Polytechnic UNIVERSITY
Brooklyn · Long Island · Westchester

Testing Web Database Applications

Yuetang Deng   Phyllis Frankl   Jiong Wang

Department of Computer and Information Science

Technical Report
TR-CIS-2004-01
04/28/2004

Testing Web Database Applications

Yuetang Deng
Phyllis Frankl
Polytechnic University
Six Metrotech Center
Brooklyn NY 11201 *
{ytdeng, pfrankl}@cis.poly.edu

Jiong Wang
Information Science and Technology Dept
Donghua University
Shanghai, P. R. CHINA
wangj30@mail.dhu.edu.cn

Abstract

Commercial, scientific, and social activities are increasingly becoming dependent on Web database applications. New testing techniques that handle the unique features of these systems are needed. To that end, we have extended AGENDA, a tool set for testing relational database applications, to test web database applications. Application source code is analyzed to extract relevant information about the URLs and their parameters. This information is used to construct and simplify a graph in which nodes represent URLs and edges represent links between URLs. A set of paths through the graph is selected and test cases are generated for each path. The extracted information about the parameters to each URL (e.g., values that an application user would enter into a form), is used to guide AGENDA to generate inputs for the URLs. The URLs on a path and their inputs are stored in an XML file, which is then automatically executed. The current implementation is targeted toward web applications written as Java Servlets and uses an algorithm based on cyclomatic complexity to generate paths. Preliminary empirically evaluation based on the TPC-W benchmark is presented.

1. Introduction

Along with the tremendous growth of the World Wide Web (WWW) over the past decade, there has been a growth and transformation in the nature of services accessible over the Web. Many new services are web sites that are driven from data stored in databases. Examples of such web applications include services that provide access to large data repositories, E-commerce applications such as online stores, and business-to-business(B2B) support products. It is essential that these applications function correctly and provide suitable protection to customer data and enterprise assets.

*Supported in part by NSF Grants CCR-9988354

Most web database applications consist of three layers of application logic. At the base is a Database Management System (DBMS) and a database. At the top is the client web browser used as an interface to the application. Between the two lies most of the application logic, usually developed with a web server-side scripting language or Java extended with library that can interface with the DBMSs, and can decode and produce HTML used for presentation in the client web browser.

Application development is often driven by time-to-market concerns, with insufficient attention to correctness and security concerns. The Business Internet Group of San Francisco undertook a study to capture, assess and quantify the integrity of Web applications on 41 specific Web sites under the management of the U.S. Government. According to the report, of those 41 sites, 28 sites contained Web application failures [2].

Web applications pose new challenges to software testing. They are highly dynamic and interactive, with unpredicatable control flow, as users jump to arbitrary URLs [17]. Although they are based on the stateless HTTP protocol, application behavior is often influenced by the state of the database with which the interaction interacts and/or the value of stored cookies. They have a vast user base with varying levels of technical sophistication. (Among these, there may be malicious users submiting unexpected inputs to the applications, with the intent of comprimising other users and/or the enterprise deploying the application.)

To illustrate some of the difficulties of testing a web application, consider an online bookstore with a ”search for books” page prompting the user to search for a book by entering keywords and a price threshold. The application generates an SQL SELECT statement based on the input values and submits that query to the database. Suppose, due to a programming error, the a faulty SQL query is generated, which only considers the keywords and ignores the price threshold. Then the response to the user might show some extra books which do not meet the user's requirements (i.e., the price is too high). We would like to expose this fault during testing. To do so, we need a test case which models a user goes to the "search for books" page and entering keywords, say $k1, k2$, and a price threshhold, say p. Note that in order to get to the "search for books" page, the user may need to pass through a sequence of other pages, such as a login page. In order to expose the fault, the database must be populated with at least one book that matches keywords $k1, k2$ and has price greater than p. Thus, effective testing of web applications requires integration of techniques for navigating through a web site, supplying values for forms, selecting links, etc., with techniques for testing database applications.

To our knowledge, no previous approach to testing web applications explicitly considers the effect of the database(s) the application is using. Since the outputs generated in response to a given input are strongly affected by the contents of the database, more thorough testing can be achieved by controlling and observing the database state, along with the user inputs and application outputs. In addition, in some cases it will be easier to check that an application has responded correctly to a given input to a given form by checking the database state, rather than by checking the HTML page output by the application.

Most prior research on testing web applications takes a black box testing approach, adapting crawler technology to discover links/forms in the application under test and generating (or replaying) values for form inputs. In this paper, we describe a white box approach, in which application source is analyzed to discover the salient features of the application. This information is used, in concert with the AGENDA [4, 5] database testing tool, to populate the database with appropriate values and select inputs that are related to the database contents.

