Web Testing: a Roadmap for the Empirical Research

Filippo Ricca and Paolo Tonella
ITC-irst, Trento, Italy
ricca@itc.it, tonella@itc.it

Abstract

The quality delivered by existing Web applications is often poor. A consequence of this situation is a strong demand for techniques and tools that address the problem of the Web application quality. Lots of approaches are currently available, often coming with prototype or commercial tools implementing them. However, no attempt has been made so far to validate their effectiveness.

In this paper, we consider the available techniques for Web testing and we propose a classification into three major groups. We deal with the problem of defining the Web-specific faults. Our approach is an empirical investigation of the reported faults, abstracted into a fault model. Then, we evaluate the available techniques against the fault model, in terms of the fault categories directly addressed by them. Finally, we sketch a roadmap for the future empirical research.

1. Introduction

Web applications are crucial components of our life. They are involved in critical activities such as economic transactions, e-commerce, etc. Correspondingly, the quality demands made upon these software systems have increased over time. However, the quality experienced by the end users and assessed by means of research studies is often poor or not completely satisfactory.

Several techniques have been proposed for the testing of Web applications. The most advanced techniques are usually available only as research prototypes, but some of them have been incorporated into commercially available tools and programming environments. It is roughly possible to categorize such techniques into three groups: (1) functional testing techniques, supporting requirement-base testing; (2) structural techniques, supporting some form of white-box testing based upon the analysis and instrumentation of the source code; and (3) model-based techniques, which exploit a navigation model of the application. Different techniques tend to focus on different kinds of defects. However, there is no well-established and widely recognized model for the faults that typically occur in Web applications. While some of these faults are expected to be associated with the same defects that can be observed in traditional software, others are specific of the Web applications and of the interaction modes that characterize these applications. It would be important to have an explicit model of Web faults available, in order to assess whether the available techniques are addressing the Web-specific faults directly. Moreover, no empirical validation of the alternative techniques has been so far conducted, so that it is difficult to guess which is most appropriate to detect the defects present in a given Web application. Design and execution of empirical studies on the fault detection capabilities of the various Web testing techniques would represent a major step forward for the discipline.

In this paper, after giving an overview of the available Web testing techniques (Section 2), we describe how we are collecting data in order to define a fault model for Web applications (Section 3). We also present the preliminary outcome of this data collection process, consisting of an initial version of the fault model. Next, we discuss the fault detection abilities of the various testing techniques, based on their underlying representations and assumptions (Section 4). In Section 5, we provide some guidelines for future empirical studies aimed at comparing the alternative techniques. Their fault detection can be evaluated by inserting artificial faults through a fault seeder and measuring the respective fraction of detected faults. Section 6 concludes the paper.

2. Related works

Several works on Web application testing have been presented in the literature and a few tools [7, 8, 9] have been proposed to support it.

Functional testing tools [7] are usually based on capture/replay facilities: they record the interactions that a user has with the graphical interface (Web browser) and repeat them during regression testing. Another approach to functional testing is based on HttpUnit [4]. When combined with a framework such as JUnit [3], HttpUnit permits programmers to write test cases that check the functioning of a Web application. A good introductory paper on the usage of HttpUnit and JUnit for functional testing of Web applications is that by Hieatt et al. [5]. In such paper some XP practices, such as "Test First", are extended to Web applications.

The structure of a Web application can be described at the high level, in terms of its composing pages and its naviga-

tion links, or at the low level, by considering the execution flow followed at the server and client side. For this reason, two kinds of structural testing are possible: at the navigational level (model-based testing) or at the code level (code coverage testing).

The first proposal of code coverage testing for Web applications was made by Liu et al. [6]. They extended traditional data flow testing techniques to Web applications. The data flow information is captured using various flow graphs. Control flow graph and interprocedural control flow graph are used to discover def-use chains present in the scripting portion of a client Web page or present in a server program, while object control flow graph and composite control flow graph permit the computation of two more types of def-use chains. The first one is associated with different function invocation sequences, depending on the user interaction (scripting events), while the second captures def-use chains between different pages, where a variable is defined in a page and is used in a server program.

