Qi XF, Wang ZY, Mao JQ *et al.* Automated testing of web applications using combinatorial strategies. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 32(1): 199–210 Jan. 2017. DOI 10.1007/s11390-017-1699-x

Automated Testing of Web Applications Using Combinatorial Strategies

Xiao-Fang Qi [1,2], *Member, CCF, ACM*, Zi-Yuan Wang [3], *Member, CCF, IEEE*, Jun-Qiang Mao [1] and Peng Wang [1], *Member, CCF, ACM, IEEE*

[1] *School of Computer Science and Engineering, Southeast University, Nanjing 211189, China*

[2] *Key Laboratory of Computer Network and Information Integration, Ministry of Education, Nanjing 211189, China*

[3] *School of Computer Science and Technology, Nanjing University of Posts and Telecommunications Nanjing 210023, China*

E-mail: xfqi@seu.edu.cn; wangziyuan@njupt.edu.cn; mjqseu@163.com; pwang@seu.edu.cn

Received March 21, 2016; revised November 11, 2016.

Abstract Recently, testing techniques based on dynamic exploration, which try to automatically exercise every possible user interface element, have been extensively used to facilitate fully testing web applications. Most of such testing tools are however not effective in reaching dynamic pages induced by form interactions due to their emphasis on handling client-side scripting. In this paper, we present a combinatorial strategy to achieve a full form test and build an automated test model. We propose an algorithm called pairwise testing with constraints (PTC) to implement the strategy. Our PTC algorithm uses pairwise coverage and handles the issues of semantic constraints and illegal values. We have implemented a prototype tool ComjaxTest and conducted an empirical study on five web applications. Experimental results indicate that our PTC algorithm generates less form test cases while achieving a higher coverage of dynamic pages than the general pairwise testing algorithm. Additionally, our ComjaxTest generates a relatively complete test model and then detects more faults in a reasonable amount of time, as compared with other existing tools based on dynamic exploration.

Keywords automated testing, combinatorial testing, web application

1 Introduction

With the widespread use of the Internet and intranets, web applications have been becoming increasingly important and popular in recent years[1]. One of key technologies employed in developing modern Web applications is AJAX, an acronym for Asynchronous JavaScript and XML[2]. AJAX, which combines JavaScript and Document Object Model (DOM) manipulation, as well as asynchronous communication, offers users high interactivity and responsiveness. Despite these advantages, AJAX web applications often exhibit dynamic and stateful behaviors, making them notoriously difficult to test. Besides handling static components (such as static hypertext links), testing modern AJAX web applications has to deal with dynamic components including client-side scripting and form interactions[1-2].

Existing commonly used web application testing tools, such as Selenium IDE, WebKing, and Sahi, run in a capture-and-replay style①~③. They first record specific user-defined scenarios, then generate test scripts mainly containing sequences of events fired by user interactions, and finally automatically replay the recorded scenarios in another test process. While such tools are capable of executing test cases, they are difficult to conduct thorough and automated testing since a large amount of manual effort is required for record-

Regular Paper

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61472076, 61472077, and 61300054.

① http://selenium.openqa.org, September 2016.

② http://www.parasoft.com/jsp/products/home.jsp?product=WebKing, September 2016.

③ http://sahi.co.in/w/, September 2016.

ing scenarios. Typically, only the functionalities of a few critical execution paths are tested in many applications. A large number of execution paths that are likely to be exercised by users are missed. In addition, if web applications are modified, previous scripts will become invalid and cannot be reused. A long time may be taken to generate new scripts from scratch.

Crawljax is a tool that supports the dynamic and automatic exploration of the state spaces for modern web applications[2]. Using Crawljax, testers can derive an automated test model, which captures the states of user interface and the possible event-based transitions between states. Since this model systematically provides possible execution paths, it could be directly used to generate test cases. Recently, Crawljax has been successfully incorporated into the automated framework for web application testing or regression testing[3-4]. Crawljax excels in handling client-side scripting, but it does not make a particular effort for forms. It fills self-generated random values in forms. As we know, dynamic pages are induced by not only client-side scripting but also form interactions. Many dynamic web pages may not be reachable unless appropriate inputs are submitted through forms. Therefore, it is difficult for Crawljax to achieve a full coverage of dynamic web pages. To derive a complete automated test model, testers should generate valid form test cases for further exploration.

Assume that a form submission request consists of k parameters. Each parameter has appropriate values supplied by using techniques such as equivalent class analysis or generated by users. One approach to generating form test cases is to derive all the possible k-way combinations of parameter values and then reach almost all possible web pages. However, such combinations are exponential and may lead to a potential state explosion problem during exploration. Pairwise testing, which covers all the combinations of values of any two parameters, has shown to be a cost-effective combinatorial strategy. It can make a more than 80% branch coverage in software testing while significantly reducing the number of test cases that need to be executed (the web pages commonly are determined by the branches in the sever-side code that process the form)[5]. Therefore, using the pairwise combinatorial strategy for generating form test cases may achieve a high coverage of dynamic web pages while substantially decreasing the number of test cases. Empirical studies also indicate that a form test using pairwise combination detects 75%∼90% faults in web applications[6-7].

However, a simple pairwise combinatorial strategy may not be practical in form test case generation due to the ubiquity of domain semantics constraints in forms[8-9]. A test case will become invalid if it violates the semantics constraints. For example, consider a login form, which consists of the user name, password, and confirm password. In this case, the value of the password must be exactly the same as that of the confirm password; otherwise the test case is invalid and the request is not passed. During form test generation, we should consider these semantics constraints and exclude invalid test cases. In addition, since a large number of users without training often fill illegal data into forms, developers have to design multiple corresponding dynamic web pages in server-side code to respond to various illegal data requests. If every illegal value is treated as a common value that will combine with other parameter values, some dynamic web pages may never be reached. The reason is that some illegal values may terminate the execution in advance and then mask other illegal or legal values. Therefore, illegal values should be handled specially.