The white box approach can potentially provide:

1. better coverage of those aspects of the application that might not be "found" by a crawler or by user data, and
2. more appropriate input values for forms.
3. better targeting of test effort, by using static analysis to determine that certain tests are necessary and/or unnecessary.

One disadvantage of white box testing is that the tools must be targeted to particular source languages. This is a particular problem in the domain of web applications, which is characterized by a wide variety of languages and rapidly changing technology. In this paper, we describe a prototype tool targeted to applications written as Java Servlets, subject to certain limitations. However, the underlying techniques are broadly applicable.

The rest of the paper is organized as follows. Section 2 introduces the background. Section 3 presents our techniques to test web database applications and describes our prototype testing tool implementation. Section 4 presents experience applying our technique to the TPC-W Benchmark. Section 5 describes related work. Section 6 concludes with our contributions and discusses the future work.

2. Background

2.1. AGENDA Tool Set

In previous work [4, 5], the authors and others have discussed issues arising in testing database systems, presented an approach to testing database applications, and described AGENDA, a set of tools based on this approach. In testing such applications, the states of the database before and after application's execution play an important role, along with the user's input and the system output. A framework for testing database applications was introduced. A complete tool set based on the framework was prototyped. The components of this system are: Agenda Parser, State Generator, Input Generator, State Validator, and Output Validator.

AGENDA takes as input the database schema of the database on which the application runs; the application source code; and "sample value files", containing some suggested values for attributes. The tester interactively selects test heuristics and provides information about expected behaviors of test cases. Using this information, AGENDA populates the database, generates inputs to the application, executes the application on those inputs and checks some aspects of correctness of the resulting database state and the application output. This approach is loosely based on the Category-Partition method [13]: the user supplies suggested values for attributes, partitioned into groups, which we call data groups. This data is provided in the sample value files. The tool then produces meaningful combinations of these values in order to fill database tables and provide input parameters to the application program. Data groups are used to distinguish values that are expected to result in different application behaviors. For example, in a payroll system, different categories of employees may be treated differently. Additional information about data groups, such as probability for selecting a value from the list of values that follows, can also be provided via sample value files.

Using these data groups and guided by heuristics selected by the tester, AGENDA produces a collection of test templates representing abstract test cases. The tester then provides information about the expected behavior of the application on tests represented by each test template. For example, the tester might specify that the application should increase the salaries of employees in the "faculty" group by 10% and should not change any other attributes. In order

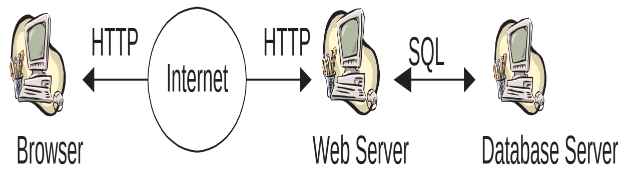

Figure 1. Typical Web Database Application Configuration

to control the explosion in the number of test templates and to force the generation of particular kinds of templates, the tester selects heuristics. Finally, AGENDA instantiates the templates with specific test values, executes the test cases and checks that the output and new database state are consistent with the expected behavior indicated by the tester.

2.2. Web application programming model

Web content is increasingly generated dynamically, a departure from the early days of the Web when virtually all content consisted of static HTML, or image files. Figure 1 is a typical web application configuration. Dynamic Web content is typically generated by a front-end Web server and a back-end database. The (dynamic) content of the site is stored in the database. The application logic provides access to that content. The client sends an HTTP request to the Web server containing the appropriate URL and some parameters. The Web server causes the application logic to be executed. Typically, the application logic issues a number of queries to the database and formats the results as an HTML page. The Web server finally returns this page as an HTTP response to the client. The application logic may take various forms, including scripting languages such as PHP, that execute as modules in the Apache Web Server; Microsoft Active Server pages that are integrated with Microsoft's IIS server; and Java-based systems that execute in a separate Java Virtual Machine. In a static Web page, the content is determined at the time the page is created. When any user accesses a static page, it always displays the same information. In a dynamic Web page, the content varies based on user inputs and data retrieved from external sources. In this article, the term *data based Web pages* [12] refers to dynamic Web pages that derive some or all of their content from data files or databases. It is useful to separate data based Web pages from pages created using client-side scripting technologies such as JavaScript or VBScript. These technologies support tasks like data verification, displaying new browser windows, and providing animated graphics and sounds, rather than interaction with files or databases.

A data based Web page is requested when a user clicks a hyperlink or a "Submit" button on a web page form. If the request comes from clicking a hyperlink, the link must either specify a web server program or a web page that calls a web server program. If the request is generated when the user clicks a web page form's "Submit" button, the web server program usually uses the form inputs to dynamically create a database query, based on parameter values supplied via the URL's parameters. In some cases, the program performs a static query. Although this query requires no user inputs, the results may vary depending on when the query is made. In either case, the web server program is responsible for formatting the query results by adding HTML tags. The web server then sends the program's output back to the client's browser as a web page.

