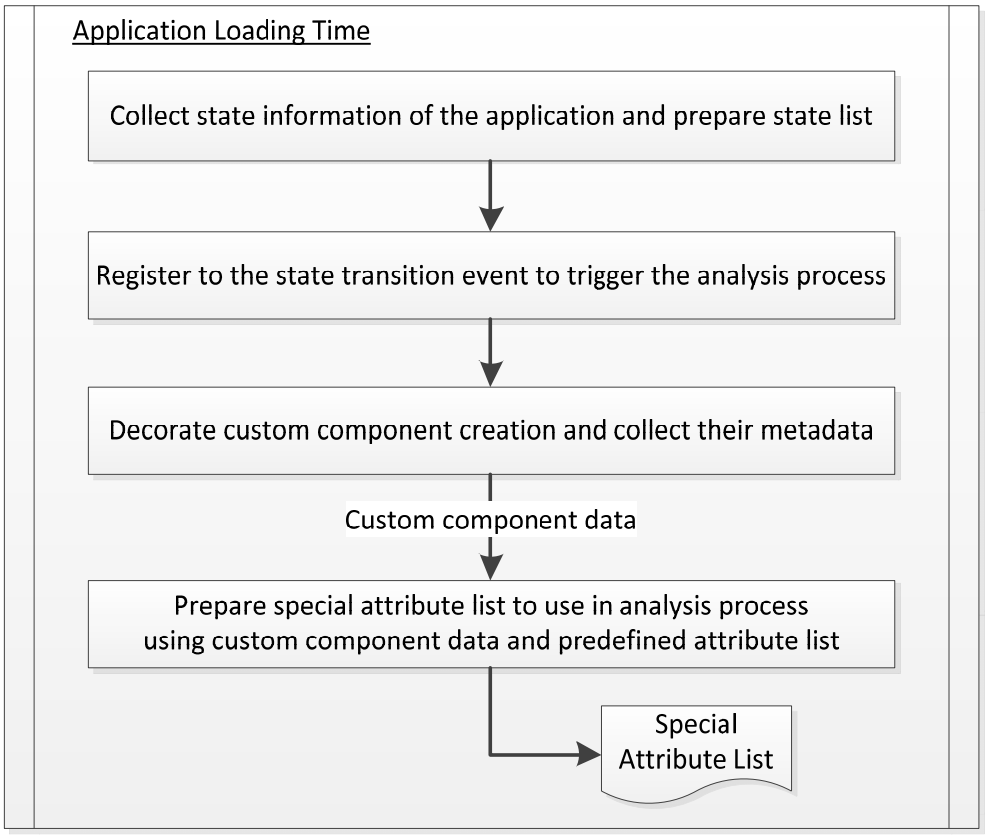

Figure 6. Executed jobs by the tool while initializing

Our approach includes finding special HTML attributes since these attributes have values. HTML elements cannot have value but they do have attributes which may have value. These values are used to bind model variables to HTML attributes. We use the word “special” because not all HTML attributes have their value as a binding to the Model. Almost every

basic HTML attributes have a static value. For example there is “type” attribute of element “button” is just a text in FlightApp (Figure 2 line 18). Another example can be “class” attribute which is applicable to almost all of the HTML elements (Figure 3 line 2).

First part of the special attributes is from framework itself. AngularJS has built-in special attributes that we need to predefine their names and their expected types (Figure 6 Step 4). So we can use them to search for their name in the view of the state. An example built-in special attribute is “`ng-click`” which we use in FlightApp (Figure 2 line 19). Simply it expects a function and invokes that function when the associated element is clicked by the user.

Second and more important part of the special attributes are defined in the application itself. As developers define custom web components, they usually define expected attribute names and their types by these custom web components. But the developers cannot ensure the right usage of the attributes in the view as these custom web components are independent, reusable templates. In addition, these components are not rendered by browser before they are encountered in the loading phase of the DOM.

For instance, in our FlightApp example, there are 3 built-in attributes and 3 attributes from custom web component. The set of special attributes;

A = {

<"ng-model", OBJECT>, <"ng-click", FUNCTION>,

<"ng-hide", BOOLEAN>, <"header", STRING>,

<"flight-list", ARRAY>, <"flight-selected", FUNCTION>

}

Here we define three functions;

Function 1 (Eval Function): Given $m \in M$, $v \in V$, and $a \in A$, the function *eval*(m,v,a) returns the value of the variable of the model which is defined in the controller c and used as attribute a in the view v.

Function 2 (TypeOf Function): Given $val \in Val$, the function *typeOf*(val) returns the type of the value val which is an element of the set Type.

Function 3 (Flatten Function): Let $v_1, v_{11}, v_{12} \in V$, $m_1, m_{11}, m_{12} \in M$, and $c_1, c_{11}, c_{12} \in C$. Given a state $s_1 : < m_1, v_1, c_1>$ in S with $s_{11} : <m_{11}, v_{11}, c_{11}>$ and $s_{12} : < m_{12}, v_{12}, c_{12}>$ as substates, *flatten*(s_1) = {s'_{11}, s'_{12}} where $s'_{11} : < m_1 \cup m_{11}, v_1 \cup v_{11}, c_1 ₱ c_{11}>$ and $s'_{12} : <m_1 \cup m_{12}, v_1 \cup v_{12}, c_1 ₱ c_{12}>$ and ₱ is the prototypical inheritance operation in JavaScript.

