Model Checking Method for SPA Page Transition Based on Component-based Framework

Naito Oshima
Graduate School of Creative Science and Engineering
Waseda University
Tokyo, Japan
always-4869@akane.waseda.jp

Tomoji Kishi
School of Creative Science and Engineering
Waseda University
Tokyo, Japan
kishi@waseda.jp

Abstract—**In recent years, because web applications have been handling increasingly important processing tasks, it is ever more important to avoid errors. Model checking is one verification method for detecting errors, whereby it is necessary to model the web application in order to verify it. However, typical web application developers may lack knowledge on creating the verification model. Furthermore, web applications have become increasingly diversified owing to web-browsing technological advancements and other factors. Among these, the single-page application (SPA) using a component-based web application framework, such as Angular, has attracted attention because of its excellent user experience. However, it makes modeling more difficult since the page structure is complicated by the intricate combination of components. In this paper, we therefore present a method to automatically generate verification models from source code and perform model checking. The method enables verification of SPA page transitions using the component-based framework. We apply our implemented automated tool to several applications. First, we experiment using sample applications that do not inject bugs and others that intentionally inject bugs. Moreover, we apply the method to real applications published on the Internet. The desired results are obtained, thereby confirming that the proposed method is effective.**

Keywords—Model Checking; Single-Page Application (SPA); Web Application Framework; Component-based; Angular.

I. Introduction

In recent years, the number of web applications handling important processing endeavors, such as online shopping, has markedly increased, thus magnifying the importance of avoiding errors. A comprehensive verification method for detecting errors is model checking [1], whereby it is necessary to model the target web application in order to verify it. However, the typical web application developer may have minimal knowledge of model checking.

Meanwhile, web applications have continued to diversify on account of the advancement of highly functional web browsers and web application development technology. Among them, a new type of single-page application (SPA) has been developed. By performing front-end processing, such as control of page transitions, which was traditionally performed on the back-end, it is possible to provide superior user experience (UX) with a fast response. Moreover, front-end frameworks—which are different from back-end frameworks, such as the conventional Ruby on Rails—are emerging to foster front-end development. Front-end frameworks include Backbone.js, Vue.js, AngularJS, and others.

The recently released Angular framework has a novel component-based architecture that is different from the conventional one. However, in SPA, with the use of a component-based web application framework (henceforth "component-based framework"), the components are intricately combined. Therefore, the framework is more complicated than in the conventional one. For example, the page structure is complex, and errors are easily mixed in the page control part. It is believed that validation using model checking is effective for such an SPA. Nevertheless, since the pages are dynamically constructed by a combination of components, it is difficult to apply the modeling method with the existing conventional static page.

In this paper, we therefore propose a method to automatically generate a verification model and formulas that verify page transitions from SPA source code using the component-based framework. Model checking is also performed. We implement a tool that automates the proposed method for SPA using the Angular component-based framework. Experiments are conducted on several applications. We herein use Simple Promela Interpreter (SPIN) as the model checker. Hence, Process Meta Language (Promela) describes the verification model, and Linear Temporal Logic (LTL) describes the verification formula.

II. Background

A. Single-Page Application (SPA)

In conventional web applications, each time an event occurs, such as a user interaction, the client synchronously requests the server (e.g., an initial request). The server responds to the client with all HTML of the corresponding page, and the client performs reloading and rendering processes.

In SPA, the server generally returns HTML, CSS, script files, and so on as a response only when responding to the initial request from the client. For subsequent requests from the client, we process and redraw using the front-end and asynchronously acquire data from the back-end in JSON format when needed. Using this mechanism, the SPA redraws

DOI reference number: 10.18293/SEKE2018-038

only the corresponding part without reloading the entire page, and it realizes page transitions as being controlled by the conventional back-end [2]. Thus, similar to a native application, the response to the user operation is fast and can provide excellent UX. Meanwhile, to realize SPA, a considerable amount of JavaScript code is necessary, and the front-end implementation and structure are complicated compared to the conventional approach [3].