2.3. Java Servlet and JDBC

Our tool is currently targeted toward web applications written as Java Servlets using JDBC for database access. A Servlet is a Java program that runs as part of a network service, typically an HTTP server and responds to requests from clients. The most common use for a Servlet is to extend a web server by generating web content dynamically. For example, a client may need information from a database.

Servlets are typically built upon a request-response model by extending the `doGet`/`doPost` method to perform the particular actions needed in response to an HTTP get or post request. A Servlet can be written that receives the request, gets and processes the data as needed by the client and then returns the result to the client.

A Servlet may access a database via Java Database Connectivity (JDBC). JDBC is part of the Java Development Kit which defines an application programming interface for Java for standard SQL access to databases from Java programs.

2.3.1 Example

Figure 2 shows a sample HTML file, which has one hyperlink *index.html*, hyperlink (anchor) tags are used to link the text/graphic in one web page to another web page. The HTML file has a FORM with two input parameters, with names *uname* and *upasswd*. When the user click the SUBMIT button, the form action specifies a web server program (in this case, it is a UserValidate Servelet,) to deal with the HTTP request. Figure 3 is the UserValidate Servlet which takes an input, accesses a database, and outputs different HTML links for different type of inputs. It has two private data members `username` and `password`. The Servlet deals with the HTTP request in the `doGet` method. Here we briefly explain this method. AgendaDbBean in line 1 is

```
<html>
  <head> <title> University Information System  </title> </head>
  <body>
    <A HREF="index.html">Return to university home page </A> <BR>
    <FORM ACTION="UserValidate_servlet" METHOD="GET" >
       Please Enter your ID and Password to log in  <BR>
        ID       <INPUT TYPE=TEXT NAME="uname" VALUE="" SIZE=20> <BR>
        PASSWORD <INPUT TYPE=PASSWORD NAME="passwd" VALUE="" SIZE=20> <BR>
                 <INPUT TYPE="SUBMIT" VALUE="Login">
     </FORM>
  </body>
</html>
```

Figure 2. Sample HTML file

a Java Bean class which handles the access to the database. Java Bean is a component technology for Java that allows developers to create reusable objects that can be shared among applications. From line 2 to line 3, the Servlet sets up the response's PrintWriter to return text to the client. Parameters are passed in via name-value pairs. In line 5 and line 6, the values of two parameters, with names *uname* and *upasswd* in Figure 2, are passed to this Servlet and stored in instance variables *username* and *password*. From line 7 to line 9 the Servlet generates some necessary HTML tags in order to respond to the client. The database has a table called `user` whose attributes include `name`, `password`, and `type`. In line 10 the Servlet generates an SQL SELECT statement to query the database in order to determine the user's type (student or faculty) based on the values of variables `username` and `password`. These variables in the host language that are used as parameters in SQL queries are called *host variables*. From line 11 to line 17, if the inputs matches some record in the database, it returns the type and outputs a URL link (student.html or faculty.html) for this type. Line 18 is the exception handler. In line 19 the Servlet generates a home URL link (index.html). Finally in line 20 and line 21 it generates some more necessary HTML tags for a complete HTML page and closes the writer; the response is done. A sample output for legal student inputs *uname* and *upasswd* is shown in Figure 4.

2.4. TPC-W Benchmark

We illustrate our technique with examples based on $TPCBenchmark^{TM}$ W (TPC-W) [15], an eBusiness benchmark that models an online bookstore. There are 8 tables: customer, address, order, order_line, credit_info, items, author, and country in the database. The order_line, order and credit_info tables store information about orders that have been placed. The item and author tables contain information about the books and authors. The customer and address tables include information about customers.

There are 14 web interactions defined in the TPC-W specification. 6 of them are read only, while the other 8 update the database. The read only interactions include accessing the homepage, listing of new products and best sellers, request for product detail, and two interactions involving searches. Read-write interactions include user registration, updates to the shopping cart, two purchase interactions, two involving order inquiry and display, and two administrative tasks.

3. Web database application testing techniques

A test case for a web application is a sequence of pages to be visited plus the input values to be provided to pages containing forms [14]. Our approach to testing web applications involves the following steps:

1. First, useful information such as URL links and inputs for each URL is extracted from the application source. Also URLs are partitioned into two categories according to their content: static page, and data-based page.

2. An application graph, where nodes represent URLs and edges represent URL links, is generated and then simplified according to URL link types.

3. Some paths through the graph are selected. Each path corresponds to one or more test case.

4. For each path, AGENDA is used to generate inputs for each URL. The path, along with these inputs constitutes a test case. An XML file representing the test case is generated.