The first proposal of model-based testing for Web applications was made by Ricca and Tonella [10]. Coverage criteria (e.g., page and link coverage) are defined with reference to a *navigational model*, i.e., a model containing Web pages, links and forms. In this approach 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. Another proposal of model-based testing for Web applications, similar to the one by Ricca and Tonella [10], was made by Andrews et al. [1]. In this case, the navigational model is a finite state machine (FSM) with constraints recovered by hand by the test engineers directly from the Web application. Test cases are generated automatically as subsequences of states in the FSM. Elbaum et al. [2] proposed a Web application testing approach that utilizes data captured in user sessions (stored in a modified log file) to create automatically test cases. The authors describe two similar implementations of this approach, and a hybrid implementation that combines their approach with the model-based approach by Ricca and Tonella [10]. Instead of inserting manually the input values for the test cases [10], they are recovered automatically from the modified log file. Different strategies can be applied to construct test cases from the collected user sessions [8, 9].

Tonella and Ricca [11] confronted model-based testing and code coverage testing on a simple Web application (Wmforum) and gave a preliminary subjective assessment of the faults revealing ability of the two techniques.

3. A fault model for Web applications

Two approaches can be taken when defining a fault model for a class of software systems. It is possible to derive the model top-down, starting from a detailed analysis of the programming features and languages that characterize the software systems under study, or it is possible to infer them in a bottom-up fashion, starting from observations collected empirically in the field. We decided to go for the second option, following our intuition that the fault model implicitly assumed by most proposed techniques may depart from the actually experienced faults. Thus, we intend to first collect empirical data and then refine them with more theoretical, top-down considerations.

We are currently in the process of gathering data about the faults reported for publicly available Web applications. Out of such data we intend to construct a fault model for the Web applications. In the following, we describe the procedure that we are using for data collection and the preliminary model constructed so far.

3.1. Procedure

Web applications have been selected among those available as open source projects on popular software distribution portals, such as sourceforge.net. We focused on applications written in PHP, Java (servlets), Javascript and JSP. We considered only applications of medium to large size. Moreover, only Web applications recording the successive versions/releases into a CVS repository and using a bug reporting system were of interest to us, since the comment information about the changes and the reported bugs is necessary for our analysis.

For each selected application, the CVS repository and the bug reporting system have been randomly visited and the modifications commented as bug fixing have been examined in detail. Typically, the bug reporting system is accessed first, while the CVS repository is accessed only when more details about the changed code fragments are necessary. A summary description of the fault and a tentative fault categorization are then produced. These are the steps of the procedure that we followed in the execution of such task:

1. Randomly select one of the reported bugs for the given application (using the bug reporting system).

2. If necessary, isolate the changes that can be regarded as bug fixing for the selected bug, using the CVS repository.

3. Produce a description of the fixed bug, based upon the user's comment and/or the inspection of the code changes.

4. Map the bug description to a fault category in the currently available fault model or create a new fault category if none of the existing ones is appropriate.

Periodically, after a few additions, fault categories are revised, in order to introduce global changes, such as category merges and splits.

Web appl.	Lang	LoC
jwma	Java/Servlet	27 890
phpMyAdmin	PHP	146 896
SEE	Java/Javascript	252 876
hipergateCRM	Java/JSP	347 721
eGroupWare	PHP	365 359

Table 1. Web applications considered.

3.2. Preliminary results

Table 1 shows the Web applications considered in the construction of our preliminary fault model. More Web applications will be added to this initial set in the near future. The number of Lines of Code (LoC) refer to the PHP, Java, Javascript and JSP code. They indicate a non trivial size. Moreover, these applications underwent lots of modifications, as apparent from the bug fixes that we found reported for them.

Among the faults that we collected for these applications, some can be classified as generic faults (not shown for space reasons), that can be found in almost every software system, with no regard to the programming language in use and to the specific features of the programming domain. On the other hand, other faults are strictly dependent on the interaction mode and on the typical programming idioms that are employed to deliver Web applications. The latter are listed in Table 2.

Every fault category is described by a short sentence and is followed by the actually encountered faults that were abstracted into the given fault category. For each fault category, *support* provides the number of instances of reported faults that falls into this category. Faults are ranked by decreasing support. In brackets is the number of distinct Web applications where the fault was found. The descriptions of the actually encountered faults have been taken from the comments in the bug reporting system and/or from the CVS. Under the column *support* we indicate for each of them the Web application where we found it.

