A Combinatorial Approach to Building Navigation Graphs for Dynamic Web Applications

Wenhua Wang, Yu Lei
Dept. of Computer Science and Engineering
University of Texas at Arlington,
Arlington, Texas
{wenhuawang, ylei}@uta.edu

Sreedevi Sampath
Dept. of Information Systems
University of Maryland, Baltimore County,
Baltimore, MD
sampath@umbc.edu

Raghu Kacker, Rick Kuhn
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD
{raghu.kacker, kuhn}@nist.gov

James Lawrence
Department of Mathematics
George Mason University
Fairfax, VA
lawrence@gmu.edu

Abstract

Modeling the navigation structure of a dynamic web application is a challenging task because of the presence of dynamic pages. In particular, there are two problems to be dealt with: (1) the page explosion problem, i.e., the number of dynamic pages may be huge or even infinite; and (2) the request generation problem, i.e., many dynamic pages may not be reached unless appropriate user requests are supplied. As a user request typically consists of multiple parameter values, the request generation problem can be further divided into two problems: (1) How to select appropriate values for individual parameters? (2) How to effectively combine individual parameter values to generate requests?

This paper presents a combinatorial approach to building a navigation graph. The novelty of our approach is two-fold. First, we use an abstraction scheme to control the page explosion problem. In this scheme, pages that are likely to have the same navigation behavior are grouped together, and are represented as a single node in a navigation graph. Grouping pages reduces and bounds the size of a navigation graph for practical applications. Second, assuming that values of individual parameters are supplied by using other techniques or generated manually by the user, we combine parameter values in a way that achieves a well-defined combinatorial coverage called pairwise coverage. Using pairwise coverage can significantly reduce the number of requests that have to be submitted while still achieving effective coverage of the navigation structure. We report a prototype tool called Tansuo*, and apply the tool to five open source web applications. Our empirical results indicate that* Tansuo *can efficiently generate web navigation graphs for these applications.*

1. Introduction

A web navigation graph, or simply a navigation graph, is a representation of the navigation structure of a web application, with nodes representing web pages and edges representing direct transitions between web pages. Navigation graphs can be used as an aid in tasks such as understanding, maintaining, and testing web applications. For example, they can be used as a model to generate test sequences during testing and/or regression testing [1]. As another example, navigation graphs can be used to facilitate impact analysis, i.e., how to identify pages that could be potentially affected by a modified page.

The main challenge of building a navigation graph is dealing with dynamic pages. (If an application only consists of static pages, its navigation graph can be built using a classical graph traversal algorithm, e.g., a depth-first search algorithm.) Unlike a static page, whose content is prescribed and stored on a web server, a dynamic page does not physically exist until a request for this page is submitted, typically through an HTML form. The existence of dynamic pages creates two problems for building a navigation graph:

- A potentially infinite number of dynamic pages may be generated in a web application. If a dynamic page is directly modeled as a node, the size of a navigation graph may be infinite. For example, after a user logs in, an application may dynamically generate a personalized page to greet the user. Since the number of users can be infinite, the number of personalized pages generated by the application can be infinite.

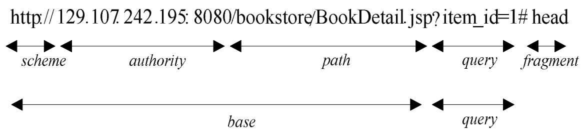
- Some dynamic pages may not be reached unless appropriate requests are supplied. In other words, in order to ensure coverage, user requests must be generated carefully during the construction of a navigation graph. For example, consider an application where a user can log in as a regular user or an administrator. Pages that can only be visited by an administrator would be missed if we do not request to log in as an administrator.

We will refer to the first problem as the page explosion problem, and the second problem as the request generation problem. Considering that a request often consists of multiple parameter values, the request generation problem can be further divided into two smaller problems: (1) How to select appropriate values for individual parameters? (2) How to effectively combine individual parameter values to generate requests? In this paper, we focus on the second aspect of the request generation problem, i.e., how to combine parameter values, assuming that these values are supplied by using techniques such as boundary value analysis and/or generated manually by the user.

To the best of our knowledge, little work has been reported on effective construction of navigation graphs. However, the above two problems have been encountered and addressed in a similar context, i.e., web crawling. Web crawling refers to discovering useful information by navigating through web applications. Many web applications store information in a database, and provide the user with an HTML form through which a query can be submitted to retrieve information of interest. Therefore, like navigation graph construction, web crawling also has to deal with the challenge of how to interact with forms. However, unlike navigation graph construction, which is interested in "structure discovery", i.e., how different pages interconnect with each other, web crawling is interested in "content discovery", i.e., how to discover useful information that is contained in those pages. This difference has a profound impact on techniques that are developed in the two different contexts. This will be discussed in detail in Section 2.

