A Method for Automated User Interaction Testing of Web Applications

Le Khanh Trinh, Vo Dinh Hieu, Pham Ngoc Hung
Faculty of Information Technology, University of Engineering and Technology, VNU
Email: trinhlk@vnu.edu.vn, hieuvd@vnu.edu.vn, hungpn@vnu.edu.vn

***Abstract* - Automated user interaction testing of Web applications has been received great attentions from the research community and industry. Currently, several available tools are proposed to partly deal with the problem. However, how to perform the automated user interaction testing of whole Web applications effectively is still an open problem. This research proposes a method and develops a tool supporting automated user interaction testing of whole Web applications. In this method, the model of each Web page of the Web application under testing which describes the user interaction (UI) is represented by a finite state automaton. The whole model that describes the behaviors of the whole Web application then is constructed by composing the models of all Web pages. After that, test paths are generated automatically based on the compositional model of the Web application so that these test paths cover all possible user interactions of the application. A tool supporting the proposed method has been developed and applied to test on some simple Web applications. The experimental results show the potential application of this tool for automated user interaction testing of Web applications in practice.**

Keywords* - *User Interface Testing, Web Applications, Model-based Testing.

1. INTRODUCTION

In model-based testing, test cases are generated automatically from a model that exactly describes the intended behavior of the application under testing [4, 10]. In fact, the model-based testing can bring success for software testing activities. This approach has potential to increase reliability, reduce implementation efforts, and tedious activities in the testing process. Especially, if we have a large number of test cases, manual testing can lead to deficiencies that will unpredictably damage the project. In this case, the model-based testing can lead to deficiencies that will unpredictably damage the project. In this case, the model-based testing has been considered as one of the potential solutions to deal with the problem. been considered as one of the potential solutions to deal with the problem.

Nowadays, Web applications have rapidly become popular in practice. Applying model-based testing for Web applications has received great attentions from the research community and industry. Testing has been recognized as a major solution in order to guarantee the quality of Web applications. Currently, there are several methods and tools that support for testing of Web applications. Most of them focus on nonfunctional requirements such as usability, performance, the load-bearing capacity, security, etc. [2, 3, 11]. On the other hand, some researches and tools can support to functional testing [5, 6]. However, these researches almost support on testing element's functions and Web architecture but there are not many researches on user interaction testing. Furthermore, the methods just allow ensuring the quality of Web applications by testing each Web page separately. In fact, UI testing of whole Web application is very important and takes a lot of time and effort, specially in the system and acceptance testing phases [7, 8, 12]. Nevertheless, this task is performed manually with very low coverage because UI testing on the whole Web applications is very complex and not possible. Consequently, there are many errors related to UI when applications are deployed in practice. Therefore, we need a solution for automated UI testing of whole Web applications. In addition, in order to apply the model-based testing, it is very difficult to obtain models that exactly

describe behaviors of the Web applications under testing. As a result, the current methods are difficult to be applied in practice.

This paper proposes a method for automated UI testing of whole Web applications in order to deal with the above issues. In this method, user interactions of a Web page are represented by a finite state automaton. The model of whole Web application is generated by composing by all models corresponding UI Web pages. The Depth First Search algorithm (DFS) then is applied to generate all test paths corresponding to user's stream activities. After that, the generated test paths are used as test cases in order find out errors/mistakes related to UI in the implementation of the Web application under test. Moreover, the proposed method also supports a GUI tool for building the UI models of Web pages visually. A tool supporting the proposed method has been developed and applied to test on some simple systems. The experimental results obtained clearly show that the proposed method and implemented tool are potential to be applied in practice.

2. UI MODELS OF WEB APPLICATIONS

In order to perform automated UI testing, we first have to build a model that exactly describes UI behaviors of the whole Web application under testing. This model specifies the user interactions by a finite state automaton. For the purpose, we first build UI models of all Web pages of the Web application. The UI model of the application then is generated by composing these UI Web page models.