5. the test case in the XML file is automatically executed and AGENDA checks the new database state and the output pages.

```
// Example 1: A Servlet which has different HMTL links in the output
based on inputs and database state.


public class UserValidate extends HttpServlet {
  private String username, password;

  public void doGet(HttpServletRequest request,
         HttpServletResponse response) throws ServletException, IOException
  {
1.     AgendaDbBean con = new AgendaDbBean();
2.     response.setContentType("text/html");
3.     PrintWriter out =response.getWriter();
4.     try {
5.       username = request.getParameter("uname");
6.       password = request.getParameter("upasswd");
7.       out.println("<html> <head>");
8.       out.println("<title> University Information System Login </title>");
9.       out.println("</head> <body>");
10.      String sql="select type from user where name="+username+ " and password="+password;
11.      ResultSet  rs = con.execSQL(sql) ;

12.      if (rs.next()) {
13.        int type = rs.getInt("type");
14.        if (type ==1 ) {
15.           out.println("<A HREF ="student.html"> Welcome to the student page <br></A>");}
16.        else if (type ==2) {
17.           out.println("<A HREF ="faculty.html"> Welcome to the faculty page </A>");}
      }
    }
18.    catch(Exception e) {out.println("Exception in database operation"); }

19.    out.println("<A HREF ="index.html"> Return to main page </A>");
20.    out.println("</body> </html>");
21.    out.close();
  }
}
```

Figure 3. Sample Servlet

The remainder of this section describes these steps in more detail and describes the prototype tool we have developed based on this approach. The tool prototype is currently targeted to the Java Servlet model, using JDBC for database access, and makes some assumptions about the programming style, detailed below. However, the basic technique is applicable to other more general Servlet styles and to other web application languages. The most fundamental limitation is the assumption that URLs are static and can be extracted from the Servlet source code. Some applications construct URLs dynamically based on user input. In such cases, our analysis will be incomplete, and consequently, some potential behaviors of the application will not be tested. Future work will explore integration of our white box (static analysis) approach with black-box crawler based approaches, to deal with such situations.

3.1. Information extracted from the application source

We extract the following information from the Servlet source:

1. URL content type, which indicates the output of this Servlet is a static page, or data-based page; request type (GET/POST).

2. URL links information, which includes all the other URLs that can be reached from the current page.

3. HTML form and its field information, if there are HTML forms in the output of this Servlet. We also extract the name/type for each field and form submission.

4. Parameter information, which consists of all the name-value pairs passed to the Servlet.

```
<html>
  <head>
     <title> University Information System Login </title>
  </head>
  <body>
    <A HREF="student.html">Welcome to the student page </A> <BR>
    <A HREF="index.html">Return to main page </A>
  </body>
</html>
```

Figure 4. Possible Output HTML file in the Sample Servlet

To extract this information, we currently assume that the Servlet follows some widely-used conventions. The Servlet's `doGet()` and `doPost()` methods, executed in response to an HTTP get or post request, are analyzed. The names in name-value pairs, representing input parameters to the Servlet are identified in the arguments to `getParameter()` method calls sent to the `HttpServletRequest` objects that are parameters to the Servlet. The URL links we consider include tradition HTML anchors and form submissions. The Servlet usually gets the response's PrintWriter, and generates the content including HTML anchors and HTML form in its println() method. We extract information about HTML anchors and HTML forms in the println() method.

We utilize SOOT [8] in our Java Servlet analysis. SOOT is a tool for analyzing and transforming Java bytecode. First SOOT transforms the Servlet class from Java bytecode to Jimple, a typed 3-address intermediate representation suitable for optimization and analysis. Analysis of Jimple is simpler than analysis of Servlet source. This approach also allows analysis when source code is not available.

The inputs for each Servlet are extracted from getParameter() method in the HttpServletRequest object in the Jimple file. The HREF tags generated in the PrinterWriter object are added to the output URL links set. Also we extract for HTML form field names and types and form submission information from each page. If there is a database access (via JDBC method) in the Servlet, this page is classified as data based page. Otherwise it is treated as a static page. Since static pages are independent of inputs, they only need to be tested once; If a page data-based, we need to test it thoroughly. Our technique focusses attention on this more difficult problem of testing dynamic pages.

In Example 1, The type of output HTML page is data-based, since it accesses the database via JDBC methods defined in the bean class. The input parameters are uname and upasswd. Totally there are three possible URL links in the output HTML page: student.html, faculty.html, and index.html. There is no form defined in the output page.

3.2. Web application graph generation and path selection

We next use some of the information extracted from the source to construct a graph representing the Web application. Each node is a URL and there is a link from URL A to URL B if URL A produces a link to URL B in the HTML page it generates.