Now we define the binding consistency rules;

Rule 1 (Definition Rule): Given an attribute a ∈ A and a non-nested state s=<m,v,c> where c ∈ C, v ∈ V, and m ∈ M which is defined in c, then *eval*(m, v, a) ≠ undefined.

Rule 2 (Type Matching Rule): Given an attribute a ∈ A and a non-nested state s=<m,v,c> where c ∈ C, v ∈ V, and m ∈ M which is defined in c, then *typeof*(*eval*(m, v, a)) = a.type

Definition: A JavaScript MVC web application <S, A> is consistent if and only if for every state s ∈ S, the above two rules are met for every attribute a ∈ A in every elements of *flatten*(s).

5.2 Approach

We present a runtime verification approach to detect the sources of inconsistencies in data bindings. We propose to employ runtime checks by leveraging the underlying execution environment whenever a state transition occurs. AngularJS fires an event for every transition as most of other frameworks do. We can register to that event or decorate the function which initializes the transition. We choose the first option as it helps to isolate the analysis logic (see the Step 2 in Figure 6).

Employing runtime checks at state transitions enables us to report errors in a state immediately when the state is loaded. Furthermore since the part of the HTML page where the transition will occur is predefined in the application, it is possible to execute analysis process only on the changing part of the HTML page preventing whole page to be analyzed again and again and gaining performance (Figure 7 Step 1). Additionally, this feature is very useful especially in the development environment of a big scale application where developers need to focus business domain logic more than finding inconsistencies in the binding between view and controller.

After registering state transitions we need to search the view for the special attributes, collect their values in the view and evaluate these values in the context of the particular model (Figure 7 Step 2). Searching these special attributes is done by using routing system to detect the changing area and then using DOM traversal utilities of AngularJS which use jQuery internally on this area. These evaluated values can refer to a Model variable directly or can be a statement which has model variables in it. In this case evaluation becomes harder. Fortunately, there is parsing mechanism both in native JavaScript and in the MVC framework. We have used parsing service of AngularJS framework since it is already loaded and ready to use while runtime verification occurs. We evaluate the model variables in the statement expressions after parsing them and locating the variable itself.

Choosing context to run the parsing service on is important but it is solved easily by running it in the context of the model itself and then on the prototype of the model. It means that parsing service takes prototypically inherited variables into consideration as well.

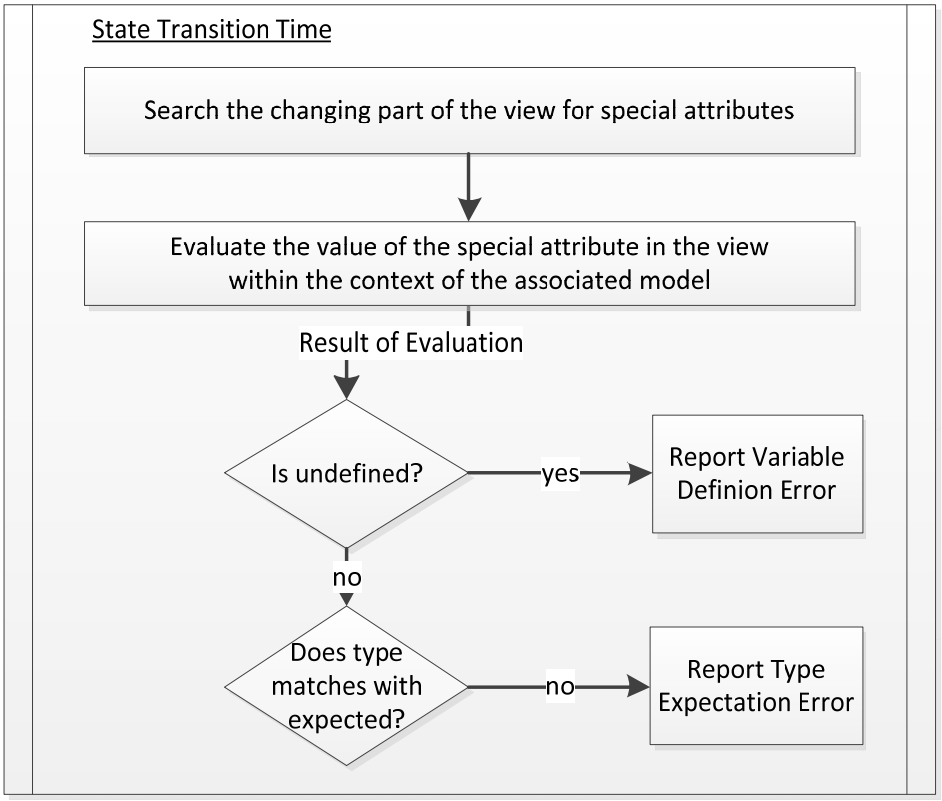

Figure 7. Executed jobs by the tool for every state transition

Checking the rules of the formal model is the next step of the analysis process. We can move to the checking phase since we have the special attribute with name and expected type, value of the attribute in the view and evaluated value of the attribute in the model. We defined two rules to use for checking purposes. First rule executes the variable definition checking process (Rule 1). This process looks for the evaluated value of the attribute and returns false if evaluated value is undefined. If result is false then the first rule fails and logs the error (Figure 7 Step 3). If result is true then Rule 2, the type checking rule, starts execution and initializes the type expectation checking process. This process looks for the type of the evaluated value and tries to match it with the expected type of the special attribute. If this matching fails then the second rule logs the error about type expectation (Figure 7 Step 4).

