Dynamic Model Extraction and Statistical Analysis of Web Applications

Paolo Tonella and Filippo Ricca
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
{tonella, ricca}@itc.it

Abstract

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.

Reverse engineering of a model from an existing Web application is useful for its understanding and evolution. However, a static analysis of its 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. Moreover, static analysis requires the ability to process multiple languages.

In this paper, a dynamic analysis technique is proposed for the extraction of a Web application model through its execution. The HTML code produced during execution on proper input values is subject to a static analysis. Availability of statistical data about the accesses to the pages produced by the Web application is exploited for statistical testing and for the analysis of the navigation habits of the users. Anomalous behaviors can be detected and indicated for an improvement intervention.

1 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 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 UML (Unified Modeling Language [1]) 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. Moreover, navigation statistics can be analyzed, to determine the average number of hyperlinks followed before reaching a given page, and the usage of navigation facilities offered by the browser instead of the hyperlinks of the site.

The proposed model can be semi-automatically extracted from an existing Web application, by exploiting the algorithms detailed in Section 2. Section 3 deals with statistical analysis and testing, for which the Web application model is enriched with statistical data. Related works are discussed in Section 4, while conclusions and future work are presented in the last section of the paper.

2 Model extraction

2.1 Web application modeling

The aim of the 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 the Web application.

The Web site model closer to ours is that proposed by Conallen [2]. Web pages are considered first-class elements, and are represented as objects, using UML. Similarly, all other architecturally relevant entities as, for example, links, frames, and forms, are explicitly indicated in the model. The main difference between Conallen's and our UML model of Web applications is in the emphasis given to design vs. analysis. In fact, the model by Conallen aims at describing the site from a logical point of view, as required when it is being designed. On the contrary, we focus our model on the implementation of the site, which is the starting point for analysis and testing.

In the following, a Web application is identified with the set of pages that can be accessed from a given Web server. Documents accessed from different servers are considered external to the given application. Testing of a Web application as well as the extraction of its model are supposed to be conducted in the development environment. Consequently, some information that cannot be accessed by external users browsing the site is considered available.

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 a *Form* and is passed to a server program, which processes it, through a *submit* link. A Web page can include any number of forms; accordingly, the cardinality of this link is arbitrary. Each

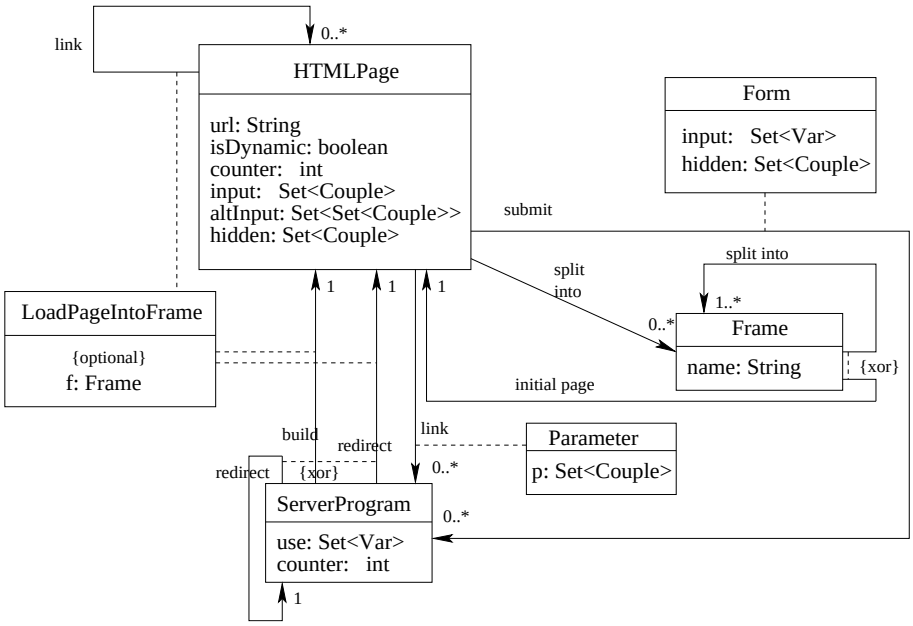

Figure 1. *Meta model of a generic Web application structure. The model of a given application is an instance of it.*

form is characterized by the input variables that are provided by the user through it (data member *input*). 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 (a variant is represented by the cookies).