2.1. UI Web Page Model

In fact, each Web page is implemented corresponding to some certain functions of the Web application. Web pages can link together by the way that moves to the another page such as: click actions on links, buttons, tab, etc. In this research, UI behaviors of each Web page are represented by a finite state automaton (FSA) $M = <S, s_0, \Sigma, \delta, F>$, where:

- S is the non-empty finite set of states of the Web page,
- $s_0 \in S$ is the initial state,
- Σ a collection of events in the form *[guard] event*,
- δ is the state transition function $\delta : S \times \Sigma \rightarrow S$, and
- $F \subseteq S$ is the set of final states.

Note 1. The form *[guard] event* means that the event occurs if and only if guard holds. We use $M = \Pi$ to denote the empty state automaton, where its set of state is empty (i.e., $S = \emptyset$).

2.2. Generating UI Models of Web Applications

The UI model of whole Web application is generated by composing all UI Web page models. This method selects a model to be the home page in order to compose UI Web page models. The home page is the first UI Web page model which is used to compose other models, there must have a test sequence in which each UI Web page model is visited in the given order. Therefore, in this research, we use the sequence composition operator in order to combine two UI Web page models. The sequence composition between two UI models $M_1 = <S_1, s_{01}, \Sigma_1, \delta_1, F_1>$ and $M_2 = <S_2, s_{02}, \Sigma_2, \delta_2, F_2>$ is a UI model $M = <S, s_0, \Sigma, \delta, F>$, denoted $M = M_1 \| M_2$, is defined as follows. If $M_1 = \Pi$, $M_2 = \Pi$, or $s_{01} \notin F_2$ and $s_{02} \notin F_1$ then $M = M1||M2 = \Pi$. Otherwise, $M = M_1||M_2$ is a UI model, where $S = S_1 \cup S_2$, $\Sigma = \Sigma_1 \cup \Sigma 2$, $\delta = \delta_1 \cup \delta_2$, $F = F_1 \cup F_2$, and the initial state is identified by following rules:

- If $s_{01} \notin F_2$ and $s_{02} \notin F1$ then $M = \Pi$.
- If $s_{02} \in F_1$ then M_1 is the home page of the Web application, and $s_0 = s_{01}$.
- Otherwise, $s_0 = s_{02}$.

Note 2. We use $M = \Pi$ to denote the empty state automaton, where its set of state is empty (i.e., $S = \emptyset$).

For example, Figure 1 shows two models M_1 and M_2, where:

- If $s_{01} \notin F_2$ and $s_{02} \notin F_1$ then $M = \Pi$.
- The initial state (s_{01}) of model M_1 is *S_index*.
- The model M_1 has three final states (F): *Admin_main*, *Normal_main*, *PDT_main*.

•The initial state (s_{02}) of model M_2 is *Normal_main*.

• The model M_2 has three final states (F): *Report*.

• M_2's initial state is *Normal_main*.

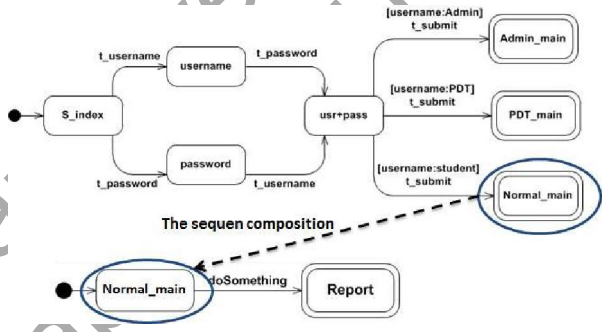


Figure 1. The models of M_1 and M_2

Figure 2 illustrates the compositional model is composed between M1 and M_2.