In this paper, we propose a combinatorial approach to building navigation graphs. To address the page explosion problem, we use the notion of an abstract URL. Conceptually, a (concrete) URL [2] can be broken into two components, *base* and *query*, as shown in Fig. 1. The *query* component is optional, and typically consists of a set of parameter-value pairs. Given a (concrete) URL u, the abstract URL for u is obtained by removing the values, but retaining the parameter names, in the *query* component. For example, given a URL u = "*http://test.com/foo.jsp?x=1&y=2*", the abstract URL is "*http://test.com/foo.jsp?x&y*". In our approach, pages that have the same abstract URL are represented as a single node in a navigation graph. The rationale behind this abstraction is that these pages are likely to be generated from the same template, and are thus similar in their structures and associated navigation behaviors (i.e., they have the same set of predecessor/successor pages). For practical applications, this abstraction allows us to bound the number of nodes, and in turn the size of the navigation graph, while largely preserving the navigation structure.

To address the request generation problem, we assume that individual parameter values are given, and use a combinatorial strategy to combine the values. Assume that a form has k parameters, each with d possible values. To reach every possible page that could be generated by submissions of this form, we could try to submit the form with every possible combination of values of those parameters. Doing so, however, can be prohibitively expensive, due to a potentially large number of combinations. In our approach, we submit the form with a subset of parameter value combinations that achieves a well-defined combinatorial coverage, namely pairwise coverage [3][4]. That is, given any two out of the k parameters, we ensure that every combination of the two parameters is covered in at least one submission. (In the remainder of the paper, we will refer to a combination of values of all k parameters as a *submission test*, and a combination of values of any two parameters as a *combination*, unless otherwise specified.) Pairwise coverage has been shown to be very effective for general software testing, while dramatically reducing the number of tests that need to be executed [3][4][5]. In particular, empirical studies indicate that pairwise coverage often leads to more than 80% branch coverage in general software testing [6]. Since the pages that could be generated by submitting a form are often determined by the branches that exist in the server code processing the form, achieving pairwise coverage can lead to a high coverage of those pages while significantly reducing the number of submission tests.


http://129.107.242.195:8080/bookstore/BookDetail.jsp?item_id=1#head
scheme
authority
path
query
fragment
base
query
Figure 1: An example of a URL

We have developed a prototype tool, called *Tansuo*, that implements our approach, and have applied this tool to five open source web applications. These applications include *Bookstore*, *Bug Tracking System* (*BugTrack*), *Classifieds*, *Links* and *Portal* [7]. The results show that *Tansuo* can effectively build navigation graphs that achieve a high degree of coverage of the navigation structure for those applications. In addition, we have compared *Tansuo* with two existing tools, namely WebSphinx [8] and Link Checker Pro [9]. The two existing tools do not interact with forms, and thus do not deal with dynamic pages. (We were unable to obtain access to tools that deal with dynamic pages, like VeriWeb [10]). The comparison shows that *Tansuo* can build navigation graphs that are dramatically more complete than the two existing tools. This suggests that the ability to handle forms is vital to achieve high-coverage navigation graphs.

The remainder of the paper is organized as follows: Section 2 discusses related work. Section 3 describes our approach. In particular, we present an algorithm that implements our approach. Section 4 introduces the architecture of *Tansuo*. Section 5 presents the empirical results of applying *Tansuo* to five web applications. Section 6 concludes this paper and discusses future work.

2. Related work

Web crawling has been an active research area in recent years [8][9][11][12][13][14]. Web crawling is related to our work because it navigates through a collection of web pages, and needs to deal with dynamic pages. (There are two types of web crawling: surface or regular web crawling, which does not interact with forms, and deep web crawling, which interacts with forms. We focus our attention on deep web crawling.) However, web crawling is about "content discovery", i.e., it aims to discover as much information as possible from different pages, while our work is about "structure discovery", i.e., it aims to capture the navigation relationship among different pages. Consequently, web crawling employs techniques that are very different from ours. Specifically, web crawlers often deal with the page explosion problem by picking pages that are information-rich and by discarding the others [11][12]. This is different from our work, where pages are abstracted based on their URLs, instead of their contents. To address the request generation problem, most web crawlers have focused on the problem of how to select values for individual parameters. One common approach used by those crawlers is to build a pre-defined list of values for the parameters that are frequently encountered. This approach is also useful in our work. The problem of how to effectively combine those parameter values has been largely left open. The very recent work by Madhavan et al. [11] is an exception. Their approach uses a bottom-up search strategy to identify parameter combinations that could lead to the largest number of distinct response pages. This differs from our work, where we generate combinations to achieve a well-defined coverage criterion.

Our work is also related to web application testing techniques in which test sequences are generated on the fly and by navigating through an application [10][15]. In particular, our work is closely related to VeriWeb [10]. VeriWeb applies a general software model checking technique called VeriSoft [16] to web applications. To test a web application, VeriWeb tries to explore all possible navigation paths in a systematic manner. VeriWeb uses an exploration algorithm that is very similar to ours. That is, both VeriWeb and our approach explore in a depth-first manner, and restore states by re-visiting the sequence of pages on the stack. However, the two approaches significantly differ in the way they handle the page explosion and request generation problem. Specifically, VeriWeb controls the page explosion problem by allowing a limit to be set on the length of navigation paths it explores. This is different from our use of abstract URLs. VeriWeb addresses the request generation problem by allowing the user to supply and reuse pre-defined parameter values. It is unclear which strategy is used to combine parameter values during exploration.