In this paper, we present a combinatorial strategy for making a full form test, hence exploring more states and building a complete automated test model of a web application. In this combinatorial strategy, we propose an algorithm called pairwise testing with constraints (PTC), which uses pairwise coverage and handles the issues of semantic constraints and illegal values. We have implemented a prototype tool ComjaxTest, which is capable of discovering various dynamic web pages induced by form interactions and systematically exploring the state space of a web application. Unlike Crawljax which fills random values in forms, ComjaxTest uses the PTC algorithm to generate an adequate form test to reach more dynamic web pages. Moreover, during exploration, ComjaxTest can automatically detect various types of faults including hyperlinks, database accesses, and JavaScript running exceptions. Other error-checking components, e.g., invariants and assertion checking, are also easily plugged into ComjaxTest to make a further test. We have applied ComjaxTest, Crawljax and other automatic testing tools based on dynamic exploration to five representative web applications and conducted an empirical study. We evaluated these testing tools in terms of dynamic web page coverage, generated states, fault detection capability, and performance. Experimental results show that our PTC algorithm generates less form test cases while achieving a higher coverage of dynamic pages than the general

pairwise testing algorithm. As compared with other existing tools based on dynamic exploration, our ComjaxTest generates a relatively complete test model and detects more faults in a reasonable amount of time.

The remainder of this paper is organized as follows. Section 2 describes our exploration algorithm and PTC algorithm for form test generation. Section 3 discusses the implementation of our automated testing tool ComjaxTest. Section 4 reports experimental results. Section 5 reviews related work. Section 6 concludes this paper and provides the guidelines of the future work.

2 Automatically Exploring Web Applications

In this section, we first present the definition of state flow graph (an automated test model) and related basic concepts, then describe our exploration algorithm for automatically building a state flow graph, and finally provide our PTC algorithm for generating form test cases.

2.1 Basic Concepts

Unlike a traditional web application in which a user interface state is represented by a URL and the corresponding page, a user interface state in an AJAX web application is represented by a browser's dynamically built DOM tree. An AJAX state change is defined as a change on the DOM tree, caused either by client-side events handled by the AJAX engine or by server-side state changes propagated to the client[2]. In this paper, a state in a web application exactly refers to a user interface state. For description purposes, state and user interface state are used interchangeably.

Typically, a user interacts with an AJAX web application by clicking on an element or by bringing the mouse over an element, etc. Such actions may generate events and change the state of the application. A DOM element that has an event listener attached to it and can cause a user interface state change is called a clickable element. To model the user interface state changes in an AJAX web application, a state flow graph, which provides the user interface states and the possible transitions between them, is defined as follows.

Definition 1. *A state flow graph (SFG) for a web application A, is a triple $G = (N, E, n_0)$ where N is a set of nodes representing runtime DOM states in A, n_0 is the initial state after A has been fully loaded into the browser, and E is a set of edges representing the possible transitions between nodes. Each edge (n_1, n_2) represents a clickable element c iff n_2 is reached by executing c in state n_1.*

A state flow graph is constructed from the initial state. New states are added incrementally during exploration. Fig.1 shows the state flow graph (SFG) of an example. The state denoted as index page is the initial state, from which three states, namely n_1, n_2 and n_3, are reached by firing three different events. Each edge between states is labeled with the type of an event and the identification of a clickable element, e.g., an XPath expression or an ID attribute. As shown in Fig.1, clicking on the //A[3] element and the //DIV[1]/IMG[2] element in the initial state results in state n_1 and state n_3 respectively while bringing mouse over the id:c_2 element generates state n_2. In state n_2, clicking on the //SPAN[1]/A[1] element leads to a transition, which returns to the initial state.

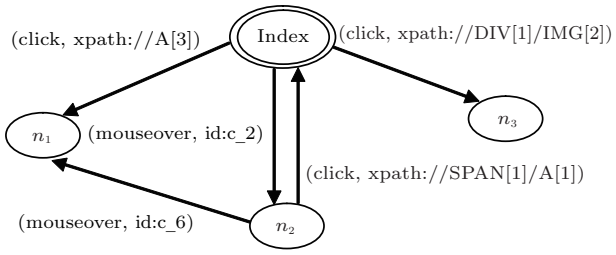


Fig.1. State flow graph of an example.

2.2 State Exploration

Algorithm 1 shows our exploration algorithm BuildStateFlowGraph, which automatically explores the state space of a web application. Given the URL of the homepage in an application, the algorithm builds a state flow graph of the application in a depth-first manner. The procedure Main (lines 1∼6) first initializes an embedded browser, generates the initial state of the state flow graph, namely *rs*, and then starts a recursive call procedure ExploreState (line 5). In the procedure ExploreState, when a newly discovered state is generated, it invokes ExploreState recursively (line 29). Once a recursive call comes to its end and begins to backtrack, the execution path and the related information are recorded. And then we restore the database, reload the application, and re-execute the events from the initial state to the desired state (line 31) such that the browser could be put back into the state which it was in before the call.

In the procedure ExploreState, Algorithm 1 first deals with the forms in the current state *cs*. For each form, it retrieves the possible values of each parameter

ps and the constraints *ct* that are supplied in a test database, and generates the set of submission requests T (lines 9∼10) using PairTestWithConstraints algorithm, which will be discussed in Subsection 2.3. After that, it identifies the elements in the current DOM tree that commonly attach event listeners as candidate clickable elements. In our implementation, the default tag types include <A>, <BUTTON>, and <INPUT type=submit>. Users can extend or exclude the selection of candidate clickable elements by defining element properties, attributes and their values, and text values. Then each candidate clickable element or form submission request is fired (line 14). The resulting DOM *dom* in the browser is compared with each corresponding DOM of visited states in *sf* (line 17). If there is a state *s* in *sf* and its DOM is similar to *dom*, then the two corresponding states are merged into one state, namely *s*, and add an edge from *cs* to *s* (lines 17∼19). Otherwise, a new state *ns* is created and added to *sf*. The corresponding edge from *cs* to *ns* is also added to *sf* (lines 23∼27). Finally, if the state *ns* is viable in exploring more states, the procedure ExploreState is recursively called to detect more possible states reachable from *ns* (lines 28∼29).