The fault categories with the highest support are related to problems with the authentication of the users, with the support to multiple languages, with the navigation (hyperlinks) and with the differences among the browsers. Problems with forms, cookies and session management are also reported. Finally, a few cases deal with protocol problems (SSL, character encoding).

Overall, they indicate that the implementation of some important features remains problematic with the technology currently available for Web application development. Most of these problems descend from the lack of an explicit support for the related features, which forces programmers to implement them through ad-hoc solutions, often reinvented from scratch. This absence of standard, reusable solutions is true also at the level of the programming languages. For example, different browsers may interpret the same code differently.

Although most of the generic faults reported in can be found also in traditional software systems, the way they appear in Web applications is somewhat specific. For example, lots of string manipulation problems are reported. This is due to the extensive manipulation of the text that is usually required in Web applications. Thus, character mapping or conversion errors, incorrect usage of string libraries or incorrect assumptions on string formats are quite recurring faults. Among the other reported problems, those related to file system and database access are also expected to occur frequently in Web applications. In fact, these software systems are typically an intermediate layer between the user (i.e., a Web browser) and some data.

4. Testing techniques for Web applications

We illustrate the differences among Functional testing, Code coverage testing and Model-based testing through a running example, the Web application of a travel agency showed in Figure 1. This simplified Web application, accessible starting from *http://star.itc.it/ricca-/php/travelAgencyCookies/index.php*, was obtained by abstracting the main characteristics present in a set of real travel agency applications downloaded from the Web. This application uses cookies to maintain the session of a client with the server and a MySQL database to store available hotels, flights and cars.

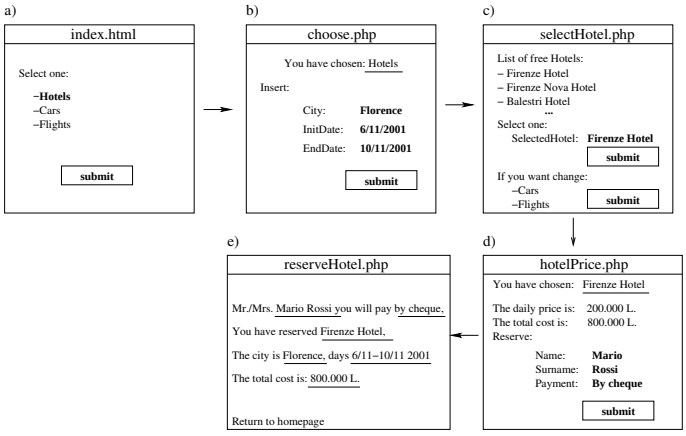

Figure 1. *A simple Travel Agency Web application.*

4.1. Travel Agency Web Application

In this Web application the user can select hotels, cars and flights and reserve/rent/book them on-line. For the sake of simplicity, the pages displaying error messages are not

Web faults	
Fault category	support
1 – Authentication problem	4 (3)
New user without password can not login	SEE
Not checked the ACL data in the servlet	SEE
Anonymous user should never get a home-page	eGroupWare
Check the privileges of the anonymous user, too.	phpMyAdmin
2 – Incorrect multi-language support	4 (3)
In the English release the System ACL data is in Chinese	SEE
This bug rises from using a wrong translation method	SEE
Multi-language support missing (missing lang(" "))	eGroupWare
Home page cannot be customized for Italian version	hipergateCRM
3 – Hyperlink problem	4 (4)
URLs are wrong. They are all in absolute path of the server	SEE
`HREF=\"viewContact('"` must be replaced by `HREF=\"#\"onclick=\"viewContact('"`	hipergateCRM
html:base in JSPs breaks proxying	jwma
Duplicate top menu hyperlinks	phpMyAdmin
4 – Cross-browser portability problem	3 (1)
Browser-specific capabilities not taken into account calendar.php (2.6 vs. 2.5)	phpMyAdmin
Top left frame in Safari 1.2.3 does not load left.php (2.38 vs. 2.37)	phpMyAdmin
Safari can't handle location.replace() redirects correctly	phpMyAdmin
5 – Incorrect form construction	2 (2)
The password is displayed. The type of the input box should be changed	hipergateCRM
Incorrect default form values	jwma
6 – Incorrect cookie value/setting	2 (2)
The skin cookie is not properly set by login_chk.jsp when using Mozilla	hipergateCRM
Incorrect cookie value	jwma
7 – Incorrect session management	1 (1)
GET var assumed as only possibility, while a COOKIE can be used as an alternative	eGroupWare
8 – Incorrect generation of error page	1 (1)
Incorrect generation of error page	jwma
9 – Incorrect usage of SSL	1 (1)
SSL configuration not applied properly	jwma
10 – Incorrect protocol	1 (1)
Data encoded as UTF-8 while SQL requests characters in ISO-8859-1	hipergateCRM

