Tsinghua Science and Technology

Volume 26 | Issue 1 Article 5

2021

Behavior Model Construction for Client Side of Modern Web Applications

Weiwei Wang
the College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China.

Junxia Guo
the College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China.

Zheng Li
the College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China.

Ruilian Zhao
the College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China.

Recommended Citation

Weiwei Wang, Junxia Guo, Zheng Li et al. Behavior Model Construction for Client Side of Modern Web Applications. Tsinghua Science and Technology 2021, 26(1): 112-134.

TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 10/10 pp112–134
DOI: 10.26599/TST.2019.9010043
Volume 26, Number 1, February 2021

Behavior Model Construction for Client Side of Modern Web Applications

Weiwei Wang, Junxia Guo, Zheng Li, and Ruilian Zhao*

Abstract: Most of the behavior models with respect to Web applications focus on sequencing of events, without regard for the changes of parameters or elements and the relationship between trigger conditions of events and Web pages. As a result, these models are not sufficient to effectively represent the dynamic behavior of the Web 2.0 application. Therefore, in this paper, to appropriately describe the dynamic behavior of the client side of Web applications, we define a novel Client-side Behavior Model (CBM) for Web applications and present a user behavior trace-based modeling method to automatically generate and optimize CBMs. To verify the effectiveness of our method, we conduct a series of experiments on six Web applications according to three types of user behavior traces. The experimental results show that our modeling method can construct CBMs automatically and effectively, and the CBMs built are more precise to represent the dynamic behavior of Web applications.

Key words: web applications; client-side behavior model; user behavior trace

1 Introduction

Web applications have been the most widespread applications because they can offer rich interactivity and responsiveness with the help of multiple languages and interaction manipulations on the client and server side. As is known, model-based testing is one of the most effective techniques to ensure the quality and reliability of Web applications[1–4]. However, existing dynamic inter-dependencies among different languages, distributed asynchronous client/server nature, and event-driven property pose many challenges in analyzing and modeling Web applications[5–9], including the following: (1) JavaScript (JS) and Document Object Model (DOM) are widely employed in modern Web 2.0 applications such that the variations of user interfaces are determined dynamically through runtime changes in DOM trees[10, 11]; (2) in Web 2.0 applications, a user event may result in different Web pages due to different execution conditions and cause different changes on parameter(s) or DOM elements. Thus, the changes on Web pages are related to the conditions triggered by events and the follow-up operations on parameter(s) or DOM elements. However, the existing modeling techniques cannot capture such changes and further represent the relationship between the execution conditions and Web pages completely.

Obviously, a precise behavior model for Web applications helps to reduce testing costs and improve software quality. To date, many models have been proposed to portray the behavior of Web applications[12–14]. For instance, Schur et al.[15, 16] mined explicit behavior models of Web applications as a Finite State Automaton (FSA), where nodes denote abstract states of an application whereas transitions indicate actions that are performed by users acting in different roles to change the state. Qi et al.[11] and Haraty et al.[17] constructed a state flow graph for Web applications, where states refer to Web pages and edges between states stand for the associated events. However, these models focus on Web pages and associated events only,

• Weiwei Wang, Junxia Guo, Zheng Li, and Ruilian Zhao are with the College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China. E-mail: vivioe_wang@163.com; gjxia@mail.buct.edu.cn; lizheng@mail.buct.edu.cn; rlzhao@mail.buct.edu.cn.
∗ To whom correspondence should be addressed.
Manuscript received: 2019-07-28; accepted: 2019-08-28

neglecting the event-handlers, which process events and implement client side functions. Thus, these models omit a large amount of valuable information about Web applications such that further analyzing and testing Web applications based on the models are difficult.

Moreover, Alimadadi et al.[5,10] proposed a graph-based behavior model for Web applications, which considers the client-side event-handlers. In this model, a node is a set of the trigger event, related event-handlers, and DOM changes, while an edge signifies the execution order of two events. Mao et al.[18] defined a function behavior model on the basis of state machine for JS/Web applications, in which states correspond to internal JS functions or the calls to application programming interfaces, and transitions correspond to the trigger events. Although these models consider event handlers, they have no insight into the relationship between trigger conditions of events and Web pages. As a result, these models are insufficient to accurately represent the dynamic behavior of Web 2.0 applications. Furthermore, the test cases generated from these models tend to be infeasible because the conditions for event triggering and follow-up operations are unknown. Thus, a novel behavior model is essential to depict dynamic behaviors of Web 2.0 applications completely.

Therefore, this paper defines a novel Client-side Behavior Model (CBM) for Web 2.0 applications, and presents a model construction approach based on the user behavior traces. The user behavior trace contains Web pages, related events, the trigger conditions, and follow-up operations to depict users' dynamic behaviors in Web applications. Furthermore, a user behavior trace acquisition method for Web application was discussed in detail in our previous work[19]. This paper focuses on a CBM construction and optimization based on user behavior traces. The contributions of this work are summarized as follows:

(1) It presents a CBM to represent dynamic behaviors for Web 2.0 applications, involving not only Web pages and events, but also the trigger conditions and follow-up operations.

(2) It proposes a user behavior trace-based CBM construction approach to create and optimize CBMs for Web applications. Furthermore, this study proves that the optimized and original CBMs have the same reactions to users' operations.

(3) It implements a prototype tool to automatically generate CBMs from collected user behavior traces and evaluate our method on six Web applications according to three types of user behavior traces. The results show that our model construction method is practical and efficient.

The rest of this paper is organized as follows: Section 2 describes the background of related concepts and techniques. Section 3 introduces the definition of CBM for Web applications. Section 4 details our CBM construction method for Web applications automatically. Section 5 depicts and analyzes the experiment results on six Web applications. Section 6 reviews the related work. Finally, we conclude our work in Section 7.

2 Background

This section describes several concepts that are associated with Web applications, such as Web page, event, and user behavior trace.

2.1 Basic concepts of Web applications

A typical workflow of Web applications is that users trigger events, such as clicking buttons on the browser of the client side, to update the content of Web pages, and the triggered events send requests to a server that processes the requests and replies to the client. Then, the client updates the content of the current Web page according to the received responses. As a result, the client-side browser and Web server can exchange data with each other.

Evidently, dynamic behaviors of Web applications are activated by triggered events and processed by client-side and server-side code, resulting in Web page changes. Thus, Web pages and events are two essential factors concerning the dynamic behaviors.

2.1.1 Web pages

Traditional Web applications are based on the multi-page interface paradigm. Each Web page is bound to a unique Uniform Resource Locator (URL), which can signify a Web page. However, in Web 2.0, JS and DOM are widely used on the client side for achieving rich interactivity and responsiveness, where DOM represents a Web page as a tree structure and JS code can mutate the DOM tree at runtime seamlessly. In this manner, the changes on Web pages are determined dynamically by the changes in DOM trees[20].

More concretely, a DOM tree of a Web page holds the content and structure represented by text and HTML elements, which are rendered using the style information defined in HTML attributes or Cascading Style Sheets (CSS)[16]. Thus, the types of DOM nodes mainly include element, attribute, and text nodes, where element nodes represent HTML elements, such as hyperlinks and buttons, text nodes describe the text of element nodes, and attribute nodes define attributes for element nodes. For example, an instance of DOM is shown in Fig. 1, corresponding to the Web page in Fig. 2a, where text="manage teacher" is a text node, id="add/edit/delete", class="dylist", and name="teacher" are attribute nodes, and the others are element nodes, such as <title>, <form>, and <table>. All nodes in a DOM tree can be dynamically modified through the execution of client-side JS code, thereby changing the Web page. Thus, Web pages in modern Web applications can be represented by DOM trees.

Additionally, although DOM is a tree representation for Web pages, it has low navigation capability. Moreover, an XPath is a path expression that possesses better navigation capability to locate elements, texts, and attributes in DOM trees. For example, the element node <title> in Fig.1 can be situated by the XPath "/HTML/head/title". Furthermore, a Web page can be represented by the set of all XPaths[21] from the root node to the leaf nodes of the DOM tree. An XPath is called an equality XPath if all terms in an XPath containing only equality predicates that indicate the position of the nodes. For instance, the "edit" button can be expressed by an equality XPath "/HTML/body/div/button[2]", where the predicate of term button[2] is position()=2. In contrast to equality XPath, generalized XPath is where some of the terms in the XPath contain generalized predicates, such as "/HTML/body/div/button[position()<3]" which represents the first two buttons, i.e., the "add" and "edit" buttons in Fig. 1. The generalized XPath can be generated based on pairs of equality XPaths.

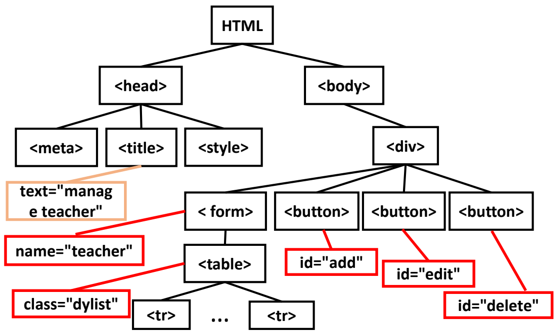
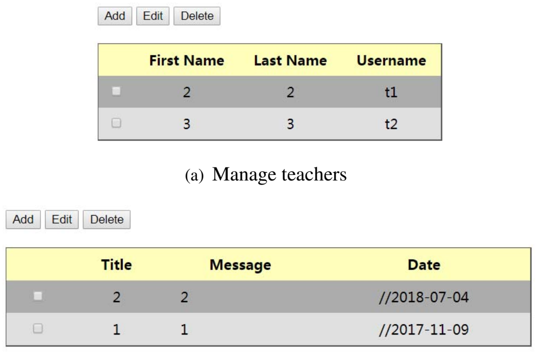


Fig. 1 A DOM instance.



	First Name	Last Name	Username
▢	2	2	t1
▢	3	3	t2

(a) Manage teachers

	Title	Message	Date
▢	2	2	//2018-07-04
▢	1	1	//2017-11-09

(b) Manage announcements

Fig. 2 Example of different Web pages in SchoolMate.

2.1.2 Event of Web application

Web applications belong to event-driven software whose behavior is activated by incoming events. The triggering of events causes the execution of client-side or server-side code, making Web application transfer from the current page to a new one. Furthermore, in modern Web applications, a set of JS functions, i.e., event handlers, are registered to handle the events, coping with user operations and implementing the client-side functions. That is to say, when an event is triggered, the executed conditions in event handlers determine which Web page (DOM) is reached and what follow-up operations are conducted on parameters or DOM elements. Thus, dynamic behaviors of modern Web applications are associated with not only Web pages and events but also trigger conditions and follow-up operations.

We take an open-source Web application called SchoolMate for example, which is a school administration system. We pay attention to the "add users" functionality in the users' management module. The corresponding HTML code is shown in Fig. 3, and the JS code is shown in Fig. 4. In this example, the event handler validate() in Fig. 4 deals with the event of clicking the "AddUser" button to verify whether the inputs of Password and Confirm-Password are equal. If the condition at Line 3 is triggered, then this request will be submitted to server-side code "./index.php" for processing. Otherwise, this page

```
<form name='adduser' action='./index.php' method='POST'>
 <table class='dynamiclist' align='center'>
  <tr class='header'>
   <th>Username</th>  <th>Password</th>
   <th>Confirm Password</th>  <th>Type</th> </tr>
  <tr class='even'>
   <td><input type='text' name='username'/></td>
   <td><input type='password' name='password'/></td>
   <td><input type='password' name='password2'/></td>
   <td> <select name='type'>
         <option value='Admin'>Admin</option>
         <option value='Teacher'>Teacher</option>
         <option value='Student'>Student</option>
         <option value='Parent'>Parent</option> </select></td></tr>
  <tr> <td> <input type='button' value='AddUser'
   onClick='validate();'> </td>
  <td>  <input type='button' value='Cancel'
   onClick='document.adduser.submit();'> </td> </tr> </table> </form>
```

Fig. 3 HTML code of AddUser.php in SchoolMate.

```
<script>
 function validate(){
   if(document.adduser.password.value ==
   document.adduser.password2.value &&
   document.adduser.password.value != ''){
    document.adduser.submit();
   }else{
    alert('Passwords do not match!');
    document.adduser.password.value = '';
    document.adduser.password2.value = '';
    document.adduser.password.select(); }}
</script>
```

Fig. 4 JavaScript code of AddUser.php in SchoolMate.

shows an alert that passwords do not match. This evidence demonstrates that events with different trigger conditions result in different Web pages and follow-up operations.

2.2 User behavior trace for Web applications

Dynamic analysis techniques are widely used to monitor the execution process of Web applications and capture the pivotal behavior information as traces[14]. Generally, a typical user behavior trace consists of sequences of events or function calls accompanied by variable values. For example, Ricca and Tonella[22] captured the trace containing information about the DOMs and event sequences. Schur et al.[15] also applied the event sequences as the traces to record the behavior of enterprise Web applications. Unlike them, the authors in Refs. [5, 10, 23] used a detailed trace that included events, DOM mutations, and all event-handler functions that were executed either directly or indirectly after an event occurred to indicate a Web application behavior.

Most of the traces record Web pages and related event sequences to indicate the behavior of Web applications. Alimadadi[10] considered event handlers but had no insight into the relationship between execution conditions and Web pages as well as the follow-up operations. However, these components are crucial for dynamic behaviors of modern Web applications. Thus, adopting a novel trace to accurately record the dynamic behavior information for Web applications is necessary.