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.

Server programs in the first category always exploit the same mechanism to produce the output (either auto-redirect, redirect or build) and generate 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, the paths in the model can still be interpreted as navigation sessions, provided that server program and related output page are replicated for all the possible 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 (*counter* in classes *HTMLPage* and *ServerProgram*). 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.

In order to define a set of analysis and testing techniques working on Web applications, it is convenient to re-interpret the model described above as a graph, whose nodes correspond to the objects in the model and whose edges correspond to the associations between objects. Labeled edges are used for the links having a *LoadPageIntoFrame, Form*, or *Parameter* relation specifier. Since the adopted model is aimed at representing navigation explicitly, the paths in the associated graph can be regarded as interaction sessions in which the user navigates inside the site, provides input data through forms and receives the results back through dynamic pages.

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

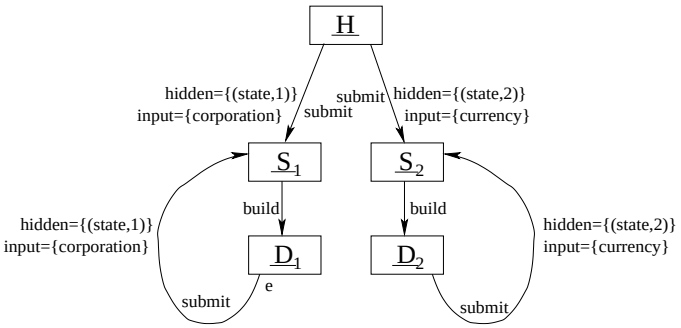

Figure 2. *Example of Web application model.*

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* association 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. To simplify the plot, the association class *Form* is not shown explicitly and its fields label the association edge. 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.

2.2 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** [11], 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 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 UML 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 a specific procedure, *DownloadDynamicPage*, described below. 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 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 UML 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, some page merging heuristics, discussed below, can be employed to try to automatically recognize equivalent dynamic pages and thus equivalent instances of the related server programs.

The procedure *DownloadDynamicPage*, sketched in Figure 4, extracts the input to be inserted into a given form

```
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              UML_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 = UML_model.recoverPageAlreadyInModel(d)
25                      UML_model.addEdgeToModel(p,q)
26                      addAltInput(q,d.input)
27                  else
28                      UML_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 3. *Pseudocode of the SPIDER procedure, that downloads a target Web site and builds the related UML model.*

from a persistent store (e.g., the text file `formInputFile.txt`), by looking for lines with a matching action (instr. 2: `line=f.action, ...`), and builds an object of type *HTMLPage*, 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 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. This knowledge may not be available a priori. In such cases, an additional operation of page merging is required to unify equivalent instances of the server programs. Three increasingly weaker page merging heuristic criteria can be used to simplify the explicit state model downloaded 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.

```
// 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 4. *Pseudocode of the* DownloadDynamicPage *procedure, that downloads a dynamic page providing proper input values to the server program.*

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 merged with the corresponding one in the model. With reference to Figure 3, page merging is checked 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 and the careful selection of the input values, as representative of the equivalence classes of inputs, results (according to the authors' experience) in a finite model.

An example of input file to be used for the extraction of the model in Figure 2 is shown in Figure 5. The *Spider* module starts its computation by adding the initial page

```
(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.

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 association 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.

3 Statistical analysis and testing

3.1 Transition probabilities

In order to apply statistical analysis and 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 UML 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.