Table 2. Preliminary classification of the Web faults.

reported. When users select an item of interest from the static page `index.html` (Figure 1.a), a dynamic page `choose.php` is created, as shown in Figure 1.b. If, e.g., the option *Hotels* was selected, users are asked to input the city where they want to go and the arrival and departure dates. After clicking the submit button the dynamic page `selectHotel.php` (Figure 1.c) appears, showing a list of available hotels in the chosen city (recovered from the database). The user can select one of them or change the option of interest passing from hotels to cars or flights. In the case of changing option, the page `selectHotel.php` transmits arrival and departure dates via form to the page `choose.php`, as well as the city. These values will be exploited by `choose.php` to propose default values to the users in the reservation of cars or flights. Otherwise, if the user chooses to select a hotel and presses the related submit button, the dynamic page `hotelPrice.php` is created, as shown in Figure 1.d. This page contains the chosen hotel, its daily price and the total cost for the entire stay. In the same page, the user can insert name, surname and payment type to reserve a room in the selected hotel. The dynamic page `reserveHotel.php` (Figure 1.e) appears after clicking the submit button in the previous page. It contains a simple report confirming the reservation. Then, the user can return to the homepage and select cars or flights.

4.2. Functional testing

Functional testing is focused on verifying the Web application's functionality. Web applications may have documents describing the requirements at the user level, such as: use-cases, functional requirements in natural language, etc. From these documents, or more simply navigating in the target Web application, it's possible to create a list of test cases. Each test case describes a scenario that can be accomplished by a Web visitor through a Web browser.

HttpUnit [4] is a Java framework based on JUnit [3], which allows the implementation of automated test scripts for Web applications. HttpUnit is well suited for functional testing. One important aspect of HttpUnit is that it can test entire Web applications. HttpUnit can manage forms, links, JavaScript, Http authentication, cookies and automatic page

redirection, thus it is possible to write test cases in HttpUnit that cover a whole session. For example, if the target Web application includes a shopping cart, a test case to try logging in, selecting an item and placing it in the cart could be written.

```
/** Verifies 'total cost'   **/
public void testTotalCost() throws Exception {
    WebConversation conv = new WebConversation();
    // index.php
    WebRequest req = new GetMethodWebRequest("
     star.itc.it/ricca/php/travelAgencyCookies/index.php");
    WebResponse resp = conv.getResponse( request );
    WebForm FormIndexPage = resp.getForms()[0];
    FormIndexPage.setParameter("x", "Hotels" );
     // choose.php
     WebResponse chHotel = FormIndexPage.submit();
     WebForm FormChHotel = chHotel.getForms()[0];
     FormChHotel.setParameter("city", "Milan" );
     FormChHotel.setParameter("initDate", "07/06/2005");
     FormChHotel.setParameter("endDate", "09/06/2005");
     // selectHotel.php
     WebResponse seHotel = FormChHotel.submit();
     WebForm FormSeHotel = seHotel.getForms()[0];
     FormSeHotel.setParameter("selectedHotel", "prince");
     // hotelPrice.php
     WebResponse hoPrice = FormSeHotel.submit();
     WebForm FormHoPrice = hoPrice.getForms()[0];
     FormHoPrice.setParameter("name", "Filippo");
     FormHoPrice.setParameter("surname", "Ricca");
     FormHoPrice.setParameter("payment", "Cash");
     // reserveHotel.php
     WebResponse reHotel = FormHoPrice.submit();
     // Recover Total Cost
     String reHotelStr = reHotel.getText();
     int pos = reHotelStr.indexOf("the total cost is:");
     int BR = reHotelStr.indexOf("<BR>", pos);
     String t_cost = resHotelStr.substring(pos, BR-1);
     // assertion
     assertEquals( t_cost, "the total cost is: 400");
 }
```