Algorithm 1 . BuildStateFlowGraph

```
Input:  URL
Output: the state flow graph
 1: proc Main( )
 2:     rs ← InitEmbededBrowser(URL)
 3:     sf ← InitStateFlowGraph()
 4:     sf.AddState(rs)
 5:     ExploreState(rs)
 6: end proc
 7: proc ExploreState(State cs)
 8:     for each form f in cs do
 9:         retrieve the parameter values ps and the constraints
    ct
10:         T ← GenPairTest(ps, ct)
11:     end for
12:     Set C ← GetCandClick(cs)
13:     for each e in T ∪ C do
14:         FireEvent(e)
15:         dom ← browser.GetDom()
16:         for each s in sf do
17:             flag ← IsSimilar(s.getDom(), dom)
18:             if flag == true then
19:                 sf.AddEdge(cs, s)
20:                 break
21:             end if
22:         end for
23:         if flag == false then
24:             ns ← NewState(dom)
25:             sf.AddState(ns)
26:             sf.AddEdge(cs, ns)
27:         end if
28:         if Enabled(ns) then
29:             ExploreState(ns)
30:         end if
31:         ResetToState(cs)
32:     end for
33: end proc
```

With regard to state comparison (line 17), our algorithm computes the similarity between two DOM trees in terms of structure and content. For any two DOMs, d_1 and d_2, we first find all leaf nodes in the two DOMs, and compute the structure similarity by comparing the XPaths of the leaf nodes in the two DOMs. A structure similarity threshold is used to determine whether d_1 is similar to d_2 or not in structure. If d_1 is not similar to d_2 in structure, we say d_1 is not similar to d_2. If d_1 is similar to d_2 in structure, the content similarity is further computed by comparing the content between the two leaf nodes with the same XPath. A content similarity is also used to finally determine whether d_1 is similar to d_2. If d_1 is similar to d_2 in both structure and content, we say d_1 is similar to d_2.

Fig.2 shows how our exploration algorithm builds the SFG of the example application in Fig.1. First, the initial index state is generated, and then ExploreState(*index*) is invoked. Afterwards, our algorithm fires the first event on the clickable element e_1 and generates the state n_1. Since n_1 is a new state, ExploreState(n_1) is invoked. Yet n_1 contains no candidate clickable. Then our algorithm backtracks to its previous index state. In Fig.2, each solid line shows an exploring path while each dotted line shows a backtracking path and the annotated number denotes the corresponding backtracking order. To go from n_1 to index, ResetToState(*index*) is invoked, i.e., our algorithm restores the database and reloads the browser so that it lands on the index state. The index state does contain two unexplored events on the clickable elements, namely e_2 and e_3. Then, it explores the (e_2, e_4) path, reaches n_1 (not a new state), and backtracks to n_2. Similarly, to go from n_1 to n_2, our algorithm restores the database, reloads the browser, and re-executes events on the clickable elements from the initial index state to n_2. After landing on n_2, it fires e_5 and reaches the index state again. Then it reloads the browser and lands on the initial index state. Finally it fires e_3, and then backtracks to the index state.

2.3 Form Test Generation

In this subsection, we propose a PairTestWithConstraints (PTC) algorithm to generate form submission requests. Our PTC algorithm, which handles constraints and illegal values, is capable of being practically applied to web applications. To effectively handle constraint issues, we have defined a simple language called FormDataSpec to specify semantics constraints

for parameters in forms and have accordingly developed a constraint checking component. For description purposes, we first provide the definition of FormDataSpec, discuss the issues of constraint checking, and then present our PTC algorithm.

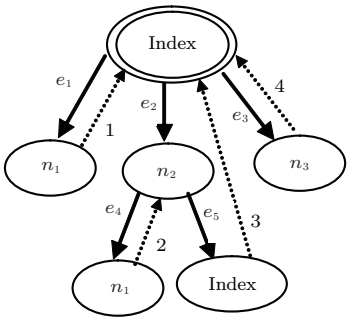


Fig.2. Exploration process of the example in Fig.1.

2.3.1 Constraint Checking

FormDataSpec presents constraints as propositional formulas. FormDataSpec supports two kinds of constraints, namely unconditional and conditional constraints. An unconditional constraint, which declares an always-valid limitation, is described using a logical expression. A conditional constraint, which declares a limitation in some condition, is described in the form of IF-THEN or IF-THEN-ELSE. FormDataSpec mainly contains the following syntax:

1) *cstrts* → *cstrt* | *cstrt cstrts*
2) *cstrt* → *comp*;
   |IF *comp* THEN *comp*;
   |IF *comp* THEN *comp* ELSE *comp*;
3) *comp* → *term* | *comp logOp term*
4) *term* → *relExp* | (*comp*) | NOT *comp*
5) *relExp* → *param relOp param*
   |*param relOp val* | *param* IN *valSet*.

In 2), unconditional and conditional constraints are described. In 3) and 4), a logical expression *comp* is defined as that in a common programming language. Logical operator *logOp* can be AND and OR while relation operator *relOp* can be =, <>, <, <=, >, >=, etc. For each parameter, a limited number of values are generated to represent many possible values determined typically by equivalent class partition or its nature. In 5), a parameter is allowed to be compared with another parameter or a parameter value. FormDataSpec offers two commonly used data types, i.e., numeric and string. A numeric parameter can be compared with a number, and a string parameter to another string.

Since a logical expression may contain multiple relation expressions, FormDataSpec is capable of describing complicated semantics constraints for multiple parameters or parameter values naturally. Generally, constraints are gathered by studying domain knowledge, requirement documents, etc.

