JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2000; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2002/09/23 v1.02]*

Research

Statistical Testing of Web Applications

Paolo Tonella*,† and Filippo Ricca†

ITC-irst, Centro per la Ricerca Scientifica e Tecnologica, 38050 Povo (Trento), Italy

SUMMARY

The World Wide Web, initially intended as a way to publish static hypertexts on the Internet, is moving toward complex applications. Static Web sites are being gradually replaced by dynamic sites, where information is stored in databases and non trivial computation is performed.

In such a scenario, ensuring the quality of a Web application from the user's perspective is crucial. Techniques are being investigated for the analysis and testing of Web applications for such a purpose. However, a static analysis of the source code may be extremely difficult (and, in general, infeasible) because of the presence of dynamic generation of the HTML code that is part of the application under analysis.

In this paper, a dynamic analysis technique is proposed for the extraction of a Web application model through its execution. Availability of statistical data about the accesses to the pages generated by the Web application is exploited for statistical testing, based on the recovered model. Test cases can be prioritized, so as to exercise the most frequently followed paths first. Moreover, statistical reproduction of the user's navigation paths allows for an estimation of the reliability of the application.

KEY WORDS: Web applications, code analysis, statistical testing, model extraction.

INTRODUCTION

Web sites – collections of static hyper-documents encoded in the HTML language – are being gradually replaced by Web applications – server side programs that dynamically generate hyper-documents in response to some input from the user. Correspondingly, the motivation behind being present on the Web is changing. While in the past it was a matter of following the trend and advertising activities and products, it is now becoming a viable alternative to

*Correspondence to: ITC-irst, Centro per la Ricerca Scientifica e Tecnologica, 38050 Povo (Trento), Italy
†E-mail: {tonella, ricca}@itc.it

Received 31 December 2002
Revised 23 March 2003

the traditional ways of selling goods and providing services. For such tasks, static Web sites are insufficient and more dynamism is required on the server side.

Static analysis of highly dynamic Web applications is a difficult task. In fact, the HTML code displayed by the browser is not fixed, being produced at run-time by server programs. While in the simplest cases a fixed HTML skeleton is filled-in with values computed dynamically, in more complex applications even the structure of the resulting HTML page is not given a priori, and is constructed dynamically. In such a situation, a static analysis of the server programs generating the Web pages can hardly result in a useful model of the application. In fact, the problem of determining the HTML code produced by a server program is related to the problem of determining if a given execution path is feasible, which is known to be an undecidable problem. Moreover, a Web application involves several programming languages. On the server side, at least one programming language is used for the dynamic production of the HTML pages (e.g., PHP, Java, Perl, VBscript, etc.). If databases are accessed, a related query language, such as SQL, is also present. HTML statements are then generated, but typically they are not pure HTML code, and include client side code for form validation, client side computation, and graphical event handling (e.g., Javascript, Java applets, etc.). Static analysis of such a variety of languages – and of all their possible interactions – is a technological challenge.

In this paper, we propose a technique for the extraction of a model of a Web application, obtained by statically analyzing the HTML code that is dynamically generated by the server programs. Input values which cover all relevant navigations are pre-specified by the user, and downloaded pages are either unrolled or merged, in order to produce an abstraction over the set of HTML pages downloaded. The resulting model is computable in presence of high – even "extreme" – dynamism and requires the ability to parse just HTML. The problem of statically approximating the HTML code being generated is absent. On the other side, the model obtained may be partial, if the inputs used to produce it do not cover all relevant behaviors of the Web application. This is an intrinsic limitation of all dynamic analyses.

The Web application model produced in this way can be enriched with the transition probabilities, obtained from the statistical information dumped by the Web server during execution. In absence of ad-hoc data, the access log which is automatically recorded by the Web server can be used. The resulting model can be interpreted as a Markov chain, on which statistical testing can be conducted [22]. Moreover, navigation statistics can be analyzed, to determine the average number of hyperlinks followed before reaching a given page [21], and the usage of navigation facilities offered by the browser instead of the hyperlinks of the site. Test cases can be generated according to the statistics encoded in the Web application model, by performing a stochastic visit on it. The number of failures occurring in such test sessions can be used to estimate reliability, since the behavior of users is stochastically reproduced in the test cases. In addition to this, the paths in the Markov chain model can be sorted by decreasing probability, thus giving a criterion for the prioritization of the test cases.

The proposed model can be semi-automatically extracted from an existing Web application, by exploiting the algorithms detailed in section MODEL EXTRACTION. Section STATISTICAL TESTING deals with testing process, transition probability estimation, and statistical test case generation. Some experimental data obtained on an existing Web application are provided in section CASE STUDY. Related works are discussed in the

successive section, while conclusions and future work are presented in the last section of the paper.

MODEL EXTRACTION

Web application modeling

The aim of a Web application model is that of describing a Web application in terms of composing pages and allowed navigation links. Both dynamic and static pages are to be properly modelled. Dynamic pages are the result of executing a program on the Web server in response to a request from the Web browser of the user. Important interactive features that are exploited by Web applications, like forms and frames, should be part of the model, being relevant to the navigation in a Web application.

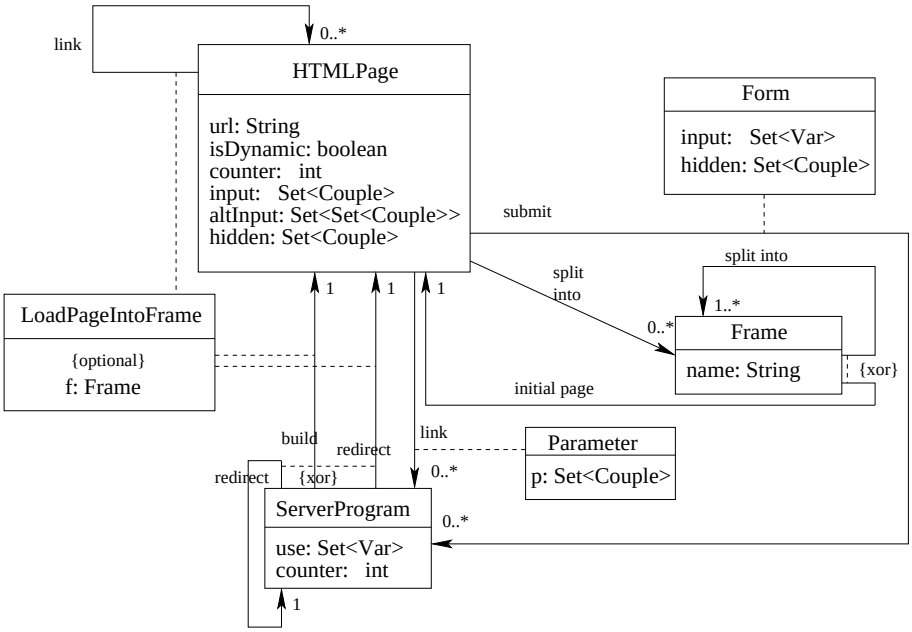


Figure 1. *Meta model of a generic Web application structure. The model of a given application is an instance of it.*

Figure 1 shows the meta model used to describe a generic Web application. The central entity in a Web application is the *HTMLPage*. An HTML page contains the information to be displayed to the user, and the navigation links toward other pages. It also includes organization and interaction facilities (e.g., frames and forms). Its URL is recorded in the attribute *url*. Navigation from page to page is modelled by the auto-association of class HTMLPage named *link*. Web pages can be static or dynamic. While the content of a *static* Web page is fixed,

the content of a *dynamic* page is computed at run time by the server (a similar distinction is proposed by Conallen [2] and Eichmann [3]) and may depend on the information provided by the user through input fields. The boolean flag *isDynamic* distinguishes the two cases. The class *ServerProgram* models the script/executable that runs on the server side and generates a dynamic HTML output. When the content of a dynamic page depends on the values of a set of input variables, the attribute *use* of class *ServerProgram* contains them. A server side program can be executed by traversing a *link* from an HTML page whose target is the server script/executable and whose attributes include a set of parameters, represented as pairs <*name, value*> or by submitting a form. The server program can either redirect the request to another server program (auto-association *redirect*), build an output, dynamic HTML page (association *build*), or simply redirect to a static HTML page (association *redirect*). The latter two cases can be distinguished only because the resulting HTML page is respectively static or dynamic. When a server program builds a dynamic page, the input and hidden variable values that have been provided to it are stored in the attributes *input* and *hidden* of the resulting page, as sets of couples <*name, value*>. Field *altInput* stores alternative inputs that generate the same dynamic page (see page merging below).