B. Component-based Framework

To improve development efficiency and quality, a web application framework is usually employed in web application development. Owing to front-end complexities, web application frameworks are likewise diversified, and front-end framework development has advanced [3]. The framework architecture has also changed, and frameworks adopting a new component-based approach are being developed.

The component-based concept is founded on Web Components [4], for which the World Wide Web Consortium (W3C) developed specifications. The following Web Component functions are the primary ones [4]:

- Custom Elements
- HTML Imports
- HTML Template
- Shadow DOM

The component-based framework usually includes these four functions. Unlike the back-end framework based on the model–view–controller (MVC) architecture [6] described for each role in the component-based framework, independent component grouping views, logic, and so on are defined for each element constituting the page.

SPA using the component-based framework dynamically constructs the whole page by combining those components. It is thereby possible to improve its reusability and other aspects. Furthermore, both page generation and page transitions are controlled by the front-end; thus, the component-based framework has a routing function to associate a component with a path (URL pattern) and to control page transitions.

C. Angular

Angular [5] is a component-based framework developed by Google. For the development language, TypeScript, a superset of JavaScript, is recommended. It is not compatible with the earlier version of "AngularJS" (version 1). Since version 2 (September 2016 release), Angular has been referenced as "Angular." In this research, an evaluation experiment is conducted on Angular 4.0.1. Additionally, SPA using Angular is defined as "Angular SPA."

Basically one Angular component consists of the following:

- HTML Template, CSS Template
- TypeScript Class
- Metadata Using a Decorator

The entire page is comprised of one or more components, including a root component, which is the first component to be called when Angular SPA is activated.

In Angular, it is possible to dynamically replace parent components under the root component according to a path by RouterModule with a routing function. It is thus possible to realize page transitions, similar to the conventional one controlled by the back-end, without reloading the entire page.

In addition, a custom element can be created as a user-specific non-standard HTML tag (CustomTag) using the selector parameter of the @Component decorator describing the meta-information of each component. By inserting the value of the selector parameter as the tag name in the other parent component, the content of the HTML template of the given custom element can be displayed in the parent component. By using the custom element, a hierarchical structure can be composed of a parent component and a child component. Additionally, several of them can be arranged on one page, the components can be reused on different pages, and one component may be used on multiple pages. Figure 1. depicts an Angular page configuration example.

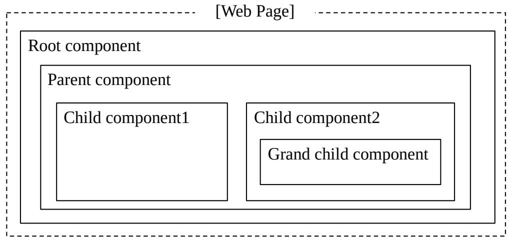

Figure 1. Example of a page structure using many components.

Thus, in general, the entire page of Angular SPA is largely divided into three elements:

- Root component
- Parent component controlled by the router (hereafter the "parent component")
- Child component group to be inserted using tags of values declared by the selector parameter of @Component decorators as the custom element (hereafter the "custom child component")

III. RELATED WORK

In this section, we briefly discuss some research related to verification for applications, focusing especially on model checking and page transition verification.

In [7], page transition diagrams that are used during the design phase are addressed. A method of model checking for one aspect of the whole application, depending on the page transition and system environment, is proposed. The approach differs from ours in that the former handles the page transition diagram at the design stage, not the implementation stage. In [8], modeling is performed in the UML model and the reachability of the page is verified. Test cases are generated; nevertheless, verification is not performed using a formal method.

In [12], a method using JPF-Android, an Android application verification tool, applies Java PathFinder (JPF) to detect errors, such as deadlocking of Android applications. Analysis of the actual source code of the application is similar to that in the present research. However, the authors of [13] target native Android applications, not web applications.