Our approach uses runtime verification methodology instead of static analysis since this method enables us to handle nested states easily. In addition, we can make use of the parsing service to evaluate variables in sentences given in HTML. For example it is possible to evaluate the variable `flightList` in Figure 5 line 18 using parsing service. Inferring the value of the variables is also not a problem in dynamic verification because we use the runtime environment to determine these values.

5.3 Implementation

We implemented our approach as a tool namely HAKI that helps developers to find about sources of inconsistencies in their code by locating errors using runtime verification. Our tool can report the errors and warnings about the page that it is working on. The tool is currently working dynamic mode only which means that the tool runs background with the application while application runs in the browser. Our tool should be loaded to the application after AngularJS framework's source file and before the source code of the application itself. This order enables us to decorate some features of MVC structure of the AngularJS and custom web component creation process (Figure 6 Step 3). Particularly there is no way to be notified about custom web component creation in the application without decorating some functions of AngularJS. So we used JavaScript as well to implement HAKI and currently it is depended on AngularJS because of the decoration process.

```
1.  /* The tool should be run manually */
2.  MANUAL: "MANUAL",
3.
4.  /* Default, The tool runs automatically,
5.  * when application transit to
6.  * that state for the first time */
7.  AUTO_ONCE: "AUTO_ONCE",
8.
9.  /* The tool runs automatically,
10. * when application transit to
11. * that state for every time */
12. AUTO_ALWAYS: "AUTO_ALWAYS",
13.
14. /* The tool runs automatically,
15. * when the application loaded */
16. ALL: "ALL"
```

Figure 8. Available modes of the tool HAKI

HAKI can be used by loading it to the application just after AngularJS. It will run automatically for every state transition by default. There are other execution strategies available for the tool. These modes and their explanation are shown in Figure 8. In addition, HAKI provides a global API

which developers can use to configure the tool. The global API provides the methods shown in Figure 9. The methods are self-explanatory.

```
/**
 * Sets the execution strategy
 * @param {string} hakiMode
 * @see haki.HAKI_MODES
 */
hakiAPI.mode = function (hakiMode) {
    haki.mode = hakiMode;
};

/**
 * Disables the analysis for given type
 * @param {string} type
 */
hakiAPI.ignoreType = function (type) {
    haki.ignoreList.add({
        type: "type",
        value: type
    });
};

/**
 * Disables the analysis for given attribute
 * @param {string} attributeName
 */
hakiAPI.ignoreAttribute = function (atrName) {
    haki.ignoreList.add({
        type: "attribute",
        value: attributeName
    });
};

/**
 * Executes the analysis for given state
 * It will run for current state,
 * If stateName is not defined
 * @param {string} stateName
 */
hakiAPI.run = function (stateName) {
    haki.analyze(stateName);
};
```

Figure 9. The global API of the tool HAKI

The tool executes the steps shown in Figure 7 at state transition time for the transitioned state. When all steps are completed, the tool marks the state as analyzed and does not try to analyze it again before reloading the application. This means a state is not analyzed if the application transits to that state the second time. It is a parametrical feature that can be turned off using the global API, but since our tool designed to be used mostly in

development or testing phase, it is not necessary to analyze a page twice. Developers usually develop pages by the following, possibly repetitive, sequence of tasks: 1) code the web application, 2) run the application, 3) check for errors, 4) if there is any error, close the application, 5) fix the error, and 6) rerun the application to see if it is error free. In addition, it is a performance enhancement to disable analyzing a page more than once.

CHAPTER 6

EXPERIMENTS AND RESULTS

We integrated our tool to the development phase of a real big scale JavaScript application. In our first experiment, we use this real application to evaluate the efficiency, reliability, and usability of the tool. In our second experiment, we used HAKI in an application, where we created and injected erroneous states, to measure the accuracy of the tool. In both of the experiments we examined whether there are any performance issues.

6.1 Real Application Tests

The application we use to evaluate our tool is a big scale web application. It is used in Turkey, mainly by manufacturers and distributors of medical devices and cosmetic products. The aim of the application is to register and track all unique medical devices and cosmetic products in the country starting with their production or import to the country and ending with their consumption or use by clients. Therefore, it is a data-heavy application and the pages that show the data should be reliable and error free.

The application uses the Java Spring Framework at server side and the JavaScript AngularJS framework at client side. There are four development teams with 28 developers and one test team with four testers working on that project. There are seven main modules and three domain independent supportive modules. Table 1 gives more detail about the size of the application.

Table 1. Metrics that shows the size of the real application

Metric	Value
Number of States	526
Number of Controllers	512
Number of Views	526

Number of JavaScript Files	1459
Number of HTML Files	642
Number of Custom Elements	59
Number of Lines of JavaScript Code	49536

6.1.1. Experiment

During the first run of the experiments, we encountered an issue about false positives. The developers of this application have a tendency to assign string or object variables directly to the Boolean attributes. Since any value other than null or zero is interpreted as true, this behavior does not result in any error during page rendering. Below is an example of this usage in our FlightApp Figure 3 line 2. It should be:

```
1. ng-hide="availableFlightList.length > 0"
```

instead of;

```
1. ng-hide="availableFlightList.length"
```

To avoid reporting false positives in these situations, we decided to add a function to the global API to disable the type checking of Boolean attributes. This setting is the default behavior of HAKI since such misuse is common among web developers. We were able to reduce the false positives with the help of this improvement. Finally, we reran the experiment and collected the results that are discussed in the following section.