In our previous work[19], we defined a kind of user behavior trace and designed a corresponding instrumentation code to capture users' dynamic behaviors. A trace represents a user's visit process that begins when a user from a new IP address sends a request to the server and ends when the user leaves the Web application. This concept is similar to user sessions[24], but the trace contains more information than traditional user sessions extracted from Web server logs[25]. During a user's visit process, once a user interacts with the Web application, causing the client-side or server-side code to execute, the interaction information together with relevant Web pages are recorded. The interaction sequence and Web pages constitute a trace, which is made up of multiple Web pages and interactions by a user for a time. The trace is defined as follows:

Definition 1 User behavior trace. A trace is composed of Web pages and associated user interactions with Web application, this is, trace $=<\mathrm{wp}_0, \mathrm{ia}_0, \mathrm{wp}_1, \mathrm{ia}_1, \ldots, \mathrm{wp}_{n-1}, \mathrm{ia}_{n-1}, \mathrm{wp}_n>$, where $\mathrm{WP} = \{\mathrm{wp}_0, \ldots, \mathrm{wp}_j, \ldots, \mathrm{wp}_n\}$ refers to the Web page sequence and $\mathrm{IA} = \{\mathrm{ia}_0, \ldots, \mathrm{ia}_j, \ldots, \mathrm{ia}_{n-1}\}$ stands for the user interaction sequence of a visiting user. Each element $\mathrm{wp}_j \in \mathrm{WP}$ corresponds to a two-tuple <URL, DOM>, and each element $\mathrm{ia}_j \in \mathrm{IA}$ is a triple <event, cond, oper>, where event represents the trigger event by users, cond represents the conditions executed in event handlers when handling the event, and oper is the follow-up operations on parameters or DOM elements caused by the user event callback or server messages.

In a trace, an interaction means that a user interacts with a Web application once. That is, a user triggers an event of a Web application, resulting in code execution on the client or server side. At this time, relevant information is recorded, such as the event type, relevant input parameters, element binding this event, current URL and DOM before executing the event, condition triggered by the event, follow-up operations on the parameters or DOM element, and reaching URL and DOM.

To ensure the integrity of the CBM model and make modeling process more effective, the collected traces are complemented based on three adequacy criteria, mainly considering the coverage of events, JS branches, and DOM. Meanwhile, an optimal minimal trace set was generated in the previous work[19]. Thus, this paper focuses on how to leverage the minimal trace set to construct the CBM model for Web applications.

3 Client-Side Behavior Model Definition for Web Applications

Web pages and events are two primary components that reflect the dynamic behavior of Web applications. An event execution may transfer to different Web pages due to various execution conditions and cause changes in parameter(s) or DOM elements. That is, the changes of Web pages are related to the conditions triggered by events and follow-up operations on the parameter(s) or

DOM elements. Thus, besides Web pages and events, the trigger conditions and follow-up operations are also essential to depict the dynamic behaviors of Web applications. However, the existing models can only express partial information. To adequately represent the dynamic behavior of Web applications, we propose a novel CBM model based on Extended Finite State Machine (EFSM) to correspond the information related to behaviors with states and transitions of the model.

3.1 Client-side behavior model

For Web applications, the new client-side behavior model is defined as follows:

Definition 2 CBM. The client-side behavior model named CBM is defined as a 4-tuple (S, I, O, T), where S is a finite set of states, I is a finite set of input declarations, O is a finite set of output declarations, and T is a finite set of transitions. Each member of S is represented as a URL and corresponding DOM, each member of I expresses an input parameter, each member of O represents an output parameter, and each member of T signifies a migration from one state to another, remarking the change of URL or DOM. Furthermore, a transition t is denoted by a 5-tuple <src, event, cond, act, trgt>, where src(t) and trgt(t) represent the source and target state of transition t, respectively; event(t) signifies the event triggered on current source state by users; cond(t) describes the triggered conditions in associated event handler functions; and act(t) indicates the follow-up operations on the parameters or DOM elements caused by user event callbacks or server responses. Specifically, an event(t) can be further expressed as event$(t$, InputList), meaning event occurs with a list of input parameters, and an act(t) can be further described as act$(t$, paraList), implying action implements with a list of input or output parameters.

A transition occurs when its event is triggered and condition is satisfied. This is, for transition t, if its event(t) is triggered and cond(t) is met, then act(t) is performed, and the state transfers from src(t) to trgt(t). The event, cond, and act parts of a transition t are optional. Essentially, the CBM of Web applications is an EFSM model.

3.2 State representation in CBM

In modern Web applications, the alterations of user interfaces are determined by changes in DOMs. Thus, DOMs can be used to represent Web pages. However, if a DOM is mapped directly to a state of the CBM, it may lead to state space explosion because minor changes in DOMs may result in an expansion of states. Thus, we consider an appropriate abstraction of concrete DOMs, which does not affect the semantics of the CBM model, that is, preserving the behavior of Web applications.

At present, most DOMs abstraction methods focus on the structure or content of DOMs, namely, abstracting DOMs by extracting their element nodes or content nodes[23, 26]. However, a pair of DOMs with the same structure are likely to indicate different functionalities and cannot be treated as one. For example, Figs. 2a and 2b show two different function pages of SchoolMate: One is the teacher management page and the other is the announcement management page. These two pages have the same DOM structure as Fig. 1, but they cannot be regarded as one state in the CBM. Thus, the methods that abstract the DOM structures are prone to fail in distinguishing this type of DOMs, making the model of Web applications inaccurate. The methods that abstract DOM contents can differentiate these two pages, but they may result in over-differentiation, that is, identifying the same page into different ones. As shown in Fig. 2a, if the data in the teacher form of two Web pages vary, these two pages will be misjudged as two states. Therefore, the existing abstraction methods are not suitable for representing DOMs.

Thus, to abstract DOMs accurately, this paper presents a new abstraction method that considers the DOM structure and attributes. In a DOM tree, attribute nodes define attributes to customize the style for elements, such as id, name, and class, while element nodes are used to bind events. As a result, they are closely related to events and Web pages. In other words, attribute nodes are indispensable to distinguish DOMs that imply different behaviors. For example, as shown in Fig. 2a, the form name of Fig. 2a is “teacher” and that of Fig. 2b is “announcements”. Although these two Web pages have the same DOM structure, their attributes indicate that they are used to implement distinct functions. Thus, these two pages can be distinguished by their attributes.

As shown from the preceding analysis, if only the structure of DOMs is considered, then inaccuracy may occur in identifying different DOMs, thereby breaking the primitive semantics of the model. Considering that attribute nodes in DOMs are essential to distinguish different behaviors, we define an abstract DOM to represent the state of CBMs. As discussed in Section 2.1.1, DOM can be expressed by a set of equality XPaths corresponding to all leaf nodes of the DOM tree. An abstract DOM is defined in the following.

Definition 3 Abstract DOM. An abstract DOM consists of a set of 2-tuples $< \text{ele}_i, (\text{attr}_0, \text{attr}_1, \ldots, \text{attr}_n) >$ corresponding to all leaf nodes, where ele_i is the i-th leaf node of DOM expressed by equality XPaths and $(\text{attr}_0, \text{attr}_1, \ldots, \text{attr}_n)$ are the associated attributes.

For example, in Fig.1, a 2-tuple instance for the edit button is <"/HTML/body/div/button[2]", <id="edit">>. Another 2-tuple for the element title is <"/HTML/head/title",$<\emptyset>$>.

3.3 Transition representation in CBM

In a Web application, some particular events may only be triggered with a sequence of user operations. For instance, in the "add users" module of SchoolMate, as shown in Fig. 3, if an administrator adds a new user, he/she needs to input the username and password first, select the role of the new user, and click the AddUser button to complete adding a new account. In this process, a single user operation can not cause the execution of event handlers and lead to the change of states. As a result, the sequence of user operations that induce the execution of client-side or server-side code should be combined into one event in transitions of the CBM. The sequence of user operations of adding a new user can be expressed by one event as <click, Xpath:(//input[@value="AddUser"])(username, password, password2, type)>, where the username, password, password2, and type are regarded as the external inputs for the event of clicking the AddUser button.

Furthermore, the JS conditions in associated event handlers triggered by events are expressed as conditions of transitions, and the follow-up operations on the parameters or DOM elements are signified as actions of transitions.

3.4 Illustrative example

We take SchoolMate as an example, which includes 12 functional modules and 4 user roles. The functional modules allow users to perform school-related tasks, including user management, school information updating, class management, registration, and so on. Four user roles are administrator, teacher, parent, and student, respectively. Now, if the administrator wants to add a new user account, firstly, he/she has to enter the login page (i.e., S_0) in Fig. 5, and then logs into the system as an administrator by inputting his/her username and password and clicking the login button to submit

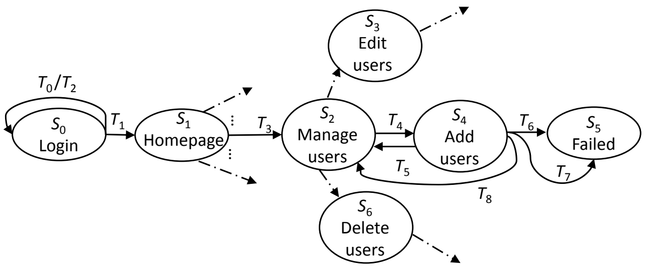


Fig. 5 Partial behavior model of SchoolMate.

the login request to the server. If the username and password are not empty and have passed the server-side verification, then the administrator is allowed to access his/her personal homepage (S_1). Otherwise, this system prompts an error message and returns to the login page. On the personal homepage, the administrator selects the user management module by clicking "manage users" link, and enters into the "manage users" page (S_2), where three operations, namely, add, edit, and delete a user account, are provided. At this time, the administrator chooses the add operation by clicking the add button, and reaches the "add users" page (S_4). On this page, a new user account can be added by inputting the username, password, and role, and sending "add a user" request to the server by clicking the "add user" button. In particular, if the passwords entered twice are inconsistent or the password is empty, then the system prompts an error message and enters the failed page (S_5). Otherwise, the server deals with the "add a user" request. Specifically, if the server determines that the new user account is invalid, then the system enters the failed page (S_5). Otherwise, the new user account is added successfully, and the system returns to "manage users" page (S_2).

The partial behavior model of user management module is depicted in Fig. 5, which mainly illustrates the process by which an administrator adds a new user account. For ease of understanding, the states are labeled with corresponding Web pages, and the transitions are labeled with corresponding migrations between pages. For example, if the administrator does not input his/her username and password but clicks the login button at the login page (S_0), then the "invalid params" is alerted and the page remains at S_0. That is, the source and target states of this transition are all S_0. Moreover, inputting the username and password and clicking the login button can be taken as event with no input, which mean that input empty can be regarded as condition, and alerting the "invalid params" can be seen as action. As a result, the transition, denoted by T_0, can be expressed as <S_0; click, Xpath: //input [@value="Login"](username, password); username. value== "||password. value=="; alert("in-valid params"); S_0>. The details of other transitions are shown in Table 1.

4 Trace-Based CBM Modeling Approach for Web Applications

In the previous section, we have defined CBMs to articulate the dynamic behaviors for Web applications, including Web pages, events, and trigger conditions as well as follow-up operations on the parameters or DOM elements. In this section, we discuss the CBM construction and optimization approach based on user behavior traces. The framework of our CBM modeling approach is shown in Fig. 6.

The overall process consists of the following two stages:

• An original client-side behavior model is constructed based on user behavior traces, including obtaining states and transitions of CBM from traces, identifying and merging identical ones, and finding and removing noise ones.

• The original CBM is further optimized by identifying equivalent states and transitions and merging them to refine the model. Furthermore, this optimized CBM is proved to be equivalent to the original CBM, namely, these two models have the same reactions to users' operations.

In the following, the CBM construction and optimization approaches are discussed in detail.

4.1 Original CBM construction based on traces

According to the definition of CBM, a state $s \in S$ represents the current URL and corresponding DOM during the execution of Web applications. A transition $t \in T$ indicates an interaction that enables migration from a state to another. In contrast to the CBM, a trace consists of Web pages WP and interaction information IA, in which each wp ∈ WP is made up of the URL and DOM, and each ia ∈ IA involves the trigger event and conditions triggered by the event and follow-up operations. Thus, a Web page wp = <URL, DOM> in a trace can be regarded as a state of the CBM. However, as discussed in Section 3.2, an abstract DOM is more appropriate for representing the state of the CBM. Thus, in the CBM, <URL, abstract DOM> is taken as a state instead of <URL, DOM>. Furthermore, an interaction ia = <event, cond, oper> is considered as the event[cond]/act of a transition, and two associated states <URL, abstract DOM> with wp_i and wp_{i+1} are taken as the states before and after the event event(ia) is triggered. In other words, for a sub-sequence $<\text{wp}_i$, ia, $\text{wp}_{i+1}>$ of a trace, the corresponding transition t can be created, namely, event[cond]/act of transition t corresponds to the event, cond, and oper of interaction ia, and src(t) and trgt(t) refer to the <URL, abstract DOM> of wp_i and wp_{i+1}, respectively.