A *frame* is a rectangular area in the currently displayed page where navigation can take place independently. Moreover, the different frames into which a page is decomposed can interact with each other, since a link in a page loaded into a frame can force the loading of another page into a different frame. This can be achieved by adding a target to the hyperlink. Organization into frames is represented by the association *split into*, whose target is a set of *Frame* entities. Frame subdivision may be recursive (auto-association *split into* within class *Frame*), and each frame has a unary association with the Web page initially loaded into the frame (absent in case of recursive subdivision into frames). When a link in a Web page forces the loading of another page into a different frame, the target frame becomes the data member of the (optional) association class *LoadPageIntoFrame*.

In HTML user input is gathered by exploiting 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).

Client side computation (e.g., embedded JavaScript code), limited to input validation and advanced presentation modes, does not affect the model. It should be reflected in the model if used to gather user input or to perform a connection with a server side page/program. Such cases are currently not handled automatically.

In a Web application, the same server program may behave differently, according to the interaction state. To clarify this situation it is convenient to classify server programs into two categories:

- Server programs with state-independent behavior.
- Server programs with state-dependent behavior.

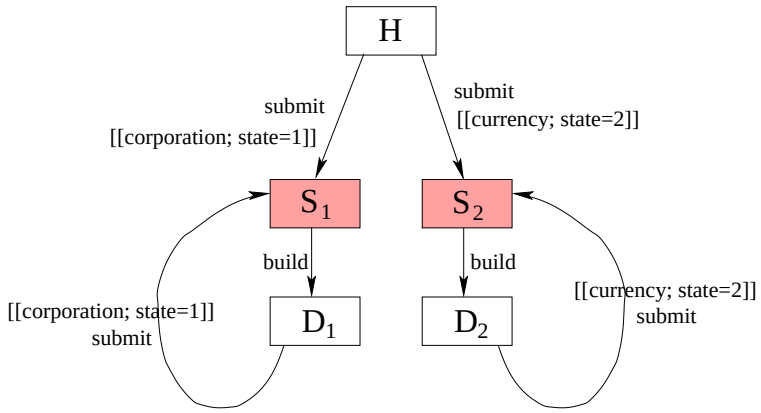


Figure 2. *Example of Web application model. Nodes with gray background are server programs, edge labels contain input variables and hidden variables, separated by a semicolon, within double square brackets.*

Server programs in the first category exploit always the same mechanism to produce the output, generating a dynamic page whose structure and links are fixed. The behavior of these server programs is the same in every interaction state. On the contrary, server programs in the second category behave differently when executed under different conditions. A server program may, for example, provide two completely different computations – and consequently different output pages – according to the value of a hidden flag recording a previous user selection.

In presence of server programs with state-dependent behavior, navigation sessions are obtained as paths in the Web application model if server programs and related output pages are replicated for all the possible behavioral variants. We call the resulting model an *explicit-state model*, differing from the alternative one, called *implicit-state model*, in that it unrolls server programs and dynamic pages with different behaviors into actually different entities, which are given a progressive identification number. In this way, the page identity is not associated to a physical entity (page or server program), but is rather differentiated according to the behavior.

Let us consider a simple financial Web application, the model of which is shown in Figure 2. This application provides two different services, related to the stock market and to the exchange rates. A single server program S provides both services. The home page of this application, H, is a static page containing some descriptive material and two links to the two services. If the user is interested in the stock market, a form gathers the name of the corporation of interest as an input value, while the name of the currency is gathered in the form leading to the exchange rates service. The dynamic page generated in response to the user request allows repeating the request, specifying a different corporation within the stock market service, or a different currency within the exchange rates service. However, when a service type has been selected, it is not possible to switch to the other one without restarting from the home page. In other words, the state of the interaction remains fixed after the initial selection.

In the model of this Web application (Figure 2), the initial static page H is connected to two replications of the server program S, S_1 and S_2, corresponding to the two different behaviors that characterize it. Correspondingly, two different dynamic pages are considered to be built in the two different interaction states, namely D_1 and D_2. When the *submit* edge from H to S with `state=1` is followed, the input variable `corporation` is passed to the server program, with the value provided by the user. When the user selects the second service, the hidden variable `state=2` is propagated through the form, together with the input variable `currency`. The two target objects, S_1 and S_2, respectively build the two dynamic pages D_1 and D_2, which display the requested information and give the user the possibility to provide an alternative `corporation/currency` in input and see the related information, obtained by the same server program S. The hidden variable `state` is propagated unchanged to the server program S_1 from D_1, and to S_2 from D_2.

Dynamic model recovery

In the context of the research project WebFAQ (Web: Flexible Access and Quality), recently launched at our research center, we developed the reverse engineering tool **ReWeb** [16], supporting the analysis of existing Web applications. One of its modules, called *Spider*, is responsible for the automatic extraction of the explicit-state model of a target Web application. The main difficulty in this operation is differentiating a same server program according to the different behaviors it may exhibit. Moreover, user input has to be simulated, so that dynamic pages can be generated and navigation can proceed beyond the purely static part of the Web site. To accomodate the latter issue, a set of input values are specified before running the *Spider*, granting the traversal of all relevant site portions. Such values are used by the *Spider*, which provides them to the server programs during the downloading of the site pages. Among the hidden and input values that are used by the *Spider* for the download, some may affect the internal state of the Web application, and consequently the behavior of the server program being invoked. Formally, the domain of input and hidden variables can be partitioned into equivalence classes, and the same behavior is expected to be obtained when the inputs belong to a same equivalence class. In the example of Figure 2, such partitioning is induced by the two possible values of the hidden variable `state`. When `state` is equal to 1, the behavior related to the stock market service is obtained, with no regard to the values of the other variables, while `state=2` characterizes the second equivalence class of inputs of this Web application. The input values provided to the *Spider* should be comprehensive enough as to cover all different behaviors of the server programs.

Figure 3 shows the pseudocode of the procedure *DownloadDynamicPage*, which is used by the *Spider* to download a dynamic page. The inputs to be used for the download are provided externally (in our implementation, in a file called `formInputFile.txt`). When an input line with a matching action (instr. 2: `line=f.action, ...`) is found, an object of type *HTMLPage* is built and inserted into the returned list D, for successive page scan. Such an object has the form *action* as URL, and the user specified inputs as *input* values. It can be noted that hidden values found in the form f are required to be the same as those specified in `formInputFile.txt` (instr. 3). This condition is necessary to ensure that inputs are used in the correct interaction state: if the same server program is activated in a different state, the

```
// scan file "formInputFile.txt"
DownloadDynamicPage(f)
1 import globalCounter
2 for_each line = (f.action, I1=V1&...&In=Vn, H1=W1&...&Hn=Wn)
                  in "formInputFile.txt"
3   if f.input = [I1, ... , In] ∧ f.hidden = [H1=W1, ... , Hn=Wn] then
4       q = new HTMLPage()
5       q.url = f.action
6       q.isDynamic = true
7       q.counter = globalCounter
8       q.input = [I1=V1, ... , In=Vn]
9       q.hidden = f.hidden
10      q.download()
11      D.addElement(q)
12      globalCounter = globalCounter + 1
13  endif
14 endfor_each
15 return D
```

Figure 3. *Pseudocode of the* DownloadDynamicPage *procedure, that downloads a dynamic page providing proper input values to the server program.*

input line from `formInputFile.txt` cannot be used. This is easily detected since the related hidden variables have different values.

Different inputs specified by the user may belong to a common equivalence class, being associated with the same behavior of the server program, or the same behavior may be obtained (and the same state may be reached) for different values of the hidden variables. When this knowledge is available a priori, only one sample tuple of values is included in `formInputFile.txt`. When this is not known, an additional operation of *page merging* is required to unify equivalent instances of the downloaded pages. Three increasingly weaker page merging heuristic criteria can be used to simplify the explicit state model constructed by the *Spider*:

1. Dynamic and static pages that are identical according to a character-by-character comparison are considered the same page in the model.
2. Dynamic pages that have identical structure, but different texts, according to a comparison of the syntax trees of the pages, are considered the same in the model.
3. Dynamic pages that have similar structure, according to a similarity metric, such as the tree edit distance, computed on the syntax trees of the pages, are considered the same in the model.