6.1.2. Results

The logs and messages created by HAKI were collected and evaluated with the developers and testers of the subject application. There were a total of 55 errors that our tool located and all of them were acknowledged as real issues that need to be fixed. Additionally, eight of the errors were marked as major errors with high priority.

Major errors include wrong variable names used in View attributes for data bindings, which result in undesirable situations. In two cases, the data entered by the user is bound to the wrong variable and the actual variable is sent to the server side with an undefined value. This binding error results in creation of inconsistent data. There are two cases with the same effect; but these are caused by using wrong types in bindings. In these cases, variables with the type of *function* are used for binding, whereas the expected type is

string. There is also a major error about type mismatch where the expected type is *date* but given type is *function*. This is a potential inconsistency within the data. The other three errors are about the pages shown to the user. The bindings expect string variables but objects are used. This situation caused the browser to attempt to convert these objects into string and display *"[object Object]"* on the page.

There were also 35 warning messages reported that show possibility of errors. Most of these warning messages are about undefined variables used for the ng-model attribute. We report this situation as warning since `ng-model` attribute can create the variable it is bound if the variable is not defined. In this case, the usages can be intentional therefore we just report a warning about the situation. Actions are taken for all of those warning messages and specific improvements are applied since they can cause errors easily if they are used as sub-states in the future. To conclude, HAKI detected a considerable amount of fault where all results are evaluated as true positives.

6.1.3. Overhead

We evaluated HAKI in a big scale project which has a big performance concern. The overhead that our tool brings to the application is very important, because the tool always runs automatically in the background as the application runs. The application is loaded with all of its components shown in Table 1 during the experiment. The average time of HAKI to analyze a state is 185 milliseconds. We measure its performance for two chosen states. The first one is chosen as the big state with at least 50 HTML elements used at depth 6 of the DOM tree. The second one, called small state, has just seven elements and one of them is a custom web component with a flat DOM tree. In addition, we measure the performance of HAKI for the most visited states which are dashboard pages (users are redirected to a dashboard page according to their role after login). The average time of HAKI to analyze these most visited states is 218 milliseconds which is satisfying for this particular case. Although these values extremely depend on the power of client's computer, the tool verifies only the changing portion of the view, not the whole DOM and it uses simple JavaScript loops for this verification, which are simple and efficient to execute. This indicates that there is a negligible overhead our tool brings to applications.

6.2 Fault Injection Tests

The application we use to inject faults is a middle size real world JavaScript application that uses AngularJS as MVC framework on the client side. It is used to help development teams to share information by creating wikis for their services and components. They use the application every time they

create a component or a public service to inform others about that new feature. Developers are responsible to maintain this information, so they also use the application for updates.

Table 2 gives more information about the size of the application:

Table 2. Metrics that shows the size of the fault injected application

Metric	Value
Number of States	42
Number of Controllers	42
Number of Views	42
Number of JavaScript Files	44
Number of HTML Files	42
Number of Custom Elements	2
Number of Lines of JavaScript Code	2911

6.2.1 Experiment

We first ran HAKI before fault injection to identify background errors. After identification and fixing of these errors, we injected 18 errors into the application as given in Table 3:

Table 3. Type of errors used for the fault injection

Error Introducing Method	Violating Rule	Number of Errors
Used undefined variable in view	Rule 1	2
Used variable from unrelated model in view	Rule 1	1
Changed the name of variable in model	Rule 1	3
Used a mistyped variable in view	Rule 2	2
Changed the type of variable in model	Rule 2	5

Deleted the definition of variable from model	Rule 1	1
Used an undefined variable in view for custom component	Rule 1	2
Used a mistyped variable in view for custom component	Rule 2	2

We tried to use as many different attributes as to make the experiment more realistic. There are also injections about type mismatch violating Rule 2. After all the errors were injected, we ran HAKI for runtime verification.

6.2.2 Results

The error messages that HAKI created were collected and analyzed. There were a total of 18 error messages which show that our tool successfully located all of the injected faults. These faults are introduced to the subject system using the methods in Table 3:

Table 3. Any other scenario that causes an inconsistency about data bindings has to be violating Rule 1 (variable definition rule) or Rule 2 (type mismatch rule) and nothing else. Since our approach covers these rules, HAKI is complete.

6.2.3 Overhead

We measured the running time performance of HAKI for all of the states in the application. The longest time it takes HAKI to analyze a state is 361 milliseconds and shortest time is 106 milliseconds. Although HAKI should mostly be used in development environments where performance is not a big concern, it can also be used in production since it carries negligible overhead.

6.3 Experiences in a Development Environment

Our tool HAKI has been used in a large-scale real project for approximately four months. We integrated the tool to the development environment of this project and developers started to use the output of the tool. This process has also provided us to enhance HAKI to support the needs of a real world software project. There are development teams and a test team in the project. HAKI provides a significant decrease the number of the failures that comes back from test team to the development teams since it enables the developers to detect more errors during the development process. It is a known fact that the earlier an error found the lower it costs to fix it.