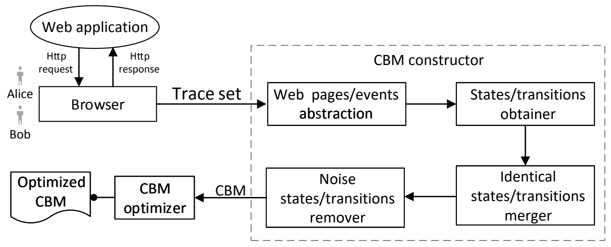

Web application
Http request
Http response
Alice
Bob
Browser
Trace set
CBM constructor
Web pages/events abstraction
States/transitions obtainer
Identical states/transitions merger
Noise states/transitions remover
CBM
CBM optimizer
Optimized CBM

Fig. 6 Framework of CBM construction and optimization.

Table 1 Details of transitions on the behavior model of SchoolMate.

Trans	src	event	cond	act	trgt
T_0	S_0	click, Xpath://input[@value='Login'] (username, password)	username.value==″|| password.value==″	alert('invalid params')	S_0
T_1	S_0	click, Xpath://input[@value='Login'] (username, password)	username.value!=″&& password.value!=″	document.login.value=1	S_1
T_2	S_0	click, Xpath://input[@value='Login'] (username, password)	username.value!=″&& password.value!=″	document.login.value=1	S_0
T_3	S_1	click, link=Users	–	–	S_2
T_4	S_2	click, Xpath://input[@value='Add']	–	document.users.submit()	S_4
T_5	S_4	click, Xpath://input[@value='Cancel']	–	document.adduser.submit()	S_2
T_6	S_4	click, Xpath:(//input[@value='AddUser']) (username, password, password2, type)	password.value!=password2. value||password.value == ″)	alert('Passwords do not match!')	S_5
T_7	S_4	click, Xpath:(//input[@value='AddUser']) (username, password, password2, type)	password.value==password2. value&&password.value !=″	document.adduser.submit()	S_5
T_8	S_4	click, Xpath:(//input[@value='AddUser']) (username, password, password2, type)	password.value==password2. value&&password.value !=″	document.adduser.submit()	S_2

However, in the acquired traces, the same Web pages and interactions often exist in different traces because users may visit the same Web pages or trigger the same events. Even in only one trace, the same Web pages or interactions may exist because a user may access one Web page or trigger one event more than once. These situations result in duplications in states and transitions of the CBM to be constructed. Thus, the identical states and transitions need to be merged to build the CBM for a Web application from traces.

Moreover, the noise information, such as navigation links, out-of-site links, and advertisements, usually appear when users access a Web application. Traces collected inevitably contain noise information, which leads to noise states and transitions in the CBM built. Thus, noise states and transitions should be further eliminated.

According to the preceding analysis, to construct CBMs for Web applications, first model components, namely, states and transitions, are obtained from user behavior traces, and then identical states and transitions are identified and merged. Meanwhile, to avoid interference from noise information, the noise states and transitions are further identified and removed. The details on how to construct the CBM are discussed in the following.

4.1.1 Obtaining model components from a trace set

States and transitions are fundamental components of the CBM model. To construct a CBM for a Web application, we first have to obtain states and transitions from a trace set. That is, for a trace $=$ $< \text{wp}_0, \text{ia}_0, \text{wp}_1, \text{ia}_1, \ldots, \text{wp}_{n-1}, \text{ia}_{n-1}, \text{wp}_n>$ in the trace set, we can take each Web page wp_j as a state s_j in CBM, denoted by s_j =<URL, abstract DOM>, where the abstract DOM is a compact representation of the DOM tree. Each user interaction ia_j can be associated with the event, condition, and action of transition t_j in CBM, i.e, event(t_j), cond(t_j), and act(t_j). The src(t_j) and trgt(t_j) of transition t_j can be tied to the corresponding states of s_j and s_{j+1}, respectively. As a result, a state list S and a transition list T can be acquired when all traces in a trace set are traversed.

4.1.2 Identifying and merging identical states and transitions

Identical states and transitions exist within one trace or among traces. Therefore, how to distinguish identical states and transitions is a critical problem in the CBM construction.

A state of CBM is expressed by <URL, abstract DOM>, so the similarity or difference between states can be determined by their URLs and abstract DOMs. Distinguishing whether two URLs are equal by string comparison is easy with the help of distance metrics, such as edit distance. However, for abstract DOMs, comparison techniques used on traditional DOMs are unsuitable because they mainly focus on structure or content similarity between DOMs[27–29], whereas abstract DOMs are made up of element and attribute nodes. Thus, the similarity between abstract DOMs should be estimated from the structures and attributes of DOM trees.

The structural similarity is reflected in the similarity of the element nodes in DOMs. Bag of XPath model is widely used to measure the structural similarity between DOMs[21]. Thus, this model is also adopted in evaluating the structural similarity between abstract DOMs as follows:

$$\text{Struct\_sim}(\text{Do}_i, \text{Do}_j) = \frac{e+s}{n+m-e} \tag{1}$$

where Do_i and Do_j represent a pair of abstract DOMs; n and m are the number of element nodes expressed by XPaths, i.e., the number of XPaths in Do_i and Do_j, respectively; e is the number of common XPaths of Do_i and Do_j; and s is the number of XPaths that do not exactly match in Do_i and Do_j but are subsumed by at least one of the generalized XPaths of the other DOMs.

Considering only the structural similarity between two abstract DOMs is insufficient, as shown in Fig. 2. Thus, whether their attributes are similar should also be considered. The more elements with the same attributes, the more similar the two DOMs are. The attribute similarity can be measured as follows:

$$\text{Attribute\_sim}(\text{Do}_i, \text{Do}_j) = \frac{e - a_{\text{diff}}}{e} \tag{2}$$

where a_{diff} indicates the number of element nodes in e whose attributes (e.g., id, class, and name) are different.

In general, Web pages with different URLs are definitely different; thus, given two states s_1 and s_2, whether they are the same is firstly determined by their URLs. That is, if the distance between s_1.URL and s_2.URL is not equal to 0, then States s_1 and s_2 are regarded as different. Otherwise, their abstract DOMs are compared to decide whether States s_1 and s_2 are equal or not. In this case, considering the structure of DOM represents all elements of the page while attributes are appended to elements, we firstly estimate the structural similarity between two abstract DOMs. That is, if the structures of two abstract DOMs are similar, namely, the structural similarity Struct_sim(s_1.dom,

s_2.dom) computed by Eq. (1) exceeds a preset threshold, then the similarity of their attributes is considered to further decide whether these two states are identical or not. In other words, if the attributes of the two abstract DOMs are similar, which means that the Attribute_sim(s_1.dom, s_2.dom) calculated by Eq. (2) exceeds the preset threshold, then these two states are regarded as identical. Otherwise, they are different. The pseudo code of judging whether a pair of states are identical is shown in Algorithm 1.

According to the definition of CBM, a transition $t \in T$ is made up of $\mathrm{src}(t), \mathrm{trgt}(t)$, and associated event( )[cond]/act, which is labeled in terms of $\mathrm{lbl}(t)$. To identify identical transitions in transition list T, we define identical transitions as follows:

Definition 4 Identical transitions. Given transitions t_1=<$\mathrm{src}(t_1), \mathrm{lbl}(t_1), \mathrm{trgt}(t_1)$> and t_2=<$\mathrm{src}(t_2), \mathrm{lbl}(t_2)$, $\mathrm{trgt}(t_2)$>, we say that:

Transition t_1 is identical to t_2 iff $\mathrm{src}(t_1)$ and $\mathrm{src}(t_2)$, $\mathrm{trgt}(t_1)$ and $\mathrm{trgt}(t_2)$, and $\mathrm{lbl}(t_1)$ and $\mathrm{lbl}(t_2)$ are identical. Here, $\mathrm{lbl}(t_1)$ and $\mathrm{lbl}(t_2)$ being identical means that every member of them is the same.

Algorithm 1 Similarity comparison of states

```
Input: State $s_1, s_2$
Output: true or false
1: Procedure Similarity ($s_1, s_2$) begin
2: NodeList1, NodeList2 = ∅
3: SimilarStruct = thrs1; SimilarAttribute = thrs2; thrs1 and
   thrs2 are preset threshold
4: count = 0; SameAttr = 0;
5: NodeList1 = NodeXpath($s_1$.dom);
6: NodeList2 = NodeXpath($s_2$.dom);
7: struct_sim = Bag of Xpath(NodeList1, NodeList2)
8: if ($s_1$.URL == $s_2$.URL & struct_similarity > SimilarStruct)
   then
9:     for ($i = 0; i <$ NodeList1.length; $i++$) do
10:        if (NodeList1[$i$] in NodeList2) then
11:            count++;
12:            if ( Attribute(NodeList[$i$], NodeList2)) then
13:                SameAttr++;
14:            end if
15:        end if
16:    end for
17:    if (SameAttr/count > SimilarAttribute) then
18:        return true;
19:    else
20:        return false;
21:    end if
22: else
23:    return false;
24: end if
```

Identifying identical transitions by comparing each component of transitions is easy. In other words, any two transitions are judged to be different as long as one component of them is different.

Therefore, based on the state list S and transition list T obtained in Section 4.1.1, the identification and merging of identical states and transitions are roughly divided into following steps. Firstly, for the states in list S, any pairs of states are compared according to Algorithm 1. If two states are identical, then one of them is preserved by randomly removing one. The transition whose source (target) state is the one removed is modified, namely, the source (target) state of this transition is replaced by the reserved one. As a result, a new state set S' without duplicate states is obtained, and a corresponding transitions list T' is formed. Then, identical transitions are removed from the transition list T'. As a result, a transition set T'' without duplicate transitions is obtained.

4.1.3 Identifying and eliminating noise states and transitions

Noise information causes redundant states and transitions in the CBMs built and reduces the efficiency of automated testing. Thus, this subsection aims to find and remove noise states and transitions in the CBMs.

The noise information in Web applications generally includes navigation links, out-of-site links, advertisements, and so on. The navigation links can be identified by Vision-based Page Segmentation (VIPS) algorithm, which is extensively used in segmenting elements of a Web page into different parts[30, 31]. In further detail, the VIPS algorithm divides the homepage of a Web application into different visual blocks. When the proportion of hyperlinks in a particular block exceeds a preset threshold, the hyperlinks in this block are determined as navigation links. Without loss of generality, a Web application possesses more than one navigation link, and each navigation link corresponds to a navigation event. As a result, for a navigation link nal, the transitions whose events belong to the navigation event of nal can be obtained, named as T_{nal}. That is, the transitions in T_{nal} are associated with the same hyperlink event, and can be regarded as redundant except one. Then, to further reduce the size of the CBM built, we only reserve the transition in T_{nal} that is the nearest to the start state of the CBM, which generally refers to the entrance of a Web application, and other transitions in T_{nal} are eliminated from the transition set T''.

Furthermore, multiple out-of-site links or advertisements usually exist in a Web application, each out-of-site link ties to an event, and so do the advertisements. Consequently, for an out-of-site link outl, the transitions whose events pertain to the event of outl can be obtained, called T_{outl}. The transitions in T_{outl} are associated with the event of outl and can be treated as redundant because they are irrelevant to the state space of a Web application. Thus, we directly eliminate those transitions in T_{outl} from the transition set T'' to achieve CBM refinement. Furthermore, the transitions T_{ads} related to the events of the advertisements ads are processed in the same manner as T_{outl}.

After processing all the transitions involved in noise information, the final transition set T_1 is formed. Then, the states that do not appear in the T_1 are deleted from state set S', thereby forming the final state set S_1. Thus far, the original CBM has been built from the state set S_1 and transition set T_1.

4.1.4 Algorithm of building CBMs for Web applications

The core idea of the CBM construction is to create a highly accurate model on the basis of user behavior traces for Web applications. According to the preceding discussions, the pseudo-code is summarized in Algorithm 2.

Algorithm 2 Construct Web application's CBM model

```
Input: a trace set traces
Output: CBM<$S_1, T_1$>//The Web application's behavior model
 1: function BuildModel()
 2:     $S, T$ = ObtainST(traces);
 3:     $S'$ = MergeIdenticalStates($S$);
 4:     $T'$ = CorrectTrans($T, S'$);
 5:     $T''$ = MergeIdenticalTrans($T'$);
 6:     Nav, OutAd = IdentifyNoiseVIPS($S'$);
 7:     $T_1$ = RemoveNoiseTrans($T''$, Nav, OutAd);
 8:     $S_1$ = RemoveNoiseStates($T_1, S'$);
 9:     Return $S_1, T_1$;
10: end function
11: function MergeIdenticalStates ($S$)
12:     set $S' = \varnothing$;
13:     boolean flag[0, ..., ($S$.length − 1)];
14:     for ($i = 0; i < S$.length; $i++$) do
15:         flag[$i$] = false;
16:     end for
17:     for ($i = 0; i < S$.length; $i++$) do
18:         if (!flag[$i$]) then
19:             $S'$.add($S[i]$);
20:             for ($j = i + 1; j < S$.length; $j++$) do
21:                 if (!flag[$j$]) then
22:                     if (Similarity($S[i], S[j]$)) then
23:                         flag[$j$] = true;
24:                     end if
25:                 end if
26:             end for
27:         end if
28:     end for
29:     Return $S'$;
30: end function
31: function CorrectTrans ($T, S'$)
32:     for $i = 0; i < T$.length; $i++$ do
33:         for $j = 0; j < S'$.length; $j++$ do
34:             if (match($T[i]$.src, $S'[j]$)) then
35:                 $T[i]$.src = $S'[j]$
36:             end if
37:             if (match($T[i]$.tgt, $S'[j]$)) then
38:                 $T[i]$.trgt = $S'[j]$
39:             end if
40:         end for
41:     end for
42:     Return $T'$;
43: end function
```

In Algorithm 2, the function BuildModel() (Lines 1–10) shows the basic steps of the CBM construction. In further detail, first, a state list S and a transition list T are obtained with the help of the function ObtainST(traces) by traversing the user behavior traces set and matching Web pages and interactions in traces with the states and transitions of the CBM. Then, the same states in state list S are identified and merged by comparing their similarity and combining the same ones to form a new state set S' by means of the function MergeIdenticalStates(S). Meanwhile, transitions whose source or target state is merged are handled by the function CorrectTrans(T, S'). That is, their original source or target states are corrected with the states in S'. As a result, a new list T' is obtained. Next, the function MergeIdenticalTrans(T') is applied to identify and merge identical transitions in T' to form a transition set T'', by comparing each component of transitions. Thereafter, the function IdentifyNoiseVIPS(S') is used to distinguish noise information according to state set S' with the aid of the VIPS algorithm. Associated noise transitions in T'' are then processed by the function RemoveNoiseTrans(Nav, OutAd), where Nav and OutAd are the events belonging to navigation and out-of-site links as well as advertisements. Thus far, a reduced set of transitions T_1 is obtained. Finally, the states that do not appear in T_1 are eliminated from the state set S' by the function RemoveNoiseStates(T_1, S'), obtaining a reduced state set S_1.