In [13], the authors focus on Apache Struts of the MVC architecture web application framework. A method, "Web Automation by Changing View," is proposed to model the behavior model of the web application. It is targeted at the implementation stage and is intended for web applications that use the Struts web application framework. However, when applying the model-checking method to the SPA page transition using the component-based framework, it is difficult to use a method of modeling one element of the MVC view as one page. Moreover, the extraction method is based on static page information.

In [9], [10], and [11], the authors focus on the implementation of a web application framework using dynamically typed languages. They respectively propose methods for extracting symbolic models to verify the data integrity of the model part and the access control security. Hence, the objectives differ from those of our research.

An example of front-end operation verification is the Rich Internet Application (RIA) (e.g., [15], [16]). The present paper differs from those works in that test cases are generated from execution traces of actual applications, attention is focused on interactions by event handlers, and search is performed by crawling. In the case of the crawling method, it cannot be verified that the back-end implementation has not been completed. However, our proposed approach focuses on the page transition part of the front-end. It can thus be verified without relying on the back-end implementation.

IV. METHODS

Our proposed method extracts necessary information from Angular SPA code for transforming the verification model and verification formulas for page transitions. Verification by model checking is performed to detect errors and improve the quality using verification models and their verification formulas. By implementing a tool that automates our proposed methods, ordinary web developers with minimal knowledge of model checking can also apply this method. The flow of the proposed method is shown as follows:

A. Extraction of information from Angular SPA

B. Construction of static page information

C. Transformation to the Promela model

D. Transformation to LTL formulas

E. Verification by SPIN

Figure 2. depicts the overall extraction input and output processes. Meanwhile, the page transitions herein are defined as follows: "Changing of the parent component embedded in the router—the router-outlet tag in the root component corresponding to a path by RouterModule—consequently changes the rendering of the entire page, similar to the transition of the conventional web application."

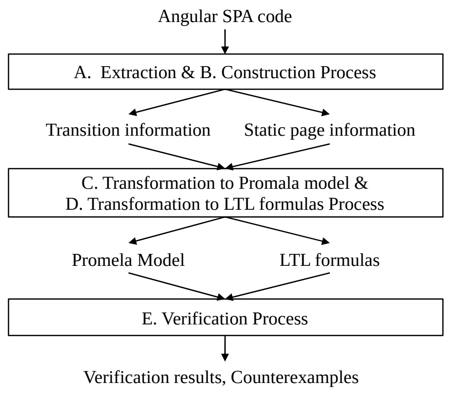
Angular SPA code
A. Extraction & B. Construction Process
Transition information
Static page information
C. Transformation to Promala model &
D. Transformation to LTL formulas Process
Promela Model
LTL formulas
E. Verification Process
Verification results, Counterexamples

Figure 2. Overview of our proposed flow.

The page transition information is a set of <before page, path, after page>, and it is divided into two sets: "page information" and "transition information." Page information indicates a path that can be transitioned from each page. It is a set of combinations of <before page, path>. Transition information indicates the transition destination page corresponding to the path described in the routing. It is a set of <path, after page>.

In this study, certain restrictions are placed on the Angular SPA description, such as not using child attributes and route parameters in routing. Since page transitions under the control of RouterModule are targeted, changes due to links to external web sites and data binding are not treated as page transitions.

A. *Extraction of Information from Angular SPA*

Transition information is included in routing defined by RouterModule and we thus extract it. Specifically, we extract path parameters and corresponding parent component names controlled by RouterModule.

Page information indicates a path that can transition from one page. In SPA, using the component-based framework, it consists of the set of "Page information in the root component" and "Page information in each component." These pieces of information are included not only in the parent component controlled by the RouterModule, but also in the custom child component that can be inserted using the selector element as descendants of the parent component. Therefore, page information is extracted from all components. Furthermore, since information indicating the parent–child relationship of each component is also necessary, the "Custom element information in each component," of which the parent component controlled by the RouterModule contains the descendant component, is also extracted.