HAKI analysis states of the application and outputs the messages about the binding errors it finds. Figure 10 shows an example output of the tool. In

this Figure it is shown that HAKI analyzed the states of the application and if it detects an inconsistency in the state, the error message about that inconsistency is printed. This error messages includes the special attribute name, value and inconsistency type. HAKI outputs only the state names that have at least one error, if silent mode of the tool is activated via global API differing from the figure below.

```
"Haki analyzing state: GIRIS_SAYFASI...",
"Haki analyzing state: ANASAYFA...",
"Haki analyzing state: BUTUN_TIBBI_CIHAZ_OZETLERI_LISTELE...",
"Haki analyzing state: TIBBI_CIHAZ_OZETI_GORUNTULE...",
"Haki analyzing state: CIHAZ_DURUM_SORGULA...",
"Haki analyzing state: BAYILIK_OZETI_LISTELE...",
"Haki analyzing state: BAYILIK_OZETI_GORUNTULE...",
"Haki analyzing state: KONTROLE_GONDERILEN_BELGE_BASVURULARINI_LISTELE...",
"Haki analyzing state: BASVURU_LISTELE_INCELEYEN_ATA...",
"Haki analyzing state: CIHAZ_BASVURU_INCELEYEN_KULLANICI_ATA...",
"--- Found: 'kullaniciHesabi', for attribute: 'ng-model', is NOT defined.",
"Haki analyzing state: INCELEDIGIM_CIHAZLARI_LISTELE...",
"Haki analyzing state: INCELEDIGIM_BELGELERI_LISTELE...",
"Haki analyzing state: UST_MAKAM_INCELEMESI_GEREKEN_KAYITLARI_LISTELE...",
"Haki analyzing state: KOZMETIK_URUN_GECMISI_DETAY_GORUNTULE...",
"Haki analyzing state: KOZMETIK_FORMULASYON_DETAYI_GORUNTULE...",
"--- Found: 'kayit.zehirMerkezineEkBilgi', for attribute: 'ng-bind', is NOT defined.",
"Haki analyzing state: KOZMETIK_BILESEN_DETAYI_GORUNTULE...",
"Haki analyzing state: BUTUN_NANOMATERYALLERI_LISTELE...",
"Haki analyzing state: KONTROLE_GONDERILEN_KOZMETIK_BASVURULARINI_LISTELE...",
"Haki analyzing state: KOZMETIK_BASVURU_LISTELE_INCELEYEN_ATA...",
"Haki analyzing state: KOZMETIK_UZERIMDEKI_BASVURULARI_LISTELE...",
"--- Found: 'startChanged()', for attribute: 'ng-change', is NOT defined.",
"Haki analyzing state: KONTROLE_GONDERILEN_IHRACAT_SERTIFIKA_BASVURUSU_LISTELE...",
"Haki analyzing state: IHRACAT_SERTIFIKA_UZERIMDEKI_BASVURULARI_LISTELE...",
"Haki analyzing state: KULLANICI_GRUBU_GUNCELLE...",
"Haki analyzing state: KULLANICI_GRUBU_KULLANICILARI_YONET...",
"--- Found: 'glyphicon-arrow-right', for attribute: 'icon', has WRONG type (number).",
"--- Found: 'glyphicon-arrow-left', for attribute: 'icon', has WRONG type (number).",
"--- Found: '10', for attribute: 'size', has WRONG type (number).",
"--- Found: '10', for attribute: 'size', has WRONG type (number).",
"Haki analyzing state: KURUM_LISTELE...",
"Haki analyzing state: KURUM_DETAYI_WIZARD...",
"Haki analyzing state: KURUM_GECMIS_DETAYI_GORUNTULE...",
"Haki analyzing state: KURUM_YETKILERINI_YONET...",
"Haki analyzing state: KURUM_YETKILI_KULLANICILARINI_YONET...",
"Haki analyzing state: YETKILI_KULLANICI_EKLE...",
"Haki analyzing state: KOZMETIK_FIRMA_BASVURUSU_LISTELE...",
"Haki analyzing state: TIBBI_CIHAZ_FIRMA_BASVURUSU_LISTELE...",
"Haki analyzing state: KULLANICI_GRUBU_SABLONU_LISTELE...",
"Haki analyzing state: KULLANICI_GRUBU_SABLONU_DETAYI_GORUNTULE...",
"Haki analyzing state: KULLANICI_GRUBU_SABLONU_GUNCELLE...",
```

Figure 10. An example output of the HAKI

HAKI has a silent mode in which it only sends a message about the states that have an error. In the development environment we enabled HAKI to send its output directly to the user interface as well as the application console since the developers activates silent mode it is usable to see the

errors directly in the upper right corner of the web page itself (Figure 11). Some of the errors that HAKI reported may not affect the functionality of the page but just the appearance. Even so it is not a desirable situation that the end users of the project see these error messages therefore developers are enforced to fix the errors that HAKI reported. In the end HAKI provides the consistency for the project.

In Figure 12 there is an error in the page about the "GMDN" value. We have highlighted the particular label; it is too hard to detect manually otherwise.