The graph may be large and complex and may have references to URLs outside of the application. We simplify the graph by removing such external links. If a link has a domain address which differs from the domain address that web application runs on, the link is removed. (If testing paths that leave the application then return is deemed to be important, such links could selectively be left in the graph at the cost of increased complexity, provided that their targets are available for analysis.) To further simplify the graph, static pages that have been tested by other means can also be removed, as long as the graph does not become disconnected. The graph for TPC-W benchmark is shown in Figure 5. Figure 6 lists the Servlets corresponding to each node in the graph and their classification.

Even this simplified application graph might still have a lot of paths and loops. The next step is selection of a set of paths that will be tested. In our current implementation a path selection algorithm based on cyclomatic complexity is applied to generate some interesting paths. Other path selection techniques could be applied as well.

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. The general formula to compute cyclomatic complexity is $V(G)=E-N+2$, where E is the number of edges, and N is the number of nodes in the graph. The cyclomatic number gives the number of independent paths, called basis paths, through the control flow graph. This means that cyclomatic number is precisely the minimal number of paths that can, in linear combination, generate all possible paths through the application. The algorithm to identify a set of basis paths we utilize is from [16]. Note that the set of basis paths is not unique.

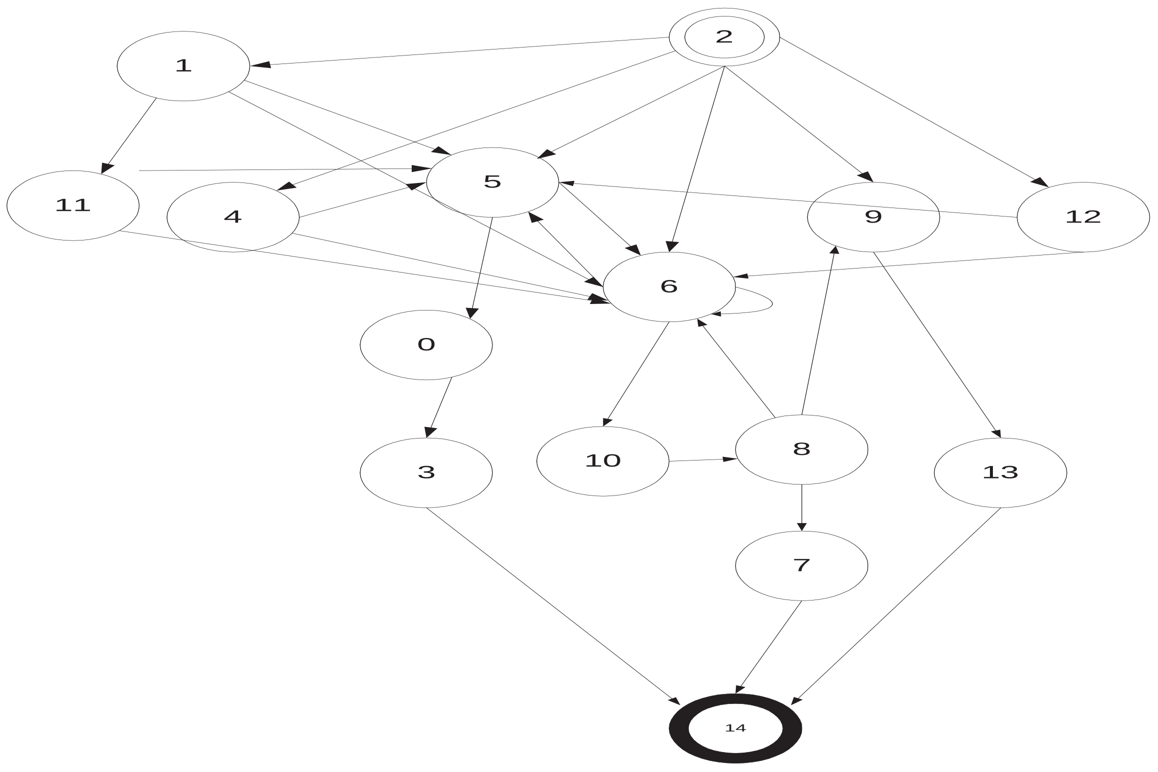

Figure 5. TPC-W Web Application Graph

id	web interaction	type
0	tpcw_admin_request_Servlet	data-based
1	tpcw_search_request_Servlet	data-based
2	tpcw_home_interaction	static
3	tpcw_admin_response_Servlet	data-based
4	tpcw_best_sellers_Servlet	data-based
5	tpcw_product_detail_Servlet	data-based
6	tpcw_shopping_cart_interaction	data-based
7	tpcw_buy_confirm_Servlet	data-based
8	tpcw_buy_request_Servlet	data-based
9	tpcw_order_inquiry_Servlet	data-based
10	tpcw_customer_registration_Servlet	data-based
11	tpcw_execute_search	static
12	tpcw_new_products_Servlet	data-based
13	tpcw_order_display_Servlet	data-based

Figure 6. TPCW web interactions

3.3. Input generation

Having selected paths to test, the next step is to generate inputs for each URL on a path. The parameters for a URL are name-value pairs. The name-value pairs are divided into three different types:

1. TYPE A: The name-value pairs are input fields in the Form. In this case we need to generate input for the field.
2. TYPE B: The name-value pairs are passed from the previous page. In this case we do not need to generate input.
3. TYPE C: The name-value pairs are generated in the application. We do not need to generate input.