Concretely, function MergeIdenticalStates(S) (Lines 11–30), firstly, initializes a flag false for each state in the state list S to indicate that all states are different (Lines 13–16). Then, for each state $S[i]$, if its flag is false (Lines 18–27), which means that no state is identical to $S[i]$, then the state is added to the new state set S' (Lines 18 and 19) and compared one by one with the states behind it whose flag is also false (Lines 20–26). If a subsequent state is recognized as the same as $S[i]$, then its flag is marked as true (Lines 22–24). If the flag of $S[i]$ is true, then it is discarded, and the next state is handled. After the preceding processing, a state set S' without duplicate states is produced (Line 29).

In CorrectTrans(T, S') (Lines 31–43), for each transition $T[i]$ in the transition list T, the source state and target state are compared with each new state $S'[j]$ in S'. If the source state $T[i]$.src or target state $T[i]$.trgt matches the state $S'[j]$, then it is replaced with $S'[j]$. As a result, a new transition list T' with states merged in S' is generated.

The implementation of the other functions is simple and similar to the functions described. Thus, the details of them are not explained in this paper.

4.2 CBM optimization

Equivalent states and transitions exist in the CBM constructed by the preceding process. Thus, further refining the original CBM by merging equivalent states and transitions is necessary. This section discusses how to identify and merge equivalent states and transitions to obtain an optimized CBM. To illustrate the existence of equivalent states and transitions in CBMs, we introduce the following two common scenarios in Web applications.

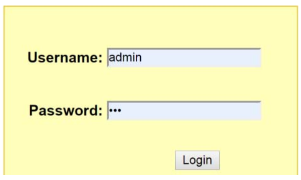
Situation 1: A user in different Web pages may trigger identical events that may result in identical subsequent Web pages.

It is a common case that users may trigger the same event from different Web pages and reach the same target page. For example, users can submit a login request from the initial login page in Fig. 7a or an error login page in Fig. 7b caused by an invalid username or password, and arrive at the same successful index page shown in Fig. 7c.

Situation 2: A user on a Web page may trigger the same event with the different values of parameters that may lead to different Web pages.

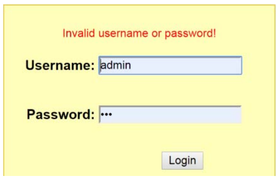
For instance, a user in the index page, as shown in Fig. 8a, may trigger the event of viewing details on different rows, namely, associated with different values, which may result in different Web pages, as shown in Figs. 8b and 8c. Although the Web pages shown in Figs. 8b and 8c are different, they have the same incoming events and derive from the same Web page.

For these two situations, the corresponding CBMs are depicted in Fig. 9. CBM-1 refers to Situation 1, where the details of transition t_1 are src(t_1)=s_1, lbl(t_1)=event[cond]/act, and trgt(t_1)=s_3, and transition t_2 means that src(t_2)=s_2, lbl(t_2)=event[cond]/act, and trgt(t_2)=s_3, respectively. The label and target state of transitions t_1 and t_2 are the same, namely,



Username:	admin
Password:	···

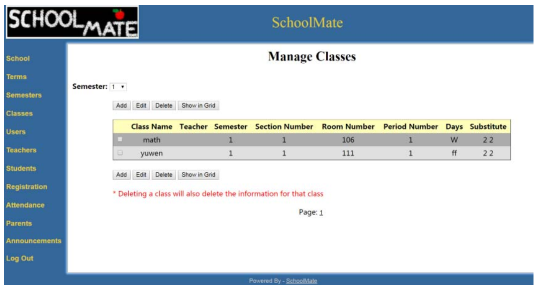
(a) Login page



Invalid username or password!

Username:	admin
Password:	···

(b) Error login page



Manage Classes

Semester: 1

	Class Name	Teacher	Semester	Section Number	Room Number	Period Number	Days	Substitute
☐	math		1	1	106	1	W	2 2
☐	yuwen		1	1	111	1	ff	2 2

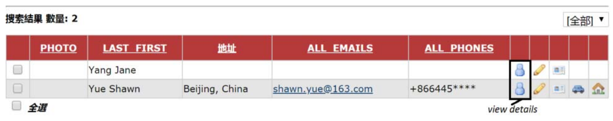
* Deleting a class will also delete the information for that class

Page: 1

Powered By - SchoolMate

(c) Reaching index page

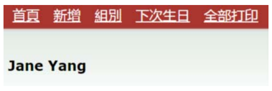
Fig. 7 Example of Situation 1 in SchoolMate.



搜索結果 數量: 2

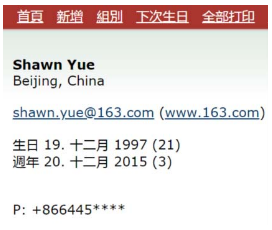
	PHOTO	LAST FIRST	地址	ALL EMAILS	ALL PHONES
☐		Yang Jane			
☐		Yue Shawn	Beijing, China	shawn.yue@163.com	+866445****

(a) Index page



Jane Yang

(b) Details of Jane



Shawn Yue
Beijing, China

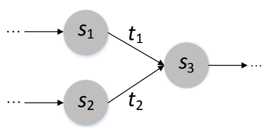
shawn.yue@163.com (www.163.com)

生日 19. 十二月 1997 (21)
週年 20. 十二月 2015 (3)

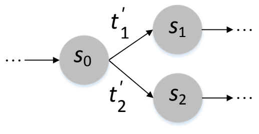
P: +866445****

(b) Details of Jane (c) Details of Shawn

Fig. 8 Example of Situation 2 in Addressbook.



(a) CBM-1



(b) CBM-2

Fig. 9 Corresponding CBMs for Situations 1 and 2.

lbl(t_1)=lbl(t_2) and trgt(t_1)=trgt(t_2). CBM-2 refers to Situation 2, where transition t_1' represents that src(t_1') = s_0, lbl(t_1') = event$'$[cond$'$]/act$'$, and trgt(t_1') = s_1, and transition t_2' is that src(t_2')=s_0, lbl(t_2')=event$'$[cond$'$]/act$'$, and trgt(t_2')=s_2. The labels and source state of transitions t_1' and t_2' are the same, namely, lbl(t_1')=lbl(t_2') and src(t_1')=src(t_2').

Intuitively, if merging states s_1 and s_2 do not affect the behavior of the CBM constructed, then these two states should be merged. Otherwise, they cannot be combined.

To further refine CBMs, we define the following state equivalence and transition equivalence in CBMs.

4.2.1 Definition of equivalent states and transitions

Definition 5 R-equivalent states. We assume the outgoing transitions from state s_1 is $\{t_{10}, t_{11}, \ldots, t_{1n}\}$ and that from s_2 is $\{t_{20}, t_{21}, \ldots, t_{2m}\}$. The states s_1 and s_2 are R-equivalent, if for each transition t_{1i} $(0 \leqslant i \leqslant n)$ from s_1, a corresponding transition t_{2j} $(0 \leqslant j \leqslant m)$ from s_2 exists such that their label lbl and target state trgt are equal, i.e., lbl(t_{1i}) = lbl(t_{2j}) and trgt(t_{1i}) = trgt(t_{2j}), and vice versa.

Based on Definition 5, the states s_1 and s_2 of CBM-1 in Fig. 9a are R-equivalent.

Definition 6 L-equivalent states. We suppose that the incoming transitions to state s_1 is $\{t_{10}, t_{11}, \ldots, t_{1n}\}$ and that to s_2 is $\{t_{20}, t_{21}, \ldots, t_{2m}\}$. The states s_1 and s_2 are L-equivalent, if for each transition t_{1i} $(0 \leqslant i \leqslant n)$ pointing to s_1, a corresponding transition t_{2j} $(0 \leqslant j \leqslant m)$ pointing to s_2 exists such that their source state src and label lbl are the same, i.e., src(t_{1i}) = src(t_{2j}) and lbl(t_{1i}) = lbl(t_{2j}), and vice versa.

Based on Definition 6, the states s_1 and s_2 of CBM-2 in Fig. 9b are L-equivalent.

Considering the definition of R-equivalent and L-equivalent states, we define the equivalent states and transitions for the CBMs of Web applications.

Definition 7 Equivalent states. Given states s_1 and s_2 of a CBM, s_1 and s_2 are equivalent, denoted by equ$<s_1, s_2>$, if they satisfy either of the following two conditions:

- For each outgoing transition from s_1, a corresponding transition from s_2 meets the requirement that their label lbl and target state trgt are equal, and vice versa.
- For each transition pointing to s_1, a corresponding transition pointing to s_2 satisfies the requirement that their source state src and label lbl are the same, and vice versa.

Definition 8 Equivalent transitions. Given transitions $t_1 = <$src(t_1), lbl(t_1), trgt(t_1)$>$ and t_2=$<$src(t_2), lbl(t_2), trgt(t_2)$>$, we say that t_1 is equivalent to t_2 iff src(t_1) and src(t_2), trgt(t_1) and trgt(t_2), and lbl(t_1) and lbl(t_2) are identical, respectively.

For an EFSM model, Androutsopoulos et al.[32] thought that a bisimulation can be formed by merging its R-equivalent and L-equivalent states. In other words, the EFSM model after merging equivalent states is equivalent in behavior to the original EFSM model. In fact, our CBM is an EFSM model, and L-equivalent states as well as R-equivalent states exist in our CBM. Therefore, the CBM can be further optimized by identifying and merging L-equivalent and R-equivalent states.

4.2.2 Identifying and merging equivalent states and transitions in CBMs

The equivalent states are distinguished by comparing their incoming and outgoing transitions. If for each incoming or outgoing transition of state s_i in a CBM, there exists a corresponding transition pointing to s_j or deriving from s_j, then states s_i and s_j are judged to be L-equivalent or R-equivalent. Thus, they can be merged by reserving one randomly, such as s_j. At the same time, the transitions associated with the deleted state s_i are modified with the reserved s_j. Equivalent transitions are identified by comparing each component of transitions and merged by reserving one randomly.

Further details of identifying and merging equivalent states and transitions in a CBM are shown in Algorithm 3. Given an original CBM$<S_1, T_1>$ with state set S_1 and transition set T_1, firstly, we consider the state set S_1. For each state $S_1[i] \in S_1$, all outgoing transitions whose source state is $S_1[i]$ are extracted from the transition set T_1 and recorded into a two-dimensional (2D) array outgoTran, where outgoTran$[i][j] = T_1[k]$ means that the j-th outgoing transition of $S_1[i]$ is $T_1[k]$ and outgoTran$[i]$ indicates all outgoing transitions from $S_1[i]$. Likewise, all incoming transitions whose target state is $S_1[i]$ are drawn out from the transition set T_1 and stored into another 2D array incomTran, where incomTran$[i][j]$=$T_1[k]$ means that the j-th incoming transition of $S_1[i]$ is $T_1[k]$ and incomTran$[i]$ presents all incoming transitions of $S_1[i]$.