We have implemented a constraint checking component called ConstraintChecker. Given a set of constraints written in FormDataSpec, ConstraintChecker is capable of parsing the constraints and determining whether a combination or a test case violates the constraints or not. Our PTC algorithm performs constraint checking by invoking ConstraintChecker.

Bryce and Colbourn demonstrated that determining whether there exists a test case that satisfies a given set of constraints is an NP-hard problem[10]. If not, such constraints are not consistent in the context of a given model. To ensure the consistency of constraints, we first check them manually. If the consistency cannot be ensured, we then use Z3, an efficient off-the-shelf satisfiability solver, to evaluate the consistency④.

2.3.2 PTC Algorithm

Given the values of parameters and constraints in a form, PTC algorithm, as shown in Algorithm 2, adopts the greedy strategy to generate a test suite in a one-test-at-a-time manner[8]. A test case is derived to cover more uncovered combinations.

In our PTC algorithm, the procedure GenPairTest first initializes *IllegList* to be the set of illegal parameter values, and initializes *UncovSet* to be the set of all 2-way parameter value combinations that does not contain invalid pairs using constraint checking (lines 2∼4). There are two loops in the PTC algorithm. The outer loop is expected to be finished until all 2-way combinations have been covered (lines 6∼19). To make the outer loop terminate in a reasonable amount of time, users can set the maximum number of iteration *MaxTimes* for the loop (line 5). At each iteration step, one test case is generated. Moreover, to avoid invalid iterations due to specific parameter values, the order for parameters is randomly shuffled (line 7). The first parameter value is selected by finding the value appearing more often than the others in *UncovSet* (line 8). The values for remaining parameters are derived in the inner loop (lines 10∼16). For each parameter, it first identifies possible values *ValSet* that do not violate the constraints (line 12), then finds the value *pv* that maximizes the number of uncovered combinations

④http://research.microsoft.com/Hen-us/um/redmond/projects/z3/, September 2016.

in *UncovSet* (line 13), and removes all 2-way combinations covered by *tc* from *UncovSet* (line 14). When the inner loop is terminated, a test case *tc* is complete.

Algorithm 2 . PairTestWithConstraints

```
Input:  the values of parameters, constraints
Output: a set of test cases
 1: proc GenPairTest(ParaVal ps, Constraint ct)
 2:     UncovSet ← ∅, IllegList ← ∅, TestCaseSet ← ∅
 3:     add each illegal parameter value in ps into IllegList
 4:     add each 2-way parameter value combination into
    UncovSet if it does not violate the constraints
 5:     times ← MaxTimes, tc ← ∅
 6:     while UncovSet ≠ ∅ and times > 0 do
 7:         PermuteParameterList()
 8:         v ← UncovSet.findMostValue()
 9:         tc.Add(v)
10:         while tc is not complete do
11:             select next parameter p
12:             ValSet ← GetValue(p, ct)
13:             pv ← UncovSet.findMostCover(ValSet)
14:             UncovSet.Remove(pv, tc)
15:             tc.Add(pv)
16:         end while
17:         add tc into TestCaseSet
18:         tc ← ∅, times − −
19:     end while
20:     for each σ in UncovSet do
21:         generate a test case tc that covers σ and does not vi-
    olate the constraints
22:         add tc into TestCaseSet
23:     end for
24:     for each v in IllegList do
25:         generate a test case tc that covers v and does not vi-
    olate the constraints
26:         add tc into TestCaseSet
27:     end for
28: end proc
```

When the outer loop is exited, there may be still uncovered pairs in *UncovSet*. One test case is constructed for each uncovered pair (lines 20∼23). Typically, since the code using an illegal value makes the control flow immediately jump into an error handling call or another program point, the subsequent code is not possible to be executed, and then the illegal value masks other errors induced by other illegal or legal values. Therefore, in our algorithm, illegal values are not combined with other parameters. For each illegal parameter value, one test case is generated (lines 24∼27).

Fig.3 shows a payment form, in which there are three parameters, namely *Total*, *Vip*, and *Pref*. *Total* represents the shopping expense. There are two legal values, namely 10 and 100, and one illegal value −100. If a customer is a VIP, the value of *Vip* is YES; otherwise, it is NO. *Pref*, which represents the type of preference, has the value of DISCOUNT or CREDIT. The last row in Table 1 shows the constraint between parameters. If *Vip* is NO, *Pref* must be CREDIT.

To ensure the generation of valid combinations or test cases, our PTC algorithm performs constraint checking by invoking ConstraintChecker. When generating the set *UnCovSet* (line 4), our PTC algorithm uses ConstraintChecker to eliminate invalid pairs, therefore ensuring that *UnCovSet* contains no invalid pairs. In lines 12, 21, and 25, new parameter values are added to generate a test case. Since some values may violate the constraints, our PTC algorithm performs constraint checking at such steps.

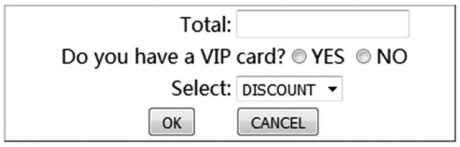


Fig.3. Payment form.

Table 1. Parameters, Values and Constraints in a Form

Parameter	Value
Total	−100, 10, 100
Vip	YES, NO
Pref	DISCOUNT, CREDIT
Constraint	IF *Vip* = NO THEN *Pref* = CREDIT

Table 2 shows the generated test cases for the example using our PTC algorithm. Initially, *IllegalList* contains (*Total*, −100), and *UncovSet* contains all 2-way combinations except the pair (NO, DISCOUNT). In the first outer iteration, the two values of *Total*, namely 10 and 100, appear four times in the uncovered pairs. We select an arbitrary value for *Total*, e.g., 100. Then we select an arbitrary value YES as the value of *Vip* since (100, YES) and (100, NO) appear once in the uncovered pairs. Finally, DISCOUNT is selected for *Pref*. After the first test case is generated, three combinations, namely (100, YES), (100, DISCOUNT), and (YES, DISCOUNT), are removed from *UncovSet*. In a similar way, we can generate the other four test cases. The five test cases cover all the pairs in the initial uncovered pairs. At the end, a test case is constructed for the illegal value, namely −100 for *Total*.