Several models have been developed and used in model-based web application testing [17][18][19]. In particular, the UML model proposed by Ricca and Tonella [19] is close to our navigation graph. A fundamental difference between their model and ours is that they do not abstract dynamic pages. That is, each dynamic page is represented as a separate node in their model. In addition, their models are designed to support a wide range of analysis such as reaching frame analysis and traditional data-flow analysis. Hence, their model contains other types of relations, e.g., the *include* relation between a frame and a web page.

3. The approach

In this section, we present a combinatorial approach to building navigation graphs. Section 3.1 gives a formal definition of a navigation graph, and uses an example to further illustrate the notion of pairwise coverage. Section 3.2 presents an algorithm that implements our approach. Section 3.3 provides additional discussion.

3.1. Basic concepts

First we define a navigation graph. Intuitively, a node in a navigation graph represents a group of web pages that have the same abstract URL. Recall from Section 1 that we abstract a URL by removing the parameter values, while retaining the parameter names, in the *query* component, if this component exists. (If a URL does not have a *query* component, its abstraction is the same as the URL itself.) Abstracting URLs helps to control the page explosion problem while preserving the navigation structure of a web application. There exists an edge from one node *n* to another node *n'* if there is a direct transition from a page *p* represented by node *n* to a page *p'* represented by node *n'*, i.e., page *p'* can be immediately visited after page *p*.

In the following, we formalize the definition of a navigation graph. Let *abs(p)* denote the abstract URL of a web page *p*. Let *pages(n)* denote the group of pages represented by a node *n*. Let $p \rightarrow p'$ denote a direct transition from a page *p* to a page *p'*. Then, a navigation graph *G* can be formally defined as follows: $G = (V, E)$, where (1) *V* is a set of nodes such that for each node $n \in V$, $\forall p, p' \in pages(n)$, $abs(p) = abs(p')$; and (2) $E \subseteq V \times V$ is a set of edges such that for each edge (n, n'), there exists at least one direct transition $p \rightarrow p'$, where $p \in pages(n)$, $p' \in pages(n')$.

In Section 1, we introduced the notion of pairwise coverage, which reduces the number of submission tests needed for each form but still achieves good coverage of dynamic pages. To illustrate pairwise coverage, consider a form that has three parameters *p1*, *p2*, and *p3*, each parameter having two values 0 and 1. Fig. 2 shows a pairwise set of submission tests for this form. Each row represents a submission test, and each column represents a form parameter (in the sense that each entry in a column is a value of the parameter represented by the column). An important property of this submission test set is that each of the three possible pairs of columns, i.e. columns *p1* and *p2*, columns *p1* and *p3*, and columns *p2* and *p3*, contains all four possible pairs of values of any two (out of three) parameters, i.e., {00, 01, 10, 11}. Thus, this submission test set achieves pairwise coverage for this form. Note that an exhaustive submission test set would consist of $2^3 = 8$ submission tests.

$$\left(\begin{array}{ccc} p1 & p2 & p3 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array}\right)$$

Figure 2: A pairwise submission test set

Many algorithms have been proposed for combinatorial test generation [20]. In particular, Tai and Lei [21] proposed a pairwise testing strategy called In-Parameter-Order (IPO). The IPO strategy generates a pairwise test set to cover the first two parameters and then extends the test set to cover the first three parameters. This process is repeated until the test set covers all the parameters. Our approach uses the IPO algorithm to generate pairwise submission test sets.

3.2. Algorithm BuildNavGraph

Fig. 3 shows algorithm *BuildNavGraph*, which implements our approach. This algorithm takes as input the URL of the *home* page of a web application, and produces as output the navigation graph of the application. Algorithm *BuildNavGraph* explores a web application in a depth-first manner, so it has a framework that is similar to that of a classic depth-first search algorithm. Therefore, we will not explain the algorithm line by line. Instead, we will focus on how algorithm *BuildNavGraph* differs from a classic depth-first search algorithm.

First, algorithm *BuildNavGraph* uses a different approach to decide whether to explore a newly encountered URL (lines 10 and 30). Specifically, a newly encountered URL is explored only if its abstraction does not yet exist in the navigation graph. In other words, we will not explore a newly encountered URL *u* if some other URL *u'*, with *abs(u')* = *abs(u)*, has been explored before. This is necessary to ensure that the exploration process comes to an end. However, it also introduces a risk of missing some pages that may be reached only if *u* is actually explored. More discussions on this risk, as well as an optimization that can reduce this risk, is provided in Section 3.3.