Then, for any two states $S_1[i]$ and $S_1[j]$ in S_1, whether they are equivalent is determined. In further detail, firstly, consider the R-equivalent, if for each transition $T_1[p]$ outgoing from $S_1[i]$ in outgoTran$[i]$,

Algorithm 3 Optimize CBM model

```
Input: the original CBM<$S_1, T_1$>
Output: the optimized CBM<$S, T$>
1: Procedure OptimizeModel() begin
2: Set $S, T = \varnothing$
3: Array outgoTran[][], incomTran[][]; //store all outgoing & incoming transitions for each state
4: flag = false //indicating whether there are equivalent states
5: do
6:     flag = false;
7:     outgoTran = ExtractOutgoTrans($S_1, T_1$);
8:     incomTran = ExtractIncomTrans($S_1, T_1$);
9:     for ($i = 0; i < S_1$.length; $i++$) do
10:        for ($j = i + 1; j < S_1$.length; $j++$) do
           //Optimize model by merging R-equivalent states
11:            if (equ(outgoTran[$i$], outgoTran[$j$])) then
12:                flag = true;
13:                MergeState($S_1[i], S_1[j]$);
14:                ReplaceState($S_1[i], S_1[j], T_1$);
15:            end if//Optimize model by merging L-equivalent states
16:            if (!flag) then
17:                if (equ(incomTran[$i$], incomTran[$j$])) then
18:                    flag = true;
19:                    MergeState($S_1[i], S_1[j]$);
20:                    ReplaceState($S_1[i], S_1[j], T_1$);
21:                end if
22:            end if
23:        end for
24:    end for
25:    for ($i = 0; i < T_1$.length; $i++$) do //merge equivalent transitions
26:        for ($j = i + 1; j < T_1$.length; $j++$) do
27:            if (equTran($T_1[i], T_1[j]$)) then
28:                MergeTran($T_1[i], T_1[j]$);
29:            end if
30:        end for
31:    end for
32: while flag = false
33: $S = S_1, T = T_1$;
34: return CBM <$S, T$>
```

there is a corresponding transition $T_1[q]$ outgoing from $S_1[j]$ in outgoTran[j] whose label lbl and target state trgt are identical to those of $T_1[p]$, then $S_1[i]$ and $S_1[j]$ are taken as R-equivalent, implemented by the function equ(outgoTran[i], outgoTran[j]). In a similar manner, the L-equivalent is considered, namely, if for each incoming transition $T_1[u]$ of $S_1[i]$ in incomTran[i], there is a corresponding transition $T_1[v]$ pointing to $S_1[j]$ in incomTran[j], whose label lbl and source state src are the same as those of $T_1[u]$, then $S_1[i]$ and $S_1[j]$ are regarded as L-equivalent, which are implemented by the function equ(incomTran[i], incomTran[j]). Otherwise, these two states are independent.

After the equivalent states are identified, they are merged into one state. In other words, if a pair of states $S[i]$ and $S[j]$ are judged to be L-equivalent, then one of them, such as $S[j]$, is deleted from the state set S_1. At the same time, the transitions in T_1, whose source or target state is the deleted state $S[j]$, are modified. That is, the deleted state $S[j]$ on these transitions is replaced with the other reserved $S[i]$. Similarly, if a pair of states are judged to be R-equivalent, then the states and transitions are handled in the same manner. The process is implemented by the function MergeState and ReplaceState.

Then, equivalent transitions are considered. That is, for each pair of transitions $T[i]$ and $T[j]$ in transition set T_1, if the two transitions are determined to be equivalent according to Definition 7, namely, equTran($T[i], T[j]$) returns true, then one of $T[i]$ and $T[j]$ is removed from T_1. As merging equivalent transitions may cause the emergence of new equivalent states, the preceding process is repeated until no equivalent states exist.

4.3 Equivalence proof of CBMs before and after optimization

To demonstrate that the original and optimized CBMs have the same behaviors, the main theorem homomorphism of model projection[32] is applied to prove it. That is, if there is a homomorphic mapping between the original and optimized CBMs, then the two CBMs are deemed to have the same behaviors, namely, they react to user operations in the same ways.

The homomorphic mapping between two CBMs is defined as follows:

Definition 9 Homomorphic mapping of models. For the original CBM$=(S, I, O, T)$ and optimized CBM$'=(S', I', O', T')$ of a Web application, a homomorphic mapping h from CBM to CBM$'$ is defined as a pair of functions $h=(h_S : S \rightarrow S', h_T : T \rightarrow T')$ that preserves the structure of CBM as well as CBM$'$ and satisfies the following constraints:

$\forall t \in T$ in the CBM whose src(t)$=s_1$, trgt(t)$=s_2$, lbl(t)$=$event[con]/act, and $s_1, s_2 \in S$, CBM$'$ has the corresponding states and transitions that meet $h_T(t) = t' \in T'$; $h_S(s_1)=s'_1$, $h_S(s_2)=s'_2 \in S'$; and src(t')$=s'_1$, trgt(t')$=s'_2$, and lbl(t')$=$event[con]/act hold.

In the following, we prove that the original CBM is homomorphic to the optimized CBM$'$, namely, a homomorphic mapping from CBM to CBM$'$ exists.

As discussed in the preceding, the optimized CBM is obtained by merging equivalent states and equivalent transitions of CBM. In other words, states from set S that satisfy L-equivalent or R-equivalent conditions are merged, and then equivalent transitions from set T are processed. Thus, to prove that CBM is homomorphic to CBM$'$, we only have to prove the following two propositions.

Proposition 1 Given an original CBM$=(S, I, O, T)$ and an optimized CBM$'=(S', I', O', T')$ produced by merging R-equivalent states of CBM, CBM is homomorphic to CBM$'$.

Proof CBM$'$ is the slice of CBM produced by merging R-equivalent states. We show the presence of a model homomorphic mapping, h, from CBM to CBM$'$.

We assume that the CBM which includes states s_1 and s_2 has identical outgoing transitions. That is, if state s_1 has an outgoing transition t_1 whose $\text{src}(t_1)=s_1$, $\text{lbl}(t_1)=\text{event[con]/act}$, and $\text{trgt}(t_1)=x$, then state s_2 also has an outgoing transition t_2 whose $\text{src}(t_2)=s_2$, $\text{lbl}(t_2)=\text{event[con]/act}$, and $\text{trgt}(t_2)=x$. According to Definition 5, states s_1 and s_2 are R-equivalent. Thus, CBM$'$ can be obtained by merging s_1 and s_2. Then, CBM$'$ is identical to CBM except that s_1 and s_2 are replaced by a single state s. The corresponding transitions t_1 and t_2 in CBM are replaced by a single transition t with $\text{src}(t)=s$, $\text{lbl}(t)=\text{event[con]/act}$, and $\text{trgt}(t)=x$, where s is either s_1 or s_2. As a result, CBM$'$ is obtained. Thus, a mapping $h=(h_S, h_T)$ from CBM to CBM$'$ exists, where $h_S : S\rightarrow S'$ means that s_1 and s_2 in S are mapped to s in S' by h_S, $h_T : T\rightarrow T'$ means that t_1 and t_2 in T are mapped to t in T' by h_T, and transitions targeting s_1 or s_2 in T map the transitions targeting s in T'. That is, CBM is clearly homomorphic to CBM$'$.

Proposition 2 Given a CBM$'=(S', I', O', T')$ and its slice CBM$''=(S'', I'', O'', T'')$, which are produced by merging L-equivalent states, CBM$'$ is homomorphic to CBM$''$.

Proof CBM$''$ is the slice of CBM$'$ produced by merging L-equivalent states. We show that a model homomorphic mapping, h', exists from CBM$'$ to CBM$''$.

We assume that CBM$'$ includes states s'_1 and s'_2, which have identical incoming transitions. That is, if state s_1 has an incoming transition t'_1 whose $\text{src}(t'_1)=x$, $\text{lbl}(t'_1)=\text{event}'[\text{con}']/\text{act}'$, and $\text{trgt}(t'_1)=s'_1$, then state s'_2 also has an incoming transition t'_2 whose $\text{src}(t'_2)=x$, $\text{lbl}(t'_2)=\text{event}'[\text{con}']/\text{act}'$, and $\text{trgt}(t'_2)=s'_2$. According to Definition 6, states s'_1 and s'_2 are L-equivalent. Thus, CBM$''$ can be obtained by merging s'_1 and s'_2. Then, CBM$''$ is identical to CBM$'$ except that s'_1 and s'_2 are replaced by a single state s'. The corresponding transitions t'_1 and t'_2 in CBM$'$ as above are replaced in CBM$''$ by a single transition t' with $\text{src}(t')=x$, $\text{lbl}(t')=\text{event}'[\text{con}']/\text{act}'$, and $\text{trgt}(t')=s'$, where s' is either s'_1 or s'_2. Besides, the transitions deriving from s'_1 or s'_2 are replaced by transitions deriving from s'. As a result, CBM$''$ is obtained. Thus, a mapping $h'=(h'_S, h'_T)$ exists from CBM$'$ to CBM$''$, where h'_S: $S' \rightarrow S''$ means that s'_1 and s'_2 in S' are mapped to s' in S'' by h'_S, h'_T: $T' \rightarrow T''$ means that t'_1 and t'_2 in T are mapped to t' in T'' by h'_T, and transitions deriving from s'_1 or s'_2 in T' map the transitions from s' in T''. That is, CBM$'$ is homomorphic to CBM$''$.

According to the proofs, a homomorphic mapping h exists from CBM to CBM$'$, and a homomorphic mapping h' occurs from CBM$'$ to CBM$''$. Obviously, homomorphism satisfies transitivity. Thus, CBM is homomorphic to CBM$''$. So, we can conclude that a homomorphic relation exists between the original CBM and the optimized CBM, that is, the two models have the same semantics. Equivalent transitions merging essentially remove duplicate transitions in the CBM; thus, the behavior of the model is not affected. Thus, the optimized CBMs are valid to represent the dynamic behavior of Web applications.

5 Empirical Study

This section focuses on the evaluation metrics and research questions, information about subject programs and experimental design, experiment results and analysis, and threats to validity.

5.1 Evaluation metrics and research questions

To verify the validity of the CBM construction method, we conducted a series of experiments on six commonly used Web applications, and the effectiveness and efficiency are evaluated on the basis of these experiments. The effectiveness is reflected in whether a CBM can accurately represent the dynamic behavior of a Web application, and the efficiency is embodied in the time cost of CBM construction. As we know, modern Web applications are mostly event-driven, and these events are responded by a set of client-side JS functions called event handlers. The execution of event handlers causes Web pages (DOMs) transferred to other pages. Furthermore, events with different trigger conditions of event handlers may result in different Web pages and follow-up operations. As a result, the client-side behaviors of Web applications are closely related to events, trigger conditions (JS branches), and DOMs.

Thus, the accuracy and integrity of CBMs in representing the client-side behavior of Web applications can be manifested by the coverage of events, JS branches, and DOMs.

Therefore, three metrics are raised to measure the effectiveness of our CBM construction method. They are the coverage of events, JS branches, and DOMs. For events and JS branches, we can obtain all of them in client-side code through static source code analysis. For DOMs, they are dynamically created in the execution of client-side and server-side code, so it is difficult to distinguish where DOMs are generated. Therefore, we regard the DOMs that appear in user behavior traces as all the DOMs. The details of these three metrics are described as follows:

• **Metric 1:** Events Coverage (EC). EC measures the proportion of the events that appear in the CBM constructed to all events in the client-side code of the Web application. For a CBM of Web application, EC can be computed as follows:

$$\mathrm{EC} = \frac{|\mathrm{M.events}|}{|\mathrm{W.events}|} \tag{3}$$

where |M.events| indicates the number of all events in the CBM model and |W.events| represents the number of all events in the Web application.

• **Metric 2:** JS branch Coverage (JC). JC estimates the proportion of the JS branches that occur in the CBM, i.e., conditions on transitions, to all JS branches in the client-side code of the Web application. For a CBM of the Web application, JC can be calculated as follows:

$$\mathrm{JC} = \frac{|\mathrm{M.JSbranches}|}{|\mathrm{W.JSbranches}|} \tag{4}$$

where |M.JSbranches| implies the number of all JS branches in the CBM and |W.JSbranches| indicates the number of all JS branches in the Web application.

• **Metric 3:** DOM Coverage (DC). DC evaluates the proportion of the DOMs that appear in the CBM to all the DOMs that occur in the user behavior traces. For a CBM of Web application, DC can be counted as follows:

$$\mathrm{DC} = \frac{|\mathrm{M.DOMs}|}{|\mathrm{W.DOMs}|} \tag{5}$$

where |M.DOMs| represents the number of all DOMs in the CBM and |W.DOMs| depicts the number of all DOMs in the user behavior traces with respect to a Web application.

As mentioned, our CBM construction is based on user behavior traces. In the previous study[19], we discussed how to obtain the trace sets according to the Web applications under test. When acquiring the trace sets, three adequacy criteria (all events coverage, all JS branches coverage, and maximum DOM coverage) are used to guide the minimal trace set generation. Under the premise of satisfying the adequacy criteria, we have obtained three minimal user behavior trace sets for each Web application, which are event-trace-set, JS-trace-set, and DOM-trace-set, respectively. The three trace sets are used to construct CBMs to further evaluate the influence of different trace sets on the CBM models built.

We implemented a prototype to assess the CBM construction approach reported in this paper. Moreover, the following research questions are raised and investigated:

RQ1. Is our CBM construction method effective for Web applications?

RQ2. Is the optimized CBM more effective than the original CBM?

RQ3. How do different trace sets affect the CBM models built? Which trace set is the most suitable for modeling a Web application?

RQ4. How efficient is our approach to CBM construction?

5.2 Experimental subjects and design

To address the given research questions, we selected five commonly used open-source Web applications from https://sourceforge.net and a laboratory management system developed by our group, called DBLab, as the experimental subjects. These applications are implemented in PHP or JSP language. Table 2 provides a brief description of these subjects, including the programming language, size of Web applications, Lines Of Code (LOC), number of events (#Events), number of

Table 2 Web applications used in the study.

App name	Version	Language	Size (KB)	LOC	#Events	#JS_branches	Functional description
SchoolMate	1.5.4	PHP	365	8181	159	161	School admin system
Addressbook	8.2.5.2	PHP	3799	47 481	50	27	Addressbook management system
Webchess	1.0.0	PHP	468	4722	28	374	Online chess game
FAQForge	1.3.2	PHP	227	1712	20	–	FAQ management tool
JCart	1.3.0	PHP	123	1188	7	13	Online shopping
DBLab	–	JSP	166	10 526	41	72	Laboratory management system
phpaaCMS	0.0.5	PHP	1659	15 949	65	105	Article management system

JS branches (#JS_branches), and a functional description of the Web applications. Although the total JS branches in Webchess are 374, a large number of them (363) are used to make rules for playing chess. The execution of this kind of JS branches does not affect the states space. So these branches are filtered when obtaining the trace set and building the CBM. In addition, subject FAQForge is a simple PHP Web application without JS branches.