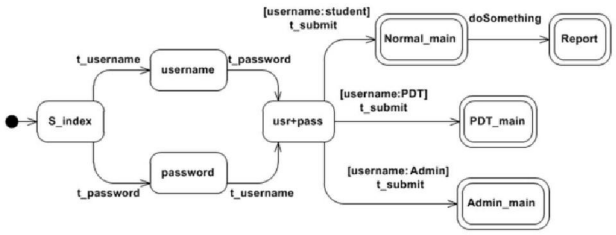


Figure 2. The sequence composition between M_1 and M_2

Suppose that a Web application has n Web page models such as: $M_1 = <S_1, s_{01}, \Sigma_1, \delta_1, F_1>$, $M_2 = <S_2, s_{02}, \Sigma_2, \delta_2, F_2>$,..., $M_n = <S_n, s_{0n}, \Sigma_n, \delta_n, F_n>$, where M_1 is the UI model of the homepage. Firstly, M_1 is composed with M_i ($2 <= i <= n$) to become M_{1i}. The model M_{1i} then is composed with Mj ($2 <= j <= n$ and $i \neq j$). In the end, all finite state automata are composed in order to obtain the compositional model of the whole Web application.

Remark 1. The sequence composition operator is associative but not commutative.

3. UI TESTING FOR WEB APPLICATION

Given the compositional model that describes UI behaviors of the whole application, the proposed method uses the DFS algorithms for finding errors/mistakes related to the implementation phase in comparison with the given design of the application.

3.1. Test Case Generation

In this research, the form of each test path is $<initial\_state> * <event_i> = <state_i> *... = <final\_state>$.

It is started by the initial state. At this state, if the $event_i$ occurs, the next state of the system will be $state_i$. The ending of each test path is a final state.

An algorithm for generating test cases has to satisfy the following requirements:

• All states and transitions of the model must be approved.

• The result is a list of distinguishing test paths. Beginning state of each test path is the initial state of the home page's FSA and the ending state is the final state of the other FSA.

Details of the Depth First Search algorithm (DFS) for generating test paths is presented in algorithm 1. The input variables are the start state named *i* and the test path is stored in the variable named P AT H. Firstly, the variable that called *backTrack* is initialized with initial value is true (line 1). After that, the algorithm will check each transition (line 2). If transition is not approved (line 4), the value of variable *backTrack* will be *false* (line 4). In this case, the algorithm will add the begin state of transition into the list named *arr* - the variable that stored temporary test paths (line 11) and notice that the transition has been approved (line 5). The algorithm is implemented recursively with the input is the end state of the transition has approved (line 7) until the state that it cannot add any more transition into test path arr. Then, the value of the variable *backTrack* is true (line 10), the variable *arr* will be appended into variable PATH (line 11). Finally, the algorithm removes all states of *arr* to create a new test path (line 8).

Algorithm 1 DFS (int *i*, path PATH)

Input: *i*: the identity of begin state; **PATH** : the set of test paths
Output: list of test paths covering all cases of UI interaction on the whole Web application.
1: **Bool** *backTrack* = true;
2: **for** all transitions **do**

```
3:     if the transition hasn't been approved then
4:         backTrack = false;
5:         the transition has been approved;
6:         add state into arr;
7:         DFS (i, PATH );
8:         remove all states from arr
9:     end if
10:    if backTrack = true then
11:        add arr into PATH ;
12:    end if
13: end for
```

Algorithm 1 is an extension of the Depth First Search algorithm. All states had been approved after done the DFS algorithm. However, in some test paths, the end state is not the final state of the Web page model. Therefore, this paper uses algorithm 2 to add more states into test path to ensure that ending state of all test paths is final state of finite state automaton. The input of algorithm 2 is variable *PATH* which is a set of test paths. The algorithm approves each test path in *PATH*, while the ending state of test path *i* is not final state (line 2), the algorithm looks up the transition in FSM which has beginning state is the ending state of this test path (line 4). After that, the ending state of this transition will be added to the test path (line 5).

Algorithm 2 ADD_Path (path *PATH*)

Input: A new key *k* to insert into the heap.
Output: The ending state of each test path is the final state.

```
1: for Approving all test paths of PATH do
2: while the ending state of test path i is not the final state do
3:   for approving all the transitions of the FSM do
4:     if the beginning state of transition is the ending state of PATH then
5:       add this ending state of transition into test path i;
6:           end if
7:         end for
8:     end while
9: end for
```

Table 1 illustrates the generated test paths from the UI model of Login Web page shown in Figure 3 using the proposed method. The first column (Table 1) is serial of test path and the second column shows the detail of each test path. It is easy to prove that the generated test case can cover all user interactions of this Web page.