Table 2. Test Cases Using PairwiseTestWithConstaints Algorithm

No.	*Total*	*Vip*	*Pref*
1	100	NO	CREDIT
2	100	YES	DISCOUNT
3	10	YES	CREDIT
4	10	YES	DISCOUNT
5	10	NO	CREDIT
6	−100	NO	CREDIT

Table 3 shows the test cases using the general pairwise testing algorithm, which also uses the same greedy strategy as our PTC algorithm but not handles constraints and illegal values. The third and the fifth test

cases are invalid since they contain (NO, DISCOUNT), which violates the constraints in Table 1. These invalid test cases cause a loss of combination coverage, i.e., (100, NO), (100, DISCOUNT), (10, NO), and (10, DISCOUNT). The first and the second test cases contain the illegal value, −100, leading to a further loss of combination coverage.

Table 3. Test Cases Using the General Pairwise Testing Algorithm

No.	*Total*	*Vip*	*Pref*
1	−100	NO	CREDIT
2	−100	YES	DISCOUNT
3	100	NO	DISCOUNT
4	10	YES	CREDIT
5	10	NO	DISCOUNT
6	100	YES	CREDIT

3 Implementation

We have developed a prototype tool called ComjaxTest in Java. ComjaxTest implements our BuildStateFlowGraph algorithm and PairTestWithConstraints algorithm in Section 2 on the top of HtmlUnit APIs. As shown in Fig.4, ComjaxTest consists of four major components, namely Embedded Browser, DOM Analyzer, Event Generator and Explorer. The interface of Embedded Browser currently supports three popular browsers, i.e., Internet Explorer, Firefox, and Chrome.

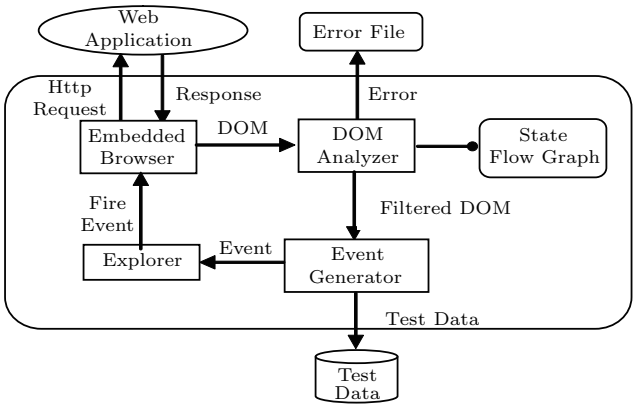


Fig.4. Architecture of ComjaxTest.

Embedded Browser is responsible for providing a common interface for accessing JavaScript engine and runtime DOMs. Through the embedded browser, ComjaxTest is capable of sending an http request to a web application and receiving the corresponding response.

DOM Analyzer checks whether a DOM tree contains general error messages, heuristic code error messages stored in error files, or other exceptions or not. If it contains any error message, the current path is recorded for debugging purposes. Otherwise, it is compared with already generated states. If it is a new state, it will be added to the state flow graph.

Event Generator extracts all DOM elements that often attach event listeners as candidate clickable elements. The default tag types include <A>, <BUTTON>, and <INPUT>. Users can customize their interest in clickable elements. For example, all elements with a tag DIV having an attribute *class* = "book" can be defined as candidate elements. If a form submission button is encountered, Event Generator will first fetch the parameter values from the Test Data database, and then generate submission requests using our PairTestWithCons algorithm.

Explorer is responsible for exploring the state space of a web application. Moreover, it simulates user actions, e.g., click, mouseover, text input, on the embedded browser, which then sends http requests to web servers.

4 Experiment

To assess the efficacy of our approach in supporting web application testing, we conducted an empirical study on five real-world web applications. The goal of this experiment is to evaluate the coverage of dynamic pages for form submissions, the completeness of the generated state flow graph, the fault detection capabilities, and the performance of our ComjaxTest, as compared with existing tools based on dynamic exploration, namely Crawljax and VeriWeb. Our research questions are summarized as follows.

*RQ*1. What is the coverage of dynamic web pages for responding to form submission using our PTC algorithm?

*RQ*2. How complete is a state flow graph built by ComjaxTest?

*RQ*3. What is the fault detection capability of ComjaxTest?

*RQ*4. What is the performance of ComjaxTest?

4.1 Subject Applications

In our experiment, we selected five open source web applications of different application domains[5]. The

[5]http://sourceforge.net, September 2016.

subject applications are implemented in JSP or PHP. Table 4 shows their properties including the number of lines of codes (LOC), forms, parameters, and parameter values.

Table 4. Experimental Web Applications

Application	LOC	Form	Parameter	Value
EShop	6 353	5	25	70
Addressbook	5 046	4	24	57
Forum	7 650	5	24	68
Schoolmate	8 181	15	67	206
Nucleus	36 697	11	83	216

EShop is an online bookstore, which allows users to create accounts, search and purchase books, and manage shopping carts and orders. Addressbook is a web application of address management. Forum is an online forum. Schoolmate is a solution for elementary, middle and high school administration. Nucleus is a personal blog application for creating blogs and replying.

4.2 Experimental Setup

Our experiment was performed on Windows 7, running on a 3.2 GHz, Intel Core i5-3470 CPU with 32 GB memory. We configured ComjaxTest, and other existing tools, namely Crawljax and VeriWeb[2,11]. Neither time nor depth was limited during exploration with these tools. Other configurations are summarized as follows.

ComjaxTest. Parameter values are generated using techniques such as equivalence class partition and boundary value analysis. Constraints for parameters are obtained by manually analyzing the source code of applications. Form submission requests are derived using our PairTestWithConstraints algorithm (PTC) and the general pairwise testing algorithm (PT) separately. The similarity threshold in structure and content is 0.8.