The CBM modeling method contains two phases: One is to build corresponding CBMs for Web applications based on their user behavior trace sets, called original CBMs, and the other is to optimize the original CBMs by merging equivalent states and transitions, called optimized CBMs. To evaluate the effectiveness of the two phases, we construct these two CBM models on the basis of three trace sets, that is, event-trace-set, JS-trace-set, and DOM-trace-set, respectively, and further analyze the difference between the original and optimized CBMs for each Web application.

In summary, for a Web application, we use three trace sets to build corresponding CBM models, and then optimize the CBMs built. As a result, six CBM models are established for each Web application, which are listed in Table 3.

All experiments are performed on Windows platform (Windows10-64 bit) with CPU i5-2470 and 8 GB of memory. The programming language is Python.

Table 3 CBM models constructed by different trace sets.

Trace set	Original CBM	Optimized CBM
Event-trace-set	$\text{CBM}_{\text{EvOrg}}$	$\text{CBM}_{\text{EvOpt}}$
JS-trace-set	$\text{CBM}_{\text{JsOrg}}$	$\text{CBM}_{\text{JsOpt}}$
DOM-trace-set	$\text{CBM}_{\text{DoOrg}}$	$\text{CBM}_{\text{DoOpt}}$

5.3 Experimental results and analysis

5.3.1 Results for RQ1

To assess the CBM construction method, we use three user behavior trace sets to build the CBMs for Web applications under test. The related statistics on user behavior traces, model components after merging the same components, noises removed, and original CBM built are summarized in Table 5. For example, for SchoolMate, the number in parentheses of event-trace-set is 34, which means that 34 traces are obtained to cover all events of SchoolMate; the number of states and transitions <#s, #t> that are included in the event-trace-set is 538 and 504, respectively; the number <#s, #t> is decreased to 65 and 352 after merging identical states and transitions; and the number <#s, #t> is further reduced to 65 and 161 after removing noise states and transitions, respectively. As a result, the original CBM consists of 65 states and 161 transitions according to event-trace-set of SchoolMate. Moreover, the CBM of SchoolMate login by administrators is shown in Fig. 10.

Table 5 Statistics on traces, merging information, noises removed, and original CBM built.

Web app	Trace set	<#s, #t> of trace	<#s, #t> merge same	<#s, #t> remove noise	<#s, #t> of CBMs
SchoolMate	event-trace-set (34)	<538, 504>	<65, 352>	<65, 161 >	$\text{CBM}_{\text{EvOrg}}$ <65, 161>
	JS-trace-set (50)	<707, 657 >	<70, 352>	<70, 186>	$\text{CBM}_{\text{JsOrg}}$ <70, 186>
	DOM-trace-set (53)	<722, 669 >	<73, 352>	<73, 203>	$\text{CBM}_{\text{DoOrg}}$ <73, 203>
Addressbook	event-trace-set (10)	<118, 108>	< 34, 84>	<34, 76>	$\text{CBM}_{\text{EvOrg}}$ <34, 76>
	JS-trace-set (11)	<122, 111>	<34, 85>	<34, 78>	$\text{CBM}_{\text{JsOrg}}$ < 34, 78>
	DOM-trace-set (11)	<123, 112>	<34, 84 >	<34, 80 >	$\text{CBM}_{\text{DoOrg}}$ <34, 80 >
Webchess	event-trace-set (10)	< 77, 67>	< 9, 33>	<9, 28 >	$\text{CBM}_{\text{EvOrg}}$ <9, 28 >
	JS-trace-set (12)	<92, 80 >	<10, 33 >	<10, 32>	$\text{CBM}_{\text{JsOrg}}$ <10, 32>
	DOM-trace-set (11)	<84, 73>	<10, 33 >	<10, 31>	$\text{CBM}_{\text{DoOrg}}$ <10, 31>
FAQForge	event-trace-set (5)	<65, 60>	<8, 35>	<8, 21>	$\text{CBM}_{\text{EvOrg}}$ <8, 21>
	JS-trace-set (–)	–	–	–	–
	DOM-trace-set (6)	<71, 65>	< 9, 36 >	<9, 23 >	$\text{CBM}_{\text{DoOrg}}$ <9, 23>
JCart	event-trace-set (2)	< 27, 25 >	<3, 13 >	<3, 10>	$\text{CBM}_{\text{EvOrg}}$ <4, 10>
	JS-trace-set (3)	< 32, 29>	<4, 15>	<4, 14 >	$\text{CBM}_{\text{JsOrg}}$ <4, 14>
	DOM-trace-set (3)	< 32, 29>	<4, 15>	<4, 14 >	$\text{CBM}_{\text{DoOrg}}$ <4, 14 >
DBLab	event-trace-set (13)	<117, 104>	<32, 80 >	<32, 67 >	$\text{CBM}_{\text{EvOrg}}$ <32, 67 >
	JS-trace-set (24)	<198, 174 >	<43, 123>	<43, 106>	$\text{CBM}_{\text{JsOrg}}$ <43, 106 >
	DOM-trace-set (40)	<276, 236>	<59, 131 >	<59, 108 >	$\text{CBM}_{\text{DoOrg}}$ <59, 108 >
Average	event-trace-set (12)	<157, 145>	<25, 99>	<25, 60>	$\text{CBM}_{\text{EvOrg}}$ <25, 60 >
	JS-trace-set (20)	<230, 210>	<32, 122>	<32, 83>	$\text{CBM}_{\text{JsOrg}}$ <32, 83>
	DOM-trace-set (21)	<218, 197>	<31, 108>	<31, 76>	$\text{CBM}_{\text{DoOrg}}$ <31, 76>

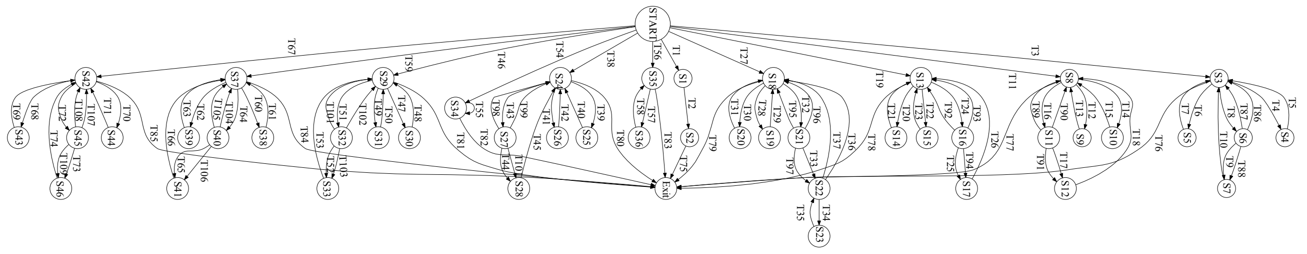


Fig. 10 CBM of SchoolMate login by administrators.

Evidently, we observe a large number of identical states and transitions as well as redundant ones in the user behavior trace sets.

Moreover, the three metrics identified (EC, JC, and DC) are applied to estimate the effectiveness of the CBMs constructed. The results are shown in Table 5. To reveal the coverage of CBMs more intuitively, we use histograms, as shown in Fig. 11, to depict the coverage

Table 5 Evaluation results of CBMs of Web applications.

Web app	<#s, #t> of CBMs	EC	JC	DC
SchoolMate	$\text{CBM}_{\text{EvOrg}}$ <65, 161>	159/159=100%	114/161=70.81%	65/73=89.04%
	$\text{CBM}_{\text{JsOrg}}$ <70, 186>	159/159=100%	161/161=100%	70/73=95.89%
	$\text{CBM}_{\text{DoOrg}}$ <73, 203>	159/159=100%	161/161=100%	73/73=100%
Addressbook	$\text{CBM}_{\text{EvOrg}}$ <34, 76>	50/50=100%	23/27=85.19%	34/34=100%
	$\text{CBM}_{\text{JsOrg}}$ < 34, 78>	50/50=100%	27/27=100%	34/34=100%
	$\text{CBM}_{\text{DoOrg}}$ <34, 80 >	50/50=100%	27/27=100%	34/34=100%
Webchess	$\text{CBM}_{\text{EvOrg}}$ <9, 28 >	28/28=100%	2/11=18.18%	9/10=90%
	$\text{CBM}_{\text{JsOrg}}$ <10, 32>	28/28=100%	11/11=100%	10/10=100%
	$\text{CBM}_{\text{DoOrg}}$ <10, 31>	28/28=100%	11/11=100%	10/10=100%
FAQForge	$\text{CBM}_{\text{EvOrg}}$ <8, 21>	20/20=100%	–	8/9=88.89%
	$\text{CBM}_{\text{JsOrg}}$ (–)	–	–	–
	$\text{CBM}_{\text{DoOrg}}$ <9, 23>	20/20=100%	–	9/9=100%
JCart	$\text{CBM}_{\text{EvOrg}}$ <4, 10>	7/7=100%	7/13=53.85%	4/4=100%
	$\text{CBM}_{\text{JsOrg}}$ <4, 14>	7/7=100%	13/13=100%	4/4=100%
	$\text{CBM}_{\text{DoOrg}}$ <4, 10>	7/7=100%	13/13=100%	4/4=100%
DBLab	$\text{CBM}_{\text{EvOrg}}$ <32, 67>	41/41=100%	35/72=48.61%	32/59=54.24%
	$\text{CBM}_{\text{JsOrg}}$ <43, 106>	41/41=100%	72/72=100%	43/59=72.88%
	$\text{CBM}_{\text{DoOrg}}$ <59, 108>	41/41=100%	72/72=100%	59/59=100%
Average	$\text{CBM}_{\text{EvOrg}}$ <25, 60>	100%	56.23%	87.03%
	$\text{CBM}_{\text{JsOrg}}$ <32, 83>	100%	100%	93.75%
	$\text{CBM}_{\text{DoOrg}}$ <31,76>	100%	100%	100%

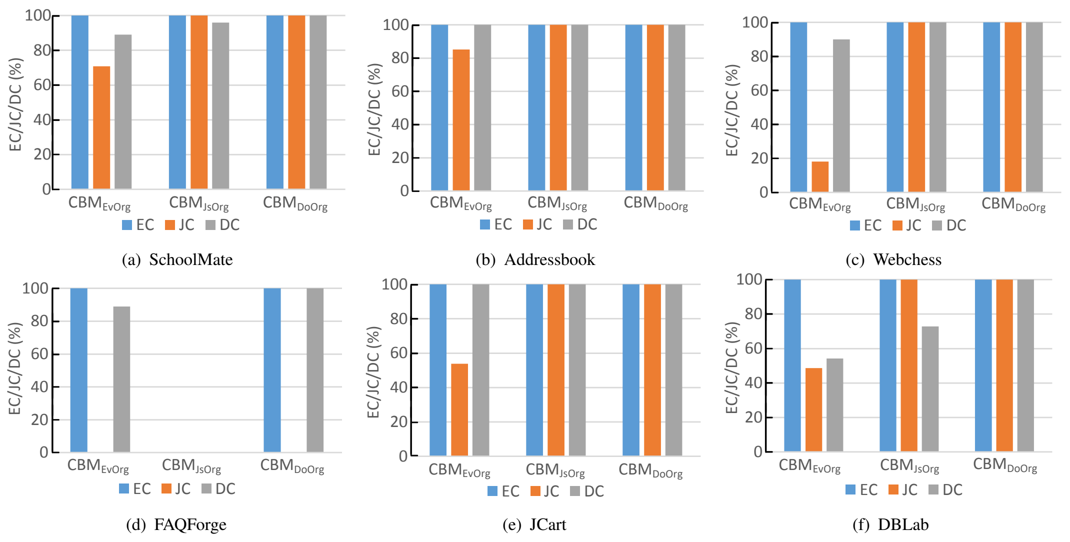


Fig. 11 EC, JC, and DC of CBM by using different trace sets.

of CBMs constructed with respect to different trace sets for six Web applications. As shown in Table 5 and Fig. 11, the EC of the original CBMs built based on the event-trace-set, i.e., $\text{CBM}_{\text{EvOrg}}$ model, is 100%; the JC of the original CBMs associated with the JS-trace-set, i.e., $\text{CBM}_{\text{JsOrg}}$ model, is 100%; and the DC of the original CBMs related to DOM-trace-set, i.e., $\text{CBM}_{\text{DoOrg}}$ model, is 100% for each Web application. These results indicate that our CBM construction method for Web applications is effective, and all events, JS branches, and DOMs can be preserved.

5.3.2 Results for RQ2

To evaluate the effectiveness of the CBM optimization method (RQ2), we compared the number of states and transitions of the original CBMs constructed and optimized CBMs for each Web application. The results are exhibited in Table 6. Furthermore, the difference between these two CBM models are analyzed. According to Situation 1 discussed in Section 4.2, a user on different Web pages triggers identical events and reaches the same follow-up pages. Then, the preceding states and corresponding transitions are merged into one in the optimized CBM. Similarly, as described in Situation 2, a user on the same Web page triggers the same event with different values but results in different follow-up pages. Then, corresponding transitions and follow-up states are combined in the optimized CBM. Thus, the optimized CBM has fewer states and transitions than the original CBM.