Figure 2. *An HttpUnit test case for the Travel Agency Web application.*

Figure 2 shows an HttpUnit test case for the Travel Agency application. This test case verifies the computation of the total cost for the entire stay in the hotel of choice (Prince). The method *testTotalCost()* uses some methods of the HttpUnit class *WebConversation* to open and mantain a connection with the Web application. The first downloaded page is `index.php`. Then, the dynamic pages `choose.php`, `selectHotel.php`, `hotelPrice.php` and `reserveHotel.php` are accessed by submitting the related forms. The parameters for these forms are set by means of the HttpUnit API method *setParameter*. Then, the total cost of the stay is extracted from the page `reserveHotel.php`, using the method *getText()* and searching the returned string. At this point, the variable `t_cost` contains the string "the total cost is:" followed by the total cost of the stay. This is compared against the expected string in the assertion that terminates the test case.

4.3. Code coverage testing

Similarly to traditional software, structural testing of Web applications is based on the knowledge about the internal structure of the system under test. The adequacy of the test cases is assessed in terms of the level of coverage of the structure they reach. This approach can be applied to the Web applications either representing the structure at the high-level, by means of the navigation model (Model-based testing), or at the low-level, by means of the control flow model (Code coverage testing). Nodes in the control flow model of a Web application represent the statements that are executed by the Web server or by the client (Web browser). Edges represent control transfer. Since the execution on the server involves one (or more) server side languages (e.g., PHP and SQL) and the execution on the client involves additional languages (such as HTML and JavaScript), the control flow model has different node kinds. In the following, the shape of the nodes indicates their kind (box is used for HTML and ellipse for PHP).

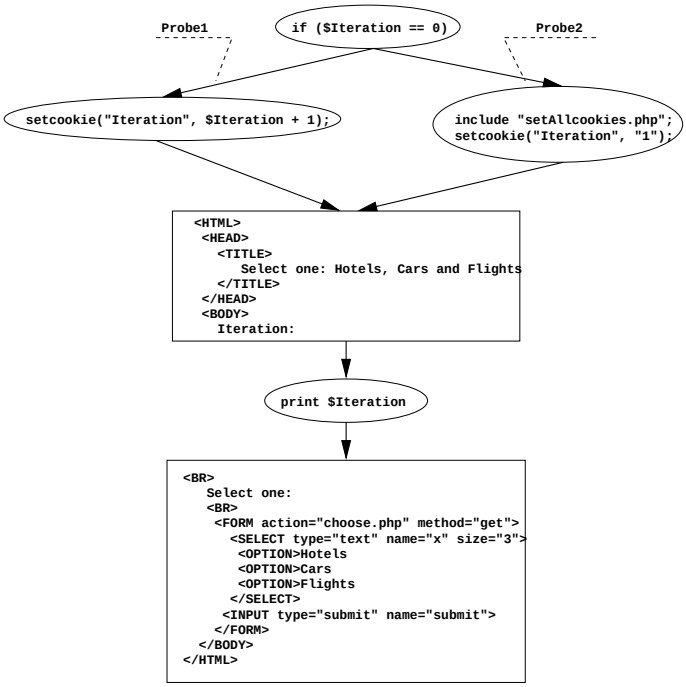

Figure 3. *Control flow model of index.php.*

Figure 3 shows the control flow model (nodes of the same type are glued together for space reasons) of the PHP/HTML page `index.php`. Several coverage criteria can be defined by taking into account the control flow model (CFM). Some of them are: path coverage (all paths in the CFM are traversed in some test case), branch coverage (all edges in the CFM are traversed in some test case) and node coverage (all nodes in the CFM model are traversed in some test case). In order to determine the level of coverage reached by a given