While moving from criterion 1 to 3, the intervention of the user to validate the automatically identified page merges becomes increasingly important, since possible errors become more and more likely. Page merging is an integral part of the *Spider* procedure, to avoid the construction of multiple copies of a conceptually same page. This contributes to producing a finite model. In fact, a page already in the model may be downloaded an infinite number of times if not

```
Spider(target_page)
1   L.addElement(target_page)
2   while not (L.isEmpty())
3       p = L.firstElement()
4       L.removeElement(p)
5       if not Pages_already_visited.contains(p) then
6           if not p.isDynamic then
11              Pages_already_visited.addElement(p)
12              p.download()
13          endif
14          Pages_found = p.scanPage()
15          for_each p' ∈ Pages_found
16              L.addElement(p')
17              model.addEdgeToModel(p, p')
18          endfor_each
19          Forms = p.retrieveForms()
20          for_each f ∈ Forms
21              Ded = p.DownloadDynamicPage(f)
22              for_each d ∈ Ded
23                  if Dynamic_pages_already_in_model.containsDynamic(d)
                    then
24                      q = model.recoverPageAlreadyInModel(d)
25                      model.addEdgeToModel(p,q)
26                      addAltInput(q,d.input)
27                  else
28                      model.addEdgeToModel(p,d)
29                      L.addElement(d)
30                      Dynamic_pages_already_in_model.addElement(d)
31                  endif
32              endfor_each
33          endfor_each
34      endif
35  endwhile
```

Figure 4. *Pseudocode of the SPIDER procedure, that downloads a target Web site and builds the related model.*

merged with the corresponding one in the model. In our experience, the hyperlink structure is less informative on the similarity between pages than the abstract syntax tree. This is why the third page merging criterion exploits the syntactic, rather than the hyperlink, structure.

Figure 4 contains the pseudocode of the *Spider* module. A list L is maintained with all the pages still to be visited. Each page in the site is considered in the body of the most external loop. If the page is static, it is just downloaded (line 12), in case it has not been visited previously. Then, hyperlinks are examined within the downloaded page (line 14, *p.scanPage()*) and the referenced pages are added to L. The model is also updated in this phase (line 17). If the page contains forms, the related dynamic pages are downloaded (line 21) by means of the procedure *DownloadDynamicPage*, described above. A dynamic page is considered to be already in the explicit state model (line 23) if its URL is the same of another page in the model and if it was obtained by passing input and hidden values to the server program which belong to the same equivalence class, i.e., which are associated to the same behavior of the server

```
(S, corporation="atd", state=1)
(S, corporation="zpm", state=1)
(S, currency="euro", state=2)
(S, currency="dollar", state=2)
```

Figure 5. *Example of input file for the* Spider. *Each input line contains the name of the invoked server program, followed by a list of input/hidden variables with respective input/state values.*

program. In this case, the dynamic page is not added to L and its input *d.input* is considered an alternative input for the page (q) already in the model. Otherwise, it is inserted into L for successive visit (line 29). When the equivalence classes characterizing the behavior of the server programs are not known a priori, the page merging heuristics discussed above can be employed to (semi-)automatically recognize equivalent dynamic pages and thus equivalent instances of the related server programs. With reference to Figure 4, page merging is executed at line 23, where potential page unification actions are detected by the *containsDynamic* method.

In principle, the *Spider* procedure is not guaranteed to terminate, because server program nodes could be replicated an infinite number of times in correspondence with the occurrence of states that are not recognized as already encountered before. In practice, the usage of the page merging heuristics results (according to the authors' experience) in a finite model. Careful selection of the input values, as representative of all equivalence classes of inputs, ensures the completeness of the model.

An example of input file to be used for the extraction of the model in Figure 2 is shown in Figure 5. Each input line contains the name of the server program to be invoked (`S`), an input variable (either `corporation` or `currency`), with associated input value, and a hidden variable `state`, with associated state value. The *Spider* module starts its computation by adding the initial page H to the list L of pages to download (*target_page* in *Spider* is H). Then, page H is extracted from L and downloaded (it is not a dynamic page). An HTML parser scans its content, giving the set of referenced pages and of contained forms. In our case, two forms are retrieved. The procedure *DownloadDynamicPage* is invoked on both. The first form has a hidden variable `state=1`, so that only the first two lines of `formInputFile.txt`, shown in Figure 5, match the condition of having the same server program as action (S) and the same values of the hidden variables. When the first line is processed, the *Spider* requests the execution of the program S on the Web server with hidden variable `state=1` and input variable `corporation="atd"`. The resulting dynamic page is downloaded by the *Spider* and added to the list of downloaded dynamic pages D. A second dynamic page is generated for the second line of `formInputFile.txt`, with the same hidden variable value and `corporation="zpm"`. Such two dynamic pages are iteratively taken into consideration at line 22 of the *Spider* procedure. While the first page is surely *not* contained in the model, and it is therefore added to L and to the model, the second is already in the model, since the same behavior of S was already observed in correspondence with the first input line used. One possibility to recognize that the second page is equivalent to the first one is to know a priory that variable `state` determines the internal state of S and partitions the input domain into equivalence classes. As a consequence,

the second page is unified with the first one, belonging to the same equivalence class. If this knowledge is not available, the two pages can be compared to recognize the possibility of merging them. The first merging criterion is expected to fail: since the two pages report data of two different corporations ("atd" and "zpm"), they are expected not to be textually equal. However, the second merging criterion is expected to succeed, in that a common page structure is expected to be used to provide the same kind of information – even if referred to different corporations. The input used for the second dynamic page is stored as an alternative input for the page previously inserted into the model. The execution of *DownloadDynamicPage* on the second form of the home page gives two dynamic pages that are unified in a similar way.

After the first iteration of the loop at line 2 in the *Spider* procedure, the list L of model elements still to be considered contains D_1 and D_2, the two dynamic pages generated by S with the inputs taken from the first and the third line of `formInputFile.txt`. Both pages contain one form. The form of D_1 has a hidden variable `state=1`. Therefore, for the download of the associated dynamic page, lines 1 and 2 of `formInputFile.txt` can be used. The first resulting page is identical to D_1, while the second one has the same structure. As a consequence, both of them can be merged with D_1 (merging criteria 1 and 2 respectively). A priori knowledge about the possibility to discriminate the equivalence classes of inputs according to the value of `state` would simplify this task. The result is an edge labeled *submit* from D_1 to S_1. Similarly, the input lines to be used for the form in D_2 are the third and fourth lines of `formInputFile.txt` and the dynamic pages obtained by form submission can be unified with D_2, giving rise to a link between D_2 and S_2. The final model produced by the *Spider* is exactly the one reported in Figure 2.

STATISTICAL TESTING

Complete testing of all paths/interactions is unfeasible for any non trivial system. Statistical testing aims at focusing on the portions of the system under test that are more frequently accessed, in order to ensure that the reliability of the delivered product be high. By verifying that the output produced in the most likely interactions of the user with the system is correct, statistical testing allows measuring and reducing the probability that an untested user interaction occurs after product delivery.

The main phases involved in statistical testing are:

1. Construction of a statistical testing model, based on available usage data.
2. Test case generation based on the statistics encoded in the testing model.
3. Test case execution and analysis of execution output for reliability estimation.

In phase 1 probabilities are assigned to the interactions a user may have with the system. In our case, the Web application model will be enriched with probabilities of navigation from a page to another one. In phase 2 the most likely execution paths can be followed. Alternatively, a random walk according to the transition probabilities encoded in the model can be conducted. Finally, in phase 3 faults are possibly revealed. Fault occurrence data are used to predict reliability.

Testing process

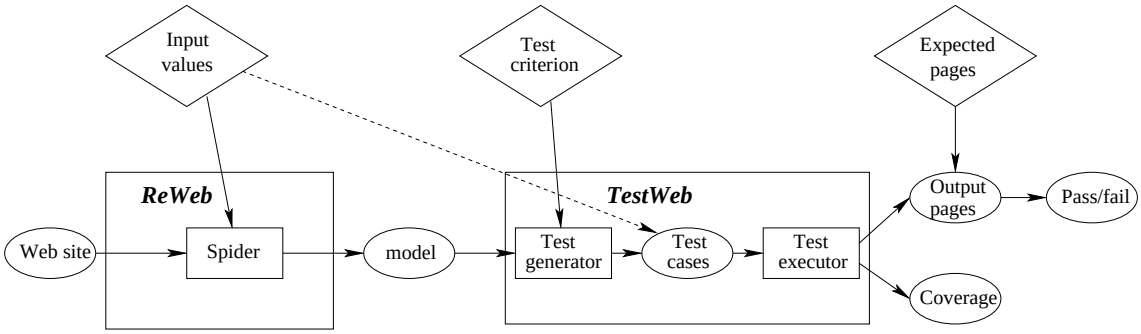
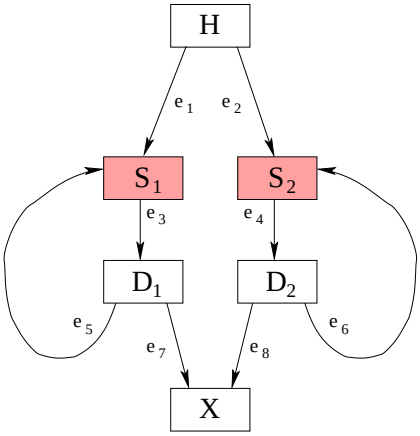


Figure 6. **TestWeb**'s *modules and their dependencies on* **ReWeb** *and the user input.*

The explicit-state model of the Web application to be tested is generated by the tool **ReWeb** (see Figure 6), which contains the *Spider* module. The operation of state merging of type 2 and 3 (see previous section) is currently performed manually, while type merging of type 1 is automated. If no input is specified for a given dynamic page, the *Spider* will not expand the model beyond the related server program. This feature of **ReWeb** is useful to analyze the functioning of portions of a Web application. Partitioning the tested functionalities by cutting the model at given nodes is important, especially for large Web applications.