Table 1. The list of test paths

Test path	Result
1	S_index * t_username = username * t_password = usr+pass * del_password = username * t_submit = error * t_back = S_index * t_submit = error
2	S_index * t_username = username * t_password = usr+pass * del_password = username * t_submit = error * t_back = S_index * t_password = password * t_username = usr+pass * t_submit = PDT_main
3	S_index * t_username = username * t_password = usr+pass * del_password = username * t_submit = error * t_back = S_index * t_password = password * t_username = usr+pass * t_submit = Admin_main
4	S_index * t_username = username * t_password = usr+pass * del_password = username * t_submit = error * t_back = S_index * t_password = password * t_username = usr+pass * t_submit = Norm_main
5	S_index * t_username = username * t_password = usr+pass * del_password = username * t_submit = error * t_back = S_index * t_password = password * t_username = usr+pass * del_username = password * t_submit = error

3.2. Test case Execution

After generating the test paths from section 3.1, the proposed method connects to the Web application by using the Selenium-WebDriver APIs in order to run automatically that for finding mistakes/errors in the implementation of the Web

application under testing. For the purpose, we use the following API.

• API connecting to Web browser

Selenium connects to a Web browser object and supplied methods to control the Web browser automatically. For example, after the tool implemented the command:

WebDriver ffdriver = newFirefoxDriver();

FireFox browser will be started. Because *ffdriver* object is used to control FireFox browser.

• Retrieving a Web page

After connecting to a browser, Selenium will use the method named get for going to a Web page. For example, the API helps to go to google.com by the command:

driver.get("http://www.google.com");

• Locating the Web Elements

Selenium-WebDriver uses the method name *findElement* to detect the location of the Web element through attributes such as *id*, *name* or *class* that is presented in Table 2.

Table 2. Locating the Web element

ID	WebElement element = driver.findElement (By.id("id value"));
Name	WebElement tests = driver.findElement (By.name("test"));
Class	List<WebElement> tests = driver. findElements (By.className("test"));

• Working with Web Elements

After determining the location of each Web element, Selenium-WebDriver provides several functions to perform mouse gestures and keyboard with this Web element. For example, the method *element.sendKeys("Test!")* is used to assign value *"Test!"* into object element and method *element.submit()* is called when A button is clicked.

In order to perform the generated test paths, the proposed method enters the sets of the values of Web elements. For each test path, the method starts from the initial state and then identifies the next state on the Web page. After that, the proposed method compares it with the next state in a test path. If these states are different, the test case corresponding to the test path is failed. Otherwise, the implemented tool will detect the next state in this test path. By the similar methodology, the process is performed repeatedly until reaching the final state of the test path. If this process doesn't have errors, the result of this test path is passed.

4. IMPLEMENTATION

4.1. Automated Testing Web Application Tool

We implement a tool to show the effectiveness of the proposed method. Figure 4 describes the tool architecture, including the following function:

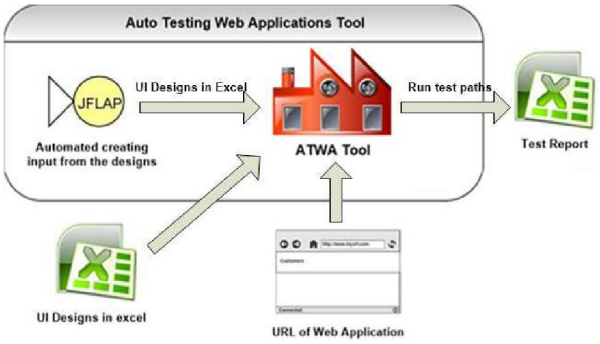


Figure 4. Architecture of the tool

• The input of the tool is a folder comprising a collection of *MS Excel* files that specifies all Web pages, a functional module or the whole of Web application. We have two ways for giving input to the tool. Firstly, the tester modifies the *MS Excel* files that describe the design models of Web pages. Secondly, the tester uses an extension called JFLAP[1] to draw the design model of Web application. JFLAP is an open source tool, we have modified that to generate automatically specifications from models which is the input of the tool.