Crawljax. In general, Crawljax fills self-generated random values in forms[2]. Yet, in this experiment, to ensure further exploration, login forms are filled in manually generated parameter values. Other parameter values are randomly generated.

VeriWeb. VeriWeb is not available now[11]. We implemented the main algorithms. It does not merge states. Each form is filled in a valid test case.

4.3 Experimental Results

4.3.1 Form Test (RQ1)

In a real web application, if the number of parameter values that can be filled in a form is infinite, a potentially infinite number of dynamic pages may be generated to respond to the form submission requests. Yet, these dynamic web pages can be classified into a few categories according to different functionalities or program logics. For description purposes, we call each category a form submission result. In this paper, we use the number of form submission results to measure the coverage of dynamic web pages responding to form submission requests. Subsequently, we manually construct a reference model to perform this evaluation.

Table 5 reports the number of form submission results in manual reference model (column "Manual"), ComjaxTest configured with PairTestWithConstraints algorithm (column "PTC"), ComjaxTest configured with general pairwise testing algorithm (column "PT"), Crawljax and VeriWeb. The average coverage of form submission results ranges from 55.0% to 96.4%. ComjaxTest with PTC achieves the highest coverage, then ComjaxTest with PT, VeriWeb, and finally Crawljax. The results of Crawljax and VeriWeb indicate that approximately half of submission results are missed if not handling forms. In contrast, the combinatorial strategy facilitates form test generation, hence reaching more dynamic pages.

Table 5. Form Submission Results

Application	Manual	PTC	PT	Crawljax	VeriWeb
EShop	8	8	5	6	6
Addressbook	7	7	7	5	5
Forum	28	27	15	8	8
Schoolmate	79	75	55	47	50
Nucleus	47	46	39	27	27
Total	169	163	121	93	96
Coverage (%)	100	96.4	71.5	55.0	56.8

However, when using the general pairwise combinatorial strategy, roughly 30% of form submission results are missed. The main reason is that it ignores constraints for parameters. In this experiment, constraints occur naturally between different parameter values in 23% of forms. When the general pairwise testing is used, some test cases may be invalid since they violate constraints. These invalid test cases would not be exercised, which leads to a less coverage of combinations between parameter values, hence decreasing the coverage of form submission results. Another reason is that the general pairwise testing does not address the issue of illegal values. Typically, many web pages are designed to respond to submission requests with various illegal values. In the five subject applications, there exist 90% of forms for which some dynamic pages are

not reachable unless illegal values are supplied. Yet, if a test case contains more than one illegal value, some illegal values may mask other illegal or legal values, hence further decreasing the coverage.

When using ComjaxTest with our PTC algorithm, almost all form submission results are covered. Six form submission results are not detected because several illegal values for exceptional cases are not provided. Table 6 shows the number of form test cases in ComjaxTest configured with PTC and PT algorithms. Our PTC algorithm generates 21.7% fewer form test cases than PT because constraint checks and the special treatment for illegal values significantly reduce the number of combinations that should be covered. As compared with PT algorithm, our PTC algorithm generates less form test cases while achieving a higher coverage of dynamic pages.

Table 6. Form Test Cases

Application	PTC	PT
EShop	53	62
Addressbook	43	45
Forum	53	64
Schoolmate	178	229
Nucleus	115	138
Total	442	538

4.3.2 State Flow Graph (RQ2)

Table 7 reports the number of nodes (column "N_{PTC}") and edges (column "E_{PTC}") in the SFGs generated by ComjaxTest configured with our PTC algorithm, the number of nodes (column "N_{PT}") and edges (column "E_{PT}") in the SFGs generated by ComjaxTest configured with PT algorithm, the number of nodes (column "N_{CR}") and edges (column "E_{CR}") in the SFGs generated by Crawljax, and the number of nodes (column "N_{VW}") and edges (column "E_{VW}") in the state flow graphs generated by VeriWeb. To measure the completeness of the state flow graphs, we manually generated complete state flow graphs for Eshop, Forum, and Nucleus by analyzing source codes and running applications. State flow graphs for Addressbook and Schoolmate are too large and complex to manually generate.

Table 7. State Flow Graph

Application	N_{PTC}	E_{PTC}	N_{PT}	E_{PT}	N_{CR}	E_{CR}	N_{VW}	E_{VW}
EShop	37	196	25	129	29	168	17	176
Addressbook	314	430	226	391	236	387	14	88
Forum	34	104	30	92	28	96	15	27
Schoolmate	138	352	102	243	93	206	25	60
Nucleus	78	204	71	195	67	196	34	72

Table 8 reports the completeness of the state flow graphs for Eshop, Forum, and Nucleus. "N_{MAN}" and "E_{MAN}" in Table 8 represent the number of nodes and edges in the manually generated state flow graphs respectively while other variables are the same as those in Table 7. The average node ratio ranges from 35.5% to 80.1%, and the average edge ratio from 38.4% to 70.3%. From Table 7 and Table 8, we can see that ComjaxTest with PTC generates the most complete graphs, then ComjaxTest with PT and Crawljax, finally VeriWeb, which roughly coincides with the coverage of form submission results. State exploration strongly depends on form handling.

Table 8. Completeness Results for Three Applications

	EShop	Forum	Nucleus	Total	Avg. Node Ratio (%)
N_{MAN}	44	45	97	186	100.0
E_{MAN}	285	186	246	717	100.0
N_{PTC}	37	34	78	149	80.1
E_{PTC}	196	104	204	504	70.3
N_{PT}	25	30	71	128	68.8
E_{PT}	129	92	195	416	58.0
N_{CR}	29	28	67	124	66.7
E_{CR}	168	96	196	460	64.1
N_{VW}	17	15	34	66	35.5
E_{VW}	176	27	72	275	38.4

The state flow graph generated by ComjaxTest with PTC missed 19.9% of nodes and 29.7% of edges. One reason is that form submission results are not completely covered. Another reason may be that some different states are merged into one state during merging states. Note that even though VeriWeb achieves a similar coverage of form submission results with Crawljax, the graphs generated by VeriWeb are much smaller than those generated by Crawljax. The reason is that VeriWeb focuses on handling traditional web applications while not coping with many dynamic features of AJAX techniques.