As depicted in Figure 6, **TestWeb** contains a test case generation engine (*Test generator*), able to produce a set of paths from the model, according to a user defined criterion, such as prioritization by likelihood, or path generation during a stochastic visit, and to generate test cases from it. Other testing criteria that the user can specify are related to structural (coverage and data flow) testing [17]. Generated test cases are sequences of URLs which, once executed, grant the achievement of the selected criterion. Input values in each URL sequence are those specified for the *Spider*. Such inputs can be marked with the unrolling index of the server program and dynamic page they allow to download. In this way, during testing it is possible to obtain exactly the dynamic page with the structure required to follow a given path. In fact, a page with the needed structure was obtained by the *Spider* by providing the same input values. Such a possibility solves one of the major problems in testing traditional software: selecting the inputs to traverse a path of interest. Testing Web applications is simpler because branch selection can be forced, being associated to the user navigation, which is an external input. Moreover, the existence of the hyperlink to be followed is granted if the dynamic page is obtained under the same conditions in which it was downloaded. This can be achieved by exploiting the same inputs that are used by the *Spider*.

TestWeb's *Test executor* can now provide the URL request sequence of each test case to the Web server, attaching proper inputs to each form. The output pages produced by the server are stored for further examination. After execution, the test engineer intervenes to assess the pass/fail result of each test case. For such an evaluation, she/he opens the output pages on a




```
h1.d1.com  H
h1.d1.com  S & state=1 & corporation="ttt"
h2.d2.com  H
h2.d2.com  S & state=2 & currency="euro"
h1.d1.com  S & state=1 & corporation="bb"
h2.d2.com  S & state=2 & currency="dollar"
h3.d3.com  H
h3.d3.com  S & state=1 & corporation="cc"
h4.d4.com  H
h4.d4.com  S & state=1 & corporation="acb"
h5.d5.com  H
h5.d5.com  S & state=2 & currency="pound"
```

Figure 7. *In the explicit-state model (left), dynamic pages D_1 and D_2 are connected to the additional exit node X. A (simplified) access log is shown on the right.*

browser and checks whether the output is correct for each given input (*functionality testing*). During regression check such user intervention is no longer required, since the oracle (expected output values) is the one produced (and manually checked) in a previous testing iteration. Of course, a manual intervention is still required in presence of discrepancies. A second, numeric output of test case execution is the level of coverage reached by the current test suite. This gives the cumulative probability of path traversal in case of statistical testing, while it gives the percentage of entities (pages, hyperlinks, data flows, etc.) covered, in case of structural testing. In Figure 6, the manual interventions required to reverse engineer and test a Web application are indicated within diamonds.

Transition probabilities

In order to apply statistical testing to a Web application, it is convenient to build its usage model. The *usage model* is a representation of the statistics involved in the executions of a given application and in the input values provided. A natural choice of usage model for a Web application is a Markov chain. In fact, each HTML page can be seen as a state and hyperlinks in the page can be regarded as Markov chain edges leading to other states. The usage model of a Web application can therefore be obtained from its explicit-state model. The only missing information to make it a Markov chain is an estimate of the transition probabilities to be associated with the edges. Values for such probabilities can be computed by using historical information, such as that contained in the log file. It represents the (conditioned) probability of navigating, at the next step, toward another page.

Let us consider the Web application model in Figure 2 (also in Figure 7, with exit node and edge labels added). An example of related access log, here slightly simplified for the sake

Edge	Count	Prob.
e_1	3	3/5
e_2	2	2/5
e_3	4	1
e_4	3	1
e_5	1	1/4
e_7	3	3/4
e_6	1	1/3
e_8	2	2/3

Table I. *Estimation of the transition probabilities.*

of presentation, is given in Figure 7 (right). In particular, time stamps are assumed to be associated to each entry of the log, although not shown. The first column contains the name of the host requesting a Web page. The next column contains the name of the requested page followed by the input provided to the Web server (via `GET`). When requests coming from the same host are found within a proper time interval (10 minutes in our case study, see next section), it is assumed that navigation from a previously accessed page to a new one is taking place. Otherwise, a direct request of a page is considered to occur. Thus, the second request of page S made by `h1.d1.com` is considered to be issued from the page downloaded with the previous request. In other words, while the first request made by `h1.d1.com` causes the traversal of the edges e_1 and e_3, the second request comes from D_1 and results in the traversal of e_5 and e_3. When a request from a host is not followed by any other request from the same host, it is assumed that the edge leading to the exit node is followed. This corresponds to termination of the navigation session. With reference to the log in Figure 7, after the second request of S from `h1.d1.com`, the edge e_7, leading to page X, is marked as traversed, in that no further request issued by the same host is present in the access log (within a reasonable time interval).

The simple analysis of the access log described above allows marking each edge in the Web site model with the number of traversals resulting from the access log. With reference to the example in Figure 7, the edge traversal count reported in the second column of Table I is obtained. Such values can be normalized into relative frequencies, approximating the probabilities of the related Markov chain, by dividing by the sum over the outgoing edges of each node. The result is an estimate of the probabilities of a Markov chain modeling the usage of the site, reflecting the *real world* requests arriving to the Web application. In practice, some page sequences in the access log may be not recognizable as following legal connections between pages, due to the absence of intermediate pages which are cached by the browser and are retrieved by means of commands such as *back*, *forward* and *go-to*. As a first approximation, they can be skipped during the analysis of the access log, giving no contribution to the edge traversal counts. A better approximation can be obtained by modeling the (minimum) sequence

Num.	Path	Prob.	Num.	Path	Prob.
1	$e_1e_3e_7$	9/20	6	$e_1e_3(e_5e_3)^2e_7$	9/320
2	$e_2e_4e_8$	4/15	7	$e_2e_4(e_6e_4)^3e_8$	4/405
3	$e_1e_3(e_5e_3)e_7$	9/80	8	$e_1e_3(e_5e_3)^3e_7$	9/1280
4	$e_2e_4(e_6e_4)e_8$	4/45	9	$e_2e_4(e_6e_4)^4e_8$	4/1215
5	$e_2e_4(e_6e_4)^2e_8$	4/135	10	$e_1e_3(e_5e_3)^4e_7$	9/5120

Table II. *Paths sorted by decreasing probability.*

of browser commands necessary to navigate from a page to its successor in the access log, when no direct link exists between them.

Statistical testing techniques

The Markov chain model of a Web application can be exploited for statistical testing with two purposes:

1. Estimating the reliability of the Web application.
2. Prioritizing the execution of test cases.

To the first aim, test cases are automatically generated according to the statistics encoded in the Markov chain. This is easily achieved by stochastically visiting the chain, i.e. by choosing which edge to traverse in accordance with the transition probabilities of the outgoing edges. The resulting test suite complies with the statistics of the usage patterns and simulates a real usage of the Web application. When test cases are executed and failures occur, the classical measures of reliability can be made, by determining the Mean Time Between Failure ($MTBF$) and estimating the probability R of correct behavior within a given time interval (reliability models) [14]. Such measures can be useful to decide when to stop testing. For the present work, we have used a very simple reliability model, based on the following formulas:

$$R = 1 - \frac{f}{n} \qquad\qquad MTBF = \frac{n}{f}$$

where n is the total number of test cases executed and f is the (estimated) number of failures.

Additional reliability estimates, such as those proposed in [22] are also possible, since a Markov chain is adopted as usage model. They include stopping criteria based on the discriminant, the probability of a failure-free realization of the testing chain and the expected number of steps between failure states.