Usually the start page does not have input. For a parameter in other pages in a path, we distinguish these three cases in the following way: Type A, if the parameter name appears as a field name in a HTML form in the previous page, we do generate values for this parameter as input. Type B, if the parameter name appears in the previous URL in the path, we think this value of this parameter in the current URL is passed from the previous page. Type C, If the parameter does not match the above two cases, we assume it belongs to TYPE C.

We only need to generate inputs for the TYPE A name-value pairs. We modified the AGENDA Input Generator tool for this purpose. In a web database application, some parameters for a URL are passed to SQL queries as host variables. Name-value pairs are associated with some attributes/values via host variables in the tables defined in the schema. Other parameters might not be associated with any tables. In our previous work Agenda generated values for the input host variables in SQL queries. Currently, we associate the parameters names in the URL to host variables in the SQL queries manually. Eventually, by utilizing program analysis techniques, the mapping could be partially automated. For those parameters which are not associated with any host variable in the SQL queries, we create a special table and make a one-to-one mapping between attribute name in the special table and these parameters. After these steps, the AGENDA Input Generator can generate inputs for web applications.

Note that when generating inputs, we need to consider the database state in order to explore various situations. For example, in the TPC-W benchmark, the order inquiry scenario needs two inputs: user id and password. There are several situations to consider:
1. invalid user id and password
2. valid user id, invalid password
3. valid user id and password, no order exists
4. valid user id and password, one order exists
5. valid user id and password, multiple orders exist

This is achieved by dividing sample values into different groups and selecting appropriate heuristics in the Agenda Input Generator. Leveraging the power of AGENDA to generate inputs for web application forms allows the tester to automatically explore a wide variety of situations the web application could face.

3.4. Test Case as an XML file

XML provides a standard, self-describing data interface which is a good mechanism to pass information in many applications. A test case is organized as an XML file. An example is shown in Figure 7. A *test* tag corresponds to a test case, which includes one or more *step* tags. A *step* tag corresponds to a URL link, which includes a *url* tag corresponding to a link, a *method* tag corresponding to a request type (GET or POST), and *parameter* tags optionally. A *parameter* tag, which includes a *name* tag and a *value* tag, corresponds to a parameter passed to the URL as name-value pair.

In Figure 7 there are three URLs: TPCW_home_interaction, TPCW_order_inquiry_Servlet, TPCW_order_display_Servlet in the path. The request type is GET for all of them. The first two URLs do not have parameters. The last URL has two parameters, UNAME, with value *myid*, and PASSWD, with value *mypassword*.

3.5. Test execution

There are two approaches to explore the web application. The first approach is to use a standard browser or web crawler to perform the exploration. The second approach is to implement the exploration using freely available HTTP libraries, DOM interfaces and JavaScript interpreters. Most of the previous work is based on the first approach and some also have techniques for automatically filling forms. In this paper we choose the second approach. Although the implementation of this approach is more complicated than that of the first approach, it is more flexible and provides the full functionality needed in many applications.

Web services, network-enabled appliances and the growth of network computing continue to expand the role of the HTTP protocol beyond user-driven web browsers, while increasing the number of applications that require HTTP support. The Jakarta HttpClient component provides an efficient, up-to-date, and feature-rich package implementing the client side of the most recent HTTP standards and recommendations [10]. We integrate the open source Jakarta HttpClient for the automatic test execution in our AGENDA system.

After a test case is generated, we utilize an XML parser to parse the XML file, extract information about URL link name, request method, and name-value pairs as parameters. Then this information is fed to HttpClient to get the response. The general process for using HttpClient consists of a number of steps : Create an instance of HttpClient; Create an instance of one of the methods (GET or POST). The URL to connect to is passed in to the the method constructor; Tell HttpClient to execute the method; Read the response; Release the connection; Deal with the response.