B. *Construction of Static Page Information*

We automatically construct static page information from the information extracted in Section IV-A. Static page information is a set of transition-capable paths contained in those pages. In this paper, each of their page names is defined by the parent component name controlled by RouterModule.

We use the information of Section IV-A from the earlier flow outline to solve the parent-child relationships among the root component, parent component, and custom child component. We construct static page information representing the page information contained in each parent component controlled by RouterModule. It expresses the parent component name (the key part on the left of Table I) controlled by RouterModule, as well as the transition-capable paths (the value part on the right of Table I). We transform the Promela model and LTL formulas based on this static page information and transition information extracted in Section IV-A.

TABLE I. EXAMPLE OF GENERATED STATIC PAGE INFORMATION

```
{
  "component1": ["/component2", "/component3"],
  "component2": ["/component1"],
  "component3": ["/component4"],
}
```

C. *Translation to the Promela Model*

Given the transition information of Section IV-A and the static page information generated in Section IV-B, we convert the verification model necessary for model checking. In this paper, since SPIN is used as the model checker, we express the verification model by Promela, the modeling language used in SPIN.

The page transitions controlled by routing are indicated by a set of "pages that can transition from one page and the page to which the path transitions," such as <page 1, path, page 2>.

An example of page information by Promela is:

```
state == page1 -> state = path
```

An example of transition information by Promela:

```
state == path -> state = page2
```

As described above, a combination of page information and transition information expresses the Promela model of the page transition. In this Promela model, the state changes alternately with page, pass, page, pass... and so on. When it is possible to transition from one page to multiple pages, it is written as if it occurs non-deterministically using the syntax of "if...fi." The process is repeatedly performed using "do ... od" of the guard command of the repeating syntax.

D. *Translation to LTL Formulas*

The properties that generally hold in web applications are the following with reference to [7]:

(1) The page reachable from the top page always has a next page in the transition (property 1).

(2) Every page is reachable from the initial page (property 2).

(3) The initial page is reachable from all pages (property 3).

(4) A page transition is triggered only after several assumed pages (property 4).

We examine the above four properties. For property 1, we do not generate verification formulas because we do not input a formula. Rather, we perform verification using the default deadlock-free of SPIN (1). For property 2, for the initial page p and arbitrary page q, we have the following LTL formula:

$$\neg \Diamond(p \,\&\, \Diamond q) \tag{2}$$

which will be verified. If a transition is possible, an error occurs, and it is confirmed that a transition from the initial page to any page is possible. By changing an arbitrary page q and repeatedly verifying all pages, it can be confirmed that the model satisfies property 2. For property 3, as with property 2, for initial page q and any page p, we have the following LTL formula:

$$\neg \Diamond(p \,\&\, \Diamond q) \tag{3}$$

If a transition is possible, an error occurs, and it is confirmed that a transition from an arbitrary page to an initial page is possible. By changing arbitrary page p and repeatedly verifying all pages, it can be confirmed that the model satisfies property 3.

As described above, in the validation of property 2 and property 3, since formulas are necessary for input, we automatically generate LTL formulas for all page names that can be transitioned from all paths defined in routing using transition information extracted by Section IV-A. This supports the verification.

In addition to the properties referencing [7], we verify property 4. For this property, in specifying one page q, and for any page p, we have the following LTL formula:

$$\neg \Diamond(p \,\&\, XXq) \tag{4}$$

To use the next operator in SPIN, we must attach "-DNXT" option at gcc compile time. It is confirmed that p is included in the next page that can be transitioned from page q.

As described above, in this research, we model to include path transitions between page transitions, such as a path from a page and another page from a path. That is why the formula contains the two next operators (XX). By changing an arbitrary page, p, and repeating the verification for all pages, we can confirm that the model satisfies property 4. Specifically, it can be checked whether the next page is directly transferred from the unintended page to page p, and whether the next page can be transitioned directly from the intended page to page p.