The coverage metrics are also estimated for the optimized CBMs. The results are described in Table 7. As Table 7 shows, the EC of the optimized CBMs built based on event-trace-set of Web applications, i.e., $\text{CBM}_{\text{EvOpt}}$, is 100%. Similarly, $\text{CBM}_{\text{JsOpt}}$ reaches 100% JC, based on the corresponding JS-trace-set. This indicates that the optimization does not affect the coverage of events and JS branches of the CBMs built. However, the merging of equivalent states results in that the states on the optimized CBMs cannot match the DOMs of Web applications one by one. Therefore, DC is not evaluated for the optimized CBM.

5.3.3 Results for RQ3

We recall that RQ3 is how different trace sets affect the CBMs and which trace set is the most appropriate for modeling Web applications. This question can be answered by analyzing the difference among the CBMs constructed by these three trace sets, namely, event-trace-set, JS-trace-set, and DOM-trace-set. Tables 5 – 7 detail the differences in the coverage and scales of the CBMs built. The more traces we use, the larger the scale of the CBM, and the higher the coverage of the CBM.

Moreover, for a Web application, the CBMs built based on three different trace sets are analyzed and

Table 6 Comparison of original and optimized CBMs.

Web app	<#s, #t> of $\text{CBMs}_{*\text{Org}}$	[#s, #t] of $\text{CBMs}_{*\text{Opt}}$
SchoolMate	$\text{CBM}_{\text{EvOrg}}$ <65, 161>	$\text{CBM}_{\text{EvOpt}}$ <64, 160>
	$\text{CBM}_{\text{JsOrg}}$ <70, 186>	$\text{CBM}_{\text{JsOpt}}$ <69, 185>
	$\text{CBM}_{\text{DoOrg}}$ <73, 203>	$\text{CBM}_{\text{DoOpt}}$ <72, 202>
Addressbook	$\text{CBM}_{\text{EvOrg}}$ <34, 76>	$\text{CBM}_{\text{EvOpt}}$ <31, 73>
	$\text{CBM}_{\text{JsOrg}}$ <34, 78>	$\text{CBM}_{\text{JsOpt}}$ <31, 75>
	$\text{CBM}_{\text{DoOrg}}$ <34, 80>	$\text{CBM}_{\text{DoOpt}}$ <31, 77>
Webchess	$\text{CBM}_{\text{EvOrg}}$ <9, 28 >	$\text{CBM}_{\text{EvOpt}}$ <9, 28>
	$\text{CBM}_{\text{JsOrg}}$ <10, 32>	$\text{CBM}_{\text{JsOpt}}$ <10, 32>
	$\text{CBM}_{\text{DoOrg}}$ <10, 31>	$\text{CBM}_{\text{DoOpt}}$ <10, 31>
FAQForge	$\text{CBM}_{\text{EvOrg}}$ <8, 21>	$\text{CBM}_{\text{EvOpt}}$ <8, 21>
	$\text{CBM}_{\text{JsOrg}}$ (–)	$\text{CBM}_{\text{JsOpt}}$ (–)
	$\text{CBM}_{\text{DoOrg}}$ <9, 23>	$\text{CBM}_{\text{DoOpt}}$ <9, 23>
JCart	$\text{CBM}_{\text{EvOrg}}$ <4, 10>	$\text{CBM}_{\text{EvOpt}}$ <4, 10>
	$\text{CBM}_{\text{JsOrg}}$ <4, 14>	$\text{CBM}_{\text{JsOpt}}$ <4, 14>
	$\text{CBM}_{\text{DoOrg}}$ <4, 14>	$\text{CBM}_{\text{DoOpt}}$ <4, 14>
DBLab	$\text{CBM}_{\text{EvOrg}}$ <32, 67>	$\text{CBM}_{\text{EvOpt}}$ <30, 66>
	$\text{CBM}_{\text{JsOrg}}$ <43, 106>	$\text{CBM}_{\text{JsOpt}}$ <41, 105>
	$\text{CBM}_{\text{DoOrg}}$ <59, 108>	$\text{CBM}_{\text{DoOpt}}$ <57, 107>
Average	$\text{CBM}_{\text{EvOrg}}$ <25, 60>	$\text{CBM}_{\text{EvOpt}}$ <24, 60>
	$\text{CBM}_{\text{JsOrg}}$ <32, 83>	$\text{CBM}_{\text{JsOpt}}$ <31, 82>
	$\text{CBM}_{\text{DoOrg}}$ <31, 76>	$\text{CBM}_{\text{DoOpt}}$ <30, 76>

Table 7 Evaluation results of optimized CBMs.

Web app	<#s, #t> of $\text{CBMs}_{*\text{Opt}}$	EC	JC
SchoolMate	$\text{CBM}_{\text{EvOpt}}$ <64,160>	**159/159=100%**	114/161=70.81%
	$\text{CBM}_{\text{JsOpt}}$ <69, 185>	159/159=100%	**161/161=100%**
	$\text{CBM}_{\text{DoOpt}}$ <72, 202>	159/159=100%	161/161=100%
Addressbook	$\text{CBM}_{\mathit{EvOpt}}$ <31, 73>	**50/50=100%**	23/27=85.19%
	$\text{CBM}_{\text{JsOpt}}$ <31, 75>	50/50=100%	**27/27=100%**
	$\text{CBM}_{\text{DoOpt}}$ <31, 77>	50/50=100%	27/27=100%
Webchess	$\text{CBM}_{\text{EvOpt}}$ <9, 27>	**28/28=100%**	2/11=18.18%
	$\text{CBM}_{\text{JsOpt}}$ <10, 32>	28/28=100%	**11/11=100%**
	$\text{CBM}_{\text{DoOpt}}$ <10, 31>	28/28=100%	11/11=100%
FAQForge	$\text{CBM}_{\text{EvOpt}}$ <8, 21>	**20/20=100%**	–
	$\text{CBM}_{\text{JsOpt}}$ (–)		
	$\text{CBM}_{\text{DoOpt}}$ <9, 23>	20/20=100%	–
JCart	$\text{CBM}_{\text{EvOpt}}$ <4, 10>	**7/7=100%**	7/13=53.85%
	$\text{CBM}_{\text{JsOpt}}$ <4, 14>	7/7=100%	**13/13=100%**
	$\text{CBM}_{\text{DoOpt}}$ <4, 10>	7/7=100%	13/13=100%
DBLab	$\text{CBM}_{\text{EvOpt}}$ <30, 66>	**41/41=100%**	35/72=48.61%
	$\text{CBM}_{\text{JsOpt}}$ <41, 105>	41/41=100%	**72/72=100%**
	$\text{CBM}_{\text{DoOpt}}$ <57, 167>	41/41=100%	72/72=100%
Average	$\text{CBM}_{\text{EvOpt}}$ <24, 60>	100%	56.23%
	$\text{CBM}_{\text{JsOpt}}$ <31, 82>	100%	100%

compared. Tables 5 and 7 show that $\text{CBM}_{\text{Ev}*}$ models constructed by the event-trace-set can cover all the events but not all the JS branches. $\text{CBM}_{\text{Js}*}$ models built by the JS-trace-set can cover all the events and JS branches but not all DOMs. $\text{CBM}_{\text{DoOrg}}$ models built by the DOM-trace-set cover all the events, JS branches, and DOMs. Here, "*" stands for original or optimized. However, as all DOMs are collected from dynamically executed traces and the traces from different users are changeable, total DOMs are undecided, which further causes uncertainty in the $\text{CBM}_{\text{Do}*}$ built. Furthermore, the changes of DOMs derive from the execution of client-side or server-side code, and it is difficult to distinguish where the changes are from. Thus, the DOM-trace-set is unsuitable for constructing CBMs for Web applications. In contrast to DOMs, the events and JS branches are obtained by static analysis from the client-side source code. They are deterministic and corresponding traces can indicate the client-side behaviors. Thus, the $\text{CBM}_{\text{Ev}*}$ and $\text{CBM}_{\text{Js}*}$ built can represent the dynamic behavior of Web applications demonstrably. As shown in Tables 5 and 7, $\text{CBM}_{\text{Js}*}$ built can cover all the events and JS branches with respect to a Web application. Therefore, we can infer that the JS-trace-set is the most appropriate for modeling a modern Web application. For Web applications without JS branches, the event-trace-set is suitable for modeling.

5.3.4 Results for RQ4

The efficiency of our CBM modelling method can be measured by the time cost. As discussed in the preceding, the JS-trace-set is the most appropriate for modelling a Web application. Thus, we build the CBMs according to the JS-trace-set and record the time cost of building the original CBM and optimizing the CBM model. For FAQForge, as no JS branches exist, the event-trace-set is used for modeling. The results are shown in Table 8. Building CBMs involves the DOM comparison; thus, much more time is needed than that of optimizing models, which is only concerned with the identification and merging of equivalent states and transitions. The maximum time cost is 12.7820 s for the six Web applications. Thus, we can say that the time cost is acceptable.

Table 8 Time cost of our methods.

(s)

Web app	<#s, #t> of CBMs	Original CBM	Optimized CBM	Total
SchoolMate	<69,185>	12.5681	0.2139	12.7820
Addressbook	<31,75>	1.1315	0.0949	1.2268
Webchess	<10, 32>	1.1865	0.0089	1.1954
FAQForge	<8, 21>	0.9612	0.0039	0.9651
JCart	<4,14>	0.5003	0.0015	0.5018
DBLab	<41, 105>	2.3442	0.1295	2.5738
Average		3.1153	0.0754	3.2075

5.4 Threats to validity

Like any empirical study, our evaluation is subject to threats to validity. The major threat is the representativeness of the selected subjects which are five open-source Web applications from https://sourceforge.net and a laboratory management system developed by our group. All these may affect the evaluation of the proposed approach. However, five open-source Web applications are popular and widely used in Web testing[26, 33, 34]. Therefore, we believe that this threat is limited.

In addition, CBMs are constructed according to corresponding user behavior trace sets. Thus, another threat relates to the integrity of trace sets. In the experiments, the total events and JS branches with respect to the Web applications under test are collected by static analysis. To reduce this threat, the uncovered events and JS branches are carefully inspected, and corresponding traces are complemented manually to cover all events and JS branches. Furthermore, the automatic generation of reasonable traces is under study.

6 Related Work

6.1 Models for Web applications

Web applications are widely used to offer various services for users[35]. A critical problem is how to ensure the security and reliability of Web applications[36]. The graph and model-based testing approach is an effective way to derive test cases based on the models constructed[7]. Thus, creating a model for Web application is essential, and precise models can support program understanding and testing. At present, many models are used to characterize the behavior of Web applications. For example, Ricca and Tonella[22] created a Finite State Machine (FSM) model in which nodes represent Web objects (Web pages, forms, frames, and others) and edges represent relationships and interactions among the objects (include, submit, split, link, and others). Andrews et al.[37] proposed the FSM with constraints for Web applications. Logical Web pages are represented by nodes in the FSM, and the transitions among logical Web pages are described by edges.

These models focus on traditional Web applications which are based on the synchronous request-response protocol. However, asynchronous requests appear in modern Web applications, which make user interfaces active and responsive due to the use of JS and DOM.

For modern Web applications, Marchetto and Tonella[9] proposed an FSM to model the behaviors of Web applications. The DOM of Web pages manipulated by the JS code is abstracted into a state of an FSM model, and the callback executions triggered by asynchronous messages from the Web server are associated with state transitions. The authors in Refs. [20, 38] described a technique for crawling Ajax-based applications through automatic dynamic analysis of user interface state changes in Web browsers. Mirshokraie et al.[39] used a state-flow graph that captured the explored dynamic DOM states and event-based transitions between them. Qi et al.[11] also constructed a state flow graph for Web applications in which a state referred to a user interface state and an edge between states was labeled with the type of an event. The authors in Ref. [2] introduced an event-flow graph for Web applications in which a node represents the Web page object and an edge represents the event that causes certain parts of the page to change. Schur et al.[15] presented an FSA in which the nodes denoted abstract individual states of the Web application and were numbered in the order that they were detected by ProCrawl, whereas the transitions denoted actions that changed the state and were performed by users acting in different roles.

In addition to the FSMs for modeling Web applications, a few approaches mined extended FSMs by combining the FSMs with data rule inference or data rule computed based on the input data. For example, the authors in Refs. [5, 6, 10, 23] proposed a graph-based behavioral model, in which a node is an information set that includes the triggered event, related event-handler function, and impact on the dynamic DOM state, whereas an edge signifies a progression of time connecting nodes. However, because a node integrates a large volume of information, it is difficult to use in generating test cases from this model. Schur et al.[16] mined explicit behavior models of Web applications as an EFSM, and a tool named PROCRAWL was given to create the model. The nodes in this method denoted abstract individual states of the Web application, whereas the transitions denoted actions that changed the states and the conditions of the transitions were computed based on the input data.

Although these models consider the new features (i.e., dynamic, event-driven, and asynchronous nature), they neither represent the parameter(s) or DOM element changes nor the relation between Web pages and execution conditions of event handlers. Furthermore, our CBM construction is based on user behavior traces while other modeling techniques, such as PROCRAWL, are based on crawlers. Thus, the coverage of the models built is related to the user behavior traces and crawlers used.