4.3.3 Fault Detection (RQ3)

Table 9 reports the number of faults automatically detected by ComjaxTest configured with PTC algorithm, ComjaxTest configured with PT algorithm, Crawljax and VeriWeb. The subject applications have been tested and released in the SourceForge net. The detected faults in Table 9 are missed by their testing tools. After carefully checking, these missing and naturally happened faults involve hyperlink, data base access, JavaScript code, etc.

Table 9. Number of Detected Faults

Application	PTC	PT	Crawljax	VeriWeb
EShop	0	0	0	0
Addressbook	7	5	5	1
Forum	1	1	1	1
Schoolmate	4	3	2	0
Nucleus	6	3	3	1
Total	18	12	11	5

The results in Table 9 show that ComjaxTest with PTC detects the most faults, then ComjaxTest with PT and Crawljax, and finally VeriWeb. Such results are consistent with the results of form submission results and state flow graphs because more form results and more states contain more faults.

4.3.4 Execution Time (RQ4)

Table 10 reports the execution time taken to generate state flow graphs by ComjaxTest configured with PTC algorithm, ComjaxTest configured with PT algorithm, Crawljax and VeriWeb. Since VeriWeb does not effectively handle the loop issue, the testing is stuck in the infinite loop, denoted by "–" in Table 10. To terminate its execution, the time limit in this experiment is 24 hours.

As shown in Table 10, in most cases, ComjaxTest configured with PTC takes the most time, then ComjaxTest with PT, and finally Crawljax. Obviously, the size of state flow graphs is an important factor influencing the execution time. Yet, as mentioned in the BuildStateFlowGraph algorithm, before firing an event, we need to restore the application, i.e., resetting the database and re-executing the events from the initial state to the desired state. This restoration is the most time-consuming activity during state exploration. It may dominate the total execution time. ComjaxTest configured with PTC and PT takes much more time than Crawljax since a large number of form submission tests dramatically increase the state restoration time. Similarly, in most cases, even though a state flow graph generated with PT is smaller than that generated with PTC, ComjaxTest configured with PT performs more form test cases, naturally taking more execution time than ComjaxTest with PTC.

Table 10. Execution Time (min)

Application	PTC	PT	Crawljax	VeriWeb
EShop	130.0	113.3	103.2	–
Addressbook	130.3	247.5	68.7	–
Forum	963.3	972.8	480.0	–
Schoolmate	152.4	189.7	119.3	–
Nucleus	296.8	320.7	181.9	–

5 Related Work

Dynamic exploration techniques play an important role in our web application testing. The framework of such techniques is also widely used in the field of web crawling[12-13]. However, web crawling and our ComjaxTest have their different goals. Web crawling intends to discover as much information as possible from various pages while our web application testing aims to derive a complete model for fully testing web applications. Consequently, web crawling uses techniques that pick information-rich pages and discard the others. In contrast, our web application testing tries to discover more user interface states and the relationship between them. With regard to form submission generation, web crawling is concerned with how to choose appropriate values for individual parameters while we generate a form test suite that covers combinations between parameters.

Like our ComjaxTest, most existing web application testing tools based on dynamic exploration use a depth-first strategy to explore the state space of a web application and restore the states by re-executing the event sequences from the initial state to the desired state[2,5,11]. The major differences between these tools and our ComjaxTest lie in how to generate form submission requests, how to control the potential state explosion, whether to support the new dynamic features of modern web applications, and other specific exploration techniques.

Benedikt *et al.* presented a tool VeriWeb for automatically exploring possible execution paths for traditional web applications[11]. As mentioned in Subsection 4.3, VeriWeb is easily stuck in infinite loop because of not coping with the loop issue. Moreover, it controls the state explosion merely by setting a limit on the length of execution paths or the execution time. Different from VeriWeb, our ComjaxTest not only handles the loop issue, but also leverages similar state merging to control the number of states. VeriWeb allows users to fill predefined parameter values in forms. Any combinatorial strategy is not mentioned in form test generation. Since it was developed earlier in 2002, it may be not able to cope with many dynamic features in AJAX, such as JavaScript and dynamic DOM manipulation.

Mesbah and Deursen proposed Crawljax for automatically exploring the user interface state spaces of modern web applications[2]. Based on the dynamic analysis of the client-side user interface states in embedded browsers, Crawljax incrementally detects DOM

states and generates state flow graphs. Recently, Crawljax has been successfully used in invariant-based testing and regression testing[3-4]. Our ComjaxTest adopts a similar exploration framework for coping with modern web applications. However, they significantly differ in form test generation. Crawljax, which fills merely self-generated random values in forms, often makes an incomplete form test and achieves the inadequate coverage of dynamic pages. Furthermore, if some parameter values are special, e.g., the password in a login form, such random values are typically not passed. In this case, the subsequent exploration is prevented. In contrast, our ComjaxTest uses PTC algorithm to generate adequate form test cases to detect more user interface states, hence deriving a relatively complete test model.

Wang *et al.* presented a tool Tansuo which allows combinatorial strategies to cope with forms and discover more dynamic web pages[5]. Different from our PTC algorithm, Tansuo adopts a general pairwise testing algorithm to generate form test cases, in which domain semantics constraints are not taken into account and illegal values are not handled specially. As for form handling, Tansuo is similar to our ComjaxTest configured with the PT algorithm, which achieves a lower coverage of dynamic pages than our PTC algorithm while using more form test cases. In addition, Tansuo represents states using abstract URL, a URL that retains parameter names but removes parameter values in the query component. However, when the attribute of a form is *post*, such abstract URL would fail to represent states since a *post* form submission request does not contain parameter information. There is no sufficient detail to demonstrate that Tansuo would be able to cope with modern web applications.