• After receiving specifications, the tool named Automated Testing Web Applications (ATWA) will run the algorithms to generation automatically test cases, one test case can include a lot of test paths. Tester then enters the address of the module that needs to test and the tool will connect to this address and run one by one test paths.

[1] www.jflap.org

• The result of the testing process will be exported to a *MS Excel* file call test report. In this file, the tester can check each test path that *PASS* or *FAIL*. If the test path F AI L, the tester can see the detail in the next column (Figure 5).

PATHS	STATE	DETAIL FAILURE
Test path 13: S_index*t_usrname=usrname*t_passwd=usr+pass*t_submit=Admin_main*t_id=id*del_id=Admin_main*t_name=name*del_name=Admin_main*t_acc=acc*del_acc=Admin_main*...=acc+pass*del_pass=acc*t_id=id+acc*del_id=acc*t_name=name+acc*del_name=acc*t_pass=acc+pass*del_acc=pass*t_del=Error*t_tryAgain=S_index	PASS	
Test path 14: S_index*t_usrname=usrname*t_passwd=usr+pass*t_submit=Admin_main*t_id=id*del_id=Admin_main*t_name=name*del_name=Admin_main*t_acc=acc*del_acc=Admin_main*....=id+name+acc+pass*t_add=Admin_main	FAIL	Real Output ("") and Expected Output ("admin") of element: "title_main" are different. FAIL STATE: "Admin_main"
Test path 15: S_index*t_usrname=usrname*t_passwd=usr+pass*t_submit=Admin_main*t_id=id*del_id=Admin_main*t_name=name*del_name=Admin_main*t_acc=acc*del_acc=Admin_main*...=name+acc+pass*del_name=acc+pass*del_pass=acc*del_acc=Admin_main	PASS	
Test path 16: S_index*t_usrname=usrname*t_passwd=usr+pass*t_submit=Admin_main*t_id=id*del_id=Admin_main*t_name=name*del_name=Admin_main*t_acc=acc*del_acc=Admin_main*...=id+name+acc+pass*del_pass=id+name+acc*t_pass=id+name+acc+pass*t_del=Admin_main	FAIL	Real Output ("") and Expected Output ("admin") of element: "title_main" are different. FAIL STATE: "Admin_main"
Test path 17: S_index*t_usrname=usrname*t_passwd=usr+pass*t_submit=Admin_main*t_id=id*del_id=Admin_main*t_name=name*del_name=Admin_main*t_acc=acc*del_acc=Admin_main*...=id+name+acc*t_pass=id+name+acc+pass*del_acc=id+name+pass*t_acc=id+name+acc+pass*t_edit=Admin_main	FAIL	Cannot match HTML element: "title_main". FAIL STATE: "Admin_main"

Figure 5. The result of the testing process

In order to apply prove the effectiveness of the proposed method, we experiment this tool to test a Web application for course registration (Fig. 6), including the following functions:

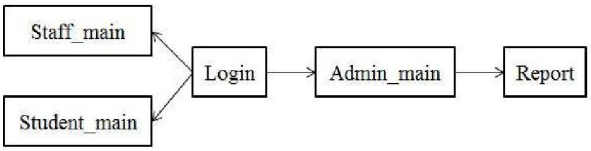


Figure 6. The site map of the course registration Web application

• Login is the home page.

• We have three pages for three different users (*Admin, Staff,* and *Student*). In this case, we test the module of user Admin. After logged in, user Admin does something to change the data of other users (in *Admin_main*) such as: showing user's information, inserting a new user, deleting an user, and editing an user. If the manipulation is fail, the Web application will move to *Report* page. User *Staff* manages subjects in *Staff* main and user *Student* registers the subjects in *Student_main.*

• For example, we perform the test cases and it finds out some errors. After fixing that errors and run again, we have a new report that shows the errors have been fixed. The detail of the implemented tool and applied application can be found at http://uet.vnu.edu.vn/~hungpn/ATWATool/.