Second, a classic depth-first search algorithm is designed to traverse all the nodes in a graph. As a result, it usually does not keep track of all the edges that are visited during the search process. In other words, a classic depth-first search is usually used to build a spanning tree of the original graph. This is different from our algorithm, which tries to capture the entire navigation graph structure. Therefore, it is important to add the corresponding edges into the resulting graph (lines 9 and 29) even if a newly encountered URL will not be explored (because its abstraction already exists in the graph).

Third, the second *for* loop (lines 12 to 33) deals with forms, which represents the key contribution of our approach. A web page may contain multiple forms, each of which is dealt with by one iteration of the *for* loop. Suppose that we are dealing with form *f* in the current iteration. We first obtain the values of individual parameters in form *f* (line 13). This can be done either interactively, i.e., asking the user to provide the values as each form is encountered, or up front, i.e., asking the user to provide the values for each possible parameter that may appear in a form. Note that the latter can be extremely useful for test automation, but requires a priori knowledge about what parameters may appear in a web application, as well as what values those parameters can take.

The first inner loop (lines 15 to 32) deals with each action in the form. An action may or may not require parameter values to be submitted to the server side. If an action does require parameter values to be submitted, we will generate a pairwise submission test set for those parameters. Each submission test is then used once to perform the action. If an action does not require any parameters to be submitted, then it can simply be performed, after which the URL of the succeeding page is added to list *l*. Note that list *l* is used to hold all the URLs of the succeeding pages that can be reached from the current page either through a static or dynamic link.

Finally, after we finish exploring a node, we need to back up to its parent node *p*. Before we explore another child node of node *p*, it is important to restore the state of the application, e.g., the session and database state, back to the state when *p* was encountered but none of its branches had been explored (line 34). This restoration ensures the exploration process to be semantically correct, as the exploration of different branches of a particular node should be independent from each other. One approach to restoring a state is to save the state when it was first encountered, and then reset the application to the saved state at the time the state is needed. However, explicit state representation can be difficult for practical applications. In our algorithm, we restore the state by re-executing all the transitions that were executed to reach this state. Note that even though algorithm *BuildNavGraph* is presented as a recursive process, it is implemented as an iterative process. Therefore, in order to restore a particular state, we only need to re-execute all the transitions that are currently on the stack.

```
Algorithm BuildNavGraph
Input: The URL of the home page of a web application
Output: The navigation graph G = (V, E) of the application

1. BuildNavGraph (URL home) {
2.   let G = <V, E> be an empty graph
3.   traverse (home, G)
4.   return G
5. }

6. function traverse (URL u, Graph G) {
7.    for each static link u' in u {
8.       add a node labeled with abs(u') into V,  if it does not exist
9.       add an edge labeled with (abs(u), abs(u')) into E,
            if it does not exist in E
10.      traverse (u', G) if abs(u') is encountered for the first time
11.   }
12.   for each form f in u {
13.      obtain the values of individual parameters in f
14.      let l be an empty list
15.      for each action a in f {
16.         if (action a requires submission of param values) {
17.            generate a pairwise submission test set s for action a
18.            for each submission test t in s {
19.               perform action a with test t
20.               add the URL of the succeeding page to list l
21.            }
22.         }
23.         else {
24.            perform action a
25.               add the URL of the succeeding page to list l
26.         }
27.         for each URL u' in list l {
28.               add a node labeled with abs(u') into V,
                    if it does not exist in V
29.               add an edge labeled with (abs(u), abs(u')) into
                       E, if it does not exist in E
30.               traverse (u', G) if abs(u') is encountered
                     for the first time
31.         }
32.      }
33.   }
34.   restore the application to the state reached right after u is
         encountered (but not traversed yet)
35.}
```

Figure 3: BuildNavGraph algorithm

Now we consider the time and space complexity of algorithm *BuildNavGraph*. The time complexity is similar to that of a classic depth-first search, except that we need to take into account the time for generating pairwise submission test sets and for performing those tests. Assume that a form has at most k parameters, each of which takes at most d values. The size of a pairwise submission test set is $O(d^2 \log k)$. The time for generating those tests is $O(d^3 k^2 \log k)$, if the IPO algorithm [21] is used. The time for performing all those tests is $O(td^2 \log k)$, where t is the longest time required to perform a submission test. Therefore, the total time complexity of algorithm *BuildNavGraph* is $O(|G|+d^3k^2\log k + td^2\log k)$. The space complexity of algorithm *BuildNavGraph* is the same as that of a classic depth-first search algorithm, i.e., $O(|G|)$.

3.3. Discussion

There are several cases in which our approach may not fully capture the navigation structure of an application. First, in our approach, pages having the same abstract URL are represented as a single node in a navigation graph. As an abstract URL drops all the parameter values in the *query* component of a (concrete) URL, we assume that the navigation behaviour of a web page, in terms of the set of pages that could be reached from this page, does not depend on specific parameter values. This may not be true for some applications. In addition, an abstract URL does not contain information about system state, e.g., the values of session variables. The same page may have different navigation behaviours depending on different system states and such navigation behaviours may not be captured by our approach.