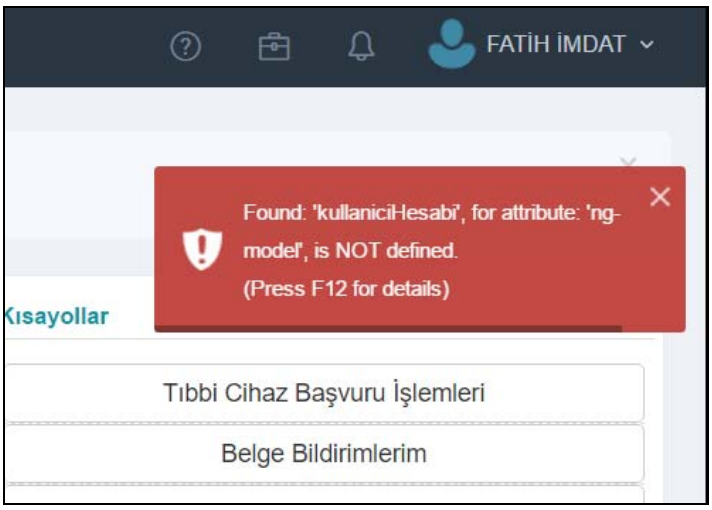

Figure 11. The output of the HAKI in the web page

Testers may think that this particular record does not have any GMDN value so that it is empty. They should access the database and check the values to match with these page. It requires a lot of effort to detect the error. Although there is not a functional error, there is deficiency of the customer requirements. In addition, it can easily cause loss of system functionality if the same error occurs in a binding in any of the forms where user inputs data to the system.

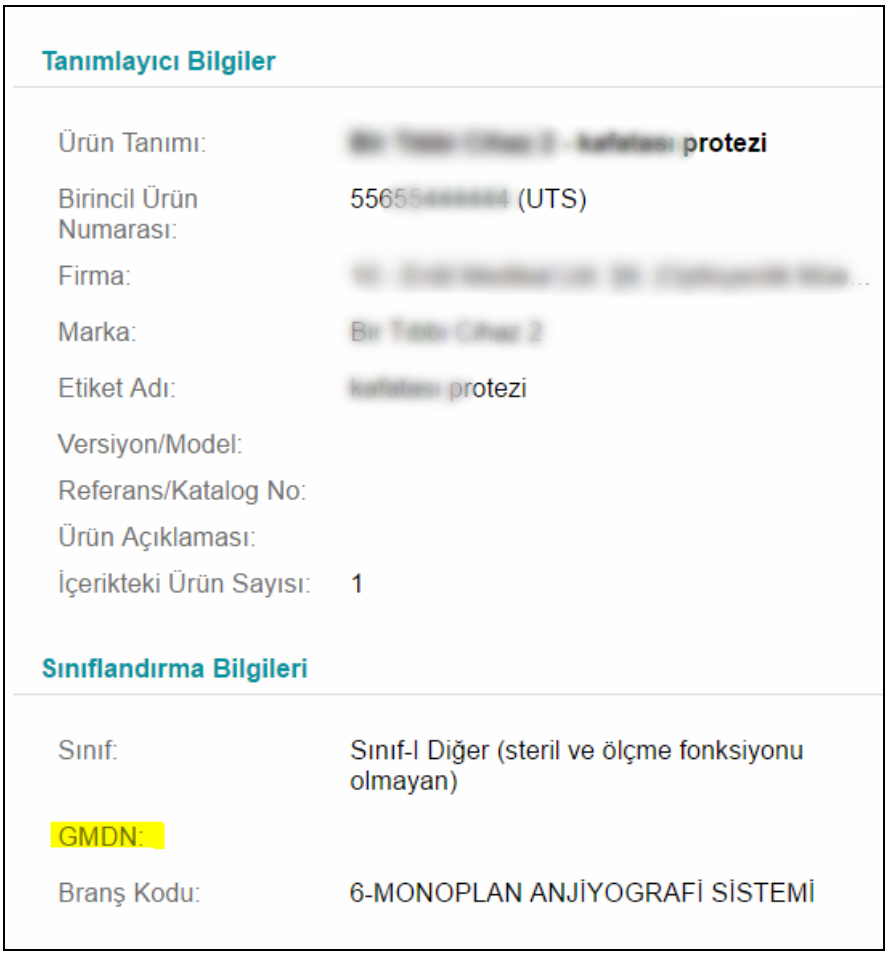

Figure 12. An example page with binding error

The root cause of this error is as follows. For this particular instance the developer of the page used four partial HTML for every header and one Controller for all of them. One of the partial HTMLs is shown in Figure 13 lines 1 to 12 and the Controller is shown in Figure 13 lines 15 to 28 (summarized). Please take into consideration that the Controller and the View are in the separate physical files. The view fragment in Figure 13 line 7 binds to `kayit.gmdnJenerik.code`. In the specific Controller in Figure 13 line 23; the `gmdnJenerik` field of the variable `kayit` is populated via a remote request. It is assumed that the `gmdnJenerik` object has a field namely `code` but the name of this field is changed by the developer who develops the remote resource to `kod` while it remains unchanged in the specific partial HTML; because of that, the page does not

render the GMDN value of the record and the users will never see that value even it is recorded in the application.

```
1. //* Partial HTML included by the view of the page -->
2. <form-block header="Sınıflandırma Bilgileri">
3.     <form-element element-name="Sınıf:">
4.        <label ng-bind="kayit.sinif"/>
5.     </form-element>
6.     <form-element element-name="GMDN:">
7.        <label ng-bind="kayit.gmdnJenerik.code"/>
8.     </form-element>
9.     <form-element element-name="Branş Kodu:">
10.       <label ng-bind="kayit.bransKodu.aciklama"/>
11.    </form-element>
12.</form-block>
13.
14.//* The controller of the page -->
15.utsApp.controller("tibbiCihazDetayController",
16.    function ($scope, URLParameters, RemoteService) {
17.
18.        var cihazId = URLParameters.get("id");
19.
20.        //* The Model of the page -->
21.        $scope.kayit = {
22.            ...
23.            gmdnJenerik: RemoteService.getGMDN(cihazId);
24.            ...
25.        };
26.
27.    }
28.);
```

Figure 13. The View and Controller of the Page in Figure 12

HAKI detected the error in this page when it analyzed the page. All four partial HTML were merged as a View and the Controller loads the Model values from remote resources. HAKI access the Model and matches with the View then parses the View and evaluates the variables respectively.