4.2. Struture of the Input

The input of the implemented tool is a folder involve *MS Excel* files, where each file specifies the finite state automaton of a Web page. The output of the tool is a report file, the content of the report are list of test paths and the result of each test path.

For example, the Login excel file include:

• Table *Element_html*: This table describe Web Elements, where:

- Column *id*: the identity of Web element, which is the increment numbers.
- Column *html id*: the identity of Web element that is showed in HTML tags.
- Column *type*: the type of Web element. In that time, the tool can test with some type such as: *textbox*, *checkbox*, *radiobox*(*radio*), *button*(*btn*), *texthtml*, *title*(*pgTitle*).
- Column *value1*, *value2* are the sets of value for elements.

id	html_id	type	value_1	value_2
0	index:txtUsername	textbox	admin	admin
1	index:txtPasswords	textbox	admin	admin
2	index:btnSubmit	btn		
3	index:btnReset	btn		
4	title_index	pgTitle	index	index
5	title_main	pgTitle	admin	admin
6	title_error_Lg	pgTitle	Error	Error
7	error:btnBack	btn		

Figure 7. Representation of Web page's elements in Excel file

Figure 7 illustrations for table Element_html

• Table *State*: This table describes all states of the Web page. The first column is the numbers to define types of state: Number *0* is the initial state. If this is a final state, the number is *1*. Number *5* is the normal state. The second column includes the name of each state. In the others, the first row is the id of Web element that was described in *Element_html* table.

The values of each cell below is corresponding with two states of Web element:

- "o": The Web element has a not null value. For example: the value of a title is "example".
- " ": The value of the Web element is empty. For example: A button do not need a value.

Figure 8 shows the list of states and the value of each Web element that makes up this state.

State	name	0	1	2	3	4	5	6	7
0	S_index					o			
5	usrname	o				o			
5	passwd		o			o			
5	usr+pass	o	o			o			
1	Norm_main						o		
1	PDT_main						o		
1	Admin_main						o		
5	ErrorLg							o	

Figure 8. States of the web page model

• Table *Event*: This table illustrations all events in each UI Web page, where:

- Column *name*: Event's name,
- Column *html _id*: the identity of Web element that is showed in HTML tags, and
- Column *action*: The action of an event: click, addtext (add some characters), deltext (delete a text), select.

Figure 9 shows the list of the event in Login page.

Event	name	html_id	action
1	t_reset	index:btnReset	click
2	t_usrname	index:txtUsername	addtext
3	t_passwd	index:txtPasswords	addtext
4	del_usrname	index:txtUsername	deltext
5	del_passwd	index:txtPasswords	deltext
6	t_submit	index:btnSubmit	click
7	t_back	error:btnBack	click

Figure 9. Events in the Web page

• Table *Transition*: The contents of this table are transitions of a Web page. The second column dis- play name of the beginning states and the first row shows name of the ending states. The formula of each cell is *[guard]event*. Therefore, each cell is showed by three type Transition

– Only event's name (*event*): having a transition,

– Guard and event's name (*[guard]event*): if guard is true, transition will be done, and

– *empty*: it does not exist transition between two states.

Figure 10 shows the list transition of a Web page.

name	S_index	usrname	passwd	usr+pass	Norm_main	PDT_main	Admin_main	ErrorLg
S_index		t_usrname	t_passwd					t_submit
usrname				t_passwd				t_submit
passwd				t_usrname				t_submit
usr+pass		del_pass			[index:txtUsername/trunghd_55]t_submit	[index:txtUsername/pdt]t_submit	[index:txtUsername/admin]t_submit	
Norm_main								
PDT_main								
Admin_main								
ErrorLg								

Figure 10. Transitions of the Web page

5. RELATED WORKS

There are many works that have been recently proposed in automated model-based testing for Web applications, by several authors. Focusing only on the most recent and closest ones, we can refer to [5 - 9], to to [12, 1].