4. Case study

We performed a small case study based on the TPC-W benchmark, which models an online bookstore. Cain and Rajwar implemented the TPC-W benchmark using Java Servlet [3]. There are over 6000 LOCs in the implementation including comments. Our case study is based on this open source implementation. Each of the 14 web interactions is implemented as a Java Servlet class and is shown in Figure 6. The third column in the table shows that the output of the Servlet is a static HTML page or a data-based page. The tool extracts all the static 14 URL names. There are static 68 parameter names in the 14 URLs. There is a loop inside which there are two parameters whose names are dynamically generated. This information can only be read during runtime.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
 <test>
   <step index="0">
       <url>http://localhost:8080/tpcw/TPCW_home_interaction</url>
       <method>get</method>
   </step>
   <step index="1">
       <url>http://localhost:8080/tpcw/TPCW_order_inquiry_Servlet</url>
       <method>get</method>
   </step>
   <step index="2">
        <url>http://localhost:8080/tpcw/TPCW_order_display_Servlet</url>
       <method>get</method>
       <parameter>
            <name>UNAME</name>
            <value>myid</value>
       </parameter>
       <parameter>
            <name>PASSWD</name>
            <value>mypasswd</value>
       </parameter>
   </step>
 </test>
```

Figure 7. Test Case as XML file

Figure 5 is the simplified TPC-W application graph. Node 2 is the TPCW-home-interaction page, which is the entry of the graph. We add a dummy exit node (Node 14). If a URL has no out links, then we add an edge from this URL to the exit node. Totally there are 28 edges in the graph. The cyclomatic complexity V(G) = 28-14+2 = 16.

Table 8 lists the 16 basis paths generated on structured testing criterion. These paths are all feasible. They include some typical activities that could happen in a web bookstore. For example, path 0 is a scenario for order inquiry. It has three URLs (2, 9, 13). Table 6 shows that the three pages are: a home page, an order inquiry page where a customer can input his id and password, and an order display page where the recent order for the given customer is displayed. A test case which corresponds to path 0 in Figure 8 is shown in Figure 7.

Path 9 is a scenario for buying books. It includes five pages: a home page, a shopping cart page where the customer can add/update/delete items, a customer registration page where both new customers and existing customers can input identification information, a buy request page where the customer can input billing and address information, and a buy confirm page.

Path 12 is a scenario for an administration task. It includes five pages: a home page, a search request page where the user can search books by title/author/subject, a search result page, an admin request page where the administrator can update the book information such as price and image, and an admin confirm page that shows the result of updates. The Servlet for searching for a book (node 11) generates different SQL queries depending on what kind of search is being done (title/author/subject). This suggests that at least three different test cases should be used for this path. AGENDA's test selection heuristics, with appropriate selection of data groups, could help assure that these different situations are covered.

Some paths might not happen in a normal execution. But it might be a good test case for testing the application security. We will discuss this issue in our future work.

We ran our tools on a Pentium 1.5Ghz laptop with 512M main memory. The time for analysis and building the web graph is 3725 ms. The time to generate all the paths is 40 ms. The time to run the tool for the 16 paths are shown in Figure 8. The column labeled "input" gives the time to generate one test case for the path. The column labeled "XML" is the time to generate the XML file for the test case. The column labeled "execution" is the time to execute the test case. The last row indicates the average run time for the 16 paths. As shown in the table, the techniques usually works in a few seconds.

path	URLs	input	XML	execution
0	2,9,13	1449	292	665
1	2,5,0,3	1765	457	835
2	2,12,5,0,3	2039	501	904
3	2,4,5,0,3	2050	451	969
4	2,6,5,0,3	1990	506	1047
5	2,1,5,0,3	1693	461	917
6	2,5,6,5,0,3	1925	550	1188
7	2,12,6,5,0,3	2256	564	1141
8	2,4,6,5,0,3	2190	516	1130
9	2,6,10,8,7	5154	575	1592
10	2,6,6,5,0,3	2100	534	1262
11	2,1,6,5,0,3	1910	512	1126
12	2,1,11,5,0,3	2206	469	1090
13	2,6,10,8,6,5,0,3	3136	748	1790
14	2,6,10,8,9,13	2227	640	1256
15	2,1,11,6,5,0,3	2387	584	1243
avg		2280	523	1135

Figure 8. Path Generation and Overhead(ms)

5. Related work

Liu et al. [11] propose WebTestModel, which considers each web application component as an object and generates test cases based on data flow between those objects. Ricca and Tonella [14] propose a model based on the Unified Modeling Language (UML), to enable web application evolution analysis and test case generation. Both these techniques, in essence, extend traditional path- based test generation and data flow adequacy assessment to the web application domain; the second also builds on the existence of popular UML modelling capabilities.

Reference [17] defines a generic analysis model that characterizes both static and dynamic aspects of web based applications. This technique is based on identifying atomic elements of dynamic web pages that have static structure and dynamic contents.

Elbaum [6] et al explore the notion that user session data gathered as users operate web applications can be successfully employed in the testing of those applications, particularly as those applications evolve and experience different usage profiles