Adding more information to the abstract URL, i.e., making the abstraction finer-grained, would help capture more navigation behaviours. However, doing so may prolong the exploration process, and may unnecessarily increase the size of a navigation graph. This is because many nodes can have the same navigation behaviour even if they have different parameter values and/or are visited at different system states.

There is an optimization that can be done to make a navigation graph more complete without adding more information to an abstract URL. In algorithm *BuildNavGraph*, a newly encountered URL is explored only if its abstract URL does not yet exist. We can change this decision so that a newly encountered URL is explored only if the abstract edge leading to the URL does not yet exist. The rationale for this change is that a page that is reached by a different edge may likely be a different page, even if another page with the same abstract URL has been visited before. The reason has been discussed in the first paragraph in this sub section. This optimization has been used in the experiments in the Section 5, and has been shown to be very effective.

There is a second case in which the navigation structure of an application may not be fully captured. In order to explore all the pages that could be generated by a form, we perform a set of submission tests that achieve pairwise coverage. Obviously, pairwise coverage does not cover all the combinations. A page would not be explored if it could only be generated by submitting one or more of the combinations that do not appear in the pairwise set. Achieving a higher degree of combinatorial coverage such as 3-way, or 4-way coverage will help to make the resulting navigation graph more complete. However, doing so will be more computationally expensive, as more submission tests will have to be performed. We note that the number of submission tests required to achieve a certain level of combinatorial coverage can grow quickly as we increase the strength of coverage.

In spite of the cases just described, our experimental results, as presented in the Section 5, suggest that our approach produces close to complete navigation graphs for the applications we studied.

4. Tansuo: a prototype tool

We have implemented our approach in a prototype tool called *Tansuo*. *Tansuo* is written in Java, and uses HTTPUnit to handle Javascripts, which allows for exploring more complete navigation structures. (HttpUnit can execute Javascripts, but it does not perform static analysis on the source code of Javascripts.) As shown in Fig. 4, *Tansuo* has seven components: *Builder*, *Fetcher*, *Parser*, *Form Handler*, *Fireeye*, *State Manager* and *Viewer*.

Builder. This component is at the core of *Tansuo*. It implements algorithm *BuildNavGraph*, and is responsible for driving the entire exploration process.

Fetcher. This component is responsible for fetching a page from the server side, upon *Builder*'s request.

Parser. This component is responsible for parsing a page, and for extracting static and dynamic links in the page.

Form Handler. This component is responsible for interacting with forms. Specifically, *Form Handler* is responsible for three tasks: (1) Obtaining parameter values either from a user interactively or by reading from a set of XML files; (2) Generating parameter combinations by using a combinatorial test generation tool, called *Fireeye* [22]; (3) Submitting a form with those combinations. Note that in the interactive mode, *Tansuo* stores user-provided values into an XML file, so that those values can be reused later.

One challenge for *Form Handler* is handling POST forms. URLs generated from POST form submissions contain only the *base* component of the URL shown in Fig. 1. The parameter name-value pairs in a POST request are not appended to the URL. Instead they are encoded in the HTTP request header. In order to distinguish different web pages generated from POST form submissions, *Tansuo* changes the *method* parameter value in the POST form to "GET". Doing so allows parameter name-value pairs to be appended to URLs generated from POST form submissions. Note that URLs with *query* components generated from POST forms may be rejected by some web applications if they do not implement a universal interface for handling GET and POST requests.

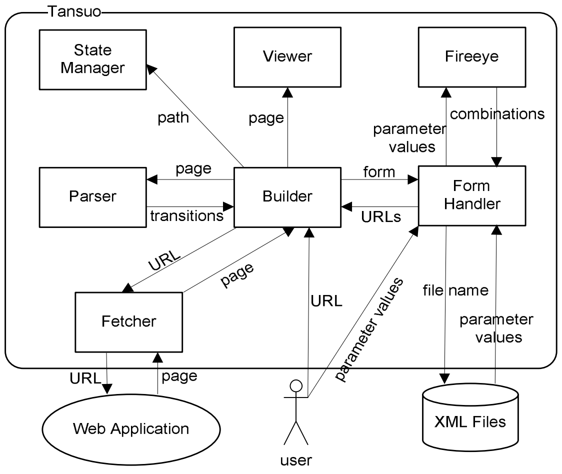


Figure 4: Tansuo's architecture

State Manager. The *Builder* calls *State Manager*, which is responsible for resetting the database to its initial state, and restoring the system state before a new path is explored. Note that we restore system states by re-visiting the sequence of pages that are on the stack.

Viewer. This component is responsible for displaying the page that is currently being explored by the *Builder*. When a user is asked to provide parameter values for a form, displaying the current page helps the user to understand the context.

Another feature of *Tansuo* is that it allows the user to specify the scope of exploration. Specifically, it allows the user to specify a base URL so that only links that begins with this base URL will be explored and included in the navigation graph that is generated.