E. *Verification by SPIN*

We input the Promela model generated in Section IV-C and LTL formulas generated in Section IV-D into SPIN and verify the model for the formulas. In the automation tool, we verify each generated LTL formula. If the verification result is false, it automatically analyzes the trail file and automatically outputs the verification result and counter example simulation result as files, respectively.

V. EXPERIMENTAL RESULTS

We employed an automated tool that implements the proposed method to conduct from the information extraction to the verification for SPA. When inputting Angular SPA, the automation tool can automatically perform all processes, from information extraction to generation of the verification model, generation of formulas, and execution of SPIN.

By applying our method to several sample applications, we checked whether the intended model was output. We then confirmed the feasibility of the flow of the proposed method and the feasibility of actually verifying it. In addition, we applied it to sample applications that intentionally incorporated errors to make properties false. We confirmed whether they could be verified correctly. Furthermore, to show the effectiveness of this method, we applied it to real applications published on the Internet.

A. *Experiment 1: Sample Applications*

We show the page transitions of sample application 1 (hereafter "sample app1") in Figure 3. First, we verified sample app1 with no errors in all properties. Next, we experimented using three of sample app2, sample app3, and sample app 4, in which errors for each property were injected.

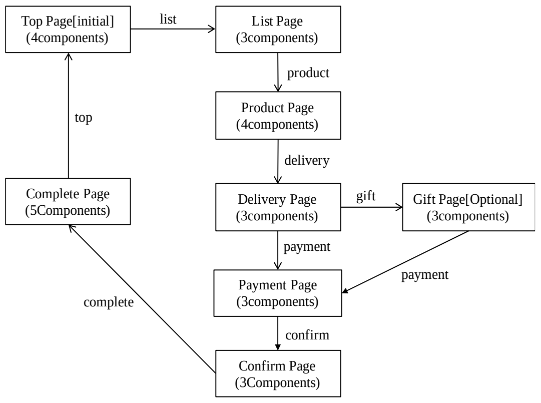
Top Page[initial] (4components)
list
List Page (3components)
product
Product Page (4components)
top
delivery
Complete Page (5Components)
Delivery Page (3components)
gift
Gift Page[Optional] (3components)
payment
payment
Payment Page (3components)
complete
confirm
Confirm Page (3Components)

Figure 3. Page transition diagram of sample app1.

1) *Verification Model*

We applied sample app1 to the automated tool and checked if the validation model was correctly generated automatically to represent the page transitions in Figure 6. As a result, the automatically generated verification model was correctly generated. It was generated automatically, as assumed from extraction to the modeling. Similarly, for sample app2, 3, and 4, the assumed verification model was correctly generated. Next, we verified the four properties using this verification model.

2) *Verification without Injecting Errors*

We verified property 1 against the verification model of sample app1 automatically generated by (1). As a result, no error was output, and it was confirmed that there was no problem in its properties, as expected. Likewise, we verified property 2, property 3, and property 4.

3) *Verification with Injecting Errors*

We verified sample app2, which intentionally injected the error of property 1 into sample app1. Specifically, in sample app2, there was no transition from "Complete Page" to "Top Page" of sample app1. We confirmed that the result was an error. As a result of verifying that property 1 was deadlock-free, it was possible to detect an intended error. Furthermore, a counter example was simulated using the output trail file. As a result, it was confirmed that the transition from "Complete Page" to "Top Page" was not completed and it stopped at "Complete Page," as expected.

Similarly, we verified sample app3, which intentionally injected the error of property 2 into sample app1. Specifically, in sample app3, there is no transition from "Confirm Page" to "Complete Page." Since the verification result of that part don't result in an error, it is observed that there was no path to reach "Complete Page" from the initial page "Top Page," and the intended error can be detected. As a result of the verification, the verification was completed without verifying the transition as an error. Therefore, it could detect the intended error.