Besides the explicit behavior models for Web applications, a large number of studies focus on the event dependency analysis over program variables and event-handler functions of the various DOM elements, which can be treated as implicit models for Web applications. For example, Sung et al.[40] proposed the first constraint-based static analysis method for computing dependencies across event-handlers and between HTML DOM elements. Nguyen et al.[41] provided a call graph for client-side code while it was still embedded in server-side code. The nodes in the call graph referred to code elements with corresponding origin locations in string literals of the server-side code. The edges represented possible jumps between the nodes and may have conditions. Wang et al.[42] proposed an Event Handler Tree (EHT) model to assist the test case generation process, in which the node of EHT is the event handler and the relationship between two event handlers is the dependency.

6.2 Trace-based model inference techniques

To model Web applications, dynamic analysis techniques are extensively applied in model establishing, which take a set of traces as input and infer a model based on these traces[14, 43]. For example, Marchetto and Tonella[9] created an FSM model by traces, aiming at detecting Ajax faults caused by the execution order of semantically interacting events. Thus, the trace only contained information about the DOM states and event sequences causing transitions from state to state. This approach considers neither the parameters or DOM element changes nor the relation between the conditions of event-handler functions and Web pages.

The authors in Refs. [5,10,23] proposed a graph-based behavior model created by traces. The objective was to facilitate developers' understanding of the client-side behavior of the Web application. A detailed trace of a Web application's behavior was captured, including all the event-handler functions that were executed either directly or indirectly after an event occurred. Although the researchers considered the event-handler functions, the trace did not analyze the internal structure of these functions and the relation between the conditions of the event-handler functions and Web pages.

Schur et al.[16] applied the event sequences as the observed application executions traces to mine an explicit FSA model of enterprise Web applications. They incrementally learned a model by generating program runs(traces) and observing the application behavior through the user interface. The relation between conditions and Web pages was computed based on the input data of traces. However, if the Web pages depend on specific features of the supplied data, which can trigger the corresponding conditions, PROCRAWL is unlikely to explore this relation through guessing.

7 Conclusion

This paper proposed a novel CBM to represent dynamic behaviors for modern Web applications, which can depict not only the Web pages and trigger events but also the trigger conditions and follow-up operations. To generalize CBMs automatically for Web application, we captured the user behavior traces and recorded them in advance. Then, based on the traces, the original CBMs were constructed and further optimized by identifying and merging equivalent states and transitions. Moreover, we prove that there exists a homomorphism relation between the original and optimized CBMs, that is, they exhibit the same behavior. The experiments show that our CBM construction method is effective and the JS-trace-set is most appropriate to guide the generation of CBMs for Web applications.

Acknowledgment

The work was supported by the National Natural Science Foundation of China (Nos. 61672085, 61702029, and 61872026).

References

[1] H. Javed, N. M. Minhas, A. Abbas, and F. M. Riaz, Model based testing for Web applications: A literature survey presented, *Journal of Software*, vol. 11, no. 4, pp. 347–361, 2016.

[2] E. Habibi and S. H. Mirian-Hosseinabadi, Event-driven Web application testing based on model-based mutation testing, *Information and Software Technology*, vol. 67, pp. 159–179, 2015.

[3] X. S. Dong, K. Patil, J. Mao, and Z. K. Liang, A comprehensive client-side behavior model for diagnosing attacks in Ajax applications, in *Proc. 18th Int. Conf. Engineering of Complex Computer Systems*, Singapore, 2013, pp. 177–187.

[4] P. Liu and Z. N. Xu, MTTool: A tool for software modeling and test generation, *IEEE Access*, vol. 6, pp. 56222–56237, 2018.

[5] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, Understanding JavaScript event-based interactions, in *Proc. 36th Int. Conf. Software Engineering*, Hyderabad, India, 2014, pp. 367–377.

[6] A. Mesbah, Software analysis for the Web: Achievements and prospects, in *IEEE 23rd Int. Conf. Software Analysis, Evolution, and Reengineering*, Suita, Japan, 2016, pp. 91–103.

[7] Y. F. Li, P. K. Das, and D. L. Dowe, Two decades of Web application testing: A survey of recent advances, *Information Systems*, vol. 43, pp. 20–54, 2014.

[8] C. H. Liu, C. J. Wu, and H. M. Chen, Testing of AJAX-based Web applications using hierarchical state model, in *IEEE 13th Int. Conf. e-Business Engineering*, Macau, China, 2016, pp. 250–256.

[9] A. Marchetto and P. Tonella, Using search-based algorithms for Ajax event sequence generation during testing, *Empirical Software Engineering*, vol. 16, no. 1, pp. 103–140, 2011.

[10] S. Alimadadi, Understanding behavioural patterns in JavaScript, in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, Seattle, WA, USA, 2016, pp. 1076–1078.

[11] X. F. Qi, Z. Y. Wang, J. Q. Mao, and P. Wang, Automated testing of Web applications using combinatorial strategies, *Journal of Computer Science and Technology*, vol. 32, no. 1, pp. 199–210, 2017.

[12] K. Hossen, R. Groz, C. Oriat, and J. L. Richier, Automatic generation of test drivers for model inference of Web applications, in *IEEE 6th Int. Conf. Software Testing, Verification and Validation Workshops*, Luxembourg, Luxembourg, 2013, pp. 441–444.

[13] A. Van Deursen, A. Mesbah, and A. Nederlof, Crawl-based analysis of Web applications: Prospects and challenges, *Science of Computer Programming*, vol. 97, pp. 173–180, 2015.

[14] N. Walkinshaw, R. Taylor, and J. Derrick, Inferring extended finite state machine models from software executions, *Empirical Software Engineering*, vol. 21, no. 3, pp. 811–853, 2016.

[15] M. Schur, A. Roth, and A. Zeller, Mining behavior models from enterprise Web applications, in *Proc. 9th Joint Meeting on Foundations of Software Engineering*, Saint Petersburg, Russia, 2013, pp. 422–432.

[16] M. Schur, A. Roth, and A. Zeller, Mining workflow models from Web applications, *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1184–1201, 2015.

[17] R. A. Haraty, N. Mansour, and H. Zeitunlian, Metaheuristic algorithm for state-based software testing, *Applied Artificial Intelligence*, vol. 32, no. 2, pp. 197–213, 2018.

[18] J. Mao, J. D. Bian, G. D. Bai, R. L. Wang, Y. Chen, Y. H. Xiao, and Z. K. Liang, Detecting malicious behaviors in JavaScript applications, *IEEE Access*, vol. 6, pp. 12 284–

12 294, 2018.

[19] W. W. Wang, J. X. Guo, Z. Li, and R. L. Zhao, EFSM-oriented minimal traces set generation approach for Web applications, in *IEEE 42nd Annu. Computer Software and Applications Conf.*, Tokyo, Japan, 2018, pp. 12–21.

[20] A. Mesbah, A. Van Deursen, and S. Lenselink, Crawling Ajax-based Web applications through dynamic analysis of user interface state changes, *ACM Transactions on the Web*, vol. 6, no. 1, pp. 1–30, 2012.

[21] S. Joshi, N. Agrawal, R. Krishnapuram, and S. Negi, A bag of paths model for measuring structural similarity in Web documents, in *Proc. 9th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, New York, NY, USA, 2003, pp. 577–582.

[22] F. Ricca and P. Tonella, Analysis and testing of Web applications, in *Proc. 23rd Int. Conf. Software Engineering*, Toronto, Canada, 2001, pp. 25–34.

[23] A. M. Fard, M. Mirzaaghaei, and A. Mesbah, Leveraging existing tests in automated test generation for Web applications, in *Proc. 29th ACM/IEEE Int. Conf. Automated Software Engineering*, Vasteras, Sweden, 2014, pp. 67–78.

[24] S. Elbaum, S. Karre, and G. Rothermel, Improving Web application testing with user session data, in *Proc. 25th Int. Conf. Software Engineering*, Portland, OR, USA, 2003, pp. 49–59.

[25] X. L. Xu, H. Jin, S. Wu, L. X. Tang, and Y. H. Wang, URMG: Enhanced CBMG-based method for automatically testing Web applications in the cloud, *Tsinghua Science and Technology*, vol. 19, no. 1, pp. 65–75, 2014.

[26] A. Marchetto, P. Tonella, and F. Ricca, State-based testing of Ajax web applications, in *Proc. 1st Int. Conf. on Software Testing Verification & Validation*, Lillehammer, Norway, 2008, pp.121–130.

[27] T. Gowda and C. A. Mattmann, Clustering Web pages based on structure and style similarity (application paper), in *IEEE 17th Int. Conf. Information Reuse and Integration*, Pittsburgh, PA, USA, 2016, pp. 175–180.

[28] A. H. Kulkarni and B. M. Patil, Template extraction from heterogeneous Web pages with cosine similarity, *International Journal of Computer Applications*, vol. 87, no. 3, pp. 4–8, 2014.

[29] B. Biswas, K. Jain, V. Mittal, and K. K. Shukla, Exploiting tree structure of a Web page for clustering, *International Journal of Knowledge & Web Intelligence*, vol. 1, no. 1/2, pp. 81–94, 2009.

[30] M. E. Akpinar and Y. Yesilada, Vision based page segmentation algorithm: Extended and perceived success, in *Proc. 13th Int. Conf. Web Engineering*, Aalborg, Denmark, 2013, pp. 238–252.

[31] T. T. Wei, Y. H. Lu, X. J. Li, and J. L. Liu, Web page segmentation based on the hough transform and vision cues, in *2015 Asia-Pacific Signal and Information Processing Association Annu. Summit and Conf.*, Hong Kong, China, 2015, pp. 865–872.

[32] K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li, Model projection: Simplifying models in response to restricting the environment, in *Proc. 33rd Int. Conf. Software Engineering*, New York, NY, USA, 2011, pp. 291–300.

[33] J. Thomé, A. Gorla, and A. Zeller, Search-based security testing of Web applications, in *Proc. 7th Int. Workshop on Search-Based Software Testing*, Hyderabad, India, 2014, pp. 5–14.

[34] N. Alshahwan and M. Harman, Automated Web application testing using search based software engineering, in *Proc. 26th IEEE/ACM Int. Conf. Automated Software Engineering*, Lawrence, KS, USA, 2011, pp. 3–12.

[35] M. G. Li, L. Y. Li, and F. P. Nie, Ranking with adaptive neighbors, *Tsinghua Science and Technology*, vol. 22, no. 6, pp. 733–738, 2017.

[36] S. Liang, Y. Zhang, B. Li, X. J. Guo, C. F. Jia, and Z. L. Liu, SecureWeb: Protecting sensitive information through the Web browser extension with a security token, *Tsinghua Science and Technology*, vol. 23, no. 5, pp. 526–538, 2018.

[37] A. A. Andrews, J. Offutt, and R. T. Alexander, Testing Web applications by modeling with FSMs, *Software & Systems Modeling*, vol. 4, no. 3, pp. 326–345, 2005.

[38] N. Alshahwan, M. Harman, and A. Marchetto, Crawlability metrics for web applications, in *Proc. 5th Int. Conf. on Software Testing Verification and Validation*, Montreal, Canada, 2012, pp. 151–160.

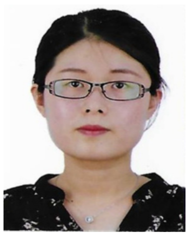
[39] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, JSEFT: Automated JavaScript unit test generation, in *IEEE 8th Int. Conf. Software Testing, Verification and Validation*, Graz, Austria, 2015, pp. 1–10.

[40] C. Sung, M. Kusano, N. Sinha, and W. Chao, Static DOM event dependency analysis for testing Web applications, in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, Seattle, WA, USA, 2016, pp. 447–459.

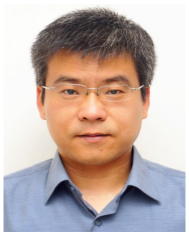
[41] H. V. Nguyen, C. Kästner, and T. N. Nguyen, Building call graphs for embedded client-side code in dynamic Web applications, in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, New York, NY, USA, 2014, pp. 518–529.

[42] B. Wang, B. B. Yin, and K. Y. Cai, Event handler tree model for GUI test case generation, in *IEEE 40th Annu. Computer Software and Applications Conf.*, Atlanta, GA, USA, 2016, pp. 58–63.

[43] D. Lorenzoli, L. Mariani, and M. Pezzè, Automatic generation of software behavioral models, in *Proc. ACM/IEEE 30th Int. Conf. Software Engineering*, Leipzig, Germany, 2008, pp. 501–510.



Weiwei Wang received the BS degree from Beijing University of Chemical Technology (BUCT), China in 2014. Currently, she is a PhD candidate at BUCT. Her research interests are in the area of software testing, focusing on Web application modeling and test case generation.



Zheng Li received the PhD degree from King's College London, CREST centre in 2009 under the supervision of Mark Harman. He is a professor at Beijing University of Chemical Technology. He has worked as a research associate at King's College London and University College London. He has worked on program testing and source code analysis and manipulation. More recently, he is interested in search-based software engineering and slicing state-based model.

Junxia Guo received the PhD degree in computer science from Tokyo Institute of Technology in 2013. She is now an associate professor at Beijing University of Chemical Technology. Her primary research interests include software testing and Web-user's behavior analysis.

Ruilian Zhao received the PhD degree in computer science from Chinese Academy of Sciences in 2001. She is now a professor and PhD supervisor at Beijing University of Chemical Technology. Her primary research interests include software testing and fault-tolerant computing.