5. Experiments

We used *Tansuo* to generate navigation graphs for five web applications and measured its runtime performance. We evaluated the completeness of the generated web navigation graphs on two of the five applications. We also compared *Tansuo* with WebSphinx and Link Checker Pro for all the subject applications.

5.1. Research questions

RQ1. How complete is a navigation graph built by *Tansuo*?

RQ2. How efficient is *Tansuo* in terms of time and memory usage?

RQ3. How does *Tansuo* compare with existing tools?

5.2. Metrics

For RQ1, the completeness of a generated navigation graph is measured by the number of missed nodes and edges when compared with a complete navigation graph generated by a manually performed static analysis of the source code. For RQ2, we evaluate the runtime performance of *Tansuo* by measuring the time taken to generate navigation graphs and the memory used during the navigation graph generation process. For RQ3, we compare *Tansuo* with two other tools - WebSphinx and Link Checker Pro - by recording the number of nodes and the number of edges generated by each tool to capture navigation structures of web applications.

5.3. Experimental Setup

Subject Applications: We used five web applications [7]: *Bookstore*, *Bug Tracking System (BugTrack)*, *Classifieds*, *Links* and *Portal* in our study. Table 1 shows some server-side characteristics of the subject applications, including the number of non-commented lines of code (NLOC), classes, methods, and branches in the five applications. Table 2 shows some client-side characteristics of the subject applications, including the number of forms, actions, parameters (params), the average number of parameters per action (APA), and the average number of values per parameter (AVP). Note that these client-side factors affect the size of navigation graphs, as well as the time for building navigation graphs of these applications. We note that all of the applications are implemented in JSP.

Table 1: Server-side characteristics of subject applications

Subject	**Characteristics**			
	NLOC	Classes	Methods	Branches
Bookstore	18385	27	925	4392
BugTrack	8094	13	438	1946
Classifieds	11599	18	618	2730
Links	8849	13	499	2074
Portal	17621	27	915	4084

Table 2: Client-side characteristics of subject applications

Subject	Characteristics				
	Forms	Actions	Params	APA	AVP
Bookstore	18	63	66	1.05	3.35
BugTrack	8	19	27	1.42	6.15
Classifieds	11	29	27	0.93	5.07
Links	11	24	26	1.08	5.77
Portal	19	39	95	2.44	3.40

Machine Configuration: The experiments were carried out on a computer with the following configuration: CPU: 1.66GHz, RAM: 2G, Hard disk: 80G. The machine was running Windows XP SP2, the Resin 2.1.8 web server, Apache 2.0.48, and the MySQL Server 4.1.

5.4. Results and discussion

RQ1: Completeness

We used *Tansuo* to generate navigation graphs for the five subject applications. Table 3 shows the characteristics of those graphs, including the number of nodes, edges, and connectivity (Conn). Connectivity is the average number of incoming and outgoing edges per node. All the graphs were generated using normal form input data, and those data were identified manually. Malicious input data, such as SQLInjection data and penetration data, were not used in these experiments. (Navigation graphs generated with both malicious input data and normal input data will likely be more complete than the navigation graphs generated with only normal input data.) The navigation graph of the largest application in terms of NLOC, *Bookstore* (with 18K NLOC), was represented with 93 nodes and 484 edges. The navigation graph for the largest application in terms of forms, actions and parameters, *Portal*, was represented with 80 nodes and 652 edges.

We evaluated the completeness of the navigation graphs of the two most complex subject web applications, *Bookstore* and *Portal*. (Other applications were not evaluated due to time constraints.) To perform this evaluation, we manually generated complete web navigation graphs, in terms of abstract nodes and edges, from the source code for *Bookstore* and *Portal*. Each JSP file in the web application source code was studied and abstract URLs were extracted. Abstract URLs were generated from either form actions or from the value of the attribute *href* of the *anchor (<a>)* tag. Each such abstract URL became a node in the resulting navigation graph. For each abstract URL, the corresponding page was studied to find transitions to other pages. These transitions became edges in the resulting navigation graph.

Table 4 shows the number of nodes and edges present in the manually generated graphs for *Bookstore* and *Portal*. The navigation graphs generated by *Tansuo* missed 12.1% of the nodes and 22.0% of the edges. After carefully studying the nodes and edges generated by the manual exploration and *Tansuo*'s exploration, we found that the reason for the missed nodes and edges is that *Tansuo* did not capture the navigation structures for page-flipping. For example, the *OrderGrid* page, in *Bookstore*, lists orders placed by a user. If total orders are no more than 20, all of them will be listed in one page. But, if a user places 25 orders, the first 20 orders will be listed in the current page and the last 5 orders will be listed in the second page. In this case, page-flipping is needed for users to browse all these orders. *Bookstore* places a link in the current page so that the user can navigate to the second page by clicking this link. Initially, there is no order listed in the *OrderGrid* page. During exploration, *Tansuo* only placed one order to drive the exploration for the reason of efficiency. As a result, there was only one order listed in the *OrderGrid* page, and *Tansuo* failed to capture the navigation structure related to page-flipping.