Next, a bug injection experiment of property 3 was performed using sample app2 above. In sample app2, there was no transition from "Complete Page" to "Top Page" on the initial page. Therefore, contrary to property 3, it could not transition from all pages to the initial page. As a result, verification was completed without causing errors for all pages. Therefore, it was confirmed that an intended error was detected.

Finally, we tested sample app4, which intentionally injected the error of property 4 into sample app 1. Sample app4 added the transition from "Product Page" to "Payment Page" to sample app1. We confirmed the pages that could transition directly to "Payment Page." From the verification results, transitions from "Delivery Page," "Gift Page," and the additional "Product Page" were possible. Specifically, we verified that it was possible to transition from "Product Page" to the next "Payment Page." As a verification result, an intended error was detected. Owing to the simulation of the counter-example, we confirmed that it was possible to transition from "Product Page" to the next "Payment Page," as intended.

B. *Experiment 2: Real Applications*

To further demonstrate the effectiveness of our method, we applied the experiment to two different real applications (hereafter "Small App" and "Large App"), whose source code is published on GitHub [17]. We examined properties 1, 2, 3, and 4. In the case of property 4, we selected one of the parent component names defined in each routing and conducted the experiment. The scale of the two applications is shown below.

TABLE II. SCALE OF REAL APPLICATIONS

	LOC	Number of pages
Small App	1824	6
Large App	6893	23

In this experiment, we automatically generated the automatic verification model and formulas using the automated tool. The verification results using the automatically generated verification model and formulas are shown below.

TABLE III. RESULTS OF EXPERIMENTS FOR REAL APPLICATIONS

	Property 1	Property 2	Property 3	Property 4
Small	No error	No error	No error	No error
Large	No error	2 errors	2 errors	No error

First, as a result of verifying property 1, it was confirmed that there was no problem in its properties because no error was output in either application. Similarly, we verified property 2. As a result, in Large App, there were two pages for which no error was output, and bugs were detected in the transition to two pages. We confirmed that part of the application, the reachable transition to that page, was described in none of the pages.

Next, we examined property 3. As a result, similar to property 2, in Large App, there were two pages wherein no error was output, and an error was detected in the transition from two pages. We confirmed that aspect of the application. Finally, we verified property 4. For each Small App and Large App, a single subsequent page was specified and verified. We visually checked whether the SPA could actually transition directly to that screen for pages that were made transition-capable by the verification result.

In property 4, it was self-evident that no bug existed. However, when the developer actually verifies it, it can be considered effective because it can detect whether the SPA directly transitions to an unintended page.

C. Discussion

We conducted experiments on several applications using automated tools that we implemented. First, in the experiment on sample applications, it was possible to automatically correctly from the information extraction to verification of Angular SPA. In experiments with applications that did not inject bugs, and with applications that intentionally injected bugs, we obtained the desired verification results. Furthermore, even when we applied this method to applications published on the Internet, we could perform the task correctly. In practice, one application could detect multiple errors. Thus, our approach showed greater effectiveness.

Based on several experimental results, we confirmed that the proposed method enables correct generating and verifying of the verification model for the page transitions of Angular SPA. Moreover, by using our automated tool, it is considered that this method can be applied, even by ordinary developers who have minimal knowledge of model checking. As stated above, this study assumed certain constraints. These constraints mainly come from the first step of our method, i.e., the extraction of information from Angular SPA. By improving the analysis of SPA, these constraints can be decreased.

VI. Conclusion

In this paper, we proposed a model checking method for page transitions of SPA using a component-based framework. In addition, we implemented an automated tool that applies this method of automatically generating verification models and formulas from extraction from source code of SPA. The tool additionally performs the verification. By using the tool, even typical developers with minimal model checking knowledge can apply the proposed method. Furthermore, it was confirmed that there was no problem in the flow of the proposed method by using real applications intentionally mixed with errors and those that actually showed the source code.