```
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 6. *Example of simplified access log.*

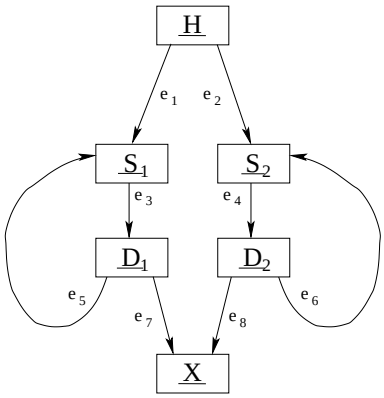

Figure 7. *Dynamic pages D_1 and D_2 are connected to the additional exit node X.*

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 of presentation, is given in Figure 6. 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, 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 6, 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 and the access log in Figure 6, the edge traversal count reported in the second column of Table 1 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, or to the usage of browser 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.

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 1. *Estimation of the transition probabilities.*

3.2 Statistical analysis

Properties of the Markov chains can be exploited to analyze a Web application model from a statistical point of view. Specifically, it is possible to determine the probability that the user be in a given Web page (stationary probabilities) and the average length of a path leading to each Web page from the initial one.

Stationary probabilities account for the probability of being in a given state, in an arbitrary number of Markov chain traversals. They are obtained by solving the following set of equations:

$$\pi_j = \sum_i \pi_i U_{ij} \tag{1}$$

where U_{ij} is the transition probability from state i to j.

The mean number of state transitions that occur before reaching a given state j from state i, in an arbitrary number of interactions, is obtained by solving the following set of equations:

$$m_{ij} = 1 + \sum_{k \neq j} U_{ik} m_{kj} \tag{2}$$

$$m_{jj} = \frac{1}{\pi_j} \tag{3}$$

With reference to the example in Figure 7 and the transition probabilities in Table 1, the stationary probabilities and

Page(s)	Prob.	Avg	Short.	Ratio
H	5/17	17/5	0	-
S_1, D_1	4/17	10/3	1	3.3
S_2, D_2	3/17	6	1	6.0
X	5/17	12/5	2	1.2

Table 2. *Stationary probabilities; average and shortest paths.*

the average paths reported in Table 2 are determined. Page H has the highest stationary probability (5/17) (equal to that of X), giving the probability that during an arbitrary interaction the user be in page H. The average number of transitions to be performed before reaching pages S_1 and D_1 (they are not distinguished since the former generates the latter, without any user interaction) is 10/3 (approx. 3.3). It is associated to a set of interactions that can either start from H and proceed along edge e_2, never reaching S_1, or start from H and reach S_1, traversing exactly 1 edge before S_1 is encountered. The average is 3.3 transitions (corresponding to mouse clicks) for S_1, and 6 for S_2. The shortest path to both pages is 1. As expected, it is lower than the average path.

The statistical analysis described above can be useful to determine the most/least accessed pages (stationary probabilities). Moreover, the comparison between average and shortest path to a page (see Table 1, last column) may suggest improvement interventions on the site structure. In fact, if the ratio between average and shortest path is high, navigation toward the given page appear to be requested less frequently than designed in the site structure. Possible actions are moving the page far from the home page or increasing its visibility in the home page, so that more users reach it. On the other side, a low ratio between average and shortest paths for pages which have a medium/long shortest path (say, more than 3-4 clicks) indicate that navigation toward such pages is frequently performed along the shortest path. If such pages are requested often (see stationary probabilities), it may be appropriate moving them closer to the home page (or introducing shortcuts to them).

Another interesting statistics that can be derived from the analysis of the access log is about the navigation modes followed by the user. In fact, the presence of requests of pages not connected by a hyperlink from a same host (and within a reasonable time interval) indicates that the user performed part of the navigation by means of browser commands (such as *back, forward* and *go-to*). Then, a new page was requested to the Web server. The presence of a high number of navigation sequences exploting browser commands may indicate that the structure of the site does not include important hyperlinks to some of the previous/related pages.