Currently, the most commonly used web application testing tools, such as Selenium IDE, WebKing, and Sahi, run in a capture-and-replay style. As mentioned in Section 1, they are not capable of making a full test.

Ricca and Tonella developed a tool ReWeb for generating a UML model of a web application to support traditional data-flow analysis and reaching frame analysis[14].

Recently, a feedback-directed web application exploration technique has been proposed to reduce the size of a test model[15]. In addition, some novel types of DOM-based test adequacy criteria have been provided for web application testing[16-17]. Mutation testing has also been extended to web applications[16]. Both static and dynamic program analyses are leveraged to guide the mutation generation.

6 Conclusions

In this paper, we presented an algorithm called pairwise testing with constraints (PTC) to fully test forms. Our PTC algorithm uses pairwise coverage and handles the issues of semantic constraints and illegal values. We implemented a prototype tool ComjaxTest, which is capable of systematically exploring the state space of a web application. The experimental results indicated that ComjaxTest configured with PTC algorithm achieves a high coverage of dynamic web pages induced by form interactions, generates a relatively complete test model, and detects more faults in a reasonable amount of time.

In the future work, to avoid irrelevant or insignificant exploration, we plan to study heuristic strategies for guiding ComjaxTest to explore only relevant or significant states. The generated test model is incomplete, but it has adequate functionality, code or DOM state coverage, hence easing the state explosion and improving its testing efficacy. Additionally, we will leverage some state or edge coverage criteria to automatically derive test suites for web application testing or regression testing by traversing the corresponding state flow graphs.

Acknowledgment We thank the anonymous JCST reviewers for their valuable comments on this paper.

References

[1] Mesbah A. Advances in testing JavaScript-based web applications. *Advances in Computers*, 2015, 97: 201-235.

[2] Mesbah A, Deursen A V, Lenselink S. Crawling AJAX-based web applications through dynamic analysis of user interface state changes. *ACM Trans. the Web*, 2012, 6(1): 3:1-3:29.

[3] Mesbah A, Deursen A V, Roest D. Invariant-based automatic testing of modern web applications. *IEEE Trans. Softw. Engin.*, 2012, 38(1): 35-53.

[4] Roest D, Mesbah A, Deursen A V. Regression testing AJAX applications: Coping with dynamism. In *Proc. the 3rd Int. Conf. Software Testing, Verification, and Validation*, April 2010, pp.127-136.

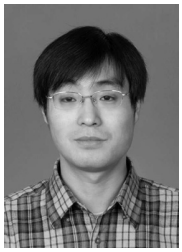
[5] Wang W, Lei Y, Sampath S *et al.* A combinatorial approach to building navigation graphs for dynamic Web applications. In *Proc. the 25th Int. Conf. Software Maintenance*, September 2009, pp.211-220.

[6] Kuhn D R, Reilly M J. An investigation of the applicability of design of experiments to software testing. In *Proc. the 27th Annual NASA Goddard Software Engineering Workshop*, December 2002, pp.91-95.

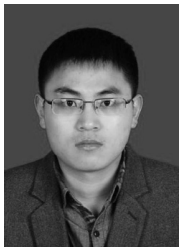
[7] Kuhn D R, Wallce D R, Gallo A M. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 2004, 30(6): 418-421.

[8] Nie C, Leung H. A survey of combinatorial testing. *ACM Computing Surveys*, 2011, 43(2): 11:1-11:29.
[9] Ostrand T J, Balcer M J. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 1988, 31(6): 676-686.
[10] Bryce R C, Colbourn C J. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 2006, 48(10): 960-970.
[11] Benedikt M, Freire J, Godefroid P. VeriWeb: Automatically testing dynamic Web sites. In *Proc. the 11th Int. Conf. World Wide Web*, May 2002, pp.654-668.
[12] Madhavan J, Ko D, Kot Ł *et al*. Google's deep-web crawl. *Proceedings of the VLDB Endowment*, 2008, 1(2): 1241-1252.
[13] Cai R, Yang J M, Lai W *et al*. iRobot: An intelligent crawler for web forums. In *Proc. the 17th Int. Conf. World Wide Web*, April 2008, pp.447-456.
[14] Ricca F, Tonella P. Analysis and testing of web applications. In *Proc. the 23rd Int. Conf. Software Engineering*, May 2001, pp.25-34.
[15] Fard A M, Mesbah A. Feedback-directed exploration of web applications to derive test models. In *Proc. the 24th Int. Symp. Software Reliability Engineering*, November 2013, pp.278-287.
[16] MirzaAghaei M, Mesbah A. DOM-based test adequacy criteria for web applications. In *Proc. Int. Symp. Software Testing and Analysis*, July 2014, pp.71-81.
[17] Zou Y, Chen Z, Zheng Y *et al*. Virtual DOM coverage for effective testing of dynamic web application. In *Proc. Int. Symp. Software Testing and Analysis*, July 2014, pp.60-70.

Xiao-Fang Qi received her Ph.D. degree in computer science from Southeast University, Nanjing, in 2008. She is currently an associate professor in the School of Computer Science and Engineering, Southeast University, Nanjing. Her research interests include software analysis and testing, and software engineering.



Zi-Yuan Wang received his Ph.D. degree in computer science from Southeast University, Nanjing, in 2009. He is currently an associate professor in the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications, Nanjing. His research interests include software testing and programming language.



Jun-Qiang Mao received his B.S. degree in computer science from Nanjing University of Information Science and Technology, Nanjing, in 2014. He is currently pursuing his M.S. degree in the School of Computer Science and Engineering, Southeast University, Nanjing. His research mainly focuses on software testing.

Peng Wang received his Ph.D. degree in computer science from Southeast University, Nanjing, in 2008. He is currently an associate professor in the School of Computer Science and Engineering, Southeast University, Nanjing. His research interests include software engineering, knowledge graph, and social network. He has published many papers at premium conferences and journals such as IJCAI, and Science China: Information Science.