test suite, the Web application under test must be instrumented and the branches in the control flow model that are traversed during test case execution must be traced. Program transformation tools can be used to insert tracing instructions (probes) inside the original code. If we apply a PHP code instrumentor to `index.php` two probes are inserted, one for each branch of the if-statement (see Figure 3). Execution of the test cases defined to achieve code coverage can be still obtained by means of HttpUnit. However, when the GUI interactions are required in order to execute some client-side (e.g., JavaScript) code, automation becomes more problematic (capture-replay tools may be used in these cases). The problem of covering as much as possible of the structure of a given Web application is orthogonal to the problem of verifying that the results of execution are correct. For each test case, an oracle must be defined with the expected output (displayed information in resulting page). During the first run of the test cases, correctness of the output is usually assessed manually. During re-execution of the test cases (regression testing), the oracle can be obtained from the previous run and the correctness check can be automated.

4.4. Model-based testing

The structure of a Web application can be described at the high level, in terms of its composing pages and its navigation links. Both dynamic and static pages are properly represented. Dynamic pages are the result of executing a program on the Web server in response to a request from the Web browser. Important interactive features that are exploited by Web applications, like forms and frames, are part of the model, being relevant to the navigation.

In HTML, user input is gathered by forms and is passed to a server program, which processes it, in response to a *submit* event. A Web page can include any number of forms. Each form is characterized by the input variables that are provided by the user through it. Additional *hidden* variables are exploited to record the state of the interaction. They allow transmitting pairs of the type *<name, value>* from page to page. Typically, the constant value they are assigned needs be preserved during the interactive session for successive usage. Since the HTTP protocol is stateless, this is the basic mechanism used to record the interaction state (variants are represented by cookies and URL parameters).

Figure 4 shows the navigation model of the Travel Agency Web application. This model is a simplification (without pages containing error message and informative static pages) of the model produced by **ReWeb** [10], applied to the considered site.

The only conditional edges present in the model are those outgoing from the node `choose.asp`. Their labels contain the respective existence conditions, depending on the value of the variable `x`, inserted by the user in the form contained in `index.html`, or in the form inside `selectHotel.php`, `selectCar.php` and `selectFlight.php`, when the user decides to change the selected option (Hotels, Cars or Flights).

In this testing technique, coverage criteria are defined by taking into account the navigation model. The criteria are similar to those used with code coverage testing.

Node coverage of the Travel Agency navigation model is reached, for example, considering the following six test cases. Three test cases are devoted, respectively, to reserve a hotel, rent a car and book a flight; the other three are similar to the previous ones, but differ from them because the selected option (Hotels, Cars or Flights) in the page `choose.php` is changed. An example of test case in the **TestWeb** format [10], belonging to the last group, is the following:

```
index.html # x=Hotels
choose.php # city=Milan & initDate=1/1/02 & endDate=1/1/02
selecHotel.php # x=Hotels
choose.php # city=Milan & initDate=1/1/02 & endDate=3/1/02
selecHotel.php # selectedHotel=prince
hotelPrice.php # name=Filippo & surname=Ricca & payment=cash
```

As with code coverage testing, a suite of test cases is executed to cover the Web application model. For each test case, assertions must be inserted to verify that the results of the executions are correct.

4.5. Fault detection ability

The three testing techniques briefly described above are now evaluated in terms of their ability to expose the fault categories in our preliminary fault model (we have considered only Web specific faults, Table 2). For each technique we consider whether a given fault type is explicitly addressed by the technique or not.

Technique	**Web fault category**									
	1	2	3	4	5	6	7	8	9	10
Functional		×						×		
Code coverage						×		×		
Model-based			×		×			×		

Table 3. Fault detection ability of the three considered testing techniques.

Table 3 shows our assessment of the three Web testing techniques. It should be noted that this assessment involves some subjective judgments and that some decisions we made could be deemed questionable. However, the overall picture seems not so much dependent on the evaluators themselves. It is in fact clear that most fault types are not addressed explicitly by any available testing technique. For example, no technique aims explicitly at exercising authentication, portability, session management, security layers and data transfer

protocols. Of course, each technique might be able to address them indirectly. For example, code coverage testing might end up exercising the portion of a Web application which contains some of these problems. However, this is not a direct consequence of the technique itself, being rather a side effect. Moreover, this could be highly sensitive to the chosen input values. In fact, the adequacy criteria of the three techniques may be achieved using quite different values, and while some of them could (indirectly) expose the problems listed above, others may not.