J.Ernits et al. proposed an algorithm for automatic testing of Web applications by replacing the usual client (for example, a user with a web browser) with a test tool [5]. The tool generates test cases that correspond to client behaviors, including sending requests to the server and checking the server's responses to determine whether it passes or fails each test. Although the motivation of this work is different, systems are specified using non-deterministic extended state machines (ESMs) with arbitrarily rich states, inputs, and outputs. Based on this work, the work proposed in [6] presents a novel automated model-based testing system for on-the-fly testing of thin-client Web applications. The key idea of this method is the systems are specified using non-deterministic extended state machines (ESMs) with arbitrarily rich states, inputs, and outputs. For a Web application, this implies that the testing system performs an input from the current page, e.g., pressing a button or editing a text field, and

receiving a new page in HTML. So, this paper can test automatically a Web page based-on Extended State Machines and outputs are the list of other Web pages. This work has been extended in [9] for the analysis and comprehension of traditional software exist that could be fruitfully adapted to the study of Web applications. However, Web applications always have several Web pages; some Web pages need to get some information from others. In this case, this research cannot build the ESMs for the problem.

An approach to automated testing of GUI applications was proposed by Atif M. Memon et al. in [7]. This work aims to demystify testing in event-driven systems by presenting an overview on the state of the art in GUI testing, with lectures on modeling, test generation and replay, as well as a discussion of the important factors that should be considered for controlling flakiness. The work suggests two demonstrations of the art in GUI testing. The first demonstration on the basics of automated GUI testing using the open source framework GUITAR [8], with examples from various domains such as desktop, Web, and mobile applications. A second demonstration on benchmarking and experimentation uses artifacts from the COMET (COMmunity Event-based Testing) Web applications with a focus on repeatability of test results. It is difficult to apply this idea to testing for UI Web applications.

Wenhua et al. in [12] proposes an algorithm for automatic testing of Web applications by models by the way that is generated some paths to detect the errors. This paper assumes they had two models of two Web pages. Afterward, a propose method will generate a small number of test sequences that are effective for detecting interaction faults. It cannot cover all cases of the system.

Finally, the work in [1] evaluates a solution that uses constraints on the inputs to reduce the number of transitions, thus compressing the FSM. Web applications are rapidly becoming more widespread, larger, more interactive, and more essential to the international use of computers. It is well understood that web applications must be highly dependable, so this research uses a straightforward technique to model Web applications as finite state automata. However, large numbers of input fields, input choices and the ability to enter values in any order combine to create a state space explosion problem. Therefore, the work presents an analysis of the potential savings of the compression technique and reports actual savings from two case studies. All researches above [1, 5 - 9] do not have a satisfying solution to test the whole Web applications.

6. CONCLUSION

We have presented a method for testing user interaction of whole Web applications automatically. At first, the proposed method represents each Web page by a finite state automaton. After that, the model of whole Web application is generated by composing the models of all Web pages using the sequence composition operator. Then, all possible test paths are generated automatically by applying the Depth First Search algorithm. Finally, the test paths as test cases are used to test on the implementation in order to find out errors/mistakes. If all the generated test cases are executed successfully, the result will be PASS. Otherwise, the result will be FAIL and the report file will show the detail of the FAIL test path.

The best advantage of the proposed method is high coverage. Generating test paths can cover all possible cases of user interactions. This work tests based on user interaction of the Web application that means it does not depend on the programming language. By sequence composition operator, the proposed method can test on whole Web applications automatically. Moreover, the extension named JFLAP helps tester to create the models of each Web page for the implemented tool visually. Moreover, the advantages of the method are potential to be applied in practice and become an useful tool for testing Web applications. Besides these advantages, the method still have some limitations. Tools currently just been tested with small and medium size Web applications with the basic Web element. The element must have the attribute id or name. In addition, we do not have a satisfactory resolution for testing pop-up interfaces. Finally, we need to have the detailed comparisons of the performance with other automated testing of Web application tools.

Currently, we also implement a tool which has been applied to test some simple Web applications

and received positive feedbacks about the usability as well as the ability to cover all probable cases of applications. We are applying this tool for more complex systems to demonstrate the effectiveness of the proposed method. Meanwhile, we continued deployment the tool with partners to get more feedbacks in order to improve its performance. At the same time, we are going to improve the implemented tool supports for covering more Web elements such as: pop-up interfaces, dynamic elements, etc. Comparing our tool to related existing tools will help upgrade more features to ensure that the testing becomes easier, more convenient, and efficient.