To the second aim, paths in the Markov chain are ordered according to their probabilities, given by the product of the probabilities on the traversed edges. With reference to the example in Figure 7 and the probabilities in Table I, the paths following e_1 as first edge and looping n times through e_3 and e_5 have probability $3/5(1/4)^n3/4$, while the paths following e_2 as initial edge and looping n times through e_4 and e_6 have probability $2/5(1/3)^n2/3$. The sorted

list of the top ten paths is given in Table II. The next path, not included in the Table, has probability 4/3645 (approx. 0.1%), while the overall probability of the top ten paths is approximately 99.78%. This means that after executing 10 test cases for the paths in Table II, the probability that the user exercises a path not seen during testing is 0.22%.

CASE STUDY

Model recovery

In the following, statistical testing of the Web application **ITC-publications** (`http://-publications.itc.it/`), providing the on-line database of the ITC-irst (our research center) publications, will be presented and discussed. The richness of its dynamic structure makes it an interesting case study for modeling and testing. Papers in the ITC-publications database can be accessed directly from the initial page of the site, `db.htm`, by means of three different forms collecting user inputs. The first form permits a search by **author**, **title**, **reference number** and **abstract**. The second form permits a free text search, referred to all document fields (input variable **query**), while the third form produces a page with links to all the publications available in the database. After submitting the first form, filled-in with correct inputs (for example with abstract `slicing` or author `ricca` and title `web`), the dynamic page `ext-search.idc` appears, showing the list of papers matching the inputs inserted. For each paper, a button, implemented in HTML as a form containing a hidden variable with the paper reference number, is displayed in the resulting Web page. In case of selection of a paper, the dynamic page `ext-view.idc` is loaded. This page contains details on the chosen paper, such as abstract and complete list of authors. Its links permit returning to the initial page and displaying a help page (`help.htm`). The results of submitting the second and third forms are respectively the dynamic pages `ext-search2.idc` and `ext-search3.idc`. The structure of these pages is very similar to `ext-search.idc`. Like `ext-search.idc`, they show the list of papers from which one can be selected, resulting in the generation of the dynamic page `ext-view.idc`. In case the input values inserted into any of the three forms are not correct, the result is a blank page and the only way to return to input insertion is using the back button of the browser and going to the initial page `db.htm`. Figure 8 shows the explicit-state model of the Web application **ITC-publications** as recovered by **ReWeb** with the inputs indicated in Figure 9.

In total, construction of the model required 3 page merging operations. Two page mergings of type 1 have been automatically detected and executed, while the other one (page `ext-search.idc`, type 3), has been identified and performed manually: the three inputs specified by the user belong to a common equivalence class, `author=ricca & title=web`, `abstract=slicing`, and `ref_number=0102-01`, (see Figure 9). The resulting pages have a similar structure, thus suggesting that the related server program generates them in a similar internal state. Actually, these three dynamic pages are produced in exactly the same way by the server program, although with slightly different inputs. No page merging was incorrectly detected by the implemented heuristics.

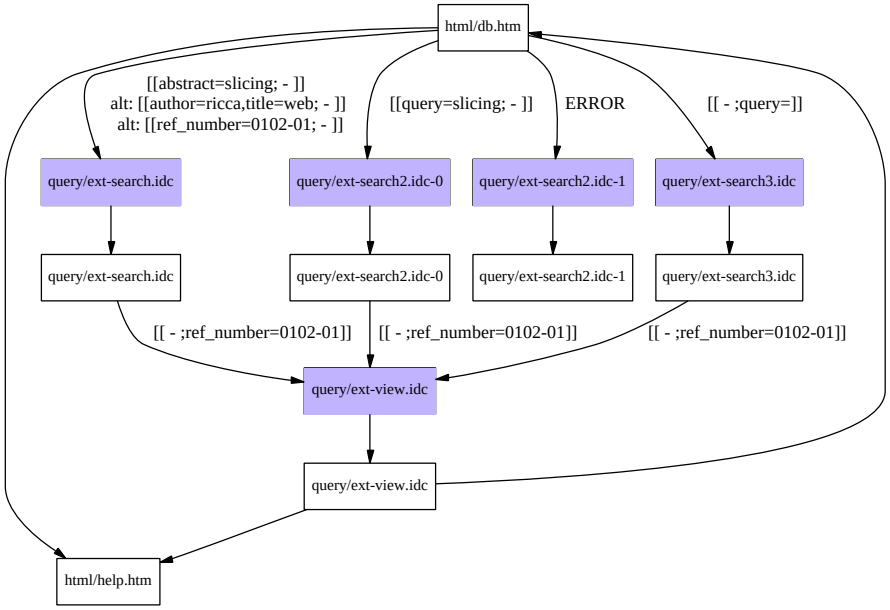


Figure 8. *Explicit-state model of the Web application* **ITC-publications**. *Nodes with gray background are server programs, edge labels contain input variables and hidden variables, separated by a semicolon, within double square brackets. The dash represents the empty list.*

```
(/query/ext-search.idc, abstract=slicing, -)
(/query/ext-search.idc, author=ricca&title=web, -)
(/query/ext-search.idc, ref_number=0102-01, -)
(/query/ext-search2.idc, query=slicing, -)
(/query/ext-search2.idc, query=#ERROR#, -)
(/query/ext-search3.idc, -, query=)
(/query/ext-view.idc, -, ref_number=0102-01)
```

Figure 9. *File* `formInputFile.txt`, *used to download the Web application* **ITC-publications**.

Usage model

In order to apply statistical testing to **ITC-publications**, it is necessary to build its usage model. The server log file of **ITC-publications** has been analyzed to estimate the transition probabilities associated with the model edges. The log file analyzed contains user connections during about two years, from 18-04-2000 to 12-02-2002. Each line of the log file provides, for each user connection, the following information: date, time, name of the host requesting the Web page (IP-number), name of the requested page, HTML status code (404 for resource not found, 200 resource found) and User-Agent (browser) used.

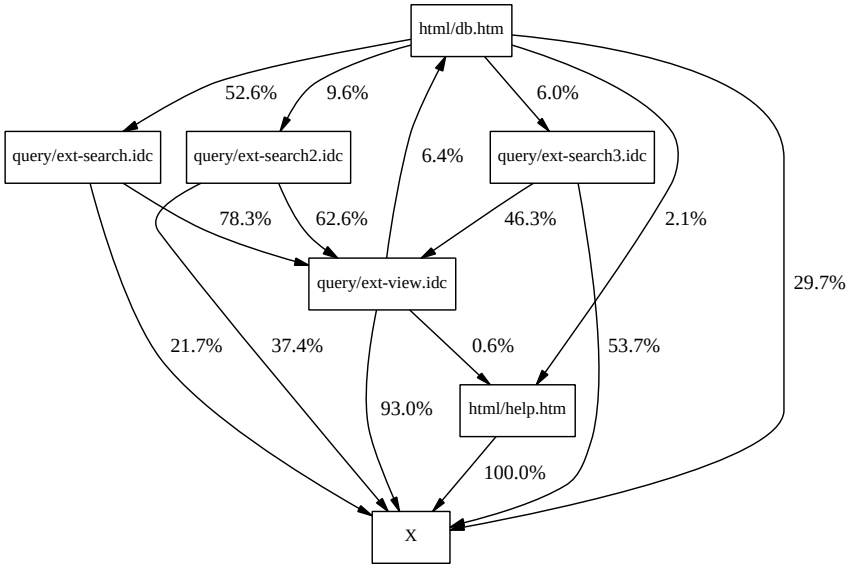


Figure 10. *Usage model of the Web application* **ITC-publications**, *computed ignoring navigation through cached pages.*

The first task was a preprocessing to remove all log entries with suffix gif, jpeg, mpeg etc., i.e. images, sounds and video files. Then all log entries with HTML status code equal to 404 have been deleted as well. The next task was session identification. Sessions can be precisely identified if the logged information includes session/user identifiers (propagated to the server as URL parameters or hidden variables), or, when cookies are used, if the related data are stored in the log file. In absence of such kind of information, a session can be approximated, in a log file, as a set of subsequent accesses to the Web site by a user with the same IP-number and the same User-Agent, within a given time period (time-out). In the session identification phase this value is quite important. In our implementation the time-out is fixed to 10 minutes, a reasonable time in order for a session to be closed. It has been determined by manually estimating the sessions, using the access log information and trying to avoid the loss of a session tail (too short time-out) as well as the mix of two or more separated sessions (too long time-out).

The analysis of the log file highlighted the presence of several consecutive page requests, within a same session, that are not feasible according to the edges in the extracted model. 869 out of 2046 sessions exhibit such a problem. For example, many (precisely, 445) requests of the page `ext-view.idc` are followed by requests of the page `ext-search.idc`, although in the model (see Figure 8) there does not exist an explicit hyperlink between these two pages.