To conclude HAKI reduced the number of the errors caused by the data bindings in the testing phase since it enables developers to detect and resolve every inconsistency in the development phase. In addition, it removes these kinds of errors from the production environment as well; considering some of the errors had not been detected in the testing phase before our tool.

CHAPTER 7

CONCLUSION

We have presented a runtime verification approach for JavaScript MVC web applications. We implemented the proposed approach as a tool named HAKI. The key insight of our work is that despite the challenging dynamic features of the JavaScript MVC frameworks, it is possible to locate inconsistencies between JavaScript code and HTML code with a predefined rule set using runtime verification. We showed that our tool is capable of detecting real world bugs in very large scale JavaScript MVC applications with a minor performance overhead.

7.1 Limitations

We have a few limitations in our approach as explained in Section 5.2. The implementation of our approach is for AngularJS and the current version is limited to AngularJS (Section 5.3). But it can be applied to another JavaScript MVC framework with little modification, since most of the implementation is independent from AngularJS. The core analysis process is written in pure JavaScript. Core analysis is triggered when an AngularJS dependency is detected. Different frameworks manage page transitions differently. However, there are mechanisms to catch those transitions in almost all of the frameworks. Therefore, little effort is required for applying HAKI to another JavaScript MVC framework.

Since we employ runtime analysis techniques, our approach lacks the analysis of the states which load partial HTML contents asynchronously. We execute the runtime verification process after the state is loaded, so if there is asynchronous loading in a state, it cannot be analyzed by HAKI. In a similar way, any HTML content rendered and added to the DOM after the state transition occurs should be missed too since there are a few special attributes which behaves that way like `ng-switch`. To conclude HAKI has just a few limitations since it is designed to handle many cases like plurization, interpolation, inheritance, filters, and custom components, which are commonly used features of web application frameworks.

7.2 Future Work

In future work, we will focus on improving the tool to make it more generic and independent from the MVC framework itself. It may be possible by adding a specific analysis logic for every framework. Another direction of future work is to implement runtime verification process of generic custom web components, which are not dependent on framework implementation and can work even with native JavaScript. Since custom component JavaScript applications are considered as the future of the web development (Section 2.4), working on further improvements in this area will have significant benefits for the developer community.

REFERENCES

[1] S. O'Grady, "The RedMonk Programming Language Rankings", June 2015, http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15

[2] D. Rowinski, "JavaScript is eating the world", https://arc.applause.com/2015/11/06/javascript-is-eating-the-world, 2015

[3] M. Cantelon, T.J. Holowaychuk and M. Harter, the book: "NodeJS in Action", August 2013

[4] EngineIO, https://github.com/socketio/engine.io, last visited November 2016

[5] MochaJS, http://www.mochajs.org, last visited October 2016

[6] ShouldJS, http://shouldjs.github.io, last visited October 2016

[7] EmberJS, http://www.emberjs.com, last visited November 2016

[8] BackboneJS, http://www.backbonejs.org, last visited November 2016

[9] KnockoutJS, http://knockoutjs.com/documentation/introduction.html, last visited November 2016

[10] AngularJS, http://www.angularjs.org, last visited October 2016

[11] C. Wilson, "Componentizing Web Applications", https://www.w3.org/TR/NOTE-HTMLComponents

[12] A. Lerner, the book: "ng-book The Complete Book on AngularJS" https://www.ng-book.com, last visited September 2016

[13] B. Bibeault, The book: "JQuery in Action", 2008

[14] D. Crockford, the book: "JavaScript: The Good Parts", "Unearthing the Excellence in JavaScript", pages 110-132, May 2008

[15] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In PLDI, June 2009

[16] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In Proceedings of the Usenix Security Symposium, Aug. 2009

[17] S. Guarnieri and B. Livshits. Gulfstream: Incremental static analysis for streaming JavaScript applications. In Proceedings of the USENIX Conference on Web Application Development, June 2010

[18] M. Sridharan, J. Dolby, S. Chandra, M. Schaefer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In ECOOP, 2012

[19] W. Li, M. Harrold, and C. Gorg, "Detecting user-visible failures in AJAX web applications by analyzing users' interaction behaviors," in IEEE/ACM Conference on Automated Software Engineering, 2010, pp. 155–158.

[20] K. Pattabiraman and B. Zorn, "DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing," in IEEE Intl. Symposium on Software Reliability Engineering (ISSRE), 2010, pp. 191–200.

[21] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of AJAX user interfaces," in Intl. Conference on Software Engineering, 2009, pp. 210–220.

[22] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for AJAX intrusion detection," in Intl. Conference on World Wide Web, 2009, pp. 561–570.

[23] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in Intl. Conference on the World-Wide Web (WWW), 2011, pp. 805–814.

[24] J. Mickens, J. Elson, and J. Howell, "Mugshot: deterministic capture and replay for JavaScript applications," in 7th USENIX Conference on Networked Systems Design and Implementation, 2010, pp. 11–11.

[25] S. Andrica and G. Candea, "WaRR: High Fidelity Web Application Recording and Replaying," in IEEE Intl. Conference on Dependable Systems and Networks, 2011.