If we look at the fault categories explicitly addressed by the three techniques, we can see that they are quite complementary. The model-based technique addresses navigation problems such as hyperlink (3), form (5) and error page (8) problems. By exercising thoroughly the control flow paths, the code coverage technique is expected to have some ability of exposing faults related to incorrect cookie values (6), while the functional technique could succeed exposing multi-language problems (2), under the assumption that the multi-language requirements are given. The incorrect generation of error pages (8) is potentially spotted by all techniques, since error pages are expected to be specified in the requirements, error conditions should be exercised during code coverage and some nodes in the Web application model represent them.

5. Future empirical work

In our future work we intend to gather more empirical data in support to our fault model for Web applications. Moreover, we will contrast the fault model constructed bottom-up from observations with the fault model that can be defined top-down, based upon the abstract features that Web applications are expected to share with each other.

Even though a theoretical evaluation, such as the one carried out in this paper, of each Web testing technique against a fault model is useful, the actual fault detection ability of each technique needs also be assessed empirically. Thus, an important part of our future work will be devoted to such kind of empirical studies. Specifically, we aim at measuring the fault detection ability of each technique. For such a purpose, it is possible to exploit a *fault seeder*, which inserts known, artificial faults into a given Web application, taken from a benchmark. The ability of each technique to reveal the artificially inserted faults provides a measure for the empirical validation of the technique.

5.1. Benchmark

Defining a benchmark of software systems for an application domain so wide as the Web application is a difficult and challenging task. In fact, the variety of application domains is immense and the range of technologies involved is also very large. Since the source code of the Web applications is required by some techniques, the natural choice is to sample the applications from those available as open source projects. This is a common practice also in testing of traditional software. While for sure a large number of technologies and domains can be covered in this way, it might be questionable the extension of the results to closed source, commercial Web applications. Ideally, the latter should be also studied, in order to see if the results obtained on the former generalize to them.

5.2. Fault seeder

Once a comprehensive fault model is available for the Web applications, it is possible to exploit it for the development of a fault seeder. Each fault category can be instantiated into a set of typical coding errors that give raise to the given fault. Then, automated program transformations can be used to insert samples of faults from a given category into the source code of the Web applications in the benchmark.

The main difficulty in the realization of a fault seeder for Web applications is the variety of programming languages and technologies that can be used to implement a given feature (say, for example, session management). A fault seeder should be able to recognize the implementation used in a given Web application and should be able to break it, introducing a bug. Programming idioms usually adopted to implement certain features should be encoded in the seeder, together with the possible variants.

Availability of the fault seeder would complete the picture sketched in this paper. After building a fault model, both inferring it from the observations and deriving it top-down, and after evaluating the available techniques against the model, an empirical assessment would become possible. The expected results would hint for future developments of techniques that address the Web specific problems more directly.

6. Conclusions

In this paper we have described a preliminary fault model for the Web applications. We obtained such a model by conducting a series of observations of faults reported for existing Web applications. The available Web testing techniques have been classified into three main groups: Functional, Code coverage and Model-based techniques. Each of them was assessed in terms of the faults directly addressed by it. Our preliminary results indicate that most faults observed on real Web applications are addressed only indirectly by the available testing techniques. This clearly demands for novel techniques which more aggressively attack those fault types. When we compared the alternative techniques, their complementary nature emerged. This suggests that one single technique is usually not enough and a combination of techniques is expected to reveal faults which belong to differ-