ACKNOWNLEDGEMENT

This work is supported by the project no. QG.13.20 granted by Vietnam National University, Hanoi (VNU).

REFERENCE

[1]. A. A. Andrews, J. Offutt, C. Dyreson, C. J. Mallery, K. Jerath, and R. Alexander. Scalability issues with using fsmweb to test web applications. Inf. Softw. Technol., 52(1):52–66, Jan. 2010.

[2]. M. Anisetti, C. A. Ardagna, E. Damiani, and F. Saonara. A test-based security certification scheme for web services. ACM Trans. Web, 7(2):5:1–5:41, May 2013.

[3]. A. Armando, R. Carbone, L. Compagna, K. Li, and G. Pellegrino. Model-checking driven security testing of web-based applications. In Proceedings of the 2010,Third International Conference on Software Testing, Verification, and Validation Workshops, ICSTW ’10, pages 361–370, Washington, DC, USA, 2010. IEEE Computer Society.

[4]. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner. Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[5]. J. Ernits, R. Roo, J. Jacky, and M. Veanes. Model-based testing of web applications using nmodel. In Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop, TESTCOM ’09/FATES ’09, pages 211–216, Berlin, Heidelberg, 2009. Springer-Verlag.

[6]. P. Koopman, R. Plasmeijer, and P. Achten. Model-based testing of thin-client web applications. In Proceedings of the First Combined International Conference on Formal Approaches to Software Testing and Runtime Verification, FATES’06/RV’06, pages 115–132, Berlin, Heidelberg, 2006. Springer-Verlag.

[7]. A. M. Memon and M. B. Cohen. Automated testing of gui applications: Models, tools, and controlling flakiness. In Proceedings of the 2013 International Conference on Software Engineering, ICSE ’13, pages 1479–1480, Piscataway, NJ, USA, 2013. IEEE Press.

[8]. B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. Guitar: An innovative tool for automated testing of gui-driven software. Automated Software Engg., 21(1):65–105, Mar. 2014.

[9]. F. Ricca and P. Tonella. Analysis and testing of web applications. In Proceedings of the 23rd International Conference on Software Engineering, ICSE ’01, pages 25-34, Washington, DC, USA, 2001. IEEE Computer Society.

[10]. M. Utting and B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Pub- lishers Inc., San Francisco, CA, USA, 2007.

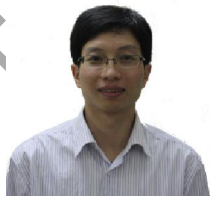
[11]. A. Vernotte. Research questions for model-based vul- nerability testing of web applications. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, ICST’13, pages 505–506, Washington, DC, USA, 2013. IEEE Computer Society.

[12] W. Wang, S. Sampath, Y. Lei, and R. Kacker. An interaction-based test sequence generation approach for testing web applications. In Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium, HASE ’08, pages 209–218, Washington, DC, USA, 2008. IEEE Computer Society.

\

AUTHORS' BIOGRAPHIES

Le Khanh Trinh received his B.S. degree from University of Engineering and Technology, Vietnam National University, Hanoi (2014). He is now an assistant teaching at Faculty of Information Technology, University of Engineering and Technology, Vietnam National University, Hanoi. His research interests include assume-guarantee verification, model-based testing, and software evolution.



Vo Dinh Hieu is working at Faculty of Information Technology, University of Engineering and Technology, VNU Hanoi. He received PhD degree from Japan Advanced Institute of Science and Technology. His research interests include Web services, service composition, and service-oriented architectures.

Pham Ngoc Hung received his B.S. degree from University of Engineering and Technology, Vietnam National University, Hanoi (2002), M.S. and PhD. degrees from Japan Advanced Institute of Science and Technology (2006, 2009). He is now a lecturer at University of Engineering and Technology, Vietnam National University, Hanoi. His research interests include model checking, assume-guarantee verification, model-based testing, and software evolution.