Table 3: Navigation graphs of five subject applications

Subject	Characteristics		
	Nodes	Edges	Conn.
Bookstore	93	484	10.17
Bug Track	43	175	7.85
Classifieds	50	313	12.53
Links	52	259	9.72
Portal	80	652	17.77

Table 4: Completeness results for Bookstore and Portal

Subject	Manual		Tansuo			
	Nodes	Edges	Nodes	%	Edges	%
Bookstore	97	596	93	95.9	484	81.2
Portal	91	836	80	87.9	652	78.0

RQ2: Efficiency

Table 5 shows the time taken to generate a navigation graph for each web application. From the results in Table 5, we see that the application state restoration process is the most time consuming activity. The time taken to restore the application state includes the time taken to reset the database and the time to re-exercise the path from the *home* page to the current page. A large number of submission tests for a form can significantly increase the state restoration time, since system state has to be restored before each submission test is performed (except for the first one).

We note that the exploration time of *Bookstore* is much higher than the other 4 web applications. The reason is that *Bookstore* stores images for books. When a large number of images are present in an application, heavy database access and image download, especially during the application state restoration process, dramatically increase the exploration time of *Tansuo*. The other 4 applications in our study did not contain a large number of images (e.g., a search result page for *Bookstore* contained 20 images, whereas a search result page for *Portal* contained no images). *Tansuo*'s exploration time for image-intensive applications can be reduced by ignoring image retrievals when retrieving a page.

Recall that *Tansuo* explores a web application in a depth-first manner. The maximum memory usage occurs at one of those back-up points, i.e., after *Tansuo* finishes exploring the current path, and right before it backs up to explore the next path. We recorded the memory usage at each of these back-up points and reported the highest of these values in Table 6. From Table 6, we see that, in general, the memory usage is consistent with the scale of the web navigation graph that is generated.

Table 5: Time (hours) taken to generate navigation graphs

Subject	**Total Time**	**State Restoration Time**
Bookstore	33.4415	27.7654
BugTrack	0.1321	0.0641
Classifieds	0.2999	0.2123
Links	0.1275	0.0581
Portal	1.2218	0.9519

Table 6: Memory usage (M bytes)

Subject	**Memory Usage**
Bookstore	42.6328
BugTrack	19.5625
Classifieds	39.0078
Links	19.4570
Portal	80.3554

RQ3: Comparison to other tools

We compared *Tansuo* with WebSphinx [8] and Link Checker Pro (LCP) [9]. Note that WebSphinx and LCP do not handle forms. (It would be better if the comparison were made to tools that handle forms. Unfortunately, we were not able to obtain access to such tools.) In addition, they do not make any page abstraction. For example, "*http://test.com/BookDetail.jsp?item_id=1*" and "*http:/test.com/BookDetail.jsp?item_id=2*" are identified as two different pages in their navigation graphs. If there are thousands of such pages in a web application, the navigation graphs generated by WebSphinx and LCP will be very large, while contributing little to represent the navigation structure of the application.

Table 7 shows the results of our comparison. Both WebSphinx and LCP generated similar numbers of nodes and edges, while *Tansuo* generated significantly more nodes and edges than WebSphinx and LCP. This suggests that the ability to interact with forms is vital to build high-coverage navigation graphs.

Table 7: Comparison to WebSphinx and LCP

Subject	**WebSphinx**		**LCP**		**Tansuo**	
	Nodes	Edges	Nodes	Edges	Nodes	Edges
Bookstore	11	11	11	11	93	484
BugTrack	7	7	7	7	43	175
Classifieds	15	16	9	9	50	313
Links	11	12	11	11	452	259
Portal	17	22	17	22	80	652

5.5. Threats to validity

Although our study investigates *Tansuo* with 5 medium to large web applications, the number, the size (in terms of NLOC), and the specific technologies (HTML, JSP, MySQL) of the subject applications prevent a generalization of our results to the entire domain of web applications.

We manually explored the source code to generate the web navigation graph for answering RQ1. Although extreme care was taken to accurately model the navigation graph, the human involved in the exploration process could have made errors when analyzing the source code, which may affect the completeness of the web navigation graphs generated by *Tansuo*.

6. Conclusion

In this paper, we have presented an approach to building navigation graphs of dynamic web applications. Our approach has two salient features. First, each node in a navigation graph represents a group of pages that are likely to display the same navigation behavior. Grouping pages allows us to reduce and bound the size of a navigation graph for practical applications, while still preserving the navigation structure. Second, a combinatorial strategy is employed to interact with forms. Specifically, when we encounter a form, we perform a pairwise set of submission tests on the form, in order to reach the dynamic pages that can be generated by the form. This can significantly reduce the number of submission tests that have to be performed while still achieving a high degree of coverage of dynamic pages. We have described a prototype tool, namely *Tansuo*, and have applied the tool to five open-source web applications. The results indicate that our approach is effective for generating navigation graphs of these applications.