Focus on the front-end page without reliance on the back-end is a strength of our proposed approach. However, the information extraction part of our implementation tool is directed to Angular SPA and thus assumes certain constraints. Therefore, it may be challenging to improve the tool and expand the application scope. Our future work will address this issue. Additionally, we will apply the method to various more complex Angular SPAs. Moreover, the information extraction part of the tool depends on Angular. Thus, we will consider implementing automation tools for SPAs using other component-based frameworks, such as Aurelia, or component-based libraries, such as React.

References

[1] Edmund M. Clarke; Orna Grumberg; Doron Peled, “Model Checking,” The MIT Press, 1999.

[2] Madhuri A. Jadhav; Balkrishna R; Sawant, Anushree Deshmukh, “Single Page Application using AngularJS,” International Journal of Computer Science and Information Technologies (IJCSIT), Vol.6, No.3, pp.2876-2879, 2015.

[3] Ning Zhang; Yizhen Cao; Shengyan Zhang, “Research of web front-end engineering solution in public cultural service project,” Computer and Information Science (ICIS), IEEE/ACIS 16th International Conference on, pp.623-626, 2017.

[4] Web Components, https://www.webcomponents.org/, accessed: 2018-03-01.

[5] Angular, https://angular.io/, accessed: 2018-03-01.

[6] Glenn E. Krasner; Stephen T. Pope, “A cookbook for using the model-view controller user interface paradigm in Smalltalk-80,” Journal of Object-Oriented Programming, Vol.1, No.3, pp.26-49, 1988.

[7] Kei Homma; Satoru Izumi; Kaoru Takahashi; Atsushi Togashi, “Modeling, Verification and Testing of Web Applications Using Model Checker,” IEICE Transactions on Information and Systems, Vol.94, No.5, pp.989-999, 2011.

[8] Filippo Ricca; Paolo Tonella, “Analysis and testing of Web applications,” Proceedings of the 23rd International Conference on Software Engineering (ICSE), Vol.47, No.6, pp.25-34, 2001.

[9] Joseph P. Near; Daniel Jackson, “Rubicon: bounded verification of web applications,” Proceedings of ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering(FSE), No.60, pp.1-11, 2012.

[10] Joseph P. Near; Daniel Jackson, “Finding security bugs in web applications using a catalog of access control patterns,” Proceedings of IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp.947-958, 2016.

[11] Ivan Bocić; Tevfik Bultan, “Symbolic Model Extraction for Web Application Verification,” Proceedings of IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp.724-734, 2017.

[12] Heila van der Merwe, “Verification of android applications,” Proceedings of the 37th International Conference on Software Engineering (ICSE), Vol. 2, pp.931-934, 2015.

[13] Shoji Yuen; Keishi Kato; Daiju Kato; Daiju Kato; Kiyoshi Agusa, “Web automata: A behavioral model of web applications based on the MVC model,” Information and Media Technologies, Vol.1, No.1, pp.66-79, 2006.

[14] Alessandro Marchetto; Paolo Tonella; Filippo Ricca, “State-based testing of Ajax web applications,” Software Testing, Verification, and Validation (ICST), 1st International Conference on, pp. 121-130, 2008.

[15] Domenico Amalfitano; Anna Rita Fasolino; Porfirio Tramontana, “Rich internet application testing using execution trace data,” Software Testing, Verification, and Validation Workshops (ICSTW), Third International Conference on, p. 274-283, 2010.

[16] Frederik Nakstad; Hironori Washizaki; Yoshiaki Fukazawa, “Finding and Emulating Keyboard, Mouse, and Touch Interactions and Gestures while Crawling RIAs,” International Journal of Software Engineering and Knowledge Engineering (SEKE), Vol.25, pp.1777-1782, 2015.

[17] GitHub, https://github.com/, accessed: 2018-03-01.