HED [7] is a model for the maintenance of web database applications. HED model decomposes a web database application into three diagrams, hyperlink diagrams, entity-relationship diagrams, and data-flow diagrams, which are used to represent different aspects of a web database application. Based on the HED model, the program files affected by a program change can be identified precisely via the structure and database analyses. In addition, a maintenance tool is implemented to demonstrate the capability of the HED model.

VeriWeb [1] is a tool for automatically discovering and systematically exploring web-site execution paths that can be followed by a user in a web application. Unlike traditional crawlers which are limited to the exploration of static links, VeriWeb can navigate automatically through dynamic components of web sites, including form submissions and execution of client-side script.

Based on a number of software-testing techniques including dynamic analysis, black-box testing, fault injection, and behavior monitoring, WAVES [9] is a tool for assessing web application security.

6. Discussion and future work

The procedure of most previous work is like this: download a candidate page, analyze its links, choose one of the links as the next candidate page. Repeat this procedure until some condition is satisfied.

Instead, in this paper, we use static analysis techniques to extract useful information in a web database application; construct the application graph; systematically generate interesting paths based on the graph. A test case is organized as an XML file and automatically executed.

Compared with the previous work, our white-box approach has several potential advantages. First, the static analysis techniques might find more candidate URL links, since some URL links are dynamically generated, based on inputs, thus might not be found in a crawling approach. Second, since the path generation algorithm is based on the structured testing criterion, each decision is evaluated independently at least once; thus our approach might explore some paths that might not be detected in black box approaches. Third, the test case as XML file greatly facilitates the automatic execution. Moreover, in our approach the database state is carefully constructed to include many different possible situations and inputs to URLs are generated to exercise a variety of situations.

Currently, our analysis focuses on web database applications implemented as Java Servlets and HTML pages. There are some limitations in our analysis tool.

1. We assume the URL links are static text in the Servlet. The URL links can be extracted from the Servlets directly. We did not consider URL rewriting in the current implementation, and more generally, the URL links including URL name could be generated dynamically.

2. Parameters are represented as name/value pairs. If the parameter names are dynamically generated in the application, static analysis can not figure out this information.

3. The HTML page is generated directly in the doGet/doPost method. It is possible the doGet/doPost could call other methods to generate the HTML page content.

Preliminary empirical evaluation based on the TPC-W benchmark is presented and demonstrates that our approach is efficienct enough to allow automatic generation and execution of large test suites for large application programs. To overcome some of the limitations, we are exploring hybrid techniques that combine static analysis, where it's practical, with crawler based approaches for URLs that cannot be deduced from the source code. Future work will also consider the role of session and cookies. Finally we plan to extend our tool to test web application security.

References

[1] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *11th International World Wide Web Conference*. ACM Press, 2002.

[2] BIGSF. *Government Web Application Integrity*. The Business Internet Group of San Francisco, 2003.

[3] H. Cain and R. Rajwar. An architectural evaluation of java tpc-w. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*. IEEE Computer Society Press, 2001.

[4] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker. A framework for testing database applications. *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pages 147–157, Aug. 2000.

[5] D. Chays, Y. Deng, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker. An agenda to test relational database application. *Journal of Software Testing, Verification and Reliability*, Mar. 2004.

[6] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proceedings of the 25rd international conference on Software*. IEEE Computer Society Press, 2003.

[7] C.-L. Hsu, H.-C. Liao, J.-L. Chen, and F.-J. Wang. A web database application model for software maintenance. In *The Fourth International Symposium on Autonomous Decentralized Systems*. IEEE Computer Society Press, 1999.

[8] http://www.sable.mcgill.ca/soot/. *Soot: a Java Optimization Framework*. 2002.

[9] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *12th International World Wide Web Conference*. ACM Press, 2003.

[10] Jakarta Commons HttpClient. *http://jakarta.apache.org/commons/httpclient/*. 2003.

[11] C.-H. Liu, D. Kung, P. Hsia, and C.-T. Hsu. Structural testing of web applications,. In *11th International Symposium on Software Reliability Engineering*. IEEE Computer Society Press, 2000.

[12] M. Morrison and J. Morrison. *Database-Driven Web Sites*. 2002.

[13] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31(6):676–686, June 1988.

[14] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd international conference on Software*. IEEE Computer Society Press, 2001.

[15] Transaction Processing Performance Council. *TPC-Benchmark W*. www.tpc.org, 2002.

[16] A. H. Watson. *Structured Testing: Analysis and Extensions*. PhD thesis, Computer Science, Princeton University, 1996.

[17] Y. Wu and J. Offutt. Modeling and testing web-based applications. In *CS Technical Report*. George Mason University, 2002.