There are a number of venues to continue our work. First, some combinations of parameter values may be invalid based on domain semantics. These combinations need to be excluded when we generate a pairwise submission test set. We will enhance *Tansuo* with constraint support so that invalid combinations can be specified and excluded during submission test generation. Second, we plan to develop automated or semi-automated techniques to help identify values of individual parameters. In particular, we intend to leverage existing work that handles similar problems in the context of web crawling. Third, we currently restore an application state by re-executing a path that previously reached the state, which can be time-consuming. We plan to explore the use of checkpoints to improve the time efficiency of state restoration. The challenge is to identify state objects that need to be included in a checkpoint in an application-independent manner. Finally, we plan to improve the user interface of our tool so that it is accessible to average users. It is our goal to make the tool publicly available on the Web.

Acknowledgements: We would like to thank Prof. Richard Carver at George Mason University for his comments on an earlier version of this paper. This work is partly supported by a grant (Award No. 60NANB6D6192) from the Information Technology Lab (ITL) of National Institute of Standards and Technology (NIST).

7. References

[1] W. Wang, S. Sampath, Y. Lei and R. Kacker, "An Interaction-based Test Sequence Generation Approach for Testing Web Applications", *Proc. of 11th Int'l IEEE HASE Symposium*, pp. 209-218, 2008.

[2] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): General Syntax", *http://labs.apache.org/webarch/uri/rfc/rfc3986.html.*

[3] D.M. Cohen, S.R. Dalal, J. Parelius and G.C. Patton, "The Combinatorial Design Approach to Automatic Test Generation", *IEEE Software*, 13(5): 83-88, 1996.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Transactions on Software Engineering*, 23(7): 437-444, 1997.

[5] D. R. Kuhn, D. Wallace, A. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Transactions on Software Engineering*, 30(6): 418-421, 2004.

[6] K. Burr, and W. Young, "Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage", *Proc. of Int'l Conf. on Software Testing Analysis and Review*, pp. 503-513, 1998.

[7] Open Source Web Applications with Source Code, *http://www.gotocode.com*, Feb. 3, 2009.

[8] R.C. Miller, and K. Bharat, "SPHINX: A Framework for Creating Personal, Site-specific Web Crawlers", *Proc. of 7th Int'l Conf. on WWW*, pp. 119-130, 1998.

[9] Link Checker Pro, *http://www.link-checker-pro.com*, Feb. 3, 2009.

[10] M. Benedikt, J. Freire, and P. Godefroid, "VeriWeb: Automatically Testing Dynamic Web Sites", *Proc. of 11th Int'l Conf. on WWW*, 2002.

[11] J. Madhavan, D. Ko, Ł. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy, "Google's Deep-Web Crawl", *Proc. of the VLDB Endowment*, 1 (2): 1241-1252, 2008.

[12] R. Cai, J.M. Yang, W. Lai, Y. Wang and L. Zhang "iRobot: An Intelligent Crawler for Web Forums", *Proc. of 17th Int'l Conf. on WWW*, pp. 447-456, 2008.

[13] S. Raghavan, and H. Garcia-Molina, "Crawling the Hidden Web", *Proc. Of 27th Int'l Conf. on Very Large Data Bases*, pp. 129-138, 2001.

[14] B. He, M. Patel, Z. Zhang, and K.C. Chang, "Accessing the Deep Web", *Communications of the ACM*, 50 (5):94-101, 2007.

[15] G.D. Lucca, A. Fasolino, F. Faralli, and U.D. Carlini, "Testing Web Applications", *Proc. of the 18th ICSM*, pp. 310-319, 2002.

[16] P. Godefroid, "Model Checking for Programming Languages using VeriSoft", *Proc. of 24th ACM Int'l Symp. on POPL*, pp. 174-186, 1997.

[17] A. Andrews, J. Offutt, and R. Alexander, "Testing Web Applications by Modeling with FSMs", *Journal of Software and Systems Modeling*, 4 (3): 326-345, 2005.

[18] C.-H. Liu, D.C. Kung, P. Hsia, and C.-T. Hsu, "Structural Testing of Web Applications", *Proc. of 11th ISSRE*, pp. 84-96, 2000.

[19] F. Ricca, and P. Tonella, "Analysis and Testing of Web Applications", *Proc. of 23rd ICSE*, pp. 25-34, 2001.

[20] M. Grindal, J. Offutt, and S.F. Andler, "Combination Testing Strategies: A Survey", *Software Testing, Verification, and Reliability*, 15 (2): 167-199, 2005.

[21] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPO-D: Efficient Test Generation for Multi-way Combinatorial Testing", *Software Testing, Verification, and Reliability*, 18 (3): 125-148, 2007.

[22] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A General Strategy for T-Way Software Testing", *Proc. of Int'l Conf. and Workshops on the Engineering of Computer-Based Systems*, pp. 549-556, 2007.