[26] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in ACM Conference on Programming Language Design and Implementation, ser. PLDI '10, 2010, pp. 1–12.

[27] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "JSMeter: Measuring JavaScript behavior in the wild," Usenix Conference on Web Application Development (WebApps), 2010.

[28] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers, "A limit study of JavaScript parallelism," in IEEE Intl. Symposium on Workload Characterization (IISWC), 2010, pp. 1–10.

[29] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the web," in Intl. Conference on World Wide Web (WWW), 2009, pp. 961–970.

[30] D. Jang, R. Jhala, S. Lerner, and H. Shacham, “An empirical study of privacyviolating information flows in JavaScript web applications,” in ACM Conference on Computer and Communications Security, 2010, pp. 270–283.

[31] R. Cartwright and M. Fagan. Soft typing. SIGPLAN Notices, 39(4):412–428, 2004

[32] C. Anderson and P. Giannini. Type checking for JavaScript. In WOOD S04, volume WOOD of ˇ ENTCS. Elsevier, 2004. http://www.binarylord.com/ work/js0wood.pdf, 2004

[33] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In Proceedings of the European Conference on Object-Oriented Programming, pages 429–452, July 2005

[34] R. Thiemann. Towards a type system for analyzing JavaScript programs. European Symposium On Programming, 2005.

[35] A. Mesbah, F. Ocariza, and K. Pattabiraman. “Detecting inconsistencies in javascript mvc applications”. In Proceedings of the 37th International Conference on Software Engineering-Volume 1, pages 325--335. IEEE Press, 2015

[36] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, “An empirical study of client-side JavaScript bugs,” in Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE Computer Society, 2013, pp. 55–64

[37] F. Ocariza, K. Pattabiraman, and A. Mesbah, “AutoFLox: an automatic fault localizer for client-side JavaScript,” in Proceedings of the International Conference on Software Testing, Verification and Validation (ICST). IEEE Computer Society, 2012, pp. 31–40

[38] F. Ocariza, K. Pattabiraman, and A. Mesbah, “Vejovis: suggesting fixes for JavaScript faults,” in Proceedings of the International Conference on Software Engineering (ICSE). ACM, 2014, pp. 837–847

[39] A. Leff and J. T. Rayfield, “Web-application development using the model/view/controller design pattern,” in Proceedings of the International Enterprise Distributed Object Computing Conference (EDOC). IEEE Computer Society, 2001, pp. 118–127

[40] J. L. Singleton and G. T. Leavens, “Verily: a web framework for creating more reasonable web applications,” in Companion Proceedings of the International Conference on Software Engineering (ICSE). ACM, 2014, pp. 560–563.

[41] S. Halle, T. Ettema, C. Bunch, and T. Bultan, “Eliminating navigation ´ errors in web applications via model checking and runtime enforcement of navigation state machines,” in Proceedings of the International

Conference on Automated Software Engineering (ASE). ACM, 2010, pp. 235–244.

[42] J. Nijjar and T. Bultan, “Bounded verification of Ruby on Rails data models,” in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). ACM, 2011, pp. 67–77.

[43] R. Morales-Chaparro, M. Linaje, J. Preciado, and F. Sanchez-Figueroa, ´ “MVC web design patterns and rich internet applications,” Proceedings of the Jornadas de Ingenierıa del Software y Bases de Datos, pp. 39–46, 2007.

[44] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding Bugs in Dynamic Web Applications. In Proceedings of ISSTA 2008, pages 261–272. ACM, 2008

[45] C. Braband, A.Moller, and M. Schwartzbach. Static validation of dynamically generated HTML. In PASTE 2001.

[46] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static analysis tool for detecting Web application vulnerabilities (short paper). In Security and Privacy, 2006.

[47] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In ICSE, 2008.

[48] J. Wojciechowski, B. Sakowicz, K. Dura, and A. Napieralski, “MVC model, struts framework and file upload issues in web applications based on J2EE platform,” in Proceedings of the International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET). IEEE Computer Society, 2004, pp. 342– 34

TEZ FOTOKOPİSİ İZİN FORMU

ENSTİTÜ

Fen Bilimleri Enstitüsü	☐
Sosyal Bilimler Enstitüsü	☐
Uygulamalı Matematik Enstitüsü	☐
Enformatik Enstitüsü	X
Deniz Bilimleri Enstitüsü	☐

YAZARIN

Soyadı : Bilge
Adı : İbrahim
Bölümü : BİLİŞİM SİSTEMLERİ

TEZİN ADI (İngilizce) : HAKI: A RUNTIME VERIFICATION TOOL FOR JAVASCRIPT MVC WEB APPLICATIONS

TEZİN TÜRÜ: Yüksek Lisans X Doktora ☐

1. Tezimin tamamı dünya çapında erişime açılsın ve kaynak gösterilmek şartıyla tezimin bir kısmı veya tamamının fotokopisi alınsın. X
2. Tezimin tamamı yalnızca Orta Doğu Teknik Üniversitesi kullanıcılarının erişimine açılsın. (Bu seçenekle tezinizin fotokopisi ya da elektronik kopyası Kütüphane aracılığı ile ODTÜ dışına dağıtılmayacaktır.) ☐
3. Tezim bir (1) yıl süreyle erişime kapalı olsun. (Bu seçenekle tezinizin fotokopisi ya da elektronik kopyası Kütüphane aracılığı ile ODTÜ dışına dağıtılmayacaktır.) ☐

Yazarın imzası ..... Tarih .....