A first usage model (Figure 10) has been computed, ignoring infeasible pairs of requests in the access log. Node X in the usage model represents the end of a user interaction (end of a session). An edge from a given page to X, labeled with probability p, models the case where the user navigation terminates. Its complement (navigation to another page), has thus

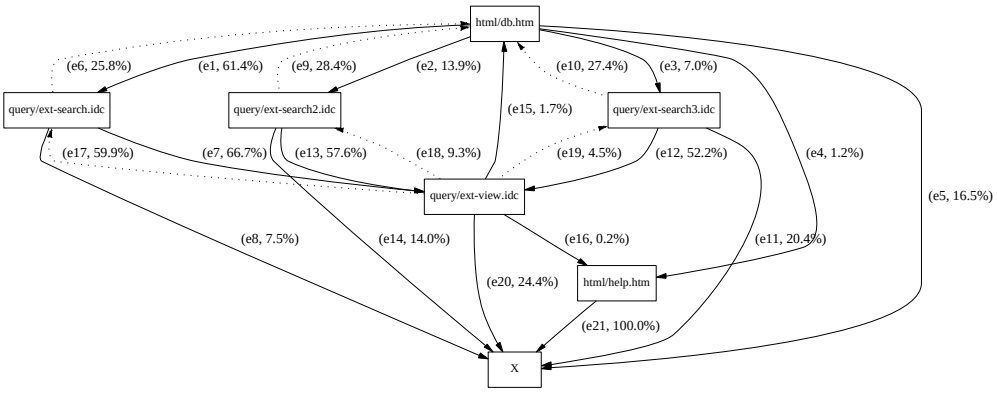


Figure 11. *Usage model of the Web application* **ITC-publications** *with back edges (dotted).*

probability $1-p$, equal to the sum of probabilities over the other outgoing edges. In the usage model, server programs (nodes with grey background in the model) are not represented, since they generate the resulting dynamic pages (shown in the usage model) with probability 1.

By manually inspecting the infeasible pairs of consecutive pages, it was evident that in such cases the navigation performed by the user comprises one or more accesses to the previously visited page (*back* button in most browsers). Since this page is stored in the browser cache, it does not generate any request on the Web server. Correspondingly, no entry is produced in the access log. A simple algorithm has been defined and implemented to estimate the pressure of the back button by the user during the navigation. It works in the following way: when two consecutive pages appear in a same session of the access log for which no edge exists in the model, the preceding pages in the same session are examined, moving backward from the infeasible pair. As soon as a page is encountered that contains a link toward the second page of the pair, the backward traversal is stopped and all traversed pages are used as a completion of the session. Correspondingly, the usage model is augmented with edges that connect a page to (one of) its predecessors, each time a backward traversal is discovered by the algorithm described above. Considering the possibility of backward navigation, a plausible explanation can be given for most of the infeasible sequences of pages in the access log (777 out of 869). For example, the infeasible sequence `db.htm`, `ext-search.idc`, `ext-search` becomes feasible by introducing a backward edge from `ext-search.idc` to `db.htm`, before the last request of `ext-search.idc`. In fact, it is very likely that after a first search (`ext-search.idc` following `db.htm`), the user pushed the back button, with the browser loading `db.htm` from the cache and no request traced in the log file, to perform a new search. Then, after the insertion of the parameters for the new search, the dynamic page `ext-search.idc` is requested for a second time. The (feasible) path followed by the user becomes: `db.htm`, `ext-search.idc`, `db.htm` (back pressed), `ext-search.idc`. Remaining infeasible sequences, not explained by the algorithm above, may be due to direct accesses to a page (*go-to* button or complete address specification) or to imprecise session identification.

Num.	Path	Prob.	Num.	Path	Prob.
1	e_5	16.5%	6	$e_2e_{13}e_{20}$	1.9%
2	$e_1e_7e_{20}$	10.0%	7	e_2e_{14}	1.9%
3	e_1e_8	4.6%	8	$e_1e_7e_{17}e_8$	1.8%
4	$e_1e_7e_{17}e_7e_{20}$	4.0%	9	$e_1(e_7e_{17})^2e_7e_{20}$	1.6%
5	$e_1e_6e_5$	2.6%	10	$e_1e_6e_1e_7e_{20}$	1.6%
				Total	46.6%

Table III. *Paths sorted by decreasing probability.*

Figure 11 shows the new usage model, inclusive of backward edges. The values of the transition probabilities associated with the edges are changed, with respect to the previous usage model, in particular in the case of edges connected to the exit node X. In fact, several sessions terminate with one or more infeasible pairs of pages. Backward edges are new outgoing edges, whose probability was previously zero. Being now greater than zero, it produces a reduction of the probabilities associated with the other outgoing edges (such as the termination edges).

Path selection

Table III shows the ten most probable paths in the usage model, sorted by decreasing probability. The overall probability of the top ten paths is approximately 46.6%. The most probable navigation (topmost path) consists of the traversal of e_5 only, the edge that connects the initial page to X. It accounts for the sessions in which the initial page is loaded and then no further page is accessed. The second topmost path accounts for a paper search (performed by means of `ext-search.idc`), followed by the selection of one entry in the search result, visualized by means of `ext-view.idc`. Then, the session terminates. The third topmost path consists of the same navigation as the second one, without entry selection.

The Total Probability (TP) of a set of paths (such as 46.6% for those in Table III) is the sum of the probabilities of the paths. If paths are interpreted as test cases for the Web application, a set of paths with a high TP is a test suite that provides a statistically more complete test than a test suite with lower TP. Correspondingly, test cases are generated from paths sorted by decreasing probability. By adding more test cases, the TP increases and with an infinite number of test cases this value tends to 100%. Table IV shows the values of TP for an increasing number of test cases (n).

For example, if we execute the 500 most probable test cases, instead of the most probable 20, the TP becomes 80.1% (from 55.3%). This means that after executing the 500 most probable test cases, the probability that the user exercises a path not seen during statistical testing is 19.9% (i.e., 100% - 80.1%). With 20000 test cases the TP becomes 92.2%.

Figure 12 gives a graphical representation of the data in Table IV. The abscissa represents the number of test cases, while the ordinate gives the TP value. It is apparent from the

Test cases (n)	TP	Test cases (n)	TP
10	46.6%	1000	83.3%
20	55.3%	1500	84.9%
30	59.7%	2000	86.0%
50	64.6%	2500	86.8%
100	70.2%	5000	88.9%
200	75.0%	10000	90.7%
300	77.4%	20000	92.2%
500	80.1%		

Table IV. *Total probability of the n most probable paths.*

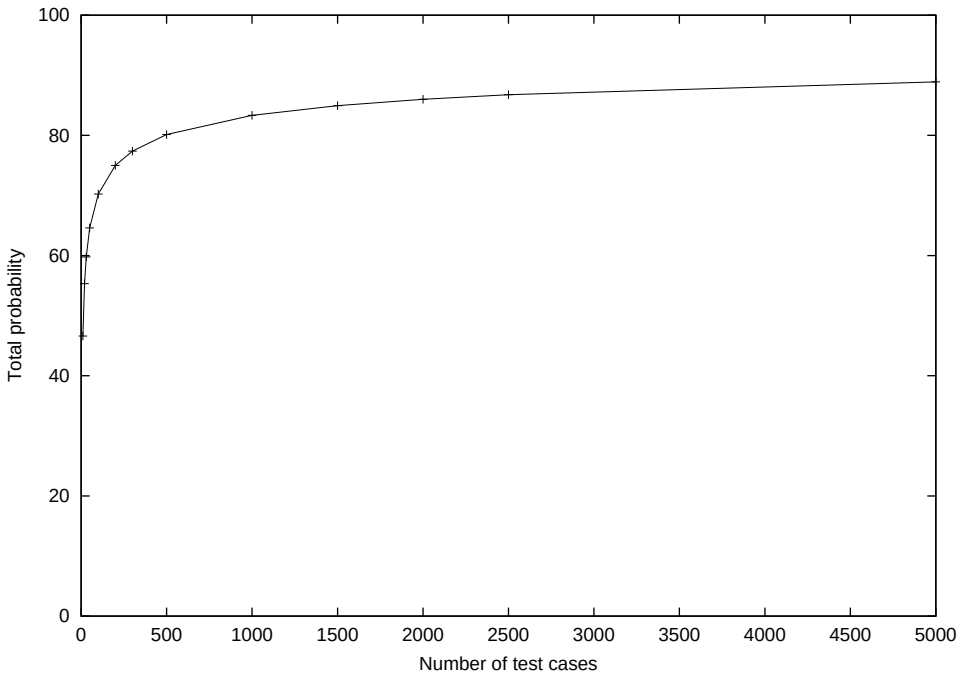


Figure 12. *Plot of the total probability vs. number of test cases .*

plot that beyond 500 test cases, the growth rate of the curve becomes smaller and smaller. Correspondingly the extra effort necessary to produce, execute and check additional test cases is not justified in terms of statistical test thoroughness.

Reliability estimation

To compute the reliability R of **ITC-publications**, one thousand test cases ($n = 1000$) have been generated, by performing a stochastic visit of the Markov chain model of the Web application. Correspondingly, the (estimated) number of failures f is 415.25 (see below), and the resulting reliability R is 58.5% ($R = 1-f/n$). This means that the existing Web application is executed 58.5% of the times without failures. In the remaining cases, one or more failures occur, resulting in a user dissatisfaction. The presence of failures indicates that defects are still present in this Web application. Their impact in terms of impossibility to satisfy a user request is remarkable: according to our user model only 58.5% of the sessions can be completed successfully. In all other cases, the defects in the Web application produce a user observable failure and the session terminates without providing the searched information. Actually, the estimated reliability applies to new users, having no prior experience with this Web application. On the contrary, returning users are expected to be aware of the defect, and correspondingly avoid the bugged input sequence.

Let us consider the observed failures in more detail. Execution of the test case number 2 in Table III highlights an anomalous behavior, associated to a defect of the server program `ext-search.idc`, used by the **ITC-publications** application. When the search is made by **abstract**, a text, expected to appear in the searched paper abstract, is specified in the first form of the initial page. The text specified by the user is assigned to the variable **abstract**. The anomalous behavior occurs each time this variable is assigned a non empty value, regardless of the values of the other variables of this form (collecting author, title and reference number, as other possible inputs). If **abstract** is a non empty string, an no other search criterion is specified, the entire database of publications is reported in the output (a very long list), with no filtering in place at all. If other search criteria are specified in addition to the **abstract**, the latter is ignored: the presence of search data about the abstract is not handled and results in an incorrect output, with no regard to the other form fields.

The main problem with the estimation of the reliability R for this Web application is that a failure occurs each time a search is made through the first form *and* the abstract is specified as search criterion. While our usage model provides an accurate estimate of the probability of a search made through the first form, we have no data from which to estimate the number of times a search includes the specification of the text expected to appear in the abstract. In fact, the values submitted by forms (via `POST`) are not saved in the access log, and no additional logging facility was in place in the time interval considered. We have consequently made a very rough estimate of the probability that a user inserts some data in the **abstract** field. Since the form permits a search by **author**, **title**, **reference number** and **abstract**, there are 16 different combinations of inputs (no field specified, field 1 specified, field 2 specified, ..., all four fields specified). Eight combinations out of 16 have the field **abstract** filled in, while the other 8 do not. If the probability of these 16 cases is considered to be uniform, the probability to fill in the field **abstract** is 50%. The main limitation of this estimate is that it might be

the case that the 16 possible combinations are not equally probable at all. If for example, the insertion of data about the abstract is less likely than the other data, a lower probability of having a failure would be obtained.

Among the considered 1000 paths, 678 submit the first form at least once (in these paths the edge e_1 is contained at least once). When e_1 appears in a path exactly once, the probability of a failure is $p = 0.5$, i.e., the probability of filling in the **abstract** field. For paths with two occurrences of e_1, the probability of *not* producing a failure is $(1-p)^2 = 0.25$ (both occurrences of e_1 do not contain the field **abstract**). Its complement, 0.75, is the probability of a path with failure. With 3 occurrences of e_1 a failure has a probability $1 - (1-p)^3 = 0.875$, and in general, when a path contains k occurrences of e_1, the probability of observing a failure is

$$P[Fail] = 1 - (1-p)^k$$

The mean number of observed failures for the 1000 test cases can thus be computed as the sum of the probabilities $P[Fail]$ over all test cases (with $P[Fail] = 0$ for test cases not including e_1, as results also from the formula above with $k = 0$). This sum is equal to 415.25. If used as the estimated value of f (total number of failures observed), it produces a reliability equal to 58.5%. This means that the analyzed application is expected to fail providing the requested information to the user about half of the times, due to the presence of e_1 in the navigation path and to the specification of the field **abstract**.

Discussion

The structure of a Web application is typically associated to a highly connected graph, which is often one strongly connected component as a whole. Back edges and hyperlinks providing alternative navigations are quite common, making the resulting structure close to a fully connected graph. On the contrary, state/structural models used for statistical testing of traditional software are typically simpler graphs. The main consequence of this difference is that the number of paths that can be followed, with a proper upper bound for the loops, tends to be much higher for Web applications than for traditional applications, since several (unstructured) ways to go back to a same starting point are available in a Web application. When the graph representation of the Web application is used for statistical testing, such a difference has a very negative impact on the ability to reach a high value of the total probability (TP) of the considered paths. In fact, if a loop is traversed k times in a path, the overall probability of the path is equal to the loop free probability multiplied by q^k, where q is the probability of traversing the loop. Since the loop factor q^k decreases exponentially with k, as soon as loop traversals are included in the considered paths, the contribution of additional paths to the TP becomes small. In other words, when the structure of the Web application is strongly connected, loop probabilities tend to make the contribution of new test cases to the TP small. The result is that it is difficult to achieve high values of TP and the user is expected to live with a Web application containing paths that have a non negligible probability of being traversed, but have never been exercised during testing. For Web applications with an internal state having dependences on the specific path followed, this results in a limited efficacy of the test phase. It is thus preferable to limit as much as possible the dependences of the internal

state on the path followed. Ideally, only the immediate predecessor should affect the internal state of the application.

For the present work, the usage model constructed from the access log consists basically of an estimate of the probabilities associated with the possible navigation paths. However, when the user navigates along a given path, input values are inserted into the forms encountered during the navigation. A more complete usage model should include also an estimate of the probabilities that a given input (or an input from a given equivalence class) be inserted into a form. In fact, the occurrence of a failure depends both on the followed path and on the inserted inputs. The absence of information about the probability of the inputs affects the statistical significance of the test cases that are produced, and the estimate of the reliability of the Web application. As regards the test cases, they should cover the most probable paths as well as the most probable inputs, while in our work we have been able to pursue only the first objective. As regards reliability, if inputs are considered uniformly distributed, both an overestimate and an underestimate are possible. If the inputs generating the failures are less likely than estimated according to a uniform distribution, reliability will be higher than estimated, Vice versa, it will be lower if failure inputs are more likely than in the uniform distribution.

RELATED WORK

Various Web application modeling methods have been proposed for different purposes, such as testing [8, 9, 11], architecture recovery [5, 10] and design [2, 6]. Each method emphasizes some particular aspect of a Web application. The representation adopted in [10] is similar to the one proposed in [2]. The latter work is focused on forward engineering and is suited for the high level specification of a Web application, while the conceptual model proposed in [10] is focused on a reverse-engineering process. In fact, it better highlights the behavior of and the dynamic interaction between the elements in an existing Web application. In the adopted representation, differences between static client pages and dynamic client pages, passive Web objects (e.g. images) and active Web objects (e.g. scripts) are remarked, and interface objects (i.e., objects that interface the Web application with a DBMS or an external system) are added. On the contrary, our model aims at explicitly representing user navigations. Consequently, internal entities such as interface objects, and passive and active Web objects, are not considered.

Recently, some testing tools have been proposed to support functional testing of Web applications [7, 11, 12]. The black-box testing tool proposed in [12] is based on capture/replay facilities: it records the interactions that a user has with the graphical interface and repeats them during regression testing. The tool described in [7] supports the specification of test cases in XML and provides powerful facilities to encode the related oracles. Operations such as retrieving a node in the abstract syntax tree of the resulting page and checking a value (also by means of regular expressions) are available. The authors of [11] exploit a slighlty different format for the specification of test cases, based on decision tables that include input values for each execution variant and expected outputs. Execution is automated by their tool **WAT**. We share with these works the ability to automate test case execution and the focus on functionality testing, but differently from them we take also navigation statistics into consideration for test case construction.

In [9] traditional data flow testing techniques are extended 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 to compute 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.

A statistical technique is proposed in [1] for the automatic selection of the paths to be exercised in a static Web site. Similarly to our work, the number of invalid links encountered along the test paths allows estimating the site reliability, i.e., probability that a user completes the navigation without errors.

Statistical Web testing has been proposed in [8]. The main differences between our method and theirs are in the model extracted and in the type of failures considered. The approach in [8] is based on UMMs (Unified Markov Models), generated using the **FastStats** log file analysis tool. This tool analyses the data in the access log and produces the *hyperlink tree view*, a tree-structured graph that shows the Web site architecture as well as the direct hits (edge traversal count) from parent pages to child pages. In contrast, we construct an explicit-state model, which captures also the dynamic aspects of a Web application (ignored in [8]), and we perform functionality testing. The usage model of our approach, similar to their UMMs, is obtained in a second time, by decorating the explicit-state model with probabilities deduced from the log file. The failures considered by their method are those present in the access log and error log, i.e., failures such as "permission denied" (accounting for unauthorized access to a Web resource) or "file does not exist" (the requested file was not found in the system). Our technique permits to discover these types of failure. In addition, our technique detects the behavioral deviations from the user expectations (functionality testing) not reported in log files, as follows: the output pages are inspected by the test engineer to assess whether the test cases have been passed or not, going beyond the information about static failures reported in the log files.

Statistical analysis of the user access to Web resources is widely exploited in Web usage mining [4, 13, 15, 20]. The purpose of these works is quite different from ours. Data mining techniques are applied to discover usage patterns from Web data, such as those recorded in a (possibly extended) log. Obtained information is useful for Web personalization, recommendation, and navigation improvement.

In our previous works [16, 17, 18, 19] we focused on Web model definition and construction, Web reengineering, and structural testing. Our Web application model has been described in [16, 17]. Static analysis based on such a model are proposed in [16], while white box, coverage testing is investigated in detail in [17]. However, the algorithm for the dynamic recovery of a Web application model is not thoroughly described in neither of the previous works [16, 17]. Testing processes are discussed in [19] with reference to the typical life cycle of a Web application. Examples of analyses and transformations aimed at Web restructuring are provided in [18]. The present work, which extends [21], builds upon our previous works on modeling and testing. Its novel contribution is in the analysis of the navigation statistics

in order to drive testing. Moreover, it provides detailed algorithms for dynamic model and Markov chain construction, as well as a preliminary empirical assessment.

CONCLUSIONS AND FUTURE WORK

A model of dynamic Web applications has been presented and used for the definition of statistical testing techniques. The model can be extracted from an existing Web application by means of a semi-automatic procedure, requiring user involvement only in the specification of a set of input values covering all the internal states of the application. Partial models can still be constructed and are still of interest whenever full coverage of all possible internal states is not granted by the available set of inputs.

The proposed model has been recovered for an existing Web application. Information available from the access log has been used to estimate transition probabilities. A stochastic visit of the resulting Markov chain has been performed to generate test cases in agreement with the typical usage patterns, leading to an estimate of the reliability of the analyzed application, which was quite low, due to a failure occurring along a quite frequently accessed path. The statistical information encoded in the Markov chain model has been exploited also to prioritize test cases, from those exercising the most likely paths to the less likely ones.

A critical review of the results obtained on the case study highlights some directions for our future research. A test case is associated to a path in the Web application model. In the Web application we considered for testing, the ability of a test case to reveal a defect depended only and exclusively on the presence (absence) of a given edge of the model in the path, and on the input values inserted into the source page of such an edge, with no regard to the path length and to the input values inserted previously. The reason for this is that the considered application has a very simple state model, where the internal state is directly induced by the values inserted into a form, and it is reset after the execution of `ext-view.idc`, which is at most at distance 2 from the form. Previous internal states have no effect on the current one and the current one does not influence the future states, if outside the 2-step path. In such cases, it makes no sense to consider paths of length greater than 2, since they are functionally equivalent to those of length 2. In other words, it is possible to split the given Web application into a set of smaller sub-applications, consisting of three pages each: a page with a form to be filled in for the search (with the form varying according to the search criterion), the dynamic page with the search result, and the dynamic page produced by selecting an entry from the search result. Testing of the sub-applications is of course much less expensive than testing the whole application (full coverage of all paths is easily achieved for the sub-applications). The presence of a very local state of a Web application is, in our opinion, a quite general feature of several existing applications. This is partially explained by the nature of the interactions mediated by the Web, which have typically the form of an input submission followed by a resulting page, with the possibility to interrupt the navigation at every moment. Moreover, ensuring the expected behavior in case of long chains of dependences is a difficult task, due to the (typically) high connectivity of the Web application structures (see 'Discussion' in the CASE STUDY section). The possibility to identify independent portions of a Web application

that can be tested separately seems to be an important issue, and will be the subject of our future work.

Another aspect where the present work can be improved is the usage model constructed for a Web application. Currently, we are not modeling the statistics of the input values inserted by users into forms, in case of submission via `POST`, since such values are not recorded in the access log. However, the results obtained on our case study indicate that an accurate estimate of the reliability can be obtained only if the usage model includes the statistics of the input values. In order to obtain them, it is necessary to develop an instrumentation tool to automatically log the inputs submitted by the user when requesting the generation of a dynamic page. This is also a topic of our future research.

REFERENCES

1. W. K. Chang and S. K. Hon. A systematic framework for ensuring link validity under web browsing environments. In *Proc. of the 13th International Software/Internet Quality Week*, San Francisco, California, USA, 2000.
2. J. Conallen. *Building Web Applications with UML*. Addison-Wesley Publishing Company, Reading, MA, 2000.
3. D. Eichmann. Evolving an engineered web. In *Proc. of the International Workshop on Web Site Evolution*, Atlanta, GA, USA, October 1999.
4. Andreas Geyer-Schulz and Michael Hahsler. Evaluation of recommender algorithms for an internet information broker based on simple association rules and on the repeat-buying theory. In Brij Masand, Myra Spiliopoulou, Jaideep Srivastava, and Osmar R. Zaiane, editors, *Fourth WebKDD Workshop: Web Mining for Usage Patterns & User Profiles*, pages 100–114, Edmonton, Canada, July 2002.
5. A. E. Hassan and R.C. Holt. Architecture recovery of web applications. In *Proc. of International Conference on Software Engineering*, Orlando, Florida, USA, May 19-25 2002.
6. T. Isakowitz, A. Kamis, and M. Koufar. Extending RMM: Russian dolls and hypertext. In *Proc. of HICSS-30*, 1997.
7. Xiaoping Jia and Jordan Hongming Liu. Rigorous and automatic testing of web applications. In *Proc. of the DePaul CTI Research Symposium (CTIRS)*, Chicago, USA, November 2001.
8. C. Kallepalli and J. Tian. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, November 2001.
9. 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.
10. G. A. Di Lucca, M. Di Penta, G. Antoniol, and G. Casazza. An approach for reverse engineering of web-based application. In *Proc. of the 8th Working Conference on Reverse Engineering (WCRE)*, Stuttgart, Germany, October 2001.
11. Giuseppe A. Di Lucca, Anna Rita Fasolino, Francesco Faralli, and Ugo De Carlini. Testing web applications. In *Proc. of the International Conference on Software Maintenance (ICSM)*, Montreal, Canada, October 2002. IEEE Computer Society.
12. Edward 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.
13. Bamshad Mobasher, Robert Cooley, and Jaideep Srivastava. Automatic personalization based on Web usage mining. *Communications of the ACM*, 43(8):142–151, 2000.
14. J. D. Musa. *Software Reliability Engineering*. McGraw-Hill, NY, 1998.
15. Mohammad El-Ramly Nan Niu, Eleni Stroulia. Understanding web usage for effective dynamic web-site adaptation. In C. Boldyreff and P. Tonella, editors, *Fourth International Workshop on Web Site Evolution*, pages 53–62, Montreal, Canada, October 2002. IEEE Computer Society.
16. F. Ricca and P. Tonella. Web site analysis: Structure and evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 76–86, San Jose, California, USA, 2000.
17. F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proc. of ICSE 2001, International Conference on Software Engineering, Toronto, Ontario, Canada, May 12-19*, pages 25–34, 2001.

18. F. Ricca, P. Tonella, and I. Baxter. Web application transformations based on rewrite rules. *Information and Software Technology*, 44(13):811–825, 2002.
19. Filippo Ricca and Paolo Tonella. Testing processes of web applications. *Annals of Software Engineering*, 14:93–114, 2002.
20. Jaideep Srivastava, Robert Cooley, Mukund Deshpande, and Pang-Ning Tan. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explorations*, 1(2):12–23, 2000.
21. Paolo Tonella and Filippo Ricca. Dynamic model extraction and statistical analysis of web applications. In *Proc. of the International Workshop on Web Site Evolution (WSE)*, pages 43–52, Montreal, Canada, October 2002. IEEE Computer Society.
22. J. A. Whittaker and M. G. Thomason. A markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, October 1994.