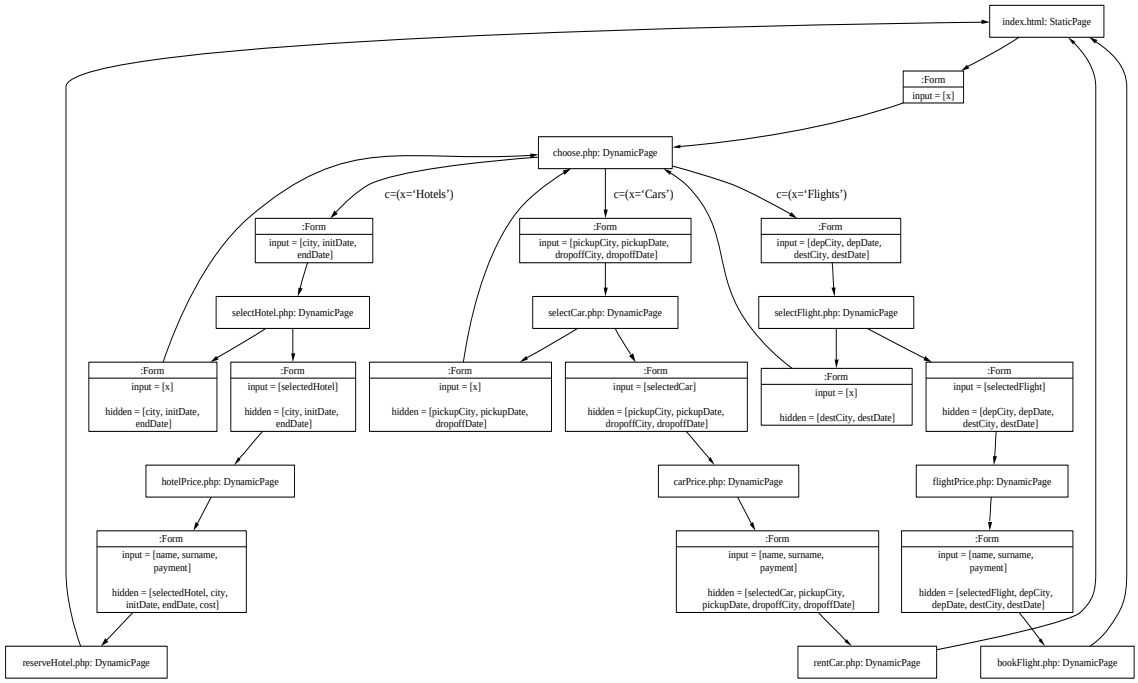

Figure 4. *Model of Travel Agency Web application.*

ent fault categories. Our fault model is still preliminary and requires further empirical and theoretical validation. Moreover, the Web testing techniques that we considered have been assessed just in terms of the directly addressed fault types. However, an empirical validation aimed at measuring the fault detection ability needs to be also carried out. To this aim, we plan to develop a fault seeder based on the fault model. Thus, our roadmap for the future empirical research in Web testing includes:

- Creation of a solid fault model, validated empirically.
- Development of a fault seeder, based on the fault model.
- Creation of a benchmark of Web applications, with faulty versions created by means of the fault seeder.
- Empirical validation of the Web testing techniques.
- Definition of novel techniques and refinement of existing techniques, based on the results of the empirical study.

References

[1] A. Andrews, J. Offutt, and R. Alexander. Testing web applications. *Software and Systems Modeling*, 2004.

[2] S. Elbaum, S. Karre, and G. Rothermel. Improving Web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 49–59, Portland, USA, May 2003. IEEE Computer Society.

[3] E. Gamma and K. Beck. *junit*. `http://www.junit.org/`.

[4] R. Gold. *httpunit*. `http://httpunit.sourceforge.net/`.

[5] E. Hieatt, R. Mee, and G. Faster. Testing the web application engineering internet. *IEEE Software*, 19(2):60–65, March/April 2002.

[6] C.-H. Liu, D. C. Kung, P. Hsia, and C.-T. Hsu. An object-based data flow testing approach for web applications. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):157–179, April 2001.

[7] E. Miller. The web site quality challenge. - companion paper: "website testing". In *Proc. of QW'98, 11th Annual International Software Quality Week*, San Francisco, CA, USA, May 1998.

[8] Parasoft. *WebKing*. `http://www.parasoft.com/`.

[9] Rational. *RobotJ*. `http://www.wilsonmar.com/`.

[10] F. Ricca and P. Tonella. Analysis and testing of Web applications. In *Proceedings of International Conference on Software Engineering (ICSE2001)*, IEEE Computer Society: Los Alamitos CA, 25-34, 2001.

[11] P. Tonella and F. Ricca. A 2-layer model for the white-box testing of web applications. In C. Kaner, editor, *Proc. of the 6th IEEE Int. Workshop on Web Site Evolution*, pages 11–19, Chicago, Illinois, USA, September 2004. IEEE Computer Society.