3.3 Statistical testing

The UML model, enriched with transition probabilities, 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 of correct behavior within a given time interval (reliability models) [10]. Such measures can be useful to decide when to stop testing. Additional reliability estimates, such as those proposed in [12] 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.

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

Table 3. *Paths sorted by decreasing probability.*

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 1, 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 3. 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 3, the probability that the user exercises a path not seen during testing is 0.22%.

4 Related work

Various Web application modeling methods have been proposed for different purposes, such as testing [6, 7], architecture recovery [4, 8] and design [2, 5]. Each method emphasizes some particular aspect of a Web application. The representation adopted by Di Lucca et al. [8] is similar to the one proposed by Conallen in [2]. Conallen's work is focused on forward engineering and is suited for the high level specification of a Web application, while the conceptual model proposed by Di Lucca et al. is focused on a reverse-engineering process. In fact, it better highlights the behavior of and the dynamic interaction between the elements in the Web application. In Di Lucca et al.'s 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 the interface objects are not considered.

Recently, a few testing tools have been proposed to support functional testing of Web applications (e.g., [9]). These black-box testing tools are based on capture/replay facilities: they record the interactions that a user has with the graphical interface and repeat them during regression testing.

Liu et al. [7] 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 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.

Statistical Web testing has been proposed by Kallepalli and Tian in [6]. The main differences between our method and theirs are in the model extracted and in the type of failures considered. Kallepalli and Tian's approach 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 UML model, which captures also the dynamic aspects of a Web application (ignored in [6]), and we perform functionality testing. The usage model of our approach, similar to their UMMs, is obtained in a second time, by decorating the UML 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.

5 Conclusions and future work

A model of dynamic Web applications was presented and used for the definition of statistical analysis and 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 resulting model solves by construction some of the typical problems related to traditional software testing. In fact, branching is not conditional to the input values for an HTML page, and the internal state that allows traversing a given path can be forced by exploiting the same values that were used for model construction. As a consequence, test case generation (and execution) can be automated.

Some experimental and implementative work is still to be done. The page merging heuristics proposed as a way to recognize the occurrence of a previously encountered behavior of a server program from its output need an empirical validation. Moreover, the effectiveness of the statistical analysis and testing approach has no yet been assessed experimentally. It requires the implementation of a tool for the estimation of the transition probabilities from the access log. Although apparently this is an easy task, there may be difficulties related to the several levels of caching interposed between client and server, such that part of the user navigation is not visible in the access log and has to be re-

constructed heuristically. Moreover, the input values passed to the server programs are not visible in the access log if parameter passing is by `POST` instead of `GET`: in such cases, additional tracing mechanisms have to be incorporated into the Web server.

The source code running on the browser (e.g., Javascript and Applets) and that executed by the Web server are not analyzed currently. Their analysis would allow treating these components as white-boxes, similarly to the Web pages, instead of considering them as black-boxes, highly improving the defect-detection capabilities offered by the proposed testing techniques. Finally, an interesting (and difficult) research area is related to the automatic generation of the input values used during model extraction and successively during testing.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language – User Guide*. Addison-Wesley Publishing Company, Reading, MA, 1998.

[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] A. E. Hassan and R. Holt. Architecture recovery of web applications. In *Proc. of International Conference on Software Engineering*, page (to appear), Buenos Aires, Argentina, May 19-25 2002.

[5] T. Isakowitz, A. Kamis, and M. Koufar. Extending rmm: Russian dolls and hypertext. In *Proc. of HICSS-30*, 1997.

[6] 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.

[7] 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.

[8] G. A. D. Lucca, M. D. 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*, Stuttgart, Germany, October 2001.

[9] 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.

[10] J. D. Musa. *Software Reliability Engineering*. McGraw-Hill, NY, 1998.

[11] 